Algorithmique parallèle

Yves Robert

Année 2000-01

Table des matières

Chapitre 1

Introduction

Un vieux proverbe de l'Ouest assure qu'il est plus facile de conduire un attelage avec deux boeufs qu'avec mille poulets. Faire coopérer un grand nombre de processeurs standards, agencés en grappes et reliés par un réseau plus ou moins rapide, s'avère en effet nettement plus complexe que d'utiliser un super-calculateur Cray monolithique, encombrant et ruineux. Mais c'est tellement plus amusant ! En plus c'est l'avenir, alors pourquoi se priver du plaisir de l'algorithmique parallèle ?

Sont rassemblées dans ce document quelques notes de cours, qui viennent en complément du polycopié d'Eric Goubault [39]. Le lecteur ne trouvera pas ici une introduction complète au domaine, pour laquelle nous renvoyons au polycopié cité, ainsi qu'à des ouvrages généraux traitant des algorithmes et des architectures parallèles : voir les livres de Gengler, Ubéda et Desprez [35] (en français), ou ceux de Kumar et al. [46], Hwang et Xu [42], et Culler et Singh [27] (en anglais).

Tous les chapitres de ce polycopié sont largement indépendants, qu'ils traitent de modèles, d'algorithmique, ou de techniques de parallélisation :

Partie 1 : Modèles

C'est une partie théorique certes, mais les modèles, même éloignés de la réalité, sont indispensables pour bâtir les fondements d'une discipline.

Chapitre 2 : Machines P-RAM Ce chapitre traite des machines P-RAM. P-RAM signifie Parallel RAM, un modèle où les processeurs peuvent accéder simultanément à une (grosse) mémoire commune en une unité de temps, sans payer de coût de communication. Le chapitre propose d'abord des exemples d'algorithmes simples, mais se poursuit sur la machine à trier de Cole : pour trier par tri-fusion deux suites de taille n en temps $O(\log n)$, et comme il y a $\log n$ étapes de fusion, il faut savoir faire la fusion en temps constant ! pas facile du tout, mais très joli.

Chapitre 3 : Réseaux de tri Ce chapitre aborde les réseaux de tri, un autre modèle de calcul classique, moins général que les P-RAM, mais plus réaliste, du moins si l'on considère qu'un circuit spécialisé pour le tri peut avoir un intérêt pratique. On commence par le réseau de tri-fusion de Batcher, qui fait le pendant de la machine P-RAM de Cole du chapitre précédent, et on finit par le réseau de transposition pair-impair et ses variantes.

Chapitre 4 : Ordonnancement Ce chapitre présente quelques résultats élémentaires pour l'ordonnancement des graphes de tâches. On considère d'abord un modèle très simple, où les coûts de communications sont négligés : le problème sans limitation de ressources est polynômial, tandis que celui à ressources bornées est NP-complet, et résolu de manière sous-optimale à l'aide d'heuristiques de listes. On passe alors à un modèle plus réaliste, où les coûts de communication sont pris en compte : la difficulté s'accroît notablement, car même le problème sans limitation de ressources devient NP-complet.

Partie 2 : Algorithmique parallèle

Quelques grands classiques ! on prend un algorithme bien connu, comme le produit de deux matrices ou la décomposition LU, et on le tord dans tous les sens sur un anneau, une grille 2D, un hypercube, ou encore une grappe de processeurs hétérogènes. Le but est d'analyser les facteurs qui limitent l'efficacité (déséquilibrage de charge, coût des communications, inactivité forcée due aux dépendances), et de mener des analyses de complexité, ou de scalabilité.

Chapitre 5 : Algorithmique sur un anneau de processeurs La topologie la plus simple va nous permettre de concevoir et écrire nos premiers algorithmes parallèles, sans rien connaître aux architectures de machines. On commence par les primitives de communications globales, et on enchaîne sur le produit matrice-vecteur. Même sur un anneau, l'algorithmique peut se compliquer, comme le prouvent les exemples de la transformée en distance et de la factorisation LU.

Chapitre 6 : Communications et routage On décrit quelques réseaux d'interconnexion classiques, et les mécanismes de routage les plus usuels. Deux études de cas sont approfondies : l'analyse des propriétés topologiques de l'hypercube, et l'étude de plusieurs algorithmes pour la multiplication de deux matrices carrées, tous destinés à s'exécuter sur des grilles toriques 2D de processeurs.

Chapitre 7 Ce chapitre présente quelques algorithmes parallèles destinés à s'exécuter sur un réseau de stations hétérogène, i.e. dont les processeurs travaillent avec des vitesses différentes. Autant l'équilibrage de charge 1D reste facile (nous re-visitons les algorithmes sur l'anneau du Chapitre 5), autant l'équilibrage de charge 2D s'avère complexe (retour au produit de matrices du Chapitre 6). Le chapitre se conclut avec l'étude du produit de matrices sur un réseau hétérogène : l'équilibrage de charge redevient facile, puisqu'on peut toujours configurer le réseau en un anneau virtuel, mais l'impact des communications doit être pris en compte. NP-complétude et heuristiques sont au menu.

Partie 3 : Techniques de compilation

Les derniers chapitres sont consacrés aux techniques de compilation, dont l'importance va aller grandissante avec l'arrivée de micro-processeurs dotés de multiples unités de calcul. Le parallélisme doit d'abord être détecté, à partir de l'analyse des dépendances dans le programme, puis le code doit être transformé pour faire apparaître des boucles parallèles :

Chapitre 8 : Nids de boucles Ce chapitre est consacré aux techniques d'extraction du parallélisme dans les nids de boucles imbriquées. On commence par l'analyse de dépendance, et les différentes abstractions des dépendances : on poursuit par la présentation des algorithmes classiques de transformation de code : distribution/fusion de boucle et algorithme d'Allen-Kennedy-Allen, transformations unimodulaires et algorithme de Lamport.

Chapitre 9 : Pipeline logiciel Il s'agit d'ordonnancer un ensemble de tâches génériques qui vont être exécutées un grand nombre de fois, parce qu'elles figurent dans un corps de boucle. Il s'agit de ré-ordonner les tâches de manière à utiliser au mieux les ressources existantes, d'où le nom de pipeline logiciel. Ce chapitre utilise à la fois les résultats du Chapitre 4 sur l'ordonnancement, et du Chapitre 8 sur l'analyse de dépendances. NP-complétude et heuristiques garanties sont (encore et toujours !) au menu.

Ces notes de cours sont appelées à s'enrichir au fil des ans : vos commentaires, remarques, suggestions, et *bug reports* sont attendus à l'adresse `Yves.Robert@ens-lyon.fr`. Bonne lecture !

Remerciements

Les chapitres 4, 8 et 9 sont issus du livre *Scheduling and Automatic Parallelization*, dont les auteurs sont A. Darte, Y. Robert et F. Vivien [28]. Merci à mes co-auteurs Alain et Frédéric de m'autoriser à utiliser cette excellente source :-)

Merci aussi à F. Desprez, à qui j'ai emprunté plusieurs paragraphes du Chapitre 6, et à F. Rastello, dont la thèse a fortement inspiré le Chapitre 7. Merci enfin à O. Beaumont, avec qui je partage le cours *Algorithmes et architectures parallèles* à l'ENS Lyon, et qui a relu une bonne partie de ce pensum.

Merci enfin à R. Cori et S. Toueg de m'avoir donné l'occasion d'enseigner un cours de parallélisme dans le cadre de la Majeure 2 d'Informatique. Le cours est partagé avec E. Goubault ; pour des raisons de calendrier, nous n'avons pas pu rédiger le polycopié ensemble. Mais ce sera chose faite l'an prochain, promis !

Première partie

Modèles

Chapitre 2

Modèle P-RAM

2.1 Introduction

La machine de Turing est un modèle classique pour le calcul séquentiel, qui permet de donner une définition *précise* du coût d'un algorithme, et, partant, est à la base des études de complexité (polynômial vs. NP-complet, etc).

En parallélisme, la recherche d'un modèle général se heurte à la variété des architectures à considérer. Comment définir le coût d'une communication ? Finalement, il s'est avéré que le plus raisonnable était ...de l'ignorer ! On obtient alors un modèle imparfait, et le coût d'un algorithme avec ce modèle n'aura qu'un lointain rapport avec son coût d'exécution sur machine réelle. Mais ce modèle permet tout de même de donner une *classification* des algorithmes parallèles, et de mener des analyses de complexité (pour montrer qu'un algorithme donné est optimal, ou pour établir la complexité minimale d'un problème). En un mot, le modèle permet d'extraire, ou de déterminer, le parallélisme *maximal* contenu dans un algorithme ou un problème.

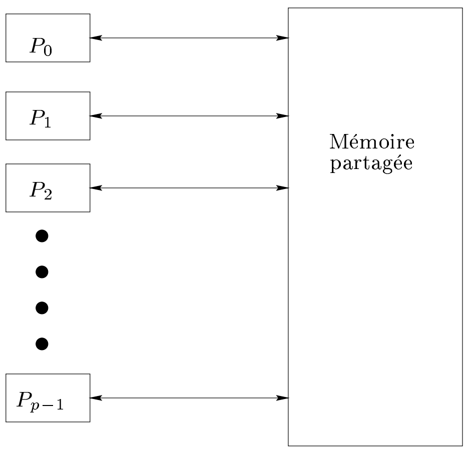


Fig. 2.1 – Schéma d'une machine P-RAM.

Le modèle P-RAM, pour Parallel RAM (Random Access Machine) se compose d'une mémoire centrale partagée, à laquelle accèdent les divers processeurs, comme illustré Figure 2.1. Le nombre de processeurs n'est pas borné, pas plus que la taille de la mémoire. L'hypothèse simplificatrice du modèle est que les processeurs accèdent à n'importe quelle case de la mémoire centrale en une unité de temps. L'accès simultané de plusieurs processeurs à autant de cases mémoire distinctes est possible en un seul pas de temps. Il y a des règles pour spécifier ce qui se passe en cas d'accès concurrent à la même case :

CREW (Concurrent Read Exclusive Write) C'est le modèle le plus usuel. Le nombre de lectures simultanées d'une même case n'est pas borné ; par contre l'accès en écriture n'est possible que par un seul processeur à la fois.

CRCW (Concurrent Read Concurrent Write) C'est le modèle le plus puissant. Le nombre d'écritures simultanées dans une même case n'est pas borné ; voici plusieurs variantes pour déterminer le résultat d'une écriture simultanée :
- mode consistant :tous les processeurs doivent écrire la même valeur,
- mode arbitraire : on prend la valeur du dernier processeur qui écrit,
- mode fusion : on effectue au vol le calcul d'une fonction commutative et associative appliquée aux différentes valeurs écrites, comme un ET ou un OU booléens, un maximum ou une somme d'entiers, etc.

EREW (Exclusive Read Exclusive Write) C'est le modèle le plus restrictif, mais aussi le plus proche des machines réelles.

2.2 Techniques de pointeurs

Trois exemples simples d'algorithmes P-RAM sont présentés dans ce paragraphe.

2.2.1 Calcul du rang

On a une liste chaînée L de taille n, et on veut calculer, pour chaque élément i de la liste, sa distance $d[i]$ à la fin de la liste :

$$d[i] = \begin{cases} 0 & \text{if } next[i] = \text{NIL} \\ d[next[i]] + 1 & \text{if } next[i] \neq \text{NIL} \end{cases}$$

On pourrait propager séquentiellement les valeurs d à partir de la fin de la liste, en temps $O(n)$. Avec une P-RAM, on associe un processeur P_i à chaque élément i de la liste, et on peut obtenir une solution en temps $O(\log n)$:

Initialisation
  for each processor i in // **do**
    if $next[i] = \text{NIL}$ **then** $d[i] = 0$ **else** $d[i] = 1$
Boucle principale
  while ($\exists$ objet i t.q. $next[i] \neq \text{NIL}$) **do**
    for each processor i in // **do**
      if $next[i] \neq \text{NIL}$ **then** $\begin{cases} d[i] \leftarrow d[i] + d[next[i]] \\ next[i] \leftarrow next[next[i]] \end{cases}$

On peut suivre l'exécution de l'algorithme sur la Figure 2.2. Le principe est simple : chaque pas de la boucle dédouble les listes courantes en listes des objets en positions paires et ceux en en positions impaires. La taille de chaque liste est donc réduite de moitié à chaque étape, d'où un nombre d'itérations égal à $\lceil \log n \rceil$.

La preuve de correction de l'algorithme ne présente qu'une difficulté : à chaque étape, les mises à jour doivent être synchronisées. La sémantique d'une boucle **for** parallèle, appelée aussi **forall**, est la suivante :

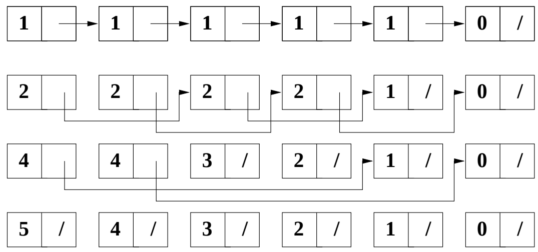


FIG. 2.2 – Exécution de l'algorithme de calcul du rang.

$$\begin{array}{l} \textbf{forall } i \textbf{ do} \\ \quad A(i) = B[i] \end{array} \iff \begin{array}{l} \textbf{for } i = 1 \textbf{ to } n \textbf{ do} \\ \quad A(i) = temp[i] \\ \textbf{for } i = 1 \textbf{ to } n \textbf{ do} \\ \quad temp[i] = B[i] \end{array}$$

En clair, toutes les lectures ont lieu avant la première écriture. Ce point étant réglé, l'analyse du nombre d'itérations nous permet de réécrire la boucle

$$\textbf{while } (\exists \text{ objet } i \text{ t.q. } next[i] \neq \text{NIL}) \textbf{ do}$$

en une boucle

$$\textbf{for } \text{étape } = 1 \textbf{ to } \lceil \log n \rceil$$

En effet le test de la condition globale sur les objets ne peut pas être fait en temps constant sur une P-RAM de type autre que CRCW. Au passage, comment ferait-on sur une P-RAM CRCW ? A la fin d'une étape, chaque processeur pourrait écrire dans une même case mémoire fini $\leftarrow$ TRUE ou fini $\leftarrow$ FALSE selon son statut, et on prendrait le ET booléen des écritures en mode fusion. Pour une P-RAM CREW, le temps nécessaire à la propagation d'une telle information globale est $O(\log n)$, où n est le nombre de processeurs en jeu.

Alors, quel type de P-RAM est nécessaire pour exécuter l'algorithme du calcul de rang en $O(\log n)$? chaque processeur P_i écrit dans $d(i]$ et $next[i]$, en mode exclusif donc. Pour les accès en lecture, si on ne précise pas davantage, deux processeurs P_i et P_j, où $next[j] = i$, peuvent accéder simultanément à la même valeur $d[i]$. Deux solutions pour passer en mode lecture exclusive : synchroniser les accès en deux sous-étapes, ou introduire un tableau de temporaires : on remplace

$$d[i] \leftarrow d[i] + d[next[i]]$$

par

$$\begin{array}{l} temp[i] \leftarrow d[next[i]] \\ d[i] \leftarrow d[i] + temp[i] \end{array},$$

et on obtient enfin une exécution en mode EREW.

2.2.2 Calcul des préfixes

Etant donnée une suite $(x_1, x_2, \ldots, x_n)$, il s'agit de calculer la suite $(y_1, y_2, \ldots, y_n)$ définie par $y_1 = x_1$ et, pour $1 < k \leq n$, par

$$y_k = y_{k-1} \otimes x_k = x_1 \otimes x_2 \otimes \ldots \otimes x_k.$$

Ici $\otimes$ désigne une opération binaire associative quelconque.

Avec une P-RAM à n processeurs, et en représentant la suite $(x_1, x_2, \ldots, x_n)$ sous la forme d'une liste chaînée (avec $next(x_i) = x_{i+1}$ pour $i < n$ et $next(x_n) = \text{NIL}$), on obtient l'algorithme suivant :

for each processor i in // **do**
  $y[i] \leftarrow x[i]$
  while ($\exists$ objet i t.q. $next[i] \neq$ NIL) **do**
    for each processor i in // **do**
      if $next[i] \neq$ NIL **then** $\begin{cases} y[next[i]] \leftarrow y[i] \otimes y[next[i]] \\ next[i] \leftarrow next[next[i]] \end{cases}$

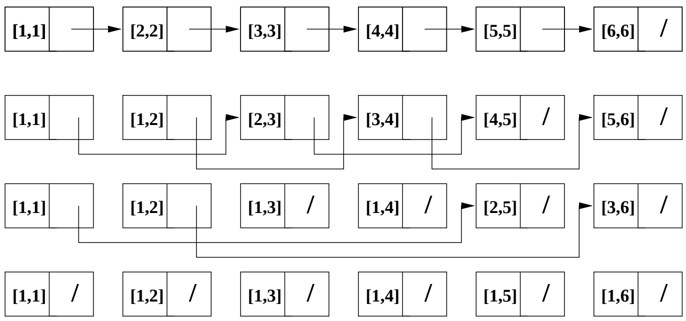


FIG. 2.3 – Exécution de l'algorithme de calcul des préfixes. On note $[i, j] = x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$ pour $i \leq j$.

L'exécution de l'algorithme est schématisée Figure 2.3. Mêmes commentaires que pour le calcul du rang : il y a $\lceil \log n \rceil$ itérations, on peut remplacer le test de la condition globale du *while* par une boucle **for**, et on peut éliminer facilement les deux accès en lecture concurrents : on obtient alors un algorithme EREW en $O(\log n)$.

2.2.3 Tour d'Euler

On donne ici une application du calcul des préfixes : étant donné un arbre binaire à n sommets, on veut calculer la hauteur de chaque sommet, i.e. sa distance à la racine de l'arbre. La structure de données pour représenter l'arbre est simple : chaque sommet i à trois champs de données $parent[i]$, $left[i]$ et $right[i]$, qui pointent respectivement sur son père, son fils gauche et son fils droite (et sur NIL s'ils n'existent pas).

Un algorithme P-RAM naïf serait de balayer l'arbre en largeur, ce qui prendrait $O(d)$ étapes, où d est la hauteur de l'arbre. C'est satisfaisant si l'arbre est équilibré, car alors $d = O(\log n)$, mais pas si l'arbre est un peigne ($d = O(n)$). La technique du tour d'Euler permet d'obtenir un coût en $O(\log n)$ quelle que soit la hauteur de l'arbre.

On associe 3 processeurs à chaque sommet de l'arbre, notés A, B, et C, et on effectue un parcours préfixe (père, fils gauche, fils droit) comme indiqué à la Figure 2.4(a). La tête de la liste chaînée est le processeur A de la racine, et sa queue est le processeur C de celle-ci. Pour un noeud donné :

- le processeur A pointe sur le processeur A de son fils gauche, s'il existe, et sinon sur son propre processeur de type B,
- le processeur B pointe sur le processeur A de son fils droit, s'il existe, et sinon sur son propre processeur de type C,

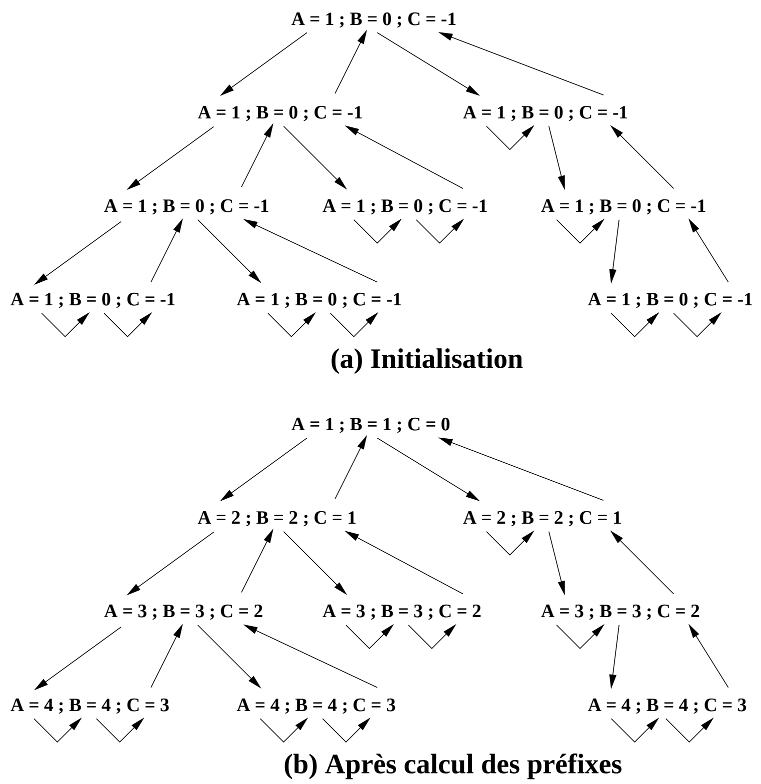


FIG. 2.4 – Tour d'Euler : (a) création du chemin eulérien et initialisation des processeurs ; (b) résultats après le calcul des préfixes.

– le processeur C pointe sur le processeur B de son père s'il s'agit d'un fils gauche, et sur le processeur C de son père s'il s'agit d'un fils droit (le processeur C de la racine de l'arbre pointe sur NIL).

Chacun des $3n$ processeurs i est responsable d'une valeur $x[i]$ initialisée à 1 pour les n processeurs de type A, à 0 pour ceux de type B et à -1 pour ceux de type C. On effectue alors un calcul de préfixes sur la liste, en utilisant l'addition comme opérateur associatif $\otimes$. Le résultat, i.e. la hauteur de chaque noeud, se trouve dans les valeurs finales des processeurs de type C (voir Figure 2.4(b)). Pour le voir, il suffit de remarquer que les valeurs $x[i]$ sont placées dans les processeurs A, B et C de telle sorte que la visite d'un sous-arbre n'ajoute rien à la somme courante :

– le processeur A d'un noeud i contribue un 1 à la somme courante du sous-arbre enraciné en son fils gauche, reflétant le fait que $d(left[i]) = d[i] + 1$ ($d[i]$ est la profondeur du sommet i),
– le processeur B contribue 0, car la hauteur du fils gauche est celle du fils droit,
– enfin le processeur C contribue -1

Tout comme le calcul des préfixes, l'algorithme est de type EREW.

2.3 Simulation

Dans ce paragraphe, nous discutons la *puissance* comparée des modèles EREW, CREW et CRCW, et nous définissons *l'efficacité* des algorithmes P-RAM.

2.3.1 Calcul du maximum

Voila un algorithme pour le calcul en temps constant du maximum d'un tableau A de n éléments, sur une P-RAM CRCW avec $O(n^2)$ processeurs :

```
for all i = 1 to n in // do
   m[i] ← TRUE
for all i, j = 1 to n in // do
   if A[i] < A[j] then m[i] ← FALSE
for all i = 1 to n in // do
   if m[i] = TRUE then max ← A[i]
```

La P-RAM opère en mode CRCW consistant, car tous les processeurs écrivent la même valeur FALSE. L'algorithme n'et pas très bien écrit : en fait à la première et troisième étapes on utilise seulement n processeurs, tandis qu'à la deuxième étape on a $n(n-1)$ processeurs P_{ij}, $i \neq j$, chacun responsable d'une comparaison $A[i] < A[j]$.

2.3.2 Séparation des modèles

Une P-RAM CRCW est-elle plus *puissante* qu'une P-RAM CREW ? Le problème de la comparaison des modèles s'énonce ainsi : peut-on trouver un algorithme tel que, pour un même nombre de processeurs, on obtienne avec une P-RAM CRCW un temps d'exécution qu'il est impossible d'atteindre pour une P-RAM CREW ?

L'algorithme du maximum nous fournit la réponse. Plus généralement, une P-RAM CRCW de n processeurs est capable de calculer en temps constant la fusion $\bigotimes_{1\leq i\leq n} e_i$ de n éléments $(e_1, \ldots, e_n)$, où $\otimes$ est une loi associative. Cette opération de réduction ne peut pas s'effectuer en moins de $O(\log n)$ étapes sur une P-RAM CREW : avec ce modèle, en temps constant on peut fusionner au plus un nombre constant d'éléments en une seule valeur.

Il est naturel de se poser la même question de séparation pour les modèles CREW et EREW. Voici un problème pour les séparer : déterminer si un élément e donné se trouve dans un ensemble de n éléments $(e_1, \ldots, e_n)$ tous distincts. Ce problème peut être résolu en temps constant sur une P-RAM CREW avec n processeurs : tout d'abord, on initialise un booléen à faux : $res \leftarrow$ FALSE ; ensuite, en parallèle, chaque processeur i compare l'élément e à e_i (il y a donc des lectures simultanées) et positionne le résultat res à TRUE s'il y a égalité. Comme les e_i sont différents deux à deux, au plus un processeur va écrire dans la case mémoire qui contient le booléen res, et l'algorithme peut bien s'exécuter en mode écriture exclusive.

Sur une P-RAM EREW, les processeurs ne peuvent pas accéder simultanément à l'élément e. Pour que les comparaisons puissent se faire en temps constant, il faut un nombre de copies de e de l'ordre de n, ce qui ne peut pas être réalisé en moins de $O(\log n)$ pas de calcul. Plus généralement, sur une P-RAM EREW la diffusion d'une information à n processeurs nécessite $O(\log n)$ étapes, car le nombre de processeurs "au courant" ne peut pas faire plus que de "doubler" (en fait être multiplié par une constante) à chaque étape.

2.3.3 Théorème de simulation

Nous venons de séparer les modèles, en exhibant un algorithme pour lequel un facteur $O(\log p)$ sépare les temps d'exécution avec p processeurs d'un même problème. Ce facteur $O(\log p)$ est maximal :

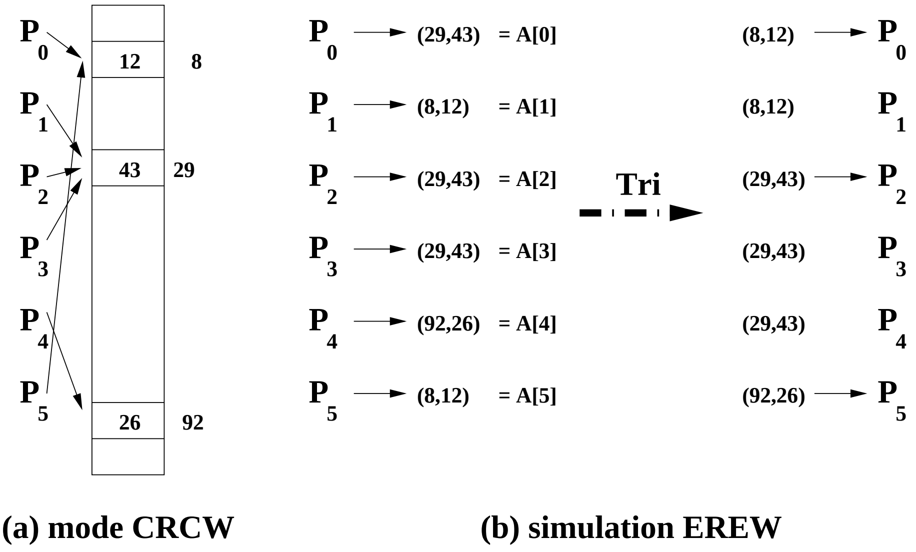


FIG. 2.5 – Simulation en mode exclusif d'écritures concurrentes.

Théorème 1 *Tout algorithme sur une machine P-RAM CRCW à p processeurs ne put pas être plus de $O(\log p)$ fois plus rapide que le meilleur algorithme P-RAM EREW à p processeurs pour le même problème.*

Preuve On supposera que la P-RAM CRCW opère en mode consistant (seule les écritures simultanées d'une même valeur sont autorisées). On va montrer comment simuler les écritures concurrentes en mode écriture exclusive, le cas des lectures se traitant de manière similaire.

Considérons un pas de l'algorithme CRCW à p processeurs : on va le simuler en $O(\log p)$ pas d'un algorithme EREW. La puissance de calcul étant la même, on se concentre sur les accès mémoire. L'algorithme EREW utilise un tableau auxiliaire de taille p, comme indiqué à la Figure 2.5. Quand un processeur P_i de l'algorithme CRCW écrit une donnée x_i à l'adresse l_i en mémoire, le processeur P_i de l'algorithme EREW effectue l'écriture (exclusive) $A[i] = (l_i, x_i)$. On trie alors le tableau A suivant la première coordonnée en temps $O(\log p)$, du moins si l'on suppose disposer d'un algorithme de tri EREW qui trie p données avec $O(p)$ processeurs en temps $O(\log p)$. Finissons la preuve avant de revenir sur cette hypothèse. Une fois le tableau A trié, chaque processeur P_i de l'algorithme EREW inspecte les deux cases adjacentes $A[i] = (l_j, x_j)$ et $A[i-1] = (l_k, x_k)$, où $0 \leq j, k \leq p-1$. Si $l_j \neq l_k$ ou si $i = 0$, le processeur P_i écrit la valeur x_j à l'adresse l_j, sinon il ne fait rien. Comme A est trié suivant la première coordonnée, l'écriture est bien exclusive.

Reste donc à exhiber un algorithme de tri, capable de trier p données avec $O(p)$ processeurs EREW en temps $O(\log p)$: c'est l'objet du Paragraphe 2.4. En fait, ce paragraphe présente seulement la solution pour une machine CREW, mais celle-ci peut être modifiée pour un mode EREW [24, 30]. ■

2.3.4 Théorème de Brent

Théorème 2 *Soit A un algorithme comportant un nombre total de m opérations et qui s'exécute en temps t sur une P-RAM (avec un nombre de processeurs indéterminé). Alors on peut simuler A en temps $O(\frac{m}{p} + t)$ sur une P-RAM de même type avec p processeurs.*

Preuve A l'étape i, A effectue $m(i)$ opérations, avec $\sum_{i=1}^{n} m(i == m$. On simule l'étape i avec p processeurs en temps $\lceil \frac{m(i)}{p} \leq \frac{m(i)}{p} + 1$. On obtient le résultat en sommant sur les étapes. ■

Le théorème de Brent permet de prédire les performances quand on réduit le nombre de processeurs. Prenons par exemple le calcul du maximum sur une P-RAM EREW. On peut agencer ce calcul en temps $O(\log n)$ à l'aide d'un arbre binaire : à l'étape 1, on procède paire par paire avec $\lceil \frac{n}{2} \rceil$ processeurs, puis on continue avec les maximum des paires deux par deux, etc. C'est à la première étape qu'on a besoin du plus grand nombre de processeurs, donc il en faut $O(n)$. Formellement, si $n = 2^m$, si le tableau A est de taille $2n$ et si on veut calculer le maximum des n éléments de A en position $A[n], A[n+1], \ldots, A[2n-1]$, on obtient le résultat dans $A[1]$ après exécution de l'algorithme :

for $k = m - 1$ **downto** O **step** -1 **do**
  for all j, $2^k \leq j \leq 2^{k+1} - 1$ in // **do**
    $A[j] \leftarrow \max(A[2j], A[2j+1])$

Que se passe-t-il si on dispose de moins de $O(n)$ processeurs ? Le théorème de Brent nous dit qu'avec p processeurs, on peut simuler l'algorithme précédent en temps $O(\frac{n}{p} + \log n)$ (en effet, le nombre d'opérations total est $m = n - 1$). Si on choisit $p = \frac{n}{\log n}$, on obtient le même temps d'exécution, mais avec moins de processeurs !

2.3.5 Travail et efficacité

Donnons quelques définitions usuelles : soit P un problème de taille n à résoudre, et soit $T_{seq}(n)$ le temps du meilleur algorithme séquentiel (connu) pour résoudre P. Soit maintenant un algorithme parallèle P-RAM qui résout P en temps $T_{par}(p)$ avec p processeurs. Le facteur d'accélération (*speed-up* en anglais) est défini comme $S_p = \frac{T_{seq}(n)}{T_{par}(p)}$ et l'efficacité comme $e_p = \frac{T_{seq}(n)}{p.T_{par}(p)}$. Enfin, le travail (*work*) de l'algorithme est $W_p = p.T_{par}(p)$. Intuitivement, le travail est une surface rectangulaire de taille $T_{par}(p)$, le temps d'exécution, multiplié par p, le nombre de processeurs, et est maximal si à chaque étape de calcul tous les processeurs sont utilisés à faire des choses utiles, i.e. des opérations présentes dans la version séquentielle.

Le résultat suivant montre qu'on peut conserver le travail par simulation :

Proposition 1 *Soit A un algorithme qui s'exécute en temps t sur une P-RAM avec p processeurs. Alors on peut simuler A sur une P-RAM de même type avec $p' \leq p$ processeurs, en temps $O(\frac{t.p}{p'})$.*

Preuve Avec p' processeurs, on simule chaque étape de A en temps proportionnel à $\lceil \frac{p}{p'} \rceil$. On obtient un temps total de $O(\frac{p}{p'}.t) = O(\frac{t.p}{p'})$. ■

Une conséquence de ce résultat est que le travail d'un algorithme P-RAM est au moins de l'ordre de la complexité séquentielle (sinon la simulation de cet algorithme avec un seul processeur améliorerait cette complexité). On dit qu'un algorithme P-RAM est *efficace* quand son travail est de l'ordre de la complexité séquentielle (on n'ose pas dire *optimal*, sauf si la complexité séquentielle est établie). On ne peut pas diminuer le travail d'un algorithme efficace de plus d'un facteur constant.

2.4 Machine à trier

Cole [24] a proposé en 1986 un superbe algorithme P-RAM CREW pour trier n nombres en temps $O(\log n)$ avec $O(n)$ processeurs. Cet algorithme est optimal, puisque la complexité séquentielle

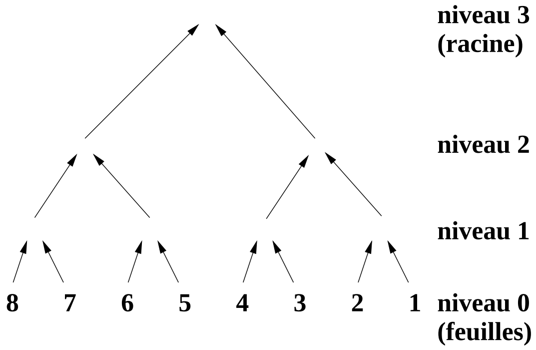


FIG. 2.6 – Arbre binaire pour le tri-fusion de Cole.

du tri (dans un modèle avec comparaisons) est de $O(n \log n)$. L'algorithme de Cole est basé sur l'algorithme de tri-fusion classique, dont la représentation en arbre binaire de la Figure 2.6 fait bien apparaître le parallélisme potentiel : toutes les fusions d'un même niveau de l'arbre peuvent être effectuées en parallèle. Imaginons qu'on progresse ainsi niveau par niveau, en parallèle. Comme il y a $\log n$ niveaux, si on veut terminer avec un temps d'exécution total en $O(\log n)$, il faut savoir traiter un niveau en temps constant. Mais comment fusionner deux listes de taille arbitraire en temps constant ? c'est là toute la beauté de l'algorithme de Cole, qui ré-utilise l'information partielle acquise lors des fusions précédentes pour préparer la fusion courante et l'exécuter en temps constant. Très joli, mais un peu difficile : courage !

2.4.1 Fusion

Pour fixer les idées, on manipule des suites d'entiers. On dit que x est *entre* a et b si $a < x \leq b$.

Définition 1 *Une suite L est un* bon échantillonnage (BE) *d'une suite J si, pour tout $k \geq 1$, entre $k+1$ éléments consécutifs (arbitraires) de $\{-\infty\} \cup L \cup \{+\infty\}$, il y a au plus $2k+1$ éléments de J.*

Intuitivement, L est une BE de J si les éléments de L sont distribués uniformément (avec une certaine tolérance) au milieu des éléments de J. Pour $k = 1$, la définition impose qu'il n'y a jamais plus de 3 éléments de J entre deux éléments consécutifs de $\{-\infty\} \cup L \cup \{+\infty\}$. Par exemple Pair$(J)$, l'ensemble des éléments d'indice pair dans J, et Impair(J), sont tous les deux des BE de J. Notons $J = \{j_1, \ldots, j_n\}$ et vérifions que Pair(J) est un BE de J : considérons $k+1$ éléments consécutifs $\{j_{2i}, j_{2(i+1)}, \ldots, j_{2(i+k)}\}$ de $\{-\infty\} \cup \text{Pair}(J) \cup \{+\infty\}$; on note éventuellement $j_0 = -\infty$ si $i = 0$, et/ou $j_{2(i+k)} = +\infty$ si $2(i+k) > n$; il y a bien $2k-1$ ou $2k$ éléments de J entre eux.

On définit le rang d'un élément x dans une suite J comme le nombre d'éléments de J plus petits que x :

$$rang(x, J) = \text{card}\{j \in J, j < x\}$$

De même, le rang croisé de A dans B est la fonction $R[A, B](e) = \text{rang}(e, B)$ pour tout $e \in A$, fonction que l'on représente sous la forme d'un tableau de taille $|A|$, dont la i-ème entrée contient le rang du i-ème élément de A dans B.

Grâce aux bons échantillonnages, on peut fusionner deux suites triées de manière rapide : soient J et K deux suites triées, et supposons que L est un BE de J et de K. On pose $l_0 = -\infty$ et $l_{|L|+1} = +\infty$.

FusionAssistée(J, K, L)

1. On partitionne J et K en $|L|+1$ sous-ensembles $J(i) = \{j \in J,\ l_{i-1} < j \leq l_i\}$ et $K(i) = \{k \in K,\ l_{i-1} < k \leq l_i\}$, $1 \leq i \leq |L|+1$. Puisque L est un BE de J et de K, chacun de ses sous-ensembles a au plus trois éléments.
2. On fusionne en parallèle les sous-ensembles :

$$\textbf{for}\ \mathsf{all}\ i,\ 1 \leq i \leq |L|+1,\ \mathsf{in}\ //\ \textbf{do}$$
$$res_i \leftarrow \mathsf{Fusion}(J(i), K(i))$$

3. On concatène les résultats partiels :

$$J|K \leftarrow res_1 res_2 \ldots res_{|L|+1}.$$

On a noté $J|K$ la fusion de J et K. Illustrons la procédure FusionAssistée sur un exemple : soit $J = [2,3,7,8,10,14,15,17,18,21]$ et $K = [1,4,6,9,11,12,13,16,19,20]$. Alors $L = [5,10,12,17]$ est un BE à la fois de J et de K : il faut le vérifier de façon exhaustive pour $k = 1$ (5 vérifications pour les paires $(-\infty,5)$, $(5,10)$, $(10,12)$, $(12,17)$, et $(17,+\infty)$), puis pour $k = 2$, etc. On obtient :
- $J(1) = [2,3]$, $J(2) = [7,8,10]$, $J(3) = \emptyset$, $J(4) = [14,15,17]$, et $J(5) = [18,21]$.
- $K(1) = [1,4]$, $K(2) = [6,9]$, $K(3) = [11,12]$, $K(4) = [13,16]$, et $K(5) = [19,20]$.
- $\begin{cases} res_1 = \text{Fusion}([2,3],[1,4]) = [1,2,3,4] \\ res_2 = \text{Fusion}([7,8,10],[6,9]) = [6,7,8,9,10] \\ res_1 = \text{Fusion}(\emptyset,[11,12]) = [11,12] \\ res_1 = \text{Fusion}([14,15,17],[13,16]) = [13,14,15,16,17] \\ res_1 = \text{Fusion}([18,21],[19,20]) = [18,19,20,21] \end{cases}$

Chacune des trois étapes peut se faire en temps constant, à condition de disposer des rangs croisés $R[L,J]$, $R[L,K]$, $R[J,L]$ et $R[K,L]$:

1. Pour partitionner J, on utilise $|J|$ processeurs. Chaque P_j, $j \in J$, lit $rang(j,L) = r$ et insère j dans $J(r)$. Comme il y a au plus 3 processeurs qui veulent écrire dans $J(r)$, on peut bien obtenir un mode d'écriture exclusive.
2. Pour la fusion parallèle, on a besoin de $|L|$ processeurs et le temps est clairement constant.
3. Pour la concaténation, connaissant $R[L,J]$ et $R[L,K]$ on peut calculer $R[L,J|K]$: pour chaque élément $l \in L$, $rang(l,J|K) = rang(l,J) + rang(l,K)$. Le processeur responsable de res_i regarde le rang r de l_{i-1} dans $J|K$ et range les éléments (au plus 6) à partir de la position $r+1$. Il suffit de L processeurs.

En résumé :

Proposition 2 *Si L est un BE des suites triées J et K et si on connaît les rangs croisés $R[L,J]$, $R[L,K]$, $R[J,L]$ et $R[K,L]$, alors* FusionAssistée(J,K,L) *s'exécute en temps constant avec $|J|+|K|$ processeurs sur une P-RAM CREW.*

2.4.2 Arbre à trier

On suppose que $n = 2^m$ et on fait fonctionner l'arbre à trier de la Figure 2.6 en pipeline ; chaque sommet de l'arbre stocke une valeur $val(t)$, qui est une suite triée qui grossit au fil des étapes. Intuitivement, on fait en sorte que $val(t)$ soit un bon échantillonnage des entrées de l'étape suivante.

Opération Au top $t = 0$, tous les noeuds stockent la suite vide, sauf les feuilles qui stockent leur élément. L'opération d'un noeud au top $t+1$ est le suivant :

1. Recevoir $X(t+1)$ du fils gauche et $Y(t+1)$ du fils droit.

2. Fusionner : $val(t+1) \leftarrow$ FusionAssistée$(X(t+1), Y(t+1), val(t))$
3. Réduire : envoyer à son père $Z(t+1) =$ Réduction$(val(t+1))$. L'opérateur de réduction conserve une donnée sur quatre ; plus précisément si $Z = \{z_1, z_2, \ldots, z_n\}$, Réduction$(Z) = \{z_4, z_8, z_{12}, \ldots\}$.

Sous réserve de la preuve de correction, on a donc $val(t) =$ Fusion$(X(t), Y(t)) = X(t)|Y(t)$ à tout instant t.

Fonctionnement Un sommet de l'arbre est dit complet quand il a toutes ses entrées, i.e. une suite triée de 2^k éléments pour un sommet de niveau k. Dès qu'elle est non nulle, la taille de ses entrées double à chaque top, pour aller de 1 à 2^k. Pour un sommet complet, l'opération de réduction est modifiée. Si le sommet est complet au top t, alors

- au top $t+1$, il envoie un élément sur 4 de $val(t+1)$ à son père, comme en régime permanent,
- au top $t+2$, il envoie un élément sur 2 (le deuxième, le quatrième, etc) de $val(t+1)$ à son père,
- au top $t+3$, il envoie tous les éléments de $val(t+1)$ à son père,
- au top $t+4$ et après, il cesse de fonctionner et n'envoie plus rien à son père.

Etude de l'exemple Nous décrivons ici en détail le fonctionnement de l'arbre pour une entrée de taille 8, comme à la Figure 2.6. Au top $t = 0$, les feuilles sont complètes, car elles possèdent une suite de taille 2^0. Au top $t = 1$ et $t = 2$ elles n'envoient qu'un élément sur quatre puis sur deux, i.e. rien. Au top $t = 3$, elles envoient leur unique valeur à leur père, puis cessent de fonctionner.

Suivons le père $[8, 7]$ des feuilles 8 et 7. Au top $t = 3$, il calcule $val(3) =$ FusionAssistée$(\{8\}, \{7\}, \emptyset)$ et devient complet (niveau 1). Au top $t = 4$ il n'envoie rien à son père. Au top $t = 5$ il envoie un élément sur deux, i.e. $\{8\}$. Au top 6 il envoie ses deux éléments puis cesse de fonctionner.

Suivons maintenant le sommet racine du sous-arbre $[8, 7, 6, 5]$. Au top $t = 5$ il reçoit $\{8\}$ et $\{6\}$ et calcule $val(5) =$ FusionAssistée$(\{8\}, \{6\}, \emptyset)$. Au top $t = 6$ il reçoit $X(6) = \{7, 8\}$ et $Y(6) = \{5, 6\}$ et calcule $val(6) =$ FusionAssistée$(\{7, 8\}, \{5, 6\}, val(5))$. Au passage on vérifie que $val(5)$ est un BE de $X(6)$ et $Y(6)$. Au top $t = 6$ le sommet est complet (niveau 2). Au top $t = 7$ il envoie $\{8\}$, et au top $t = 8$ il envoie $\{6, 8\}$. Au top $t = 9$ il envoie tous ses quatre éléments puis cesse de fonctionner.

Suivons enfin le sommet racine de l'arbre, de niveau 3. Au top $t = 7$ il reçoit $\{8\}$ et $\{4\}$ et calcule $val(7) =$ FusionAssistée$(\{8\}, \{4\}, \emptyset)$. Au top $t = 8$ il reçoit $\{6, 8\}$ et $\{2, 4\}$ et calcule $val(8) =$ FusionAssistée$(\{6, 8\}, \{2, 4\}, val(7))$. Au top $t = 9$ il reçoit $\{5, 6, 7, 8\}$ et $\{1, 2, 3, 4\}$ et calcule $val(9) =$ FusionAssistée$(\{5, 6, 7, 8\}, \{1, 2, 3, 4\}, val(8))$. On vérifie que $val(t)$ est un BE de $X(t+1)$ et de $Y(t+1)$ pour $t = 6, 7, 8$. Au top $t = 9$, le sommet est complet, le tri est terminé.

2.4.3 Preuve de fonctionnement

On commence par le temps d'exécution et le nombre de processeurs nécessaires :

Lemme 1 *L'arbre fonctionne en temps $O(\log n)$ avec $O(n)$ processeurs.*

Preuve Un sommet de niveau k est complet au top $t = 3k$: par récurrence immédiate sur k, car il faut 3 tops à ses fils pour envoyer toutes leurs données. L'arbre fonctionne donc bien en temps $O(\log n)$.

Pour le nombre de processeurs nécessaires : à chaque niveau k de l'arbre il y a $\frac{n}{2^k}$ sommets qui fusionnent chacun des listes de taille bornée par 2^k, ce qui conduit à un besoin total de $O(n)$ processeurs par niveau de l'arbre. Il faudrait alors $O(n \log n)$ processeurs pour faire fonctionner l'arbre. En fait, une analyse plus fine du fonctionnement en pipeline montre que tous les niveaux ne sont pas actifs en même temps. Les processeurs du niveau k traitent des donnée de taille 2^k au top $t = 3k$, 2^{k-1} au top $t-1$, 2^{k-2} au top $t-2$, etc. Une vision globale de l'arbre, et non plus niveau par niveau, montre donc que le nombre de processeurs réellement nécessaires est en $O(n)$. ∎

On prouve maintenant l'invariant lié au bon échantillonnage :

Lemme 2 *Soient X, X', Y, et Y' quatre suites triées. Si X est un BE de X' et Y un BE de Y', alors Réduction$(X|Y)$ est un BE de Réduction$(X'|Y')$.*

Preuve On note toujours $X|Y = \text{Fusion}(X,Y)$. Il est clair que si X est un BE de X', alors $X|W$ est encore un BE de X', pour tout ensemble W. Par contre, si X est un BE de X' et Y un BE de Y', $X|Y$ n'est pas nécessairement un BE de $X'|Y'$. Prenons par exemple $X = [2,7]$, $X' = [2,5,6,7]$, $Y = [1,8]$, et $Y' = [1,3,4,8]$. Alors $X|Y = [1,2,7,8]$ et $X'|Y' = [1,2,3,4,5,6,7,8]$, mais il y a 5 éléments de $X'|Y'$ entre 2 et 7, qui sont pourtant deux éléments consécutifs de $X|Y$. C'est ce qui explique le recours à l'opérateur de réduction.

Nous allons montrer la propriété suivante : il y au plus $2r+2$ éléments de $X'|Y'$ entre r éléments consécutifs de $X|Y$ (on supposera que X et Y contiennent $-\infty$ et $+\infty$). Prenons une suite $e_1, e_2, \ldots, e_r$ de r éléments consécutifs de $X|Y$. De ces r éléments, h viennent de X et h' de Y, avec $h + h' = r$. Sans perte de généralité, supposons que $e_1 \in X$. Il y a deux cas à considérer :

Cas 1 : $e_r \in X$ Entre e_1 et e_r il y a au plus $2(h-1)+1$ éléments de X', puisque X est un BE de X'. Par ailleurs, il y a au plus $2(h'+1)+1$ éléments de Y', parce que ces éléments sont entre $r+2$ éléments de Y, qui est un BE de Y' : pour le voir, on ajoute deux éléments frontière à la suite des h' éléments consécutifs de Y dans $e_1, e_2, \ldots, e_r$. Au total il y a bien $2(h-1)+1+2(h'+1)+1 = 2r+2$ éléments de $X'|Y'$.

Cas 2 : $e_r \in Y$ On ajoute un élément $e_0 \in Y$ prédécesseur de e_1 et un élément $e_{r+1} \in X$ successeur de e_r. Les éléments de $X'|Y'$ qui sont entre e_1 et e_r proviennent d'éléments de X' qui se trouvent entre $h+1$ éléments de X, et d'éléments de Y' qui se trouvent entre $h'+1$ éléments de Y. Il y a un total d'au plus $2h+1+2h'+1 = 2r+2$ éléments de $X'|Y'$.

On revient à la preuve du Lemme. Soit $Z = \text{Réduction}(X|Y)$ et $Z' = \text{Réduction}(X'|Y')$. Considérons $k+1$ éléments consécutifs $z_1, z_2, \ldots, z_{k+1}$ de Z. Puisque l'opérateur de réduction prend un élément sur quatre, on a $z_1 = e_{4h}$, $z_2 = e_{4(h+1)}$, …, $z_{k+1} = e_{4(h+k)}$, où les indices des éléments e décrivent $X|Y$. Il y a donc $4k+1$ éléments de $X|Y$ entre $z_1, z_2, \ldots, z_{k+1}$, et d'après la propriété précédente avec $r = 4k+1$, il y au plus $8k+4$ éléments de $X'|Y'$ entre ces $4k+1$ éléments. L'opérateur de réduction conservant un élément sur quatre, il y a au plus $\frac{8k+4}{4} = 2k+1$ éléments de Z' entre les $k+1$ éléments consécutifs de Z : Z' est bien un BE de Z. ∎

En régime permanent, un sommet de l'arbre reçoit en entrée une suite triée $X(t+1)$ de son fils gauche et une suite triée $(Y(t+1)$ de son fils droit. Il calcule $val(t+1) = \text{FusionAssistée}(X(t+1), Y(t+1), val(t)$ et transmet $Z(t+1) = \text{Réduction}(val(t+1))$ à son père. On a l'invariant que $val(t) = X(t)|Y(t)$, $X(t)$ est un BE de $X(t+1)$, $Y(t)$ est un BE de $Y(t+1)$: en effet le Lemme 2 montre que $Z(t)$ est un BE de $Z(t+1)$. Notons que pour les deux derniers envois, la propriété est encore vérifiée (on envoie un élément sur deux, puis tous les éléments).

Pour finir, il faut s'assurer d'être dans les conditions d'application de la Proposition 2 : pour calculer la fusion assistée d'un BE en temps constant, il faut disposer des rangs croisés. Nous terminons donc par quelques résultats techniques sur les calculs de rang.

A une étape donnée, en simplifiant les notations : X est un BE de X', Y est un BE de Y', $U = X|Y$, $Z = \text{Réduction}(U)$. On suppose connaître les rangs croisés $R[X', X]$ et $R[Y', Y]$. On calcule $U' = X'|Y'$ par $U' = \text{FusionAssistée}(X', Y', U)$. On veut disposer des rangs croisés $R[X', U]$, $R[Y', U]$, $R[U, X']$, et $R[U, Y']$ pour cette opération. Enfin, on calcule $Z' = \text{Réduction}(U')$ et on veut disposer de $R[Z', Z]$ pour maintenir l'invariant.

Bien sûr pour chaque suite triée S on connaît le rang $R[S, S]$, i.e. l'indice de chaque élément de la suite (disons que cela fait partie de la représentation interne de S). Voici deux propriétés faciles :

Lemme 3 *Si $S = [b_1, b_2, \ldots, b_k]$ est une suite triée, le rang rang(a, S) d'un élément a peut être calculé en temps $O(1)$ avec $O(k)$ processeurs sur une P-RAM CREW.*

Preuve On pose $b_0 = -\infty$ et $b_{k+1} = +\infty$, et on calcule le rang par la boucle :

```
for all i, 1 ≤ i ≤ k in // do
  if b_i < a ≤ b_{i+1} then rang ← i
```

Il n'y a pas de conflit d'écriture, car un seul processeur va détermine la valeur du rang. ■

Lemme 4 *Si on a deux suites triées disjointes S_1, S_2 telles que $S = S_1|S_2$ soit déjà construite, alors on peut calculer $R[S_1, S_2]$ et $R[S_2, S_1]$ en temps $O(1)$ avec $O(|S|)$ processeurs.*

Preuve Comme S est déjà construite, on connaît $R[S, S]$. Pour $a \in S_1 \subset S$,

$$\text{rang}(a, S_2) = \text{rang}(a, S) - \text{rang}(a, S_1)$$

et le résultat s'ensuit directement. ■

On peut maintenant démontrer l'invariant sur les rangs croisés pour la fusion assistée :

Lemme 5 *Si on a des suites triées X, Y, $U = X|Y$, X' et Y' telles que X est un BE de X', Y est un BE de Y', et qu'on connaît les rangs croisés $R[X', X]$ et $R[Y', Y]$, alors on peut calculer les rangs croisés $R[X', U]$, $R[Y', U]$, $R[U, X']$, et $R[U, Y']$ en temps $O(1)$ avec $O(|X|+|Y|)$ processeurs.*

Preuve Montrons d'abord comment calculer $R[X', U]$. Soit $X = [a_1, a_2, \ldots, a_k]$, et posons $a_0 = -\infty$ et $a_{k+1} = +\infty$. On partitionne X' selon X en $X'(i) = \{x' \in X',\ a_{i-1} < x' \leq a_i\}$ pour $1 \leq i \leq k+1$. Ce partitionnement s'obtient en temps $O(1)$ avec $O(|X|)$ processeurs parce qu'on connaît $R[X', X]$. On partitionne aussi U selon X (en fait Y selon X, car $U = X|Y$) : soit $U(i) = \{y \in Y, a_{i-1} < y \leq a_i\}$. On a alors la procédure suivante :

```
for all i, 1 ≤ i ≤ k+1 in // do
  for all x' ∈ X'(i) do
    calculer rang(x', U(i)) (avec le Lemme 3)
    rang(x', U) ← rang(a_{i-1}, U) + rang(x', U(i))
```

Pour chaque i on utilise $|U(i)|$ processeurs, donc un total en $O(|U|)$. Le cardinal de chaque $X'(i)$ est au plus trois, comme X est un BE de X', et la procédure s'exécute bien en temps $O(1)$. Ceci achève le calcul de $R[X', U]$. Bien sûr $R[Y', U]$ s'obtient de manière similaire.

Pour calculer $R[U, X']$ on a besoin de $R[X, X']$ et de $R(Y, X']$. Pour $R[X, X']$, prenons un élément a_i de $X \setminus X'$ et cherchons le plus petit élément a' de $X'(i+1)$: le rang de a_i dans X' est celui de a' dans X', et celui-ci est disponible car faisant partie de la représentation interne de X' (on connaît $R[X', X']$). On peut donc trouver rang(a_i, X') en temps constant avec un seul processeur. Reste à calculer $R[Y, X']$. Soit $y \in Y$. On peut calculer rang(y, X) en utilisant le Lemme 4, puisque $U = X|Y$ est déjà construite. On calcule alors rang(y, X') à partir de rang(y, X) et de $R[X, X']$. ■

Lemme 6 *Avec les notations du Lemme 5, soit Z = Réduction(U) et Z' = Réduction(U'). On peut calculer $R[Z', Z]$ en temps $O(1)$ avec $O(|X| + |Y|)$ processeurs.*

Preuve La preuve est directe : à partir de $R[X', U]$ on calcule facilement $R[X', Z]$. De même pour $R[Y', Z]$, d'où $R[U', Z]$. Comme Z' est un sous-ensemble de Z, on a le résultat. ■

Voila qui conclut la preuve de correction. En résumé, celle-ci est basée sur trois invariants principaux :

1. les tailles des entrées d'un noeud de niveau k doublent à chaque étape, de 1 jusqu'à 2^k, le noeud est alors complet,
2. si $X(t)$ est un BE de $X(t+1)$ et $Y(t)$ un BE de $Y(t+1)$, alors $Z(t)$ est un BE de $Z(t+1)$,
3. on peut calculer $R[S(t+1), S(t)]$ pour toute suite d'entrées ou de sorties $S(t)$ d'un noeud.

Formellement, nous avons prouvé le :

Théorème 3 *On peut trier une suite de n clés en temps $O(\log n)$ sur une P-RAM CREW avec $O(n)$ processeurs.*

Le travail de l'algorithme de Cole est en $O(n \log n)$, il est donc optimal (on peut le dire plutôt qu'efficace, car on connaît la complexité séquentielle du tri).

Notes bibliographiques

Les notes de ce chapitre s'inspirent des livres de Cormen, Leiserson et Rivest [25] et de Gengler, Ubéda et Desprez [35] pour les aspects introductifs, et du livre de Gibbons et Rytter [37] pour le tri de Cole. L'article original de Cole [24] contient également une version P-RAM EREW de sa machine à trier, avec les mêmes performances. Enfin, tout ce que vous voulez savoir sur les P-RAM, et même davantage, figure dans le livre de Reif [30].

Chapitre 3

Réseaux de tri

3.1 Introduction

Ce chapitre est consacré aux réseaux de tri, un modèle de calcul plus réaliste, mais moins général, que le modèle P-RAM. Le but du jeu est de disposer et d'interconnecter des comparateurs-échangeurs (comparateurs, pour faire bref) qui prennent deux nombres en entrée et renvoient le plus petit sur la première sortie et le plus grand sur la seconde sortie (voir la Figure 3.1), afin de trier des listes. On recherche une architecture qui ne dépend que de la taille des listes, et non des valeurs de leurs éléments.

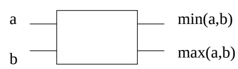


FIG. 3.1 – Fonctionnement d'un comparateur.

Le premier réseau présenté opère par fusion, et constitue donc le pendant de la machine à trier de Cole (Paragraphe 2.4). Le second réseau présenté peut être replié sur une architecture linéaire de processeurs.

3.2 Tri-fusion de Batcher

Le réseau de tri-fusion de Batcher [12] utilise récursivement un sous-réseau permettant de calculer la fusion de deux listes : nous commençons par décrire celui-ci, puis nous revenons au tri proprement dit. Dans tout ce paragraphe on ne manipule que des suites de longueur égale à une puissance de deux.

3.2.1 Réseau de fusion pair-impair

Commençons par quelques notations :

- Soit $(c_1, c_2 \ldots, c_n)$ une suite quelconque, on note $\textsc{tri}(c_1, c_2 \ldots, c_n)$ la liste triée des c_i.
- Si la suite est déjà triée, i.e. si $c_1 \leq c_2 \leq \cdots \leq c_n$, on écrit $\textsc{triée}(c_1, c_2 \ldots, c_n)$.
- Enfin, on note FUSION l'opérateur de fusion de deux listes triées : sous les les hypothèses $\textsc{triée}(a_1, \ldots, a_n)$ et $\textsc{triée}(b_1, \ldots, b_n)$,

$$\textsc{fusion}((a_1, \ldots, a_n), (b_1, \ldots, b_n)) = \textsc{tri}(a_1, \ldots, a_n, b_1, \ldots, b_n) .$$

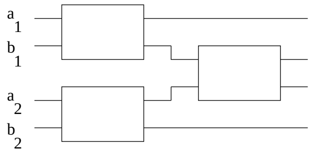


FIG. 3.2 – Réseau FUSION_1 pour fusionner deux suites triées de taille 2.

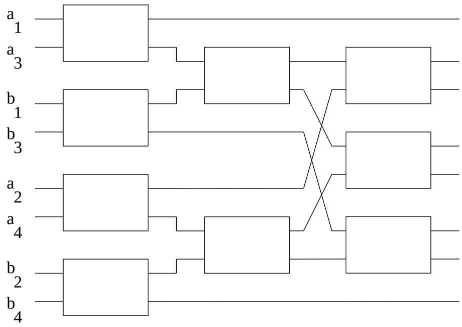


FIG. 3.3 – Réseau FUSION_2 pour fusionner deux suites triées de taille 4.

L'objectif de ce paragraphe est de construire un réseau FUSION_m pour la fusion de deux listes triées de taille 2^m. Pour $m = 0$, un seul comparateur suffit. Pour $m = 1$, supposant $\text{TRIÉE}(a_1, a_2)$ et $\text{TRIÉE}(b_1, b_2)$, on peut utiliser trois comparateurs, comme indiqué Figure 3.2. Il est immédiat de se convaincre de la validité de ce réseau FUSION_2 : par hypothèse $a_1 \leq a_2$ et $b_1 \leq b_2$, la sortie du haut est $\min(a_1, b_1)$, celle du bas est $\max(a_2, b_2)$, et on a rajouté un comparateur pour interclasser $\max(a_1, b_1)$ et $\min(a_2, b_2)$.

Pour $m = 2$, c'est déjà plus difficile : nous allons démontrer la validité du réseau FUSION_3 de la Figure 3.3. En effet, la construction est générale : pour construire le réseau FUSION_m, on utilise deux copies du réseau FUSION_{m-1} et une rangée de $2^m - 1$ comparateurs. La première copie de FUSION_{m-1} fusionne les éléments d'indice impair des deux suites en entrée, et la deuxième fusionne les éléments d'indice pair. Le miracle est qu'une simple rangée de comparateurs suffit alors pour compléter la fusion globale. Formellement, le miracle s'énonce ainsi :

Proposition 3 *Avec les hypothèses* $\text{TRIÉE}(a_1, \ldots, a_{2n})$ *et* $\text{TRIÉE}(b_1, \ldots, b_{2n})$, *en notant*

$$\begin{aligned}(d_1, \ldots, d_{2n}) &= \text{FUSION}((a_1, a_3, \ldots, a_{2n-1}), (b_1, b_3, \ldots, b_{2n-1}))\\ (e_1, \ldots, e_{2n}) &= \text{FUSION}((a_2, a_4, \ldots, a_{2n}), (b_2, b_4, \ldots, b_{2n}))\end{aligned}$$

on a

$$\text{TRIÉE}(d_1, \min(d_2, e_1), \max(d_2, e_1), \ldots, \min(d_{2n}, e_{2n-1}), \max(d_{2n}, e_{2n-1}), e_{2n}) \ .$$

Preuve Supposons les éléments tous distincts sans perte de généralité. Dans la liste conclusion, d_1 apparaît en première position, et c'est bien le plus petit élément de la liste finale ; de même, e_{2n} est bien à sa place en dernière position. Pour le cas général, d_i et e_{i-1} (pour $i \geq 2$) apparaissent dans la liste conclusion en position $2i - 2$ ou $2i - 1$. Nous allons montrer qu'ils sont bien à leur place en les "encadrant", i.e. en montrant que chacun d'entre eux domine $2i - 3$ éléments de la suite finale, et est dominé par $4n - 2i + 1$ éléments de cette suite. Leur position dans la suite est alors nécessairement $2i - 2$ ou $2i - 1$, et la comparaison effectuée entre eux deux les met à la bonne place.

Il y a donc 4 choses à démontrer :
(i) d_i domine $2i-3$ éléments ;
(ii) e_{i-1} domine $2i-3$ éléments ;
(iii) d_i est dominé par $4n-2i+1$ éléments ;
(iv) e_{i-1} est dominé par $4n-2i+1$ éléments.

Prouvons (i) : supposons par exemple que d_i appartienne à la suite A, la suite des (a_i) (le cas où d_i appartient à B est similaire). Soit k le nombre d'éléments parmi $\{d_1, d_2, \ldots, d_i\}$ qui appartiennent à A. On a alors $d_i = a_{2k-1}$, et d_i domine $2k-2$ éléments de A. Il y a $i-k$ éléments de B dans $\{d_1, d_2, \ldots, d_{i-1}\}$, le plus grand d'entre eux est donc $b_{2(i-k)-1}$, et d_i domine $2(i-k)-1$ éléments de B. Au total, d_i domine $(2k-2)+(2(i-k)-1) = 2i-3$ éléments. La preuve de (ii) est identique.

Prouvons maintenant (iv), en supposant cette fois que e_{i-1} appartient à B. Soit k le nombre d'éléments de B dans $\{e_1, e_2, \ldots, e_{i-1}\}$, alors $e_{i-1} = b_{2k}$, et e_{i-1} est dominé par $2n-2k$ éléments de B. Il y a aussi $i-1-k$ éléments de A dans l'ensemble précédent, donc e_{i-1} est dominé par $a_{2(i-k)}$ et ses successeurs, un total de $2n-2(i-k)+1$ éléments de A. En sommant avec les $2n-2k$ éléments de B on trouve bien $4n-2i+1$ éléments. Enfin, la preuve de (iii), similaire, est laissée au lecteur.

Tout ceci montre bien que la position de d_i et e_{i-1} dans la liste triée est bien déterminée à une transposition près. ■

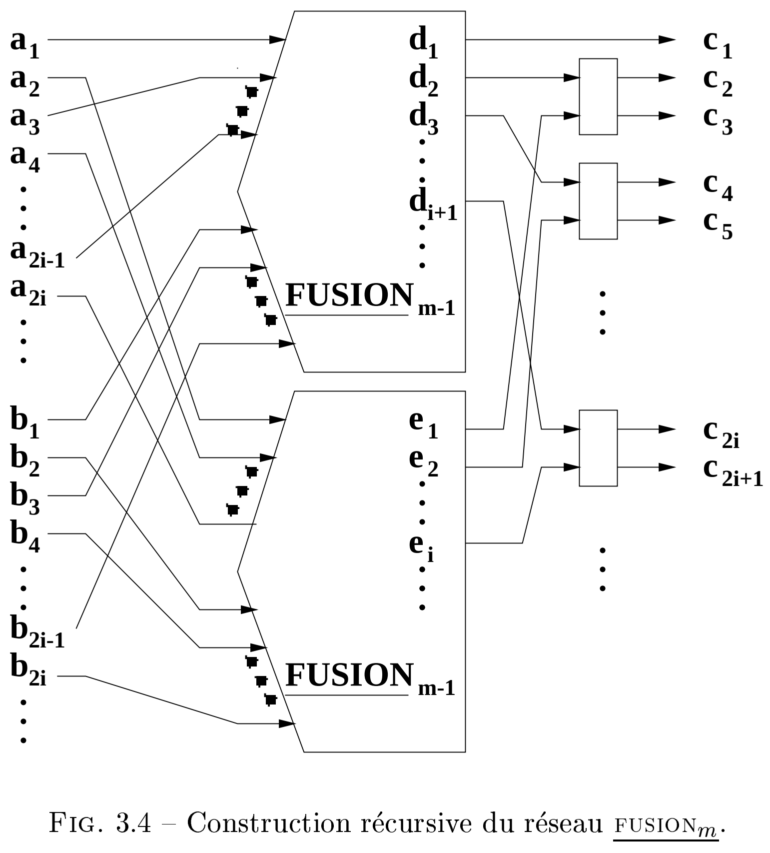


FIG. 3.4 – Construction récursive du réseau FUSION_m.

Pour constituer FUSION_m, on dispose donc de deux réseaux FUSION_{m-1}, et on connecte leurs sorties à l'aide de 2^m-1 comparateurs, comme illustré par la Figure 3.4. C'est ce qui explique a posteriori la rangée de trois comparateurs ajoutée aux deux réseaux FUSION_1 pour construire le réseau $\underline{\text{FUSION}_2}$.

Le temps de calcul t_m du réseau $\underline{\text{FUSION}_m}$ est défini comme le nombre maximal de comparateurs que doit traverser une donnée entre l'entrée et la sortie du réseau. Ainsi, $t_2 = 2$ car certaines données traversent deux comparateurs (bien que d'autres n'en traversent qu'un seul), et $t_3 = 3$. Bien sûr, plusieurs comparateurs peuvent être actifs au même moment (c'est l'idée même du parallélisme!).

Le temps de calcul t_m et le nombre p_m de comparateurs du réseau $\underline{\text{FUSION}_m}$ sont donnés par les récurrences suivantes :

Lemme 7

$$\begin{array}{llll} t_0 = 1 & t_1 = 2 & t_m = t_{m-1} + 1 & (i.e., t_m = m + 1) \\ p_1 = 1 & p_1 = 3 & p_m = 2p_{m-1} + 2^m - 1 & (i.e., p_m = 2^m m + 1) \end{array}$$

Preuve Les formules de récurrence découlent directement de la Proposition 3. L'égalité $p_m = 2^m m + 1$ se démontre directement par récurrence. ■

Ces mesures, exprimées en la taille $n = 2^m$ de chacune des deux listes d'entrée du réseau, sont en $O(\log n)$ et en $O(n \log n)$, respectivement. Le rendement du réseau est faible : si on multiplie le nombre de comparateurs par le temps de calcul, on a le travail (au sens du Paragraphe 2.3.5) $W_n = p_n \times t_n$ de l'ordre de $n(\log n)^2$, ce qui est moins bien que pour une fusion séquentielle linéaire : un seul comparateur et un temps n. Mais le temps d'exécution est très rapide.

Une parenthèse : le faible rendement s'explique par le fait que chaque comparateur ne fonctionne qu'une seule fois au cours de la fusion. On pourrait augmenter le rendement du réseau en lui demandant de fusionner plusieurs paires de listes différentes. Le même réseau peut en effet commencer à traiter deux nouvelles listes à chaque unité de temps, on dit alors qu'il fonctionne en mode pipeline. Après un délai d'initialisation égal à la profondeur du réseau (qui est un autre nom pour t_m), tous les comparateurs sont actifs à chaque top, et deux nouvelles listes fusionnées sont délivrées en sortie à chaque top également (on dit que la période du réseau est 1).

3.2.2 Réseau de tri

Il est maintenant facile de construire par récurrence un réseau $\underline{\text{TRI}_m}$ pour trier $n = 2^m$ éléments. Il suffit de placer deux réseaux $\underline{\text{TRI}_{m-1}}$ devant un réseau $\underline{\text{FUSION}_{m-1}}$, comme illustré Figure 3.5.

Le temps de calcul t'_m et le nombre de comparateurs $p'm$ sont donnés par :

Lemme 8

$$\begin{array}{lll} t'_1 = 1 & t_m = t'_{m-1} + t_{m-1} & (donc\ t'_m = O(m^2)) \\ p'_1 = 1 & p'_m = 2p'_{m-1} + p_{m-1} & (donc\ p'_m = O(2^m m^2)) \end{array}$$

Preuve Les formules de récurrence découlent directement de la construction. Comme $t_m = t'_{m-1} + t_{m-1} = t'_{m-1} + m$, on a $t'_m = O(m^2)$. Pour la seconde récurrence, $p'_m = 2p'_{m-1} + 2^{m-1}(m-1) + 1 = 2^{m-1}(1+2+3+\cdots+(m-1)) + (1+2+4+\cdots+2^{m-1})$, donc $p'_m = O(2^m m^2) + O(2^m) = O(2^m m^2)$. ■

Exprimons ces mesures en la taille $n = 2^m$ de l'entrée du réseau : le temps est en $O((\log n)^2)$, le nombre de comparateurs est en $O(n(\log n)^2)$, et le travail est en $O(n(\log n)^4)$. En pratique, on peut estimer que la taille de l'architecture est prohibitive par rapport au gain en temps, mais, à nouveau, l'idée est de faire fonctionner en pipeline ce réseau très rapide. En théorie, on peut se demander s'il existe un réseau de tri qui fonctionne en temps $O(\log n)$ pour trier une suite de taille n ; un tel réseau serait l'équivalent, pour le modèle plus rigide des réseaux de tri, de la machine à trier P-RAM de Cole (Paragraphe 2.4). La réponse est positive, mais relativement récente : en

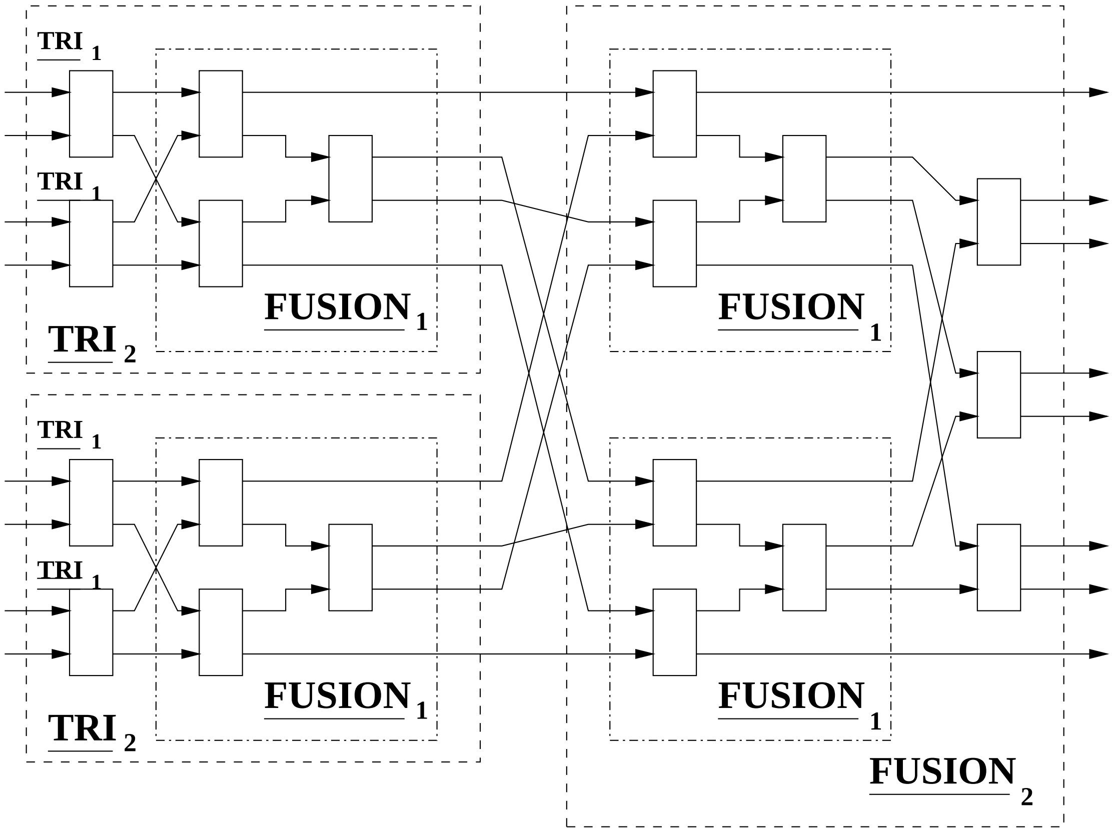


FIG. 3.5 – Construction récursive du réseau de tri TRI_m de Batcher ($m = 3$).

1983, Ajtai, Komlos et Szemeredi [2] ont proposé un réseau de tri qui fonctionne en temps $O(\log n)$ avec $O(n \log n)$ comparateurs. Les constantes qui se cachent derrière les $O(\dots)$ rendent ce résultat inutilisable en pratique. Pour conclure ce paragraphe, nous rappelons le résultat principal :

Théorème 4 *Le réseau de tri-fusion de Batcher trie une suite de taille n en temps $O(\log^2 n)$, avec $O(n(\log^2 n))$ comparateurs.*

3.2.3 Principe du 0-1

On donne ici une technique de preuve différente pour la preuve de correction des réseaux de fusion et de tri. On dit qu'une suite est de type 0-1 si elle n'est constituée que de 0 et de 1.

Proposition 4 *(Principe du 0-1) Un réseau de comparateurs implante correctement le tri si et seulement si il calcule bien cette fonction pour toutes suites de type 0-1 en entrée.*

Preuve Soit f une fonction monotone croissante. Un comparateur se comporte de la même manière sur (x_1, x_2) et sur $(f(x_1), f(x_2))$. On considère maintenant un réseau R fixé, appliqué à une suite donnée $(x_1, \dots, x_n)$. La position finale de l'entrée x_i, i.e., le fil de sortie du réseau où cet x_i aboutit, ne dépend pas de la valeur de x_i mais de sa position relative par rapport aux autres x_j. Donc, si le réseau est appliqué à $(f(x_1), \dots, f(x_n))$, $f(x_i)$ sort par le même fil de sortie que x_i plus haut. Supposons maintenant que R ne calcule pas correctement la fonction de tri. Il existe donc une suite $(x_1, \dots, x_n)$, et il existe i_1 et i_2 tels que x_{i_1} et x_{i_2} sortent de R sur deux sorties consécutives, mais $x_{i_1} > x_{i_2}$, Il suffit de définir une fonction monotone $f : \{x_1, \ldots, x_n\} \to \{0,1\}$ telle que $f(x_{i_1}) = 1$ et $f(x_{i_2}) = 0$, par exemple :

$$f(x) = \begin{cases} 0 & \text{si } x \leq x_{i_2} \\ 1 & \text{si } x > x_{i_2} \end{cases}$$

Alors R ne calcule pas correctement le tri de $(f(x_1), \ldots, f(x_n))$, puisque on trouve 1 et 0 dans cet ordre sur deux sorties consécutives. Or $(f(x_1), \ldots, f(x_n))$ est une suite de type 0-1, d'où la conclusion. ■

On peut utiliser le principe du 0-1 pour prouver la correction des réseaux $\underline{\text{FUSION}_m}$:

Nouvelle preuve de la Proposition 3 On reprend la preuve de la Proposition 3 dans un cadre simplifié. Pour toute suite triée $(x_1, \ldots, x_k)$ de type 0-1, donc de la forme $O^r 1^{k-r}$, on note $r = \text{ZÉROS}(x_1, \ldots, x_m))$. On pose $p = \text{ZÉROS}(a_1, \ldots, a_{2n})$ et $q = \text{ZÉROS}(b_1, \ldots, b_{2n})$. On distingue plusieurs cas :

- $p = 2p'$ et $q = 2q'$: Dans ce cas, on a

  $$\text{ZÉROS}(d_1, \ldots, d_{2n}) = p' + q' = \text{ZÉROS}(e_1, \ldots, e_{2n}) \ .$$

  Les suites $d_1, \ldots, d_{2n}$ et $e_1, \ldots, e_{2n}$ ont le même nombre de 0 (et de 1). Mais il y a un décalage, d_1 (resp : e_{2n}) n'est pas concerné par la rangée des $2n-1$ comparateurs finaux : le i-ème comparateur compare d_{i+1} avec e_i. La suite des couples d'entrée des comparateurs finaux est formée de $p' + q' - 1$ paires de 0, suivies d'une paire 10 (le premier 1 de $(d_1, \ldots, d_{2n})$ et le dernier 0 de $(e_1, \ldots, e_{2n})$), suivie de paires de 1. On obtient la suite triée grâce au $(p'+q')$-ème comparateur, qui est le seul "utile".
- $p = 2p'$ et $q = 2q' - 1$. On a maintenant

  $$\text{ZÉROS}(d_1, \ldots, d_{2n}) = p' + q' \qquad \text{ZÉROS}(e_1, \ldots, e_{2n}) = p' + q' - 1 \ .$$

  La suite des couples d'entrée des comparateurs finaux est formée de $p' + q' - 1$ paires de 0, suivies de paires de 1. Aucun des $2n-1$ comparateurs ne fait un travail utile, et la liste finale $O^{p+q}1^{4n-p-q}$ est obtenue par simple entrelacement.
- $p = 2p' - 1$ et $q = 2q'$. Ce cas est similaire au précédent.
- $p = 2p' - 1$ et $q = 2q' - 1$. Cette fois on a

  $$\text{ZÉROS}(d_1, \ldots, d_{2n}) = p' + q' \qquad \text{ZÉROS}(e_1, \ldots, e_{2n}) = p' + q' - 2 \ .$$

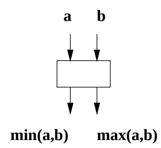
  La suite des couples d'entrée des comparateurs finaux est formée de paires de 0, suivies d'une paire 01, suivie de paires de 1. Aucun des $2n-1$ comparateurs ne fait un travail utile.

■

3.3 Tri sur réseau linéaire

3.3.1 Tri par transposition pair-impair

On décrit ici un réseau de tri très simple, connu sous le nom de réseau de transposition pair-impair dans la littérature. Pour pouvoir replier ce réseau au paragraphe suivant, nous faisons subir une rotation aux comparateurs comme indiqué ci-dessous :



Le réseau est formé d'une succession de lignes de comparateurs. Plus précisément, pour trier une liste de $n = 2p$ éléments, on positionne p copies du réseau formé de deux lignes, dont la première consiste en p comparateurs prenant en entrées les p paires de fils $2i-1$ et $2i$, $1 \leq i \leq p$ (étape "impaire"), et dont la seconde consiste en $p-1$ comparateurs prenant en entrées les $p-1$ paires de fils $2i$ et $2i+1$, $1 \leq i \leq p-1$ (étape "paire") : voir la Figure 3.6 pour $n = 8$. Il y a un total de $p(2p-1) = \frac{n(n-1)}{2}$ comparateurs dans le réseau. Pour n impair, la construction est similaire, illustrée pour $n = 7$ à la Figure 3.6, et comporte également $\frac{n(n-1)}{2}$ comparateurs.

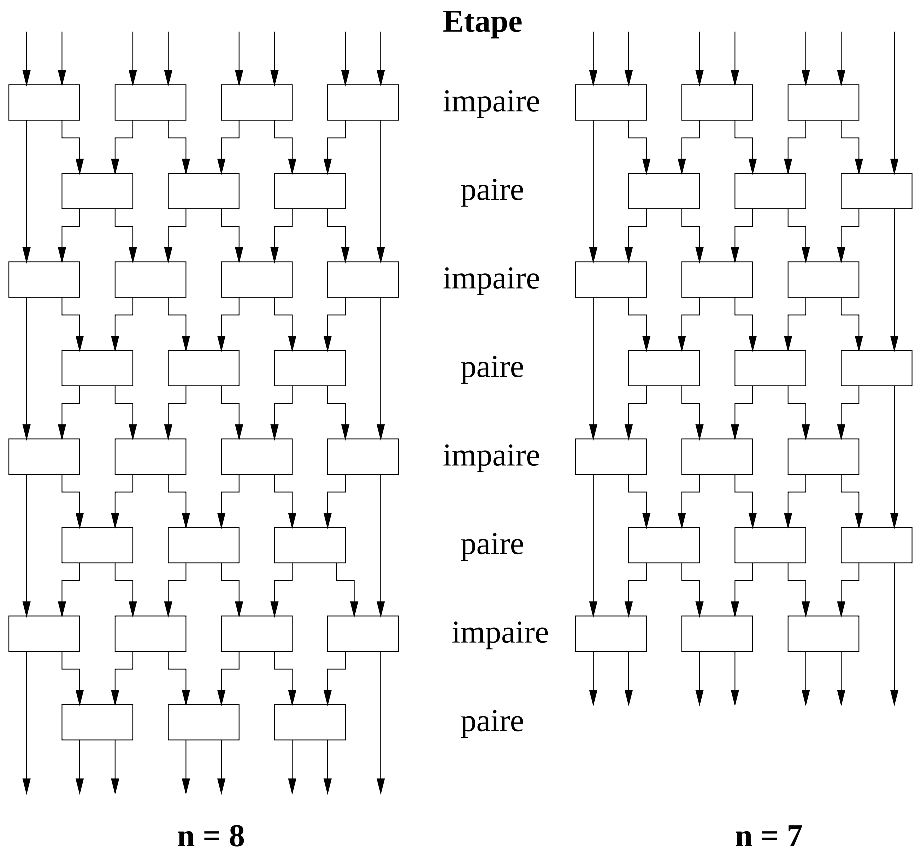


FIG. 3.6 – Construction du réseau de transposition pair-impair.

Proposition 5 *Le réseau de transposition pair-impair implante correctement le tri.*

Preuve L'idée naturelle est de faire une preuve par récurrence sur n. Cette solution s'avère assez technique, car "extraire" un réseau de taille $n-1$ d'un réseau de taille n (pour pouvoir appliquer l'hypothèse de récurrence) n'est pas évident.

Supposons donc que tout réseau de transposition pair-impair de taille $n-1$ implante correctement le tri, et considérons une suite $(a_i)_{1 \leq i \leq n}$ à trier. On note $a_i^{(r)}$ le contenu de la i-ème entrée de la r-ème ligne du réseau, juste avant la r-ème étape d'échanges (avec $a_i^{(1)} = a_i$). Si on imagine que le nombre de lignes du réseau est infini, on peut itérer le schéma de calcul pour toute valeur de r. Si r est assez grand, la suite $(a_i^{(r)})_{1\leq i\leq n}$ va finir par être triée. Mais on doit montrer que ceci est acquis pour $r=n$.

Supposons que le maximum M de la suite $(a_i)_{1\leq i\leq n}$ est à l'indice $i=2j_0$. On a donc $a_{2j_0}^{(1)}=M$, puis $a_{2j_0}^{(2)}=M$ (car l'élément d'indice $2j_0$ n'est pas modifié par la comparaison avec son voisin de gauche à la première étape), puis $a_{2j_0+1}^{(3)}=M$ (car le maximum s'est déplacé d'un cran vers la droite), puis $a_{2j_0+r-2}^{(r)}=M$ pour tout r entre 2 et $n-2j_0+2$.

Le sort du maximum étant connu, définissons le schéma $(b_j^{(r)})_{1\leq j\leq n-1}$ par $b_j^{(1)}=a_j^{(2)}$ pour $j\leq 2j_0-1$ et $b_j^{(1)}=a_{j+1}^{(2)}$ pour $j\geq 2j_0$ (on a supprimé le maximum), schéma qu'on laisse évoluer avec un processeur de moins, mais en commençant par une étape paire ($2j$ échange avec $2j+1$).

Propriété Pour $i\geq 2$, $b_j^{(i)}=\begin{cases} a_j^{(i+1)} & \text{pour} \quad j\leq 2j_0+i-2 \\ a_{j+1}^{(i)} & \text{pour} \quad j\geq 2j_0+i-1 \end{cases}$

En effet à la première étape du schéma b_j, les $2j_0-1$ premiers éléments évoluent comme ceux de $a_j^{(2)}$, tandis que les autres ne bougent pas puisqu'on recommence sur eux la première étape du schéma a_i. Donc $b_j^{(2)}=a_j^{(2+1=3)}$ pour $j\leq 2j_0$ et $b_j^{(2)}=b_j^{(1)}=a_{j+1}^{(2)}$ pour $j\geq 2j_0+1$.

Puis par récurrence, sachant que $b_j^{(i)}=a_j^{(i+1)}$ pour $j\leq 2j_0+i-2$, on a $b_j^{(i+1)}=a_j^{(i+2)}$ pour $j\leq 2j_0+i-2$, car le $2j_0+i-2$-ème élément est comparé vers la gauche et non vers la droite. De plus $b_j^{(i)}=a_{j+1}^{(i)}$ pour $j\geq 2j_0+i-1$, donc $b_j^{(i+1)}=a_{j+1}^{(i+1)}$ pour $j\geq 2j_0+i$. Reste l'élément d'indice $2j_0+i-1$, on a $b_{2j_0+i-1}^{(i+1)}=a_{2j_0+i}^{(i+1)}=a_{2j_0+i-1}^{(i+2)}$ (car échange avec le maximum). Ceci termine la preuve de la propriété. ■

Revenant à la preuve de la proposition, on utilise la propriété pour $i\geq n-2j_0+2$: pour tout j, $1\leq j\leq n-1$, $b_j^{(i)}=a_j^{(i+1)}$. Or par hypothèse de récurrence la suite $b_j^{(n-1)}$ est triée, donc $(a_j^{(n)})_{1\leq j\leq n-1}$ est triée, et le maximum M est en position n, donc $(a_j^{(n)})_{1\leq j\leq n}$ est triée. ■

Une autre preuve de la Proposition 5 Une autre preuve, moins directe, est proposée dans les exercices 36 et 37 du paragraphe 5.3.4 de Knuth [44]. La preuve fait un détour par les réseaux de tri primitifs, i.e. qui n'effectuent que des comparaisons entre éléments voisins (le réseau de transposition pair-impair est bien primitif). On modélise un réseau de tri α comme la suite de ses comparateurs, i.e. $\alpha=[x_k,y_k]\circ\ldots\circ[x_2,y_2]\circ[x_1,y_1]$. k est le nombre de comparateurs du réseau α. Pour un réseau primitif, $y_i=x_{i+1}$ La preuve passe par deux lemmes :

Propriété Soit α un réseau primitif pour n éléments. α implante correctement le tri ssi α trie le vecteur $(n,n-1,\ldots,2,1)$.

Preuve Par contradiction, soit x un vecteur d'entrée tel que $\alpha(x)_i>\alpha(x)_j$, avec $i<j$. Soit $y=(n,n-1,\ldots,2,1)$, nous allons montrer que $\alpha(y)_i>\alpha(y)_j$, ce qui établira la propriété. On procède par récurrence sur le nombre k de comparateurs de α. Soit donc $\alpha=[p,p+1]\circ\beta$, avec la propriété vraie pour β. Il faut distinguer plusieurs cas suivant les valeurs de p :

Si $p=i$ On a $\alpha(x)_i=\alpha(x)_p=\min(\beta(x)_p,\beta(x)_{p+1})>\alpha(x)_j$, $\alpha(x)_p\leq\alpha(x)_{p+1}=\max(\beta(x)_p,\beta(x)_{p+1})$, donc $j\neq p+1$, $j>p+1$ et $\alpha(x)_j=\beta(x)_j$. Ainsi $\beta(x)_p>\beta(x)_j$ et $\beta(x)_{p+1}>\beta(x)_j$, donc par récurrence $\beta(y)_p>\beta(y)_j$ et $\beta(y)_{p+1}>\beta(y)_j$, et enfin $\alpha(y)_i>\alpha(y)_j$.

Si $p = i - 1$ On a $\alpha(x)_i = \alpha(x)_{p+1} = \max(\beta(x)_p, \beta(x)_{p+1}) > \alpha(x)_j = \beta(x)_j$, donc soit $\beta(x)_p > \beta(x)_j$, soit $\beta(x)_{p+1} > \beta(x)_j$, donc par récurrence soit $\beta(y)_p > \beta(y)_j$, soit $\beta(y)_{p+1} > \beta(y)_j$, et enfin $\alpha(y)_{p+1} = \max(\beta(y)_p, \beta(y)_{p+1}) > \beta(y)_j = \alpha(y)_j$.

Si $p = j$ **ou** $p - 1$ L'argumentation est similaire aux deux cas précédents.

Autres cas Le résultat est immédiat.

■

Propriété Un réseau primitif pour n éléments qui implante correctement le tri a au moins $n(n-1)/2$ comparateurs.

Preuve Chaque comparateur du réseau α réduit le nombre d'inversions de la donnée x en entrée (considérée comme une permutation $x \to \alpha(x)$ dans $\{1, \ldots, n\}$) de 0 ou de 1. Or la permutation $y = (n, n-1, \ldots, 2, 1)$ a exactement $n(n-1)/2$ inversions. ■

Terminons maintenant la preuve de validité du réseau de transposition pair-impair. Celui-ci a exactement $n(n-1)/2$ comparateurs. Si on applique le vecteur $y = (n, n-1, \ldots, 2, 1)$ en entrée, chaque comparateur va donc effectivement réaliser un échange. On peut donc suivre chaque donnée en entrée et déterminer sa position à chaque étape. La permutation qui est répétée, disons pour n pair, $n = 2m$, est alternativement $\Pi_1 = (1,2) \circ (3,4) \circ \ldots \circ (2m-1, 2m)$ (étape impaire) puis $\Pi_2 = (2,3) \circ (4,5) \circ \ldots \circ (2m-2, 2m-1)$ (étape paire). En posant $\Pi = \Pi_2 \circ \Pi_1$, le réseau réalise la permutation Π^m. Or $\Pi^m = (1, 2m) \circ (2, 2m-1) \circ \ldots \circ (m-1, m)$, d'où le résultat. ■

Les caractéristiques du réseau de transposition pair-impair sont les suivantes : le temps de calcul est $t_n = n$ et le nombre de comparateurs est $p_n = n(n-1)/2$, d'où un travail $W_n = O(n^3)$. Rien de bien séduisant disions-nous, si ce n'est la simplicité de la solution.

3.3.2 Tri pair-impair sur réseau linéaire de processeurs

Nous abandonnons ici les réseaux de tri pour présenter un algorithme destiné à s'exécuter sur un réseau linéaire de processeurs. Cet algorithme s'inspire directement du réseau précédent : l'idée est de "replier" le réseau pour obtenir un réseau linéaire où les processeurs communiquent alternativement avec leur voisin de gauche et leur voisin de droite, comme expliqué à la Figure 3.7.

Mais profitant de la flexibilité qu'offrent des processeurs généraux, on décide de n'en utiliser que $p \ll n$ pour trier des suites de taille n. Supposons n divisible par p : chaque processeur dispose au début d'une sous-suite de taille n/p. Ces sous-suites sont triées en parallèle, séquentiellement sur chaque processeur, donc en temps $O(\frac{n}{p} \log \frac{n}{p}) = O(\frac{n}{p} \log n)$. Ensuite, l'algorithme de tri fonctionne en p étapes d'échanges alternés, selon le principe du réseau de transposition pair-impair, mais en échangeant des suites de taille n/p à la place d'un seul élément. Quand deux processeurs voisins communiquent, leurs deux suites de taille n/p sont fusionnées, le processeur de gauche conserve la première moitié, i.e. les n/p plus petits éléments, et celui de droite conserve la deuxième moitié. L'algorithme est illustré à la à la Figure 3.7. Le coût d'une étape est celui de la fusion, donc $O(n/p)$, il y a p étapes, d'où un coût en $O(n)$ pour les fusions par échange. Rajoutant le coût du tri initial, on obtient un temps de calcul en $O(\frac{n}{p} \log n + n)$, et un travail en $O(n(p + \log n))$: l'algorithme est optimal pour $p \leq \log n$.

La preuve de validité de l'algorithme découle directement de la Proposition 5. Nous avons donc prouvé la :

	P_1		P_2		P_3		P_4		P_5		P_6
init	{8,3,12}		{10,16,5}		{2,18,9}		{17,15,4}		{1,6,13}		{11,7,14}
tri local	{3,8,12}		{5,10,16}		{2,9,18}		{4,15,17}		{1,6,13}		{7,11,14}
impair	{3,5,8}	↔	{10,12,16}		{2,4,9}	↔	{15,17,18}		{1,6,7}	↔	{11,13,14}
pair	{3,5,8}		{2,4,9}	↔	{10,12,16}		{1,6,7}	↔	{15,17,18}		{11,13,14}
impair	{2,3,4}	↔	{5,8,9}		{1,6,7}	↔	{10,12,16}		{11,13,14}	↔	{15,17,18}
pair	{2,3,4}		{1,5,6}	↔	{7,8,9}		{10,11,12}	↔	{13,14,16}		{15,17,18}
impair	{1,2,3}	↔	{4,5,6}		{7,8,9}	↔	{10,11,12}		{13,14,15}	↔	{16,17,18}
pair	{1,2,3}		{4,5,6}	↔	{7,8,9}		{10,11,12}	↔	{13,14,15}		{16,17,18}

FIG. 3.7 – Tri-fusion pair-impair sur réseau linéaire de processeurs ($p = 6$).

Proposition 6 *Sur un réseau linéaire de p processeurs généraux, on peut trier une suite de taille n en temps $O(\frac{n}{p}\log n + n)$. Cet algorithme est optimal pour $p \leq \log n$.*

Notes bibliographiques

Les notes de ce chapitre s'inspirent des livres de Gibbons et Rytter [37] (tri de Batcher) et d'Akl [3] (tri linéaire). Pour le lecteur curieux, le savoir de Knuth [44] sur les réseaux de tri est, comme d'habitude, encyclopédique.

Chapitre 4

Ordonnancement

4.1 Introduction

Ce chapitre expose quelques résultats fondamentaux sur l'ordonnancement de graphes de tâches. On commence par un exemple motivant cette étude avant de brièvement passer en revue les différents modèles considérés, et les résultats de complexité associés.

4.1.1 D'où viennent les graphes de tâches ?

Considérons la méthode classique de résolution d'un système linéaire $Ax = b$, où A est un matrice triangulaire inférieure inversible de taille $n \times n$, et b un vecteur à n composantes :

```
DO i=1, n
   Tâche T_{i,i} : x(i) = b(i) / a(i,i)
   DO j = i+1, n
     Tâche T_{i,j} : b(j) = b(j) - a(j,i) * x(i)
   ENDDO
ENDDO
```

Pour une valeur donné de i, $1 \leq i \leq n$, toutes les tâches $T_{i,*}$ représentent des calculs exécutés à la i-ème itération de la boucle externe. Le calcul de $x(i)$ est effectué en premier (tâche $T_{i,i}$). Puis chaque composante $b(j)$, où $j > i$, du vecteur b, est mise à jour (tâche $T_{i,j}$).

Dans le programme, il y a un ordre de dépendance total entre les tâches. Notons $T <_{seq} T'$ si la tâche T est exécutée avant la tâche T' dans le code séquentiel initial. On a

$$T_{1,1} <_{seq} T_{1,2} <_{seq} T_{1,3} <_{seq} \ldots <_{seq} T_{1,n} <_{seq} T_{2,2} <_{seq} T_{2,3} <_{seq} \ldots <_{seq} T_{n,n}$$

Cependant, il y a des tâches indépendantes qui peuvent être exécutées en parallèle. Intuitivement, les tâches indépendantes sont celles dont on peut changer l'ordre d'exécution sans changer le résultat de l'exécution du programme. Une condition suffisante pour cela est ces tâches n'accèdent pas aux mêmes variables : elles peuvent lire la même valeur, mais ne peuvent pas écrire dans le même emplacement mémoire (sinon le résultat dépendrait de l'identité de la dernière opération). Plus formellement, chaque tâche T a un ensemble de variables d'entrée $\mathrm{In}(T)$ (valeurs lues) et un ensemble de valeurs de sortie $\mathrm{Out}(T)$ (valeurs écrites). Dans l'exemple, $\mathrm{In}(T_{i,i}) = \{b(i), a(i,i)\}$ et $\mathrm{Out}(T_{i,i}) = \{x(i)\}$. Pour $j > i$, $\mathrm{In}(T_{i,j}) = \{b(j), a(j,i), x(i)\}$ et $\mathrm{Out}(T_{i,j}) = \{b(j)\}$. Deux tâches T

and T' ne sont pas indépendantes (on écrit $T \perp T'$) si elles partagent une même variable écrite :

$$T \perp T' \Leftrightarrow \left\{ \begin{array}{ll} & \mathrm{In}(T) \cap \mathrm{Out}(T') \neq \emptyset \\ ou & \mathrm{Out}(T) \cap \mathrm{In}(T') \neq \emptyset \\ ou & \mathrm{Out}(T) \cap \mathrm{Out}(T') \neq \emptyset \end{array} \right.$$

Ces conditions sont connues sous le nom de conditions de Bernstein [14]. On montrera au Chapitre 8 comment déterminer si ces conditions sont réalisées ou non. Vérifions à la main quelques exemples : les tâches $T_{1,1}$ et $T_{1,2}$ ne sont pas indépendantes parce que $\mathrm{Out}(T_{1,1}) \cap \mathrm{In}(T_{1,2}) = \{x(1)\}$; donc $T_{1,1} \perp T_{1,2}$. De même, $\mathrm{Out}(T_{1,3}) \cap \mathrm{Out}(T_{2,3}) = \{b(3)\}$, d'où $T_{1,3} \perp T_{2,3}$.

Etant donnée la relation de dépendance $\perp$, on peut extraire un ordre partiel de l'ordre total $<_{seq}$ induit par l'exécution séquentielle du programme. Si deux tâches T et T' sont dépendantes, i.e., $T \perp T'$, on les ordonne de manière à respecter l'ordre de l'exécution séquentielle : on écrit $T \prec T'$ si on a à la fois $T \perp T'$ et $T <_{seq} T'$. La relation de précédence $\prec$ représente les dépendances qui doivent être satisfaites pour préserver la sémantique du programme initial ; si $T \prec T'$, alors T est exécutée avant T' dans le code séquentiel, et doit encore être exécutée avant T' même si on dispose d'une infinité de ressources, car T et T' partagent une variable en écriture.

Pour définir $\prec$ plus précisément en termes de relations d'ordre, on prend la fermeture transitive de l'intersection de $\perp$ et $<_{seq}$ pour déterminer l'ensemble de toutes les contraintes qui doivent être satisfaites pour préserver la sémantique du programme initial :

$$\prec = (<_{seq} \cap \perp)^+$$

où $^+$ représente la fermeture transitive. Dans un sens, $\prec$ capture la séquentialité intrinsèque du programme initial. L'ordre séquentiel total $<_{seq}$ est trop restrictif, seul l'ordre partiel $\prec$ doit être respecté. Pourquoi prendre la fermeture transitive ? Dans l'exemple, on a $T_{2,4} \perp T_{4,4}$ (ce qui n'est pas une relation de prédécesseur à successeur car il y a $T_{3,4}$ au milieu) et $T_{4,4} \perp T_{4,5}$, d'où un chemin de dépendances de $T_{2,4}$ à $T_{4,5}$, alors qu'on n'a pas $T_{2,4} \perp T_{4,5}$. Il faut suivre les chaînes de dépendances pour définir $\prec$ correctement.

On dessine un graphe orienté pour représenter les contraintes de dépendance qui doivent être satisfaites. Les sommets du graphe sont les tâches, et les arêtes expriment les contraintes de dépendance. Une arête $e : T \rightarrow T'$ dans le graphe signifie que l'exécution de T' ne peut commencer qu'après la fin de l'exécution de T, quel que soit le nombre de ressources disponibles. D'habitude on ne dessine pas les arêtes de transitivité sur le graphe, car elles représentent de l'information redondante ; si $T \prec T'$ et $T' \prec T''$, et s'il existe une dépendance $T \perp T''$, alors celle-ci sera automatiquement satisfaite. On dit que T est un prédécesseur de T' si $T \prec T'$ et s'il n'y a pas de tâche T'' entre, i.e. telle que $T \prec T''$ et $T'' \prec T'$. Dans l'exemple, les relations prédécesseur-successeur sont les suivantes :

- $T_{i,i} \prec T_{i,j}$ pour $1 \leq i < j \leq n$
  (le calcul de $x(i)$ doit être exécuté avant la mise à jour de $b(j)$ au pas i de la boucle externe).
- $T_{i,j} \prec T_{i+1,j}$ pour $1 \leq i < j \leq n$
  (la mise à jour de $b(j)$ au pas i de la boucle externe doit [1] précéder sa lecture au pas $i + 1$).

On obtient le graphe de la Figure 4.1. On utilisera ce graphe plusieurs fois dans le chapitre pour illustrer les notions introduites.

[1]En fait, le terme "doit" est légèrement exagéré, parce que l'addition est une opération associative et commutative. C'est vrai à la façon dont le programme est écrit, en ré-utilisant le même emplacement mémoire $b(j)$ pour toutes les mises à jour. Cette dépendance de sortie peut être éliminée à l'aide de techniques standard [63].

4.1.2 Tour d'horizon

Le chapitre commence par quelques résultats élémentaires (Paragraphe 4.2). On considère alors un modèle très simple, où les coûts de communications sont négligés : le problème sans limitation de ressources est polynômial (Paragraphe 4.3), tandis que celui à ressources bornées est NP-complet, et résolu de manière sous-optimale à l'aide d'heuristiques de listes (Paragraphe 4.4). On passe alors à un modèle plus réaliste, où les coûts de communication sont pris en compte (Paragraphe 4.5). La difficulté s'accroît notablement, car même le problème sans limitation de ressources devient NP-complet (Paragraphe 4.6). Le chapitre se conclut par la présentation de plusieurs heuristiques au Paragraphe 4.7.

4.2 Ordonnancement des DAGs

Définition 2 *Un système de tâches est un graphe orienté $G = (V, E, w)$ où*
- *l'ensemble V des sommets représente les tâches (V est fini).*
- *l'ensemble E des arêtes représente les contraintes de précédence entre tâches :* $e = (u, v) \in E$ *ssi* $u \prec v$.
- *la fonction de temps $w : V \longrightarrow \mathbb{N}^*$ donne le poids (ou durée) de chaque tâche. Les poids sont supposés être des entiers positifs* [2].

Pour le système triangulaire (Figure 4.1), on peut supposer que toutes les tâches ont même poids et poser $w(T_{i,j}) = 1$ pour $1 \leq i \leq j \leq n$. On pourrait aussi décider qu'une division est plus coûteuse qu'une multiplication-addition et donner un poids plus fort aux tâches diagonales tâches $T_{i,i}$.

Un ordonnancement σ d'un système de tâches est une fonction qui attribue un instant de début d'exécution à chaque tâche. On peut aussi définir une fonction d'allocation alloc qui attribue à chaque tâche un processeur responsable de son exécution :

Définition 3 *Un ordonnancement d'un système de tâches $G = (V, E, w)$ est une fonction $\sigma : V \longrightarrow \mathbb{N}^*$ telle que $\sigma(u) + w(u) \leq \sigma(v)$ pour toute arête $e = (u, v) \in E$.*

Ainsi un ordonnancement doit respecter les *contraintes de dépendance* induites par la relation $\prec$ et matérialisées par les arêtes du graphe de dépendance. Si $u \prec v$, l'exécution de u commence à l'instant $\sigma(u)$, nécessite $w(u)$ unités de temps, et le début de l'exécution de v à l'instant $\sigma(v)$ doit être postérieur à la fin de l'exécution de u.

Il y a souvent d'autres contraintes que doivent respecter les ordonnancements, à savoir *les contraintes de ressource*. Quand on dispose d'un nombre infini de processeurs [3], on dit qu'on a un problème à ressources infinies, noté Pb(∞). Quand on ne dispose que d'un nombre fini p de processeurs, on parle d'un problème à ressources bornées Pb(p). Les contraintes de ressource s'expriment comme suit : chaque tâche $v \in V$ est allouée à un processeur alloc(v) par la fonction d'allocation alloc : $V \longrightarrow \mathcal{P}$, où $\mathcal{P} = \{1, \ldots, p\}$ est l'ensemble des processeurs disponibles. La relation entre l'ordonnancement σ et l'allocation alloc est qu'un processeur ne peut effectuer (au plus) qu'une seule tâche à un instant donné :

$$\text{alloc}(T) = \text{alloc}(T') \Rightarrow \begin{cases} & \sigma(T) + w(T) \leq \sigma(T') \\ ou & \sigma(T') + w(T') \leq \sigma(T) \end{cases}$$

[2] Ce n'est pas restrictif ; les poids des tâches peuvent être des rationnels : comme il n'y a qu'un nombre fini de tâches, on peut toujours se ramener à des entiers.

[3] En fait on n'a jamais besoin d'un nombre de processeurs supérieur au nombre de tâches.

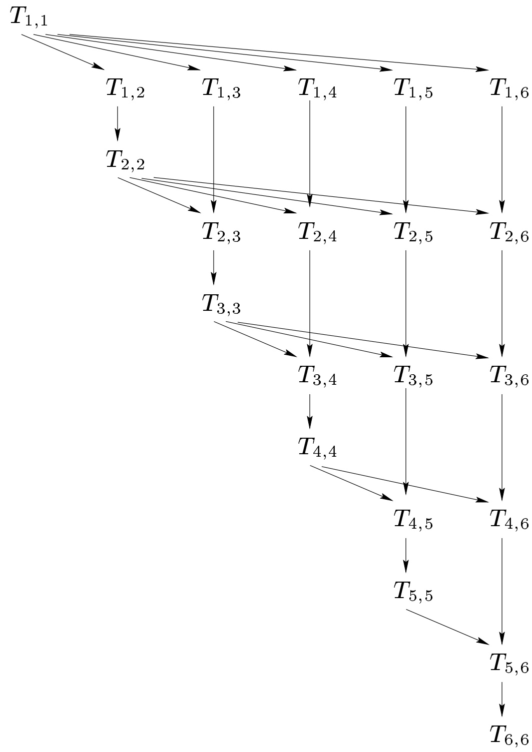


FIG. 4.1 – Graphe de tâches pour le système triangulaire ($n = 6$).

Cette condition exprime le fait que si deux tâches T et T' sont allouées au même processeur, alors l'exécution de T doit être terminée avant que l'exécution de T' puisse commencer, ou vice-versa.

Etant donné un système de tâches, l'existence d'un ordonnancement est liée à une condition élémentaire sur le graphe, indépendamment des ressources :

Théorème 5 *Soit $G = (V, E, w)$ un système de tâches. Il existe un ordonnancement ssi G est sans cycle.*

Preuve Clairement, s'il y a un cycle $v_1 \to v_2 \ldots \to \ldots \to v_k \to v_1$, alors $v_1 \prec v_1$ (v_1 dépend de lui-même), et un ordonnancement σ vérifierait $\sigma(v_1) + w(v_1) \leq \sigma(v_1)$, impossible car $w(v_1) > 0$.

Réciproquement, si G est sans cycle, alors il existe des sommets sans prédécesseur (V est fini) par lesquels on peut commencer l'exécution. Plus précisément, on fait un tri topologique des sommets et on les ordonnance l'un après l'autre, sur un seul et même processeur, en respectant l'ordre topologique. Formellement, si $v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}$ est la liste ordonnée des sommets après le tri topologique, on pose $\sigma(v_{\pi(1)}) = 0$ et $\sigma(v_{\pi(i)}) = \sigma(v_{\pi(i-1)}) + w(v_{\pi(i-1)})$ pour $2 \leq i \leq n$. Les contraintes de dépendance sont bien respectées : si $v_i \prec v_j$, alors le tri topologique nous assure que $\pi(i) < \pi(j)$. ■

On s'attache donc à trouver des ordonnancements pour les graphes orientés sans cycle (DAG, pour *Directed Acyclic Graph*), d'où la définition :

Définition 4
- *Un DAG $G = (V, E, w)$ est un système de tâches (au sens de la Définition 3) où G est un graphe orienté sans cycle.*
- *Soit $G = (V, E, w)$ un DAG.*

1. *Soit σ un ordonnancement pour G. Supposons que σ utilise au plus p processeurs (et posons $p = \infty$ si les ressources sont illimitées). La latence $MS(\sigma, p)$ (MS pour* make-span *en anglais) de σ est son temps d'exécution total :*

$$MS(\sigma, p) = \max_{v \in V}\{\sigma(v) + w(v)\} - \min_{v \in V}\{\sigma(v)\}$$

2. *$Pb(\infty)$ est le problème de déterminer un ordonnancement σ de latence minimale $MS(\sigma, \infty)$ en supposant disposer de ressources illimitées. Soit $MS_{opt}(\infty)$ la latence minimale d'un ordonnancement avec une infinité de ressources : $MS_{opt}(\infty) = \min_\sigma MS(\sigma, \infty)$.*
3. *$Pb(p)$ est le problème de déterminer un ordonnancement σ de latence minimale $MS(\sigma, p)$ en supposant disposer de p processeurs. Soit $MS_{opt}(p)$ la latence minimale d'un ordonnancement avec p processeurs.*

Quand la première tâche est ordonnancée à l'instant 0, l'expression de la latence se simplifie en $\mathrm{MS}(\sigma, p) = \max_{v \in V}\{\sigma(v) + w(v)\}$. On étend la définition des poids aux chemins de G comme d'habitude : si $\Phi = (T_1, T_2, \ldots, T_n)$ est un chemin de G, $w(\Phi) = \sum_{i=1}^{n} w(T_i)$. Parce que tout ordonnancement respecte les dépendances, on a facilement la borne suivante sur la latence :

Proposition 7 *Soit $G = (V, E, w)$ un DAG et σ a ordonnancement pour G avec p processeurs. Alors $MS(\sigma, p) \geq w(\Phi)$, pour tout chemin Φ de G.*

Preuve Soit un chemin quelconque $\Phi = (T_1, T_2, \ldots, T_n)$ de G : $e = (T_i, T_{i+1}) \in E$ pour $1 \leq i < n$. Alors $\sigma(T_i) + w(T_i) \leq \sigma(T_{i+1})$ pour $1 \leq i < n$, et donc $\mathrm{MS}(\sigma, p) \geq w(T_n) + \sigma(T_n) - \sigma(T_1) \geq \sum_{i=1}^{n} w(T_i) = w(\Phi)$. ■

Notre dernière définition introduit les notions de facteur d'accélération et d'efficacité pour les ordonnancements (voir [46] pour une discussion plus détaillée de ces notions) :

Définition 5 *Soit $G = (V, E, w)$ un DAG, et σ un ordonnancement pour G avec p processeurs :*

1. *Le facteur d'accélération (speedup) est $s(\sigma, p) = \frac{Seq}{MS(\sigma,p)}$, où $Seq = \sum_{v \in V} w(v)$ est la somme des poids de toutes les tâches.*
2. *L'efficacité est $e(\sigma, p) = \frac{s(\sigma,p)}{p} = \frac{Seq}{p \times MS(\sigma,p)}$.*

$\mathrm{Seq} = \mathrm{MS}_{opt}(1)$ est la latence minimale d'un ordonnancement avec un seul processeur. On a le résultat bien connu :

Proposition 8 *Soit $G = (V, E, w)$ un DAG. Pour tout ordonnancement σ à p processeurs,*

$$0 \leq e(\sigma, p) \leq 1$$

Preuve Considérons l'exécution de σ comme illustré sur la Figure 4.2 (c'est un exemple fictif, qui n'est pas lié à notre système triangulaire). A chaque instant durant l'exécution, certains processeurs sont actifs, et d'autres non. A la fin toutes les tâches ont été exécutées. Soit Idle le temps d'inactivité cumulé des p processeurs au cours de toute l'exécution. Comme Seq est la somme des poids des tâches, la somme Seq + Idle est égale à la surface du rectangle dans la Figure 4.2, i.e. le produit du nombre de processeurs par la latence de l'ordonnancement : $\mathrm{Seq} + \mathrm{Idle} = p \times \mathrm{MS}(\sigma, p)$. Donc $e(\sigma, p) = \frac{\mathrm{Seq}}{p \times \mathrm{MS}(\sigma,p)} \leq 1$. ■

On peut paraphraser le résultat précédent en disant que le facteur d'accélération est borné par p avec p processeurs. Pas d'accélération super-linéaire dans notre modèle ! Voici enfin un résultat

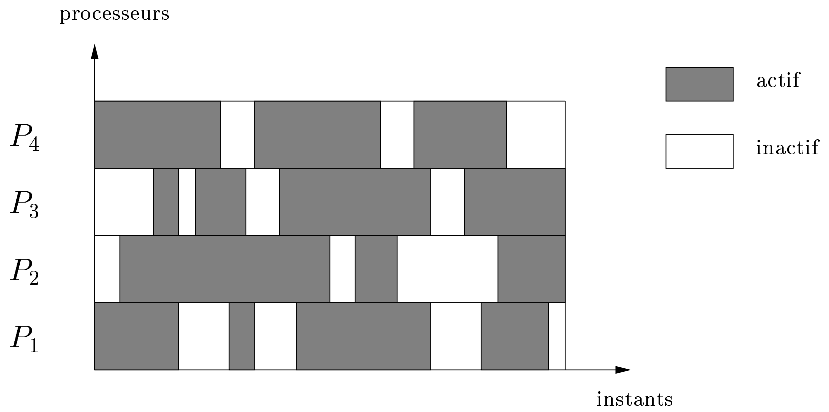


FIG. 4.2 – Processeurs actifs et inactifs au cours de l'exécution.

facile : plus on a de ressources, et plus la latence optimale diminue quand le nombre de ressources augmente :

Proposition 9 *Soit $G = (V, E, w)$ un DAG. On a :*

$$Seq = MS_{opt}(1) \geq \ldots \geq MS_{opt}(p) \geq MS_{opt}(p+1) \geq \ldots \geq MS_{opt}(\infty)$$

Preuve Soit un ordonnancement σ optimal avec p processeurs, et considérons le comme un ordonnancement avec $p+1$ processeurs, où le dernier processeur reste inactif. Alors $\mathrm{MS}(\sigma, p+1) = \mathrm{MS}(\sigma, p) = \mathrm{MS}_{opt}(p)$, donc $\mathrm{MS}_{opt}(p+1) \leq \mathrm{MS}_{opt}(p)$. ■

Précisons le résultat. Le nombre de processeurs réellement utilisés par un ordonnancement σ est $|\mathrm{alloc}(V)|$, i.e. le nombre de processeurs qui exécutent au moins une tâche. Définissons $\mathrm{MS}'(p)$ comme la latence minimale de tous les ordonnancements qui utilisent exactement p processeurs, ; on a $\mathrm{MS}'(p) = \mathrm{MS}_{opt}(p)$ pour $1 \leq p \leq |V|$, donc la Proposition 9 reste valide en remplaçant $\mathrm{MS}_{opt}(p)$ par $\mathrm{MS}'(p)$. Intuitivement, cela ne peut pas nuire à la latence d'utiliser davantage de processeurs dans un modèle où les coûts de communication ne sont pas pris en compte ! Au Paragraphe 4.6, avec des coûts de communication, on verra un exemple où $\mathrm{MS}'(p) < \mathrm{MS}'(p')$ bien que $p < p'$.

Nous voila prêts pour la recherche d'ordonnancements optimaux. Sans surprise, le problème $\mathrm{Pb}(p)$ avec ressources limitées s'avère plus difficile que $\mathrm{Pb}(\infty)$, dont nous donnons la solution au paragraphe suivant.

4.3 Résolution de Pb(∞)

Soit $G = (V, E, w)$ un DAG donné ; supposons disposer d'une infinité de ressources. Un ordonnancement σ pour G est dit *optimal* si sa latence $\mathrm{MS}(\sigma, \infty)$ est minimale, i.e. si $\mathrm{MS}(\sigma, \infty) = \mathrm{MS}_{opt}(\infty)$.

Définition 6 *Soit $G = (V, E, w)$ un graphe acyclique orienté.*

1. *Pour $v \in V$, $PRED(v)$ est l'ensemble des prédécesseurs immédiats de v, et $SUCC(v)$ l'ensemble de ses successeurs immédiats.*
2. *$v \in V$ est un sommet d'entrée (supérieur) ssi $PRED(v) = \emptyset$.*
3. *$v \in V$ est un sommet de sortie (inférieur) ssi $SUCC(v) = \emptyset$.*

4. *Pour $v \in V$, le niveau supérieur $tl(v)$ (top level) est le plus grand poids d'un chemin menant d'un sommet d'entrée à v, en excluant the poids de v.*
5. *Pour $v \in V$, le niveau inférieur $bl(v)$ (bottom level) est le plus grand poids d'un chemin menant de v à un sommet de sortie, en incluant le poids de v.*

Dans l'exemple du système triangulaire, il y a un seul sommet d'entrée, $T_{1,1}$, et un seul sommet de sortie, $T_{n,n}$. Le niveau supérieur de $T_{1,1}$ est 0, et $tl(T_{1,2}) = tl(T_{1,1}) + w(T_{1,1}) = 1$. On a

$$tl(T_{2,3}) = \max\{w(T_{1,1}) + w(T_{1,2}) + w(T_{2,2}), w(T_{1,1}) + w(T_{1,3})\} = 3$$

parce qu'il y a deux chemins du sommet d'entrée à $T_{2,3}$.

Le niveau supérieur des sommets peut être calculé en traversant le graphe ; le niveau supérieur d'un sommet d'entrée est 0, et celui d'un sommet arbitraire v (qui n'est pas un sommet d'entrée) est

$$tl(v) = \max\{tl(u) + w(u); u \in \mathrm{PRED}(v)\}$$

De même, $bl(v) = \max\{bl(u); u \in \mathrm{SUCC}(v)\} + w(v)$ (et $bl(v) = w(v)$ pour un sommet de sortie v). Le niveau supérieur d'un sommet v correspond à l'instant d'exécution de v le plus tôt possible, alors que son niveau inférieur représente une borne inférieure sur le temps total d'exécution restant quand on commence l'exécution de v.

Proposition 10 *Soit $G = (V, E, w)$ un DAG. On définit l'ordonnancement libre σ_{free} par*

$$\sigma_{free}(v) = tl(v), \forall v \in V$$

Alors σ_{free} est un ordonnancement optimal de G.

Preuve On montre d'abord que σ_{free} est bien un ordonnancement, puis on montre son optimalité. Les deux sont faciles :

1. σ_{free} respecte les contraintes de dépendance par construction ; si $(u, v) \in E$, alors $u \in \mathrm{PRED}(v)$ et la contrainte $\sigma_{free}(v) \geq \sigma_{free}(u) + w(u)$ est prise en compte dans le calcul de $tl(v) = \sigma_{free}(v)$.
2. Pour l'optimalité de σ_{free}, on fait un tri topologique sur les sommets de G, et on montre par récurrence que tous les sommets sont ordonnancés dès que possible, i.e. dès que l'exécution de tous leurs prédécesseurs est terminée.

L'ordonnancement libre σ_{free} est aussi appelé ordonnancement au plus tôt (ASAP, as soon as possible). ■

On déduit de la Proposition 10 que

$$\mathrm{MS}_{opt}(\infty) = \mathrm{MS}(\sigma_{free}, \infty) = \max_{v \in V}\{tl(v) + w(v)\}$$

Ainsi $\mathrm{MS}_{opt}(\infty)$ est simplement le poids maximal d'un chemin dans le graphe. Soulignons que σ_{free} n'est pas le seul ordonnancement optimal. Par exemple l'ordonnancement σ_{late} (as late as possible, ALAP) est aussi optimal ; σ_{late} est défini comme suit :

$$\forall v \in V, \sigma_{late}(v) = \mathrm{MS}(\sigma_{free}, \infty) - bl(v)$$

Pour comprendre cette définition, notons que $bl(v)$ est le poids maximal d'un chemin de v aux sommets de sortie, d'où la nécessité de ne pas commencer l'exécution de v plus tard qu'au top $\mathrm{MS}(\sigma_{free}, \infty) - bl(v)$ si on doit terminer l'exécution de toutes les tâches en $\mathrm{MS}(\sigma_{free}, \infty)$ unités de temps.

Corollaire 1 *Soit $G = (V, E, w)$ un DAG. $Pb(\infty)$ peut être résolu en temps $O(|V| + |E|)$.*

Preuve La Proposition 10 nous dit que l'ordonnancement optimal σ_{free} peut être calculé à l'aide des niveaux supérieurs des sommets, et que $\mathrm{MS}_{opt}(\infty)$ est le poids maximal d'un chemin dans le graphe. Comme G est acyclique, ces valeurs peuvent être obtenues en traversant le graphe, d'où une complexité en $O(|V| + |E|)$. ■

Revenons à l'exemple du système triangulaire (Figure 4.1). Toutes les tâches ayant un poids égal 1, le poids d'un chemin est égal à sa longueur plus 1. Le plus long chemin est

$$T_{1,1} \to T_{1,2} \to T_{2,2} \to \ldots \to T_{n-1,n-1} \to T_{n,-1,n} \to T_{n,n},$$

de poids $2n-1$. On peut obtenir une exécution en temps $2n-1$ avec, par exemple, $n-1$ processeurs. Soit $1 \leq i \leq n$; à l'instant $2i-2$, le processeur P_1 commence l'exécution de la tâche $T_{i,i}$, et à l'instant $2i-1$, les $n-i$ premiers processeurs P_1, P_2, ..., P_{n-i} exécutent les tâches $T_{i,j}$, $i+1 \leq j \leq n$.

4.4 Résolution de Pb(p)

Soit $G = (V, E, w)$ un DAG. Comme le problème à ressources bornées Pb(p) est NP-complet, nous introduisons des heuristiques (polynômiales) pour calculer des solutions approchées. Il se trouve que que ces heuristiques peuvent être garanties à un facteur deux du temps d'exécution optimal.

4.4.1 NP-complétude de Pb(p)

Définition 7 *Le problème de décision Dec(p) associé à Pb(p) est le suivant : étant donné un DAG $G = (V, E, w)$, un nombre de processeurs $p \geq 1$, et une borne d'exécution $K \in \mathbb{N}^*$, existe-t-il un ordonnancement σ pour G utilisant au plus p processeurs, tel que MS(σ, p) $\leq K$?*

Théorème 6 *Dec(p) est NP-complet.*

Preuve En premier lieu, Dec(p) est bien dans NP ; si on nous donne un ordonnancement σ dont la latence est inférieure ou égale à K, on peut vérifier en temps polynômial que les contraintes de dépendance et de ressource sont bien satisfaites. En effet, on doit vérifier que chaque contrainte de dépendance (chaque arête de E) est respectée, et que au plus p processeurs sont nécessaires à chaque instant. Cela peut être effectué en temps polynômial en la taille de l'instance du problème considérée. Ensuite, considérons une instance arbitraire Inst_1 de 2-PARTITION, un problème NP-complet bien connu [34] : étant donnés n entiers positifs $\{a_1, a_2, \ldots, a_n\}$, peut-on trouver un sous-ensemble I des indices tel que $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$? Nous allons montrer que 2-PARTITION peut être polynômialement réduit à Dec(p). On construit une instance Inst_2 de Dec(p) en posant $G = (V, E, w)$, avec $V = \{v_1, v_2, \ldots, v_n\}$, $E = \emptyset$, et $w(v_i) = a_i, 1 \leq i \leq n$. On pose aussi $p = 2$ et $K = \lfloor \frac{1}{2} \sum_{1 \leq i \leq n} a_i \rfloor$. La construction de Inst_2 est clairement polynômiale en la taille de Inst_1. De plus, Inst_1 a une solution ssi il existe un ordonnancement qui satisfait à la borne K, donc ssi Inst_2 admet une solution. ■

4.4.2 Heuristiques de liste

Puisque Pb(p) est NP-complet, on fait appel à des heuristiques pour ordonnancer les DAGs à ressources bornées. L'idée la plus naturelle est d'utiliser des stratégies gloutonnes : à chaque instant, on alloue autant de tâches que possible aux processeurs disponibles. Bien sûr il faut décider d'une priorité dans le cas (fréquent) où il y a davantage de tâches à allouer que de processeurs disponibles. Un résultat important du à Coffman est que toute stratégie obéissant au principe de *ne pas laisser délibérément un processeur inactif* va obtenir de bons résultats. Nous formalisons ceci après quelques définitions.

Définition 8 *Soit $G = (V, E, w)$ un DAG et σ un ordonnancement pour G. Une tâche $v \in V$ est dite libre au temps t (on note $v \in FREE(\sigma, t)$) ssi son exécution n'a pas encore commencé ($\sigma(v) \geq t$), mais tous ses prédécesseurs ont été exécutés au temps t ($\forall\, u \in PRED(v),\ \sigma(u) + w(u) \leq t$).*

Un ordonnancement de liste est un ordonnancement dans lequel aucun processeur n'est délibérément laissé inactif ; à tout instant t, si $|FREE(\sigma, t)| = r \geq 1$, et si q processeurs sont disponibles,, alors on débute l'exécution de $\min(r, q)$ tâches libres.

Théorème 7 *Soit $G = (V, E, w)$ un DAG et supposons disposer de p processeurs. Soit σ un ordonnancement de liste quelconque pour G. Soit $MS_{opt}(p)$ la latence d'ordonnancement optimal pour G. Alors*

$$MS(\sigma, p) \leq (2 - \frac{1}{p}) MS_{opt}(p)$$

Il faut souligner que le Théorème 7 est valide pour *tout* ordonnancement de liste, quelle que soit la politique de sélection des tâches libres quand il y a davantage de tâches libres que de processeurs disponibles.

Preuve On commence par un lemme :

Lemme 9 *Il existe un chemin de dépendance Φ dans G dont le poids $w(\Phi)$ vérifie :*

$$Idle \leq (p - 1) \times w(\Phi)$$

Preuve Pour la preuve du lemme, on a noté Idle le temps d'inactivité cumulé des p processeurs durant toute l'exécution. Soit T_{i_1} une tâche dont l'exécution se termine à la fin de l'ordonnancement :

$$\sigma(T_{i_1}) + w(T_{i_1}) = \mathrm{MS}(\sigma, p)$$

Soit t_1 le dernier instant avant $\sigma(T_{i_1})$ tel qu'il existe un processeur inactif durant l'intervalle de temps $[t_1, t_1 + 1[$ (on pose $t_1 = 0$ si un tel instant n'existe pas). Pourquoi ce processeur est-il inactif ? Comme σ est un ordonnancement de liste, aucune tâche n'est libre à l'instant t_1, sinon le processeur inactif débuterait son exécution. Il doit donc exister une tâche T_{i_2} qui est un ancêtre [4] de T_{i_1} et qui est en cours d'exécution à l'instant t_1 ; sinon l'exécution de T_{i_1} aurait débuté à l'instant t_1 sur le processeur inactif. Par definition de t_1, nous savons que tous les processeurs sont actifs entre la fin de l'exécution de T_{i_2} et le début de l'exécution de T_{i_1}.

On recommence la construction à partir de T_{i_2} pour obtenir une tâche T_{i_3} telle que tous les processeurs sont actifs entre la fin de l'exécution de T_{i_3} et le début de celle de T_{i_2}. Itérant le processus, on obtient r tâches $T_{i_r}, T_{i_{r-1}}, \ldots, T_{i_1}$ qui appartiennent à un même chemin de dépendance Φ de G, et telles que tous les processeurs sont actifs sauf peut-être pendant leur exécution. En d'autres mots, l'inactivité de certains processeurs ne peut se produire que pendant l'exécution de ces r tâches ; et pendant l'exécution de celles-ci, au moins un processeur est actif (celui qui exécute la tâche). Ainsi : Idle $\leq (p - 1) \times \sum_{j=1}^{r} w(T_{i_j}) \leq (p - 1) \times w(\Phi)$. ■

Revenons à la preuve du Théorème 7. On sait que $p \times \mathrm{MS}(\sigma, p) = \mathrm{Idle} + \mathrm{Seq}$, où $\mathrm{Seq} = \sum_{v \in V} w(v)$ est le temps séquentiel, i.e. la somme des poids des tâches (cf. Figure 4.2). Prenons le chemin

[4]Les ancêtres d'une tâche sont ses prédécesseurs, les prédécesseurs de ses prédécesseurs, etc.

de dépendance Φ construit au Lemme 9. On a $w(\Phi) \leq \text{MS}_{opt}(p)$, parce que la latence de tout ordonnancement est plus grand que le poids de tout chemin de dépendance dans G (simplement parce que les contraintes de dépendance sont respectées). En outre, $\text{Seq} \leq p \times \text{MS}_{opt}(p)$ (avec égalité seulement si tous les p processeurs sont actifs à tous les instants).

$$\begin{aligned} p \times \text{MS}(\sigma, p) &= \text{Idle} + \text{Seq} \\ &\leq (p-1)w(\Phi) + \text{Seq} \quad \leq (p-1)\text{MS}_{opt}(p) + p\text{MS}_{opt}(p) \\ &\qquad\qquad\qquad\qquad\quad = (2p-1)\text{MS}_{opt}(p), \end{aligned}$$

ce qui établit le théorème. ■

Fondamentalement, le théorème 7 assure que tout ordonnancement de liste est 50% de l'optimal. Avant de présenter l'heuristique de liste la plus classique pour choisir entre les tâches libres, nous montrons que la borne $\frac{2p-1}{p}$ ne peut pas être améliorée.

Proposition 11 *Soit $MS_{list}(p)$ la plus petite latence d'un ordonnancement de liste. La borne*

$$MS_{list}(p) \leq \frac{2p-1}{p} MS_{opt}(p)$$

est la meilleure possible.

Preuve Soit K un entier arbitrairement grand. On construit un DAG $G = (V, E, w)$, pour lequel tout ordonnancement de liste σ a une latence $\text{MS}(\sigma, p) \approx \frac{2p-1}{p}\text{MS}_{opt}(p)$ (voir Figure 4.3). Il y a $2p+1$ sommets, dont les poids sont : $w(T_i) = K(p-1)$ pour $1 \leq i \leq p-1$; $w(T_p) = 1$; $w(T_i) = K$ pour $p+1 \leq i \leq 2p$; et $w(T_{2p+1}) = K(p-1)$. Les arêtes de précédence sont représentées sur la figure. Il y a exactement p sommets d'entrée, donc $\sigma(T_i) = 0, 1 \leq i \leq p$ pour tout ordonnancement de liste σ. Au top 1, l'exécution de T_p est terminée, et le processeur disponible (celui qui a exécuté T_p) se verra successivement alloué $p-1$ des p tâches libres $T_{p+1}, T_{p+2}, \ldots, T_{2p}$. Ce processeur commence l'exécution de la dernière de ces $p-1$ tâches au top $1 + K(p-2)$ et termine celle-ci au top $1 + K(p-1)$. Par suite, la p-ème tâche restante sera exécutée au top $K(p-1)$ par un autre processeur. La tâche T_{2p+1} ne sera libre qu'au top $K(p-1) + K = Kp$, ce qui conduit à la latence $\text{MS}(\sigma, p) = Kp + K(p-1) = K(2p-1)$.

Il est possible d'ordonnancer le DAG en seulement $Kp+1$ tops. L'idée est de garder délibérément inactifs $p-1$ processeurs pendant l'exécution de la tâche T_p au top 0 (ce qui est interdit dans un ordonnancement de liste). Puis au top 1 chaque processeur exécute l'une des p tâches $T_{p+1}, T_{p+2}, \ldots, T_{2p}$. Au top $1 + K$ un processeur commence l'exécution de T_{2p+1} tandis que les autres $p-1$ processeurs exécutent les tâches $T_1, T_2, \ldots, T_{p-1}$. On obtient ainsi un ordonnancement de latence $1 + K + K(p-1) = Kp + 1$, ce qui est optimal car c'est le poids du chemin $T_p \to T_{p+1} \to T_{2p+1}$. D'où le rapport

$$\frac{\text{MS}(\sigma, p)}{\text{MS}_{opt}(p)} \geq \frac{K(2p-1)}{Kp+1} = \frac{2p-1}{p} - \frac{2p-1}{p(Kp+1)} = \frac{2p-1}{p} - \epsilon(K)$$

où $\lim_{K \to +\infty} \epsilon(K) = 0$. ■

4.4.3 Implémentation d'un ordonnancement de liste

Nous montrons ici comment implémenter un ordonnancement de liste "générique", ce qui signifie que nous ne décrivons pas de manière explicite le maniement de la file de priorité des tâches libres

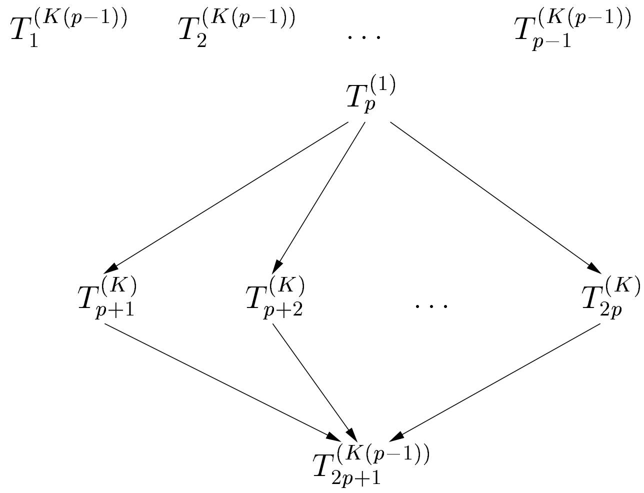


FIG. 4.3 – Le DAG utilisé pour la borne de garantie des ordonnancement de liste ; les poids des tâches sont indiqués en exposant et entre parenthèses.

(nous donnons un exemple plus tard). L'implémentation n'est pas difficile mais un peu longue à décrire. Le schéma global est le suivant :

1. *Initialisation :*
   (a) Calculer le niveau de priorité de toutes les tâches.
   (b) Constituer la file de priorité comme étant la liste des tâches libres (tâches sans prédécesseurs) triées en ordre de priorité décroissante.
   (c) Soit t l'instant courant : $t = 0$.
2. *Tant qu'il reste des tâches à exécuter* :
   (a) Ajouter les nouvelles tâches libres, s'il y en a, à la file de priorité. Si l'exécution d'une tâche se termine à l'instant t, supprimer cette tâche de la liste des prédécesseurs de chacun de ses successeurs. Ajouter les tâches dont la liste des prédécesseurs est devenue vide à la file.
   (b) S'il y a q processeurs disponibles et r tâches dans la file, supprimer les $\min(q, r)$ premières tâches de la file et les exécuter ; si T est l'une de ces tâches, poser $\sigma_{cp}(T) = t$.
   (c) Incrémenter t.

Soit $G = (V, E, w)$ un DAG et supposons disposer de p processeurs. Soit σ un ordonnancement de liste pour G. Nous allons décrire une implémentation de complexité $O(|V| \log |V| + |E|)$ pour σ. Clairement, il faut modifier le schéma précédent parce que les instants varient de $t = 0$ à $t = \mathrm{MS}(\sigma, p)$, si bien que la complexité dépendrait du poids des tâches. Pour ce faire, la Figure 4.4 présente un algorithme en pseudo-Pascal.

Voici quelques explications pour comprendre la Figure 4.4. On utilise un tas $\mathcal{Q}$ (voir par exemple [25]) pour stocker les tâches libres pour deux raisons : (i) on peut accéder à la tâche de priorité maximale en temps constant ; (ii) on peut insérer une tâche dans le tas, selon son niveau de priorité, en temps proportionnel au logarithme de la taille du tas, laquelle est bornée par $|V|$. On utilise un autre tas $\mathcal{P}$ pour gérer les processeurs actifs ; un processeur qui exécute une tâche $v \in V$ est étiqueté par l'instant auquel l'exécution de v se termine. On peut alors calculer le prochain événement en temps constant, et insérer un nouveau processeur actif dans le tas en temps $O(\log |\mathcal{P}|) \leq O(\log |V|)$ time. Quand on extrait un processeur du tas des processeurs, ce qui signifie

```
Entrée :
  le DAG G = (V, E, w), et le nombre de processeurs p, 1 ≤ p ≤ |V|.
Sortie :
  un temps d'ordonnancement σ(v) et un numéro de processeur alloc(v) pour chaque tâche v ∈ V.

  En plus de la structure de données pour représenter G, on stocke dans un tableau A
      le nombre de prédécesseurs de chaque tâche (son degré entrant)
  Initialiser le tas de priorité Q avec les tâches sans prédécesseur
  Initialiser le tas des processeurs P à Tas_Vide
  t = −1 ;
  Tant que Q ≠ File_Vide faire
    t' = Prochain_évènement(P, t) ;
    Mise_à_jour(t', A, Q) ;
    Allocation_des_tâches(t', P, Q) ;
    t = t' ;
  fin tant que
```

FIG. 4.4 – Squelette d'un algorithme d'ordonnancement de liste.

qu'une tâche v est terminée, il faut mettre à jour dans le tableau A les degrés entrant de chacun des successeurs de v. Au vol, si le degré entrant d'un successeur donné v' devient égal à zéro, on insère v' dans le tas de priorité $\mathcal{Q}$. De cette façon, on ne traite chaque arête de dépendance de G qu'une fois, pour un coût global en $O(|E|)$. En faisant le bilan, chaque tâche requiert deux insertions : la première est l'insertion de la tâche elle-même dans le tas $\mathcal{Q}$; la seconde est l'insertion du processeur qui va l'exécuter dans le tas $\mathcal{P}$. Comme chaque opération coûte au plus $O(\log |V|)$, on obtient la complexité $O(|V| \log |V| + |E|)$ pour l'ordonnancement de liste.

4.4.4 Ordonnancement basé sur le chemin critique

Nous détaillons ici un ordonnancement de liste particulier très utilisé, celui du chemin critique. Nous avons vu le principe des ordonnancements de liste, mais il reste à expliquer comment, en pratique, choisir parmi les tâches libres pour obtenir un ordonnancement. L'algorithme le plus classique repose sur les valeurs des niveaux inférieurs des tâches. Intuitivement, plus grande est la valeur du niveau inférieur, plus “urgente” est la tâche. Le *chemin critique* d'une tâche est défini comme son niveau inférieur et est utilisé pour déterminer les niveaux de priorité entre tâches. Ainsi, l'ordonnancement du chemin critique est simplement l'ordonnancement de liste dans lequel la priorité des tâches est donnée par la valeur de leur niveau inférieur. Les tâches ayant la même valeur sont départagées de manière arbitraire.

Traitons un petit exemple : sur le DAG de la Figure 4.5, il y a huit tâches, dont les poids et les chemins critiques sont donnés à la Table 4.1.

Tâches	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Poids	3	2	1	3	4	4	3	6
Chemin critique	3	6	7	3	4	4	3	6

TAB. 4.1 – Poids et chemins critiques.

Supposons disposer de $p = 3$ processeurs et soit $\mathcal{Q}$ la liste de priorité des tâches libres. A $t = 0$,

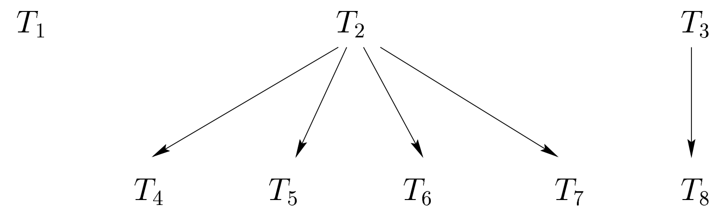


FIG. 4.5 – Un petit exemple.

$\mathcal{Q}$ est initialisée à $\mathcal{Q} = (T_3, T_2, T_1)$. Comme $q = r = 3$, on exécute ces trois tâches. A $t = 1$, on ajoute T_8 à la liste : $\mathcal{Q} = (T_8)$. Il y a un processeur disponible, qui débute l'exécution de T_8. A $t = 2$, on ajoute les quatre successeurs de T_2 à la liste : $\mathcal{Q} = (T_5, T_6, T_4, T_7)$. Nous avons départagé les ex-aequos arbitrairement, avec le numéro des tâches. Le processeur disponible pioche la première tâche T_5 dans $\mathcal{Q}$. Continuant ce schéma, l'exécution se poursuit jusqu'à $t = 10$, comme le montre la Figure 4.6.

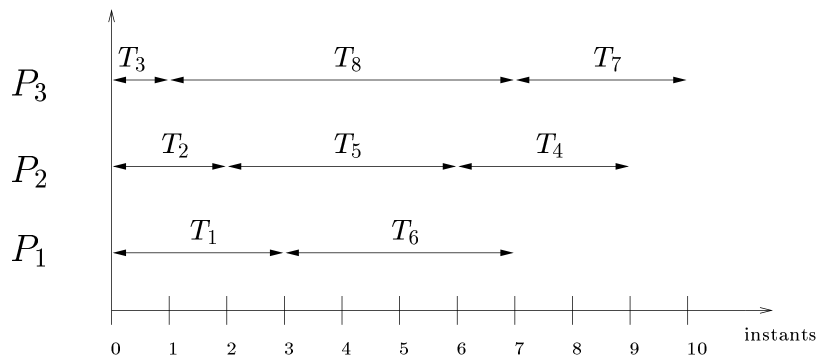


FIG. 4.6 – Ordonnancement du chemin critique pour l'exemple.

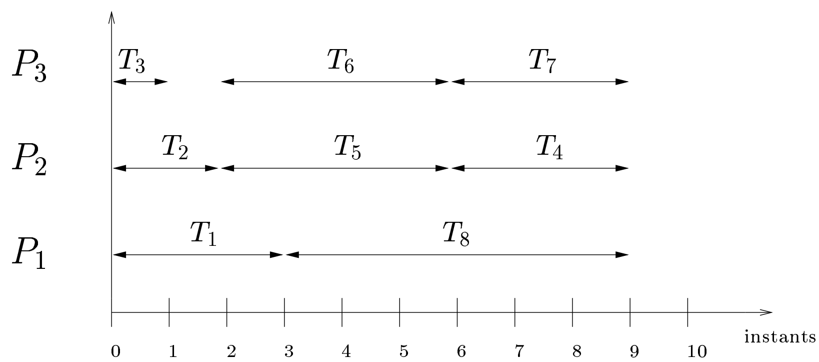


FIG. 4.7 – Ordonnancement optimal pour l'exemple.

Notons qu'il est possible d'ordonnancer le DAG en seulement neuf tops, comme le montre la Figure 4.7. Ici encore, l'astuce est de délibérément laisser un processeur inactif à $t = 1$; bien qu'elle ait le niveau supérieur le plus élevé, la tâche T_8 peut être retardée de deux tops. On donne la préférence à T_5 et T_6 pour réaliser un meilleur équilibrage de la charge entre les processeurs. Comment savons-nous que l'ordonnancement de la Figure 4.7 est optimal ? Parce que Seq = 26, et donc avec trois processeurs il faut au moins $\lceil \frac{26}{3} \rceil = 9$ unités de temps. Ce petit exemple illustre bien de manière concrète la difficulté de l'ordonnancement à ressources bornées.

4.5 Prise en compte des coûts de communication

Les machines parallèles ne sont pas tendres envers leurs utilisateurs. Il s'avère très difficile de modéliser les performances des communications. L'ancienne génération de machines utilisait un mécanisme de recopie mémoire puis transmission des messages (*store-and-forward*, voir le Paragraphe 6.3). Ce n'était pas très efficace mais pouvait facilement être modélisé par une formule du type : $\mathrm{comm}(p,p') = \mathrm{dist}(p,p') \times (\beta + L\tau)$, où L est la longueur du message envoyé par le processeur p au processeur p' ; $\mathrm{dist}(p,p')$ est la distance entre p et p' : $\mathrm{dist}(p,p') = 1$ si p et p' sont voisins (ils sont directement reliés par un lien physique), $\mathrm{dist}(p,p') = 2$ s'il y a un processeur intermédiaire entre eux, etc. Enfin, β et τ sont des paramètres machine : β est le temps d'initialisation de la communication (principalement dû aux couches logicielles, mais aussi au délai de configuration du matériel), et τ est l'inverse de la bande passante du réseau.

Les machines actuelles n'obéissent plus à ce modèle simple. Chaque unité de calcul est équipée d'un co-processeur de communication. Les messages sont découpés en paquets, qui sont routés dynamiquement entre les processeurs, deux paquets distincts pouvant emprunter deux chemins différents. Ce routage dynamique sera efficace s'il n'y a pas de contention sur certains liens de communication (*hot-spots*). La distance entre les processeurs n'est plus si importante ; plutôt, si plusieurs messages circulent simultanément sur le réseau, c'est l'aspect structuré ou non des communications en jeu qui va conduire, ou non, à de bonnes performances. En ce sens, la localité est toujours importante, mais de manière indirecte.

C'est ainsi que le modèle primitif utilisé en ordonnancement pour modéliser les communications a récemment acquis plus d'intérêt en pratique[5]. Si une tâche T doit communiquer des données à l'un de ses successeurs T', le coût est modélisé ainsi :

$$\mathrm{comm}(T,T') = \begin{cases} 0 & \textit{si } \mathrm{alloc}(T) = \mathrm{alloc}(T') \\ c(T,T') & \textit{sinon} \end{cases}$$

où $\mathrm{alloc}(T)$ est le processeur qui exécute la tâche T. Intuitivement, on considère le coût d'un accès local en mémoire comme négligeable en face d'un accès à la mémoire d'un autre processeur, et on paie le même coût de communication $c(T,T')$ pour ce dernier, indépendamment de la distance dans le réseau. Cela revient à supposer disposer d'une clique de processeurs totalement inter-connectés. Le modèle est appelé *macro-dataflow* car on suppose que : (i) une communication peut avoir lieu dès que les données sont disponibles ; (ii) il n'y a pas de limitation de ressources en matière de liens de communication. L'hypothèse (i) est raisonnable, car on peut effectuer simultanément des communications et des calculs (indépendants) sur les machines modernes. L'hypothèse (ii) est plus difficile à justifier, mais c'est le prix à payer pour obtenir des formules et des résultats significatifs. Nous concluons cette discussion en donnant la définition formelle du modèle :

Définition 9 *Un DAG avec coûts de communication (ou cDAG) est un graphe orienté acyclique $G = (V, E, w, c)$ où les sommets représentent les tâches et les arêtes représentent les contraintes de précédence. La fonction de poids est $w : V \longrightarrow \mathbb{N}^*$ et le coût de communication est $c : E \longrightarrow \mathbb{N}^*$. Tout ordonnancement σ doit préserver les dépendances, ce qui s'écrit :*

$$\forall e = (v, v') \in E, \begin{cases} \sigma(v) + w(v) \leq \sigma(v') & \textit{si alloc}(T) = \textit{alloc}(T') \\ \sigma(v) + w(v) + c(T,T') \leq \sigma(v') & \textit{sinon} \end{cases}$$

L'expression des contraintes de ressources est inchangée.

[5]Coup de chance pour les théoriciens ! Le modèle a été introduit bien avant l'évolution architecturale et est devenu plus précis : qui s'en plaindrait ?

4.6 Avec communications : NP-complétude de Pb(∞) et heuristiques

L'introduction de coûts de communication dans le modèle rend les choses plus difficiles, y compris la résolution du problème à ressources infinies Pb(∞). La raison intuitive en est la suivante : on hésite entre allouer des tâches soit à un grand nombre de processeurs (ce qui augmente le parallélisme, mais impose de communiquer beaucoup), soit au contraire à un petit nombre de processeurs (moins de communications mais également moins de parallélisme). Nous illustrons ceci à l'aide d'un petit exemples emprunté à [36].

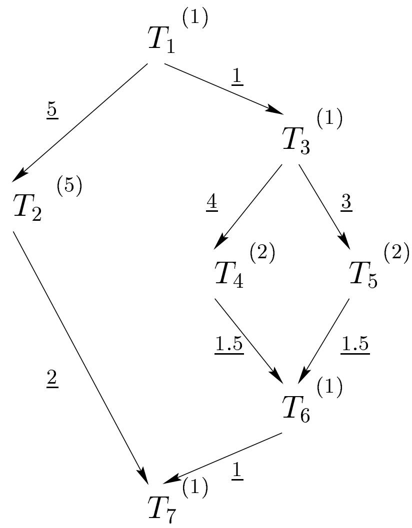


FIG. 4.8 – Exemple pour illustrer l'impact des coûts de communication.

Considérons le cDAG de la Figure 4.8. Les poids des tâches sont indiqués près de celles-ci et entre parenthèses, tandis que les coûts de communication sont représentés le long des arêtes, et soulignés. Pour pouvoir dessiner de jolies tables, nous avons deux coûts de communication qui ne sont pas entiers $c(T_4, T_6) = c(T_5, T_6) = 1.5$. Bien sûr on pourrait se ramener à des entiers en multipliant par 2 toutes les valeurs de w et c. On voit que :

- D'un côté, si on alloue toutes les tâches à un même processeur, la latence sera égale à la somme des poids de toutes les tâches, i.e., 13.
- D'un autre côté, si on dispose d'autant de processeurs que l'on veut (nous en avons besoin d'au plus sept car il y a sept tâches), on peut allouer une tâche par processeur. La latence de l'ordonnancement au plus tôt (ASAP) est alors égale à 14. Pour le voir, il est important de souligner qu'une fois que l'allocation des tâches aux processeurs est déterminée, la latence peut être calculée facilement : pour chaque arête $e : T \to T'$, on ajoute un sommet virtuel de poids $c(T, T')$ si l'arête relie deux processeurs distincts ($\text{alloc}(T) \neq \text{alloc}(T')$), et on ne fait rien sinon. On considère le nouveau graphe comme un DAG (i.e. sans coûts de communication) et on le traverse pour calculer la longueur du plus long chemin, comme expliqué au Paragraphe 4.3. Dans notre cas, puisque toutes les tâches sont allouées à des processeurs différents, on ajoute un sommet virtuel sur chaque arête. Le plus long chemin est $T_1 \to T_2 \to T_7$, de longueur $w(T_1) + c(T_1, T_2) + w(T_2) + c(T_2, T_7) + w(T_7) = 14$.

Il y a un compromis difficile à trouver entre choisir d'exécuter les tâches en parallèle (donc avec plusieurs processeurs distincts) et minimiser les coûts de communication. Dans notre exemple, il s'avère que la meilleurs solution est d'utiliser deux processeurs, avec l'ordonnancement dela

Figure 4.9, dont la latence est 9.

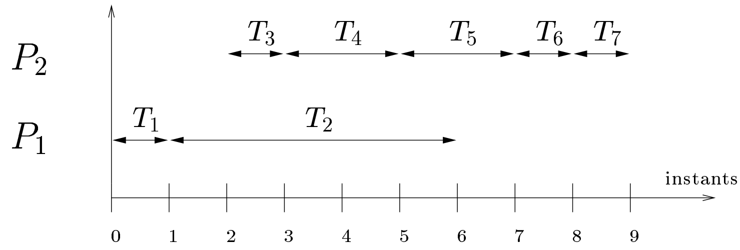


FIG. 4.9 – Un ordonnancement optimal pour l'exemple.

Notons que les contraintes de dépendance sont bien satisfaites sur la Figure 4.9. Par exemple, T_2 peut débuter au top 1 sur le processeur P_1, parce que celui-ci exécute T_1, et donc on ne paie pas le coût de communication $c(T_1, T_2)$. Par contre, T_3 est exécutée sur le processeur P_2, d'où la nécessité d'attendre le top 2 pour débuter son exécution, bien que P_2 soit inactif au top 1 : en effet $\sigma(T_1) + w(T_1) + c(T_1, T_3) = 0 + 1 + 1 = 2$.

Comment avons-nous trouvé l'ordonnancement de la Figure 4.9 ? Et comment savons nous qu'il est optimal ? Par une laborieuse étude exhaustive ! Cet exemple nous montre qu'utiliser davantage de processeurs ne diminue pas toujours le temps d'exécution. En utilisant les notations du Paragraphe 4.2,la latence minimale d'un ordonnancement faisant effectivement usage de sept processeurs est $\mathrm{MS}'(7) = 14$ alors que $\mathrm{MS}'(1) = 13$ (ou $\mathrm{MS}'(2) = 9$). La version raffinée de la Proposition 9 avec les MS' n'est plus vraie quand on prend en compte les coût de communications.

4.6.1 NP-complétude de Pb(∞)

En fait, avec des coûts de communication, même la résolution de Pb(∞), le problème à ressources illimitées, s'avère difficile :

Théorème 8 *Pb(∞) est NP-complet.*

Preuve Le problème de décision Comm(∞) associé à Pb(∞) est le suivant : étant donné un cDAG $G = (V, E, w, c)$ et une borne $K \in \mathbb{N}^*$ sur le temps d'exécution, existe-t-il un ordonnancement σ pour G tel que $\mathrm{MS}(\sigma, \infty) \leq K$? Nous allons montrer que Comm(∞) est NP-complet. D'abord, Comm(∞) est dans NP. Si on nous donne un ordonnancement σ dont la latence est inférieure ou égale à K,on peut vérifier en temps polynômial que les contraintes de dépendance sont satisfaites. Pour chaque tâche on connaît le début $\sigma(T)$ de son exécution et le processeur alloc(T) qui l'exécute, d'où une vérification directe des contraintes.

Pour la preuve de NP-complétude, on utilise 2-PARTITION comme pour le Théorème 6, mais la réduction est moins immédiate. Prenons une instance quelconque Inst_1 de 2-PARTITION : étant donnés n entiers positifs $\{a_1, \ldots, a_n\}$, existe-t-il un sous-ensemble I des indices tel que $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$? On construit une instance Inst_2 de Comm(∞) comme suit ; on définit $G = (V, E, w, c)$ comme un graphe *fork-join* (voir la Figure 4.10). Il y a $n+2$ tâches : $V = \{T_0, T_1, T_2, \ldots, T_n, T_{n+1}\}$. Les poids des tâches sont : $w(T_i) = 2 \times a_i$ pour $1 \leq i \leq n$, et $w(T_0) = w(T_{n+1}) = A$, où A est un entier positif. Il y a $2n$ arêtes, et les coûts de communication sont tous égaux : $c(T_0, T_i) = c(T_i, T_{n+1}) = C$ pour $1 \leq i \leq n$. Ici, C est un entier quelconque dans l'intervalle $]\alpha - \min_{1 \leq i \leq n} 2a_i, \alpha[$, où $\alpha = \sum_{i=1}^{n} a_i$. Cet intervalle contient bien un entier, car $\min_{1 \leq i \leq n} 2a_i \geq 2$. Enfin, soit $K = 2A + C + \alpha$. La taille de Inst_2 est clairement polynômiale en la taille size de Inst_1. La difficulté est de montrer que Inst_1 admet une solution ssi il existe un ordonnancement dont la latence est inférieure ou égale à K, i.e ssi Inst_2 admet une solution.

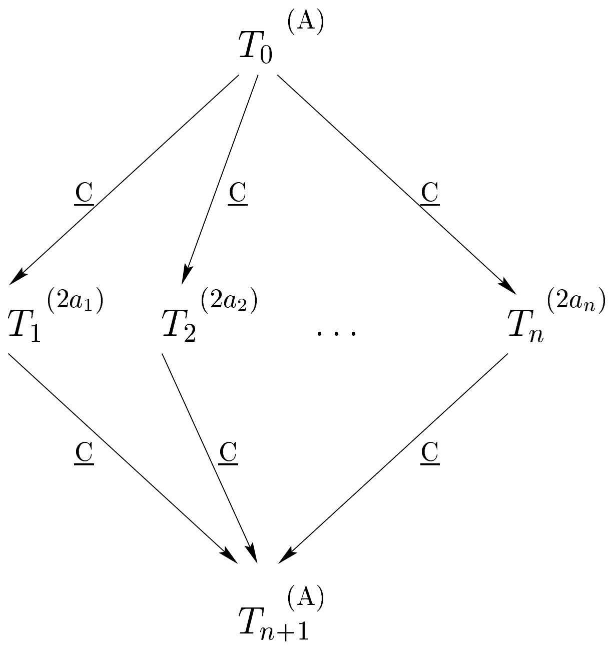


FIG. 4.10 – Réduction pour la preuve de NP-complétude.

Supposons d'abord que Inst_1 admet une solution. Soit I le sous-ensemble des indices tel que $\sum_{i\in I} a_i = \sum_{i\notin I} a_i = \frac{\alpha}{2}$. Soit $\mathcal{T}_1 = \{T_i, i \in I\}$ et $\mathcal{T}_2 = \{T_i, i \notin I\}$. Par hypothèse, $w(\mathcal{T}_1) = w(\mathcal{T}_2) = \alpha$. Considérons l'ordonnancement de la Figure 4.11.

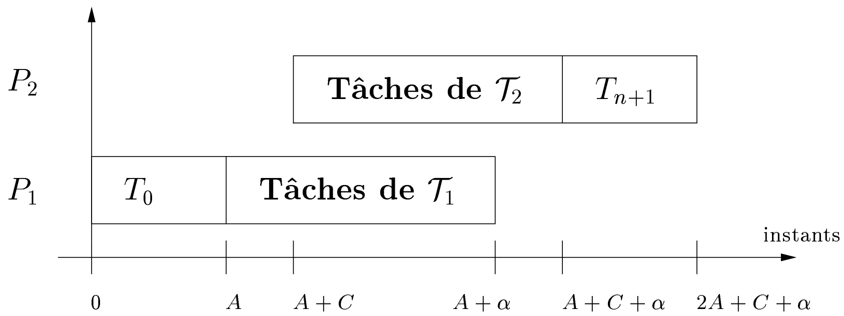


FIG. 4.11 – Un ordonnancement de latence $K = 2A + C + \alpha$.

La latence de cet ordonnancement est $K = 2A + C + \alpha$. Toutes les contraintes de dépendance sont satisfaites. En effet :

- Le processeur P_2 commence l'exécution des tâches de $\mathcal{T}_2$ au top $A + C = w(T_0) + C$.
- Le processeur P_1 termine l'exécution des tâches de $\mathcal{T}_1$ au top $A + \alpha$. Ainsi, au top $A + \alpha + C$, la tâche T_{n+1} est prête à être exécutée par le processeur P_2. Son exécution commence effectivement à cet instant, car P_2 termine les tâches de $\mathcal{T}_2$ au top $A + C + \alpha$.

Réciproquement, supposons que Inst_2 admet une solution. Soit σ un ordonnancement dont la latence $\mathrm{MS}(\sigma)$ est inférieure ou égale à K. On passe par les deux lemmes suivants :

Lemme 10 *Les tâches T_0 et T_{n+1} ne sont pas exécutées par le même processeur dans l'ordonnancement σ.*

Preuve Supposons que le même processeur P exécute à la fois T_0 et T_{n+1}. Alors P exécute toutes les n autres tâches T_i, $1 \leq i \leq n$. Sinon, soit T_{i_0} une tâche qui serait exécutée par un autre processeur. La latence de σ est au moins égale à la longueur du chemin $T_0 \to T_{i_0} \to T_{n+1}$:

$$\text{MS}(\sigma) \geq A + C + 2a_{i_0} + C + A = (2A + C) + (2a_{i_0} + C) > (2A + C) + \alpha = K$$

d'où une contradiction. Donc P exécute bien les $n+2$ tâches, ce qui est encore une contradiction, car la somme de tous les poids des tâches est $2A + 2\alpha > 2A + \alpha + C = K$. ■

Soit P_1 le processeur qui exécute T_0 et P_2 le processeur qui exécute T_{n+1}.

Lemme 11 *Chaque tâche T_i, $1 \leq i \leq n$, est exécutée soit par P_1 soit par P_2.*

Preuve Supposons qu'il existe une tâche T_{i_0}, $1 \leq i_0 \leq n$, exécutée par un processeur autre que P_1 et P_2. Alors la latence de σ est au moins égale à la longueur du chemin $T_0 \to T_{i_0} \to T_{n+1}$, contradiction comme dans le lemme précédent. ■

Soit alors $\mathcal{T}_1$ l'ensemble des tâches T_i, $1 \leq i \leq n$, exécutées par P_1. Définissons pareillement $\mathcal{T}_2$ pour P_2. La latence de σ vérifie :

$$\text{MS}(\sigma) \geq w(T_0) + w(\mathcal{T}_1) + C + w(T_{n+1}) = 2A + C + w(\mathcal{T}_1)$$

En effet, P_1 a besoin d'au moins $w(T_0) + w(\mathcal{T}_1)$ tops pour exécuter ses tâches. Puis une communication doit avoir lieu avant que P_2 ne puisse débuter l'exécution de T_{n+1}.

De même, $\text{MS}(\sigma) \geq 2A + C + w(\mathcal{T}_2)$, car P_2 doit attendre au moins $A + C$ tops avant de commencer l'exécution. Comme $\text{MS}(\sigma) \leq K = 2A + C + \alpha$, on a $w(\mathcal{T}_1) \leq \alpha$ et $w(\mathcal{T}_2) \leq \alpha$. Mais $w(\mathcal{T}_1) + w(\mathcal{T}_2) = 2\alpha$. Donc $w(\mathcal{T}_1) = w(\mathcal{T}_2) = \alpha$. Si I désigne l'ensemble des indices des tâches de $\mathcal{T}_1$, alors I est solution de Inst_1, notre instance de 2-PARTITION. ■

4.6.2 Une heuristique garantie pour $\mathbf{Pb}(\infty)$

Nous présentons ici l'heuristique de Hanen et Munier [41], pour résoudre $\text{Pb}(\infty)$. Cette heuristique est garantie à un facteur au plus $\frac{4}{3}$ de l'optimal, à la condition que tous les coûts de communication soient supposés inférieurs à tous les coûts de communication . Un tel graphe de tâches est dit “à gros grain”, ou *coarse-grain* :

Définition 10 *Soit $G = (V, E, w, c)$ un cDAG. La granularité de G est le rapport calcul/communication $g(G) = \frac{\min_{T \in V} w(T)}{\max_{T,T' \in V} c(T,T')}$. G est à gros grain si $g(G) \geq 1$.*

Avant de décrire formellement l'heuristique, nous en expliquons l'idée principale, celle des successeurs favoris.

Successeurs favoris

Soit $G = (V, E, w, c)$ un cDAG, σ un ordonnancement pour G, et $T \in V$ une tâche quelconque. Le successeur favori de T, s'il existe, est l'unique successeur immédiat T' de T tel que

$$\sigma(T') < \sigma(T) + w(T) + c(T, T') \qquad \text{(FS)}$$

S'il existe, le successeur favori de T is exécuté par le même processeur que T, sinon un coût de communication serait à payer et la condition (FS) ne serait pas vérifiée. Pour voir l'unicité en cas d'existence, supposons exister deux successeurs T' and T'' de T satisfaisant la condition (FS). T' et T'' sont exécutés par le même processeur. Sans perte de généralité, supposons T' exécutée avant T''. D'où $\sigma(T) + w(T) \leq \sigma(T')$, et $\sigma(T') + w(T') \leq \sigma(T'')$. Mais par hypothèse, $\sigma(T'') < \sigma(T) + w(T) + c(T, T'')$; donc $w(T') < c(T, T'')$, ce qui contredit le fait que G est à gros grain. La Table 4.2 donne tous les successeurs favoris pour l'ordonnancement optimal de la Figure 4.9.

Tâche	T_1	T_2	T_3	T_4	T_5	T_6	T_7
Successeur Favori	T_2	—	T_4	—	T_6	T_7	—

TAB. 4.2 – Successeurs favoris pour l'ordonnancement de la Figure 4.9.

Pour chaque arête $e = (T, T') \in E$, on introduit une variable booléenne $x_{T,T'}$: $x_{T,T'} = 0$ si T' est le successeur favori de T, et $x_{T,T'} = 1$ sinon. L'inégalité

$$\sigma(T) + w(T) + x_{T,T'} \times c(T, T') \leq \sigma(T')$$

est vraie pour toute arête $(T, T') \in E$, que T' soit le successeur favori de T ou non. Rassembler toutes ces inégalités au sein d'un problème de programmation linéaire est l'objectif qui sous-tend l'heuristique.

L'heuristique de Hanen et Munier

Définition 11 *Soit $G = (V, E, w, c)$ un cDAG. On définit le problème de programmation linéaire suivant :*

Minimiser M_∞ *sous la contrainte*

$$\begin{cases} \forall (T,T') \in E & x_{T,T'} \in \{0,1\} & (A) \\ \forall T \in V & s(T) \geq 0 & (B) \\ \forall (T,T') \in E & s(T) + w(T) + x_{T,T'} c(T,T') \leq s(T') & (1) \\ \forall T \in V t.q.\ SUCC(T) \neq \emptyset & \sum_{T' \in SUCC(T)} x_{T,T'} \geq |SUCC(T)| - 1 & (2) \\ \forall T \in V t.q.\ PRED(T) \neq \emptyset & \sum_{T' \in PRED(T)} x_{T',T} \geq |PRED(T)| - 1 & (3) \\ \forall T \in V & s(T) + w(T) \leq M_\infty & (4) \end{cases}$$

Lemme 12 *Soit $G = (V, E, w, c)$ un cDAG. La solution M_∞ du problème de programmation linéaire en nombres entiers ILP(G) est égale à la latence optimale à ressources non bornées $MS_{opt}(\infty)$.*

Preuve Montrons qu'il y a une correspondance bijective entre les ordonnancements valides pour G (à ressources non bornées) et les solutions du programme linéaire en nombres entiers ILP(G).

Soit σ un ordonnancement valide pour G. Soit $s(T) = \sigma(T)$ pour chaque tâche T, et posons $x_{T,T'} = 0$ si T' est le successeur favori de T, $x_{T,T'} = 1$ sinon. Toutes les contraintes de ILP(G) sont satisfaites :

- Les contraintes (A) et (B) sont satisfaites par construction.
- La contrainte (1) a été montrée au Paragraphe 4.6.2.
- La contrainte (2) exprime le fait que chaque tâche a au plus un successeur favori.
- La contrainte (3) exprime le fait que chaque tâche est le successeur favori d'au plus une tâche, ce qui peut être prouvé de façon tout à fait semblable à la contrainte (2).
- La définition de la latence est MS$(\sigma, \infty) = \max_{T \in V}(\sigma(T) + w(T))$, donc la contrainte (4) est vérifiée.

Soit $M_\infty(\sigma)$ la valeur renvoyée par ILP(G) quand toutes les variables $s(T)$ et $x_{T,T'}$ sont définis comme ci-dessus. Comme la contrainte (4) est la seule contrainte portant sur la fonction objective, on a $M_\infty(\sigma) \geq \text{MS}(\sigma, \infty)$.

Réciproquement, considérons une solution du problème d'optimisation ILP(G). Pour définir l'ordonnancement σ induit, nous devons déterminer pour chaque tâche un top de début d'exécution et un numéro de processeur qui l'exécute. On pose directement $\sigma(T) = s(T)$ pour toute tâche $T \in V$. On définit la fonction d'allocation par :

$$\forall e = (T, T') \in E,\ x_{T,T'} = 0 \Leftrightarrow \text{alloc}(T) = \text{alloc}(T')$$

Plus précisément, on alloue les tâches d'entrée à des processeurs distincts, et on traverse le graphe pou calculer la fonction d'allocation comme suit : si T' est un successeur immédiat de T et $x_{T,T'} = 1$, on alloue T' à un nouveau processeur,sinon on alloue T' au même processeur que T. Il n'y a pas de conflit durant ce parcours. En effet, grâce à la condition (2), pour chaque tâche $T \in V$, il y a au plus un successeur de T alloué au même processeur que T : si elle existe, l'unique tâche T' telle que $x_{T,T'} = 0$ (T' est alors le successeur favori de T. De même, grâce à la condition (3), pour chaque tâche $T' \in V$, il y a au plus un prédécesseur de T' alloué au même processeur que T'. De plus, la contrainte (1) et notre choix pour la fonction d'allocation garantissent que les contraintes de dépendance sont satisfaites : σ est bien un ordonnancement valide pour G. Enfin, la condition (4) assure que M_∞ égal à la latence de σ. ■

Etant donnée une solution de ILP(G), on peut interpréter $s(T)$ comme le niveau supérieur de T, en calculant les niveaux supérieur et inférieur de chaque tâche à l'aide de la fonction d'allocation induite par les variables $x_{T,T'}$. On ajoute le coût de communication $c(T, T')$ au poids d'un chemin qui va de T à T' ssi alloc$(T) \neq$ alloc(T'), i.e., ssi $x_{T,T'} = 1$. La condition (4) montre que la solution de ILP(G) est égale à la valeur du poids maximal d'un chemin dans le graphe de dépendance, où les poids sont calculés avec les règles précédentes.

Toute la difficulté réside dans la détermination de la fonction d'allocation, i.e., dans la détermination des valeurs $x_{T,T'}$. Comme ces valeurs sont entières (et même restreintes à 0 ou 1), le problème de programmation linéaire est à résoudre en nombres entiers, ce qui est NP-difficile [62]. Mais si on décide de chercher les $x_{T,T'}$ comme des rationnels et non plus des entiers, on obtient un programme linéaire en nombres rationnels, dont la complexité est polynômiale [62].

Définition 12 *Soit $G = (V, E, w, c)$ un cDAG :*
- *On définit le programme linéaire relaxé RLP(G) comme le programme obtenu en supprimant l'équation (A) dans la définition de ILP(G).*
- *Soit $(x^{rel}_{T,T'}, s^{rel}(T), M^{rel}_\infty)$ la solution en nombres rationnels du problème relaxé RLP(G).*

Hanen et Munier définissent leur ordonnancement σ^{hm} directement à partir de la solution de RLP(G). Soit $T \in V$ une tâche quelconque. La contrainte (2) assure qu'i y a au plus un successeur T' de T tel que $x^{rel}_{T,T'} < \frac{1}{2}$; et la contrainte (3) assure qu'i y a au plus un prédécesseur T'' de T tel que $x^{rel}_{T'',T} < \frac{1}{2}$. On pose alors $x^{hm}_{T,T'} = 0$ pour toute arête $e = (T, T') \in E$ telle que $x^{rel}_{T,T'} < \frac{1}{2}$, et $x^{hm}_{T,T'} = 1$ sinon. Pour toute tâche $T \in V$, on définit σ^{hm}_T comme le niveau supérieur de T, où les niveaux inférieur et supérieur sont calculés à partir de la fonction d'allocation induite par les $x^{hm}_{T,T'}$. On ajoute le coût de communication $c(T, T')$ au poids d'un chemin allant de T à T' ssi alloc$(T) \neq$ alloc(T'), i.e., ssi $x^{hm}_{T,T'} = 1$. Comme expliqué plus haut, ceci définit un ordonnancement valide pour G.

Théorème 9 *Soit $G = (V, E, w, c)$ un cDAG à gros grain, de granularité $g(G) \geq 1$. Soit σ^{hm} l'ordonnancement défini par Hanen et Munier. Alors*

$$MS(\sigma^{hm}, \infty) \leq MS_{opt}(\infty) \times \frac{2g(G)+2}{2g(G)+1}$$

Preuve Pour tout chemin du graphe allant d'une tâche T à l'un de ses successeurs T', on a le coût de communication $x^{hm}_{T,T'}c(T,T')$ pour l'ordonnancement de Hanen et Munier, au lieu de $x^{rel}_{T,T'}c(T,T')$ pour la solution de RLP(G). Il y a deux cas à discuter :

- $x^{hm}_{T,T'} = 0$: alors $w(T) + x^{hm}_{T,T'}c(T,T') \leq w(T) + x^{rel}_{T,T'}c(T,T')$.
- $x^{hm}_{T,T'} = 1$: alors $x^{rel}_{T,T'} \geq \frac{1}{2}$. On a

$$\frac{w(T) + x^{hm}_{T,T'}c(T,T')}{w(T) + x^{rel}_{T,T'}c(T,T')} \leq \frac{w(T) + c(T,T')}{w(T) + c(T,T')/2} = \frac{1 + \frac{c(T,T')}{w(T)}}{1 + \frac{c(T,T')}{2w(T)}} \quad \text{et}$$

$$\frac{1 + \frac{c(T,T')}{w(T)}}{1 + \frac{c(T,T')}{2w(T)}} \leq \frac{1 + \frac{1}{g(G)}}{1 + \frac{1}{2g(G)}} = \frac{2g(G)+2}{2g(G)+1}$$

Dans tous les cas, $w(T) + x^{hm}_{T,T'}c(T,T') \leq \frac{2g(G)+2}{2g(G)+1}(w(T) + x^{rel}_{T,T'}c(T,T'))$, et cette inégalité s'étend à tous les chemins dans le graphe, ce qui conduit au résultat. ■

Une conséquence immédiate du Théorème 9 est que l'heuristique de Hanen et Munier est garantie à un facteur d'approximation au plus $\frac{4}{3}$ pour les graphes à gros grain.

4.7 Avec communications : heuristiques de liste pour Pb(p)

Bien sûr le problème à ressources bornées Pb(p) reste NP-complet quand on introduit des coûts de communication. Notons que Pb(p) reste bien dans la classe NP : une fois que l'allocation est connue, la vérification des contraintes de dépendance et de ressources peut s'effectuer en traversant le graphe. Le vrai problème est de trouver une bonne allocation. L'idée la plus naturelle est d'étendre l'ordonnancement de liste basé sur le chemin critique, celui que nous avons présenté au Paragraphe 4.4.4. Nous commençons par une modification directe avant de décrire une amélioration plus efficace.

4.7.1 Version naïve du chemin critique

L'idée de base du chemin critique reste la même : c'est un algorithme de liste, pour lequel la priorité est donnée selon la valeur du niveau inférieur de chaque tâche. Le problème est que nous ne savons pas calculer les niveaux inférieurs sans connaître l'allocation : impossible de savoir quels coûts de communication doivent être pris en compte. Une approche "conservative" consiste à inclure par précaution tous les coûts de communication pour déterminer les niveaux inférieurs, ce qui revient à supposer qu'on a un processeur distinct par tâche.

Reprenons l'exemple de la Figure 4.8. Le niveau inférieur de chaque tâche avec ce mode de calcul conservatif est donné dans la Table 4.3. On peu vérifier que le chemin critique de la tâche T_1 a pour valeur 14, la latence de l'ordonnancement au plus tôt (ASAP) avec un processeur par tâche.

Construisons un ordonnancement de liste basé sur les valeurs de ces chemins critique. L'algorithme se déroule comme expliqué au Paragraphe 4.4.4, avec une différence importante cependant.

Tâche	T_1	T_2	T_3	T_4	T_5	T_6	T_7
Chemin critique	14	8	11.5	6.5	6.5	3	1

TAB. 4.3 – Chemins critiques (niveau inférieurs) pour l'exemple de la Figure 4.8.

Une tâche est libre quand tous ses prédécesseurs ont été exécutés. Mais l'exécution d' une tâche ne peut pas débuter dès qu'elle est libre, même s'il y a des processeurs disponibles : selon la décision pour l'allocation, il faudra, ou non, attendre à cause du coût de communication.

Supposons disposer de $p = 3$ processeurs P_1, P_2, et P_3 dans notre exemple. Quand il y a plus de processeurs disponibles que de tâches libres, choisissons d'allouer les tâches aux processeurs de numéros les plus petits. Soit $\mathcal{Q}$ la file de priorité des tâches libres. Nous avons une seule tâche libre ($r = 1$) au top $t = 0$: $\mathcal{Q} = (T_1)$. Le processeur P_1 exécute T_1 à $t = 0$. A $t = 1$, $\mathcal{Q}$ devient $\mathcal{Q} = (T_3, T_2)$ (T_3 reçoit priorité sur T_2 parce que son chemin critique est plus grand). Tous les processeurs sont disponibles, donc en suivant notre règle sur les numéros nous allouons T_3 à P_1 et T_2 à P_2. Nous avons de la chance d'allouer T_3 à P_1 : comme P_1 a exécuté T_1, il n'a pas de coût de communication à payer, et peut débuter T_3 au top $t = 1$. Par contre, P_2 doit attendre le top $t = 6$ pour débuter T_2. Au top $t = 2$, on a $\mathcal{Q} = (T_4, T_5)$ (on départage les ex-aequos en utilisant les numéros des tâches), et deux processeurs disponibles P_1 et P_3. En effet, bien qu'il soit inactif, P_2 est considéré comme occupé jusqu'à avoir terminé l'exécution de T_2. On alloue donc T_4 à P_1 (l'exécution peut débuter immédiatement) et T_5 à P_3 (l'exécution ne peut pas débuter avant $t = 5$). En continuant selon ce principe, on obtient l'exécution décrite à la Figure 4.12.

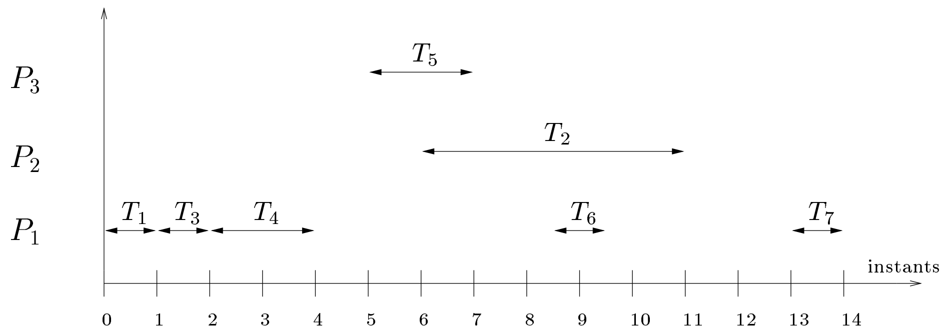


FIG. 4.12 – Exécution du chemin critique version naïve.

On obtient une latence égale à 14. Remarquons que nous aurions obtenu le même résultat avec seulement deux processeurs : T_5 aurai été exécutée par P_1 au top $t = 4$ plutôt que par P_3 au top $t = 5$: c'es la seule différence. Dans les deux cas, on obtient une latence moins bonne que l'exécution séquentielle (avec un seul processeur). Il doit y avoir une amélioration possible !

4.7.2 Chemin critique modifié

Si on analyse l'exécution de la version naïve du chemin critique dans notre petit exemple, on s'aperçoit qu'on a pris une mauvaise décision en allouant T_2 à P_2. Au top $t = 1$, $\mathcal{Q} = (T_3, T_2)$. La première allocation, celle de T_3 à P_1, est judicieuse. Mais la seconde, celle de T_2 à P_2, ne l'est pas. Il faudrait allouer T_2 à P_1 aussi, même si celui-ci n'est pas disponible : en effet, P_1 peut débuter T_2 plus tôt que P_2 ne peut le faire. La règle du chemin critique modifié (*MCP, Modified Critical Path*) est : *allouer une tâche libre au processeur qui permet de débuter son exécution au plus tôt, compte-tenu des décisions déjà prises.* Il est important de préciser la signification de la phrase "compte-tenu des décisions déjà prises". Les tâches de la file de priorité sont traitées l'une après l'autre. A tout moment, on sait quels processeurs sont disponibles et quels processeurs sont actifs. En outre, pour les processeurs actifs, on sait quand ils auront terminé l'exécution de leur tâche. On a donc tous les éléments pour sélectionner le processeur qui permettra de débuter au plus tôt l'exécution de la tache libre en cours de traitement. On peut fort bien choisir un processeur actif, comme pour la discussion sur T_2.

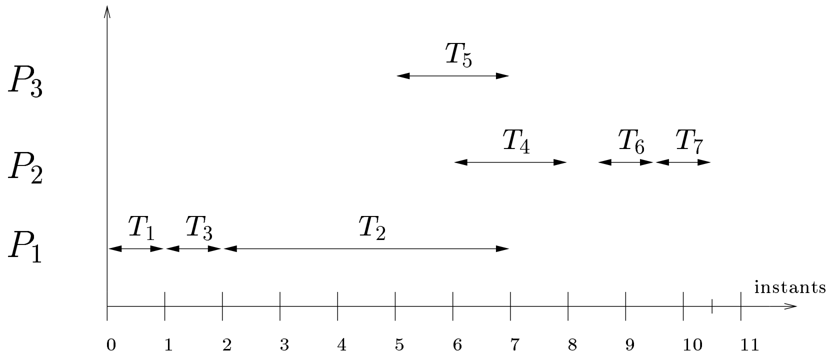


FIG. 4.13 – Ordonnancement MCP pour l'exemple de la Figure 4.8.

Plutôt que d'écrire l'algorithme MCP sous forme détaillée, nous reprenons notre exemple avec trois processeurs, et obtenons l'ordonnancement de la Figure 4.13. Comme prévu, T_2 est allouée à P_1. De même, T_6 est allouée à P_2, qui permet une exécution au top $t = 8.5$, contre 9.5 sur P_1 ou P_3. La nouvelle latence est 10.5, à comparer avec la latence 14 du CP naïf.

4.7.3 Comment comparer des heuristiques ?

La comparaison des heuristiques CP naïf et MCP pour notre petit exemple ne doit pas conduire à des conclusions trop tranchées. Il existe des cas où le CP naïf se comporte mieux que MCP. Dans le petit exemple de la Figure 4.14, le lecteur pourra vérifier qu'avec deux processeurs, la latence de MCP est égale à 15, alors que la latence du CP naïf est de 14.

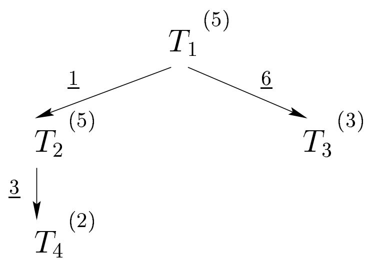
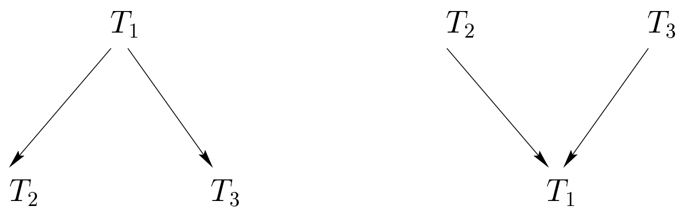


FIG. 4.14 – Un exemple où CP naïf est meilleur que MCP.

Cependant, l'intuition conduit à préférer MCP au CP naïf : MCP devrait être meilleur dans la plupart des cas. Comment quantifier cette assertion ?

Pour être moins spécifique qu'avec des exemples particuliers, on peut essayer de comparer les deux heuristiques sur des types de graphes donnés. Prenons deux cas très simples : un *fork* à deux sommets, et un *join* à deux sommets. Dans les Figures 4.15(a) et 4.15(b), on a trois tâches de même poids w. Les coûts de communication sont tous égaux à c. Supposons disposer de deux processeurs, P_1 et P_2. Deux cas sont à considérer :



(a) Fork graph (b) Join graph

FIG. 4.15 – Comparaison des deux heuristiques.

1. **Fork Graph** (Figure 4.15(a)). Le CP naïf ordonnance T_1 sur P_1 au top $t = 0$. Puis il ordonnance T_2 sur P_1 au top $t = w$, et T_3 sur P_2 au top $w + c$, d'où une latence égale à $2w + c$. MCP construit la même solution si $w > c$. Mais si $w < c$, le meilleur top de début de T_3 est $t = 2w$ sur P_1, et MCP ordonnance toutes les tâches sur P_1, avec une latence égale à $3w < 2w + c$. Conclusion : MCP surpasse CP naïf si $w < c$ et fait match égal sinon.
2. **Join Graph** (Figure 4.15(b)). CP naïf et MCP construisent la même solution. Au top $t = 0$, ils ordonnancent T_2 sur P_1 et T_3 sur P_2. Au top $t = w + c$, T_1 est ordonnancée indifféremment sur P_1 ou P_2, et la latence est $2w + c$. Ce n'est pas optimal si $w < c$; il vaut mieux ordonnancer les trois tâches sur le même processeur ! Bien sûr ceci est interdit pour un ordonnancement de liste : aucun processeur ne peut être délibérément laissé inactif.

Cette petite discussion montre qu'il est délicat de tirer des conclusions. Le problème est NP-complet, et nous ne faisons que comparer des heuristiques. D'une manière générale, il y a trois pistes possibles :

1. **Théorique :** Garantir l'heuristique avec un facteur d'approximation donné.
2. **Expérimental :** Générer des graphes aléatoires et/ou des classes de graphes "typiques des applications" pour comparer des heuristiques.
3. **Astucieux :** Démontrer l'optimalité de l'heuristique pour certaines classes de graphes : fork, join, fork-join, arbre, degré entrant/sortant borné, etc.

La première piste est la plus intéressante, au moins d'un point de vue conceptuel : on est assuré que l'heuristique donnera un résultat à une distance bornée de l'optimal, dans le pire des cas. La deuxième piste est très utile en pratique. Enfin, la troisième piste permet d'améliorer les heuristiques pour atteindre l'optimalité sur certaines classes de graphes ...et de publier quelques articles de recherche :-)

Notes bibliographiques

Ce chapitre s'inspire honteusement de l'excellent livre de Darte, Robert et Vivien [28] : en fait, nous avons traduit en français le premier chapitre et le début du deuxième chapitre de cet ouvrage immortel.

Plus sérieusement, les notions présentées ici sont pour la plupart élémentaires. Citons le livre de Coffman [23], pionnier du domaine, le livre plus récent de El-Rewini, Lewis et Ali [31], la compilation d'articles éditée par IEEE [11], et le très riche ouvrage de synthèse édité par Chrétienne et al [21].

Le théorème 8 est du à Chréetienne [22]. Picouleau [56] montre que Pb(∞) reste NP-complet même en supposant les poids de toutes les tâches et de tous les coûts de communication égaux – c'est le problème UET-UCT (Unit Execution Time-Unit Communication Time). L'heuristique de

Hanen et Munie s'étend au cas des ressources bornées, voir [41]. Pour Pb(p) avec communications, les meilleures heuristiques procèdent en deux étapes : regroupement des tâches en grappes (ou *clusters*, en supposant disposer de ressources illimitées, puis ordonnancement et allocation des clusters sur les processeurs physiques. Nous renvoyons à [28] pour une description complète de ce type d'heuristiques.

Enfin, notons que l'Appendice A5 de Garey et Johnson [34] propose une liste de problèmes d'ordonnancement NP-complets. Si cette liste date un peu, elle contient bon nombre de problèmes classiques, auxquels il est toujours bon de revenir !

Deuxième partie

Algorithmique parallèle

Chapitre 5

Algorithmique sur anneau de processeurs

5.1 Introduction

Ce chapitre présente quelques algorithmes destinés à s'exécuter sur un anneau de processeurs. Cette topologie a le mérite de la simplicité, ce qui s'avérera précieux pour nos premiers pas en algorithmique à mémoire distribuée. Mais surtout, un réseau d'interconnexion linéaire s'avère un choix naturel pour partitionner des données régulières, comme des matrices ou des images, en lignes ou en colonnes.

Le chapitre débute (Paragraphe 5.2) par quelques primitives de communication élémentaires : diffusion, échange total, ... Notre premier algorithme est un simple produit matrice-vecteur (Paragraphe 5.3). Nous enchaînons sur un balayage d'image (Paragraphe 5.4), et nous terminons avec l'incontournable décomposition LU d'une matrice dense (Paragraphe 5.5).

5.2 Macro-communications sur un anneau

On suppose disposer de p processeurs arrangés en un anneau orienté (unidirectionnel), comme indiqué Figure 5.1. Les processeurs sont numérotés de 0 à $p-1$, et chaque processeur peut connaître son numéro logique en exécutant l'appel de fonction

$$\mathsf{q} \leftarrow \mathsf{my\_num()}$$

On peut connaître le nombre total p de processeurs en appelant la fonction tot_proc_num().

Le principe de base est que tous les processeurs exécutent le même programme, mais opèrent sur des données présentes dans leur mémoire locale (on parle d'un mode de fonctionnement SPMD, *Single Program Multiple Data*). Pour accéder à des valeurs qui ne sont pas dans leur propre mémoire, les processeurs doivent communiquer par envoi et réception de messages. Chaque processeur peut envoyer un message à son successeur sur l'anneau à l'aide de la procédure

$$\mathsf{send(adr, L)}$$

où adr est l'adresse, dans la mémoire du processeur émetteur, de la première valeur à envoyer ; L est la longueur du message, exprimée en nombre de données émises (bien sûr, pour une vraie mise en oeuvre, il faudrait exprimer L en octets) : les mots du message sont supposés rangés de manière contiguë en mémoire, à partir de l'adresse de base adr. De même, pour recevoir un message, on appelle

$$\mathsf{receive(adr, L)}$$

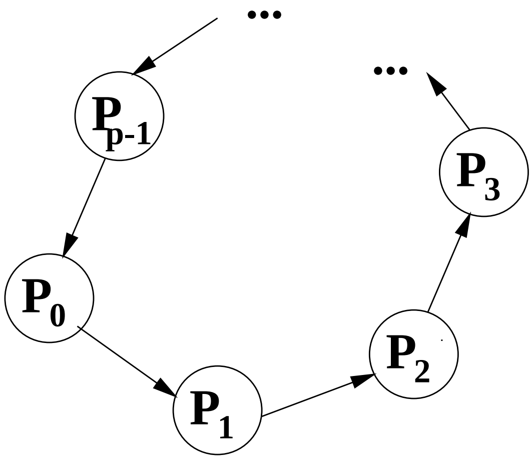


FIG. 5.1 – Anneau orienté de p processeurs.

Soulignons plusieurs points importants :

- Comme tous les processeurs exécutent le même programme, il doit y avoir un receive "en face de" chaque send : si le processeur numéro i exécute un receive, c'est que son prédécesseur (de numéro $i-1$ modulo p) exécute un send. Sinon le programme va se bloquer !
- Comme il n'y a qu'un seul lien de communication sortant, nous n'avons pas besoin d'identifier le destinataire du send : c'est le successeur du processeur émetteur. De même on sait en effectuant un receive que le message provient du prédécesseur. Pour un réseau d'interconnexion plus général, il faut préciser le numéro du lien que va emprunter le message, ou (de manière équivalente) le numéro logique du destinataire.
- L'adresse de l'en-tête du message qui est en argument d'un send est une adresse locale à la mémoire du processeur émetteur ; celle qui figure en argument d'un receive est locale à la mémoire du processeur récepteur. Même si on peut utiliser la même variable pour les adresses d'un send et d'un receive qui se font face, rien n'indique que la valeur de la variable sera la même dans les deux processeurs. Et on peut parfaitement utiliser des variables différentes, comme dans :

```
q ← mynum();
if (q=0) then send(adr1, L);
if (q=1) then receive(adr2, L);
```

Nous n'avons pas encore parlé de la sémantique des deux primitives de communication. On peut faire trois hypothèses différentes :

- Une hypothèse restrictive est de supposer chaque send et chaque receive bloquant, i.e. le processeur qui initie une primitive de communication ne peut pas continuer à exécuter son programme avant la terminaison de la communication. C'est un mode d'échange de message purement synchrone, par rendez-vous, typique des machines parallèles de première génération, sans co-processeur dédié à la communication.
- Une hypothèse classique est de garder le receive bloquant mais de supposer le send non bloquant : un processeur peut alors initier l'envoi de données puis continuer à travailler pendant que la communication s'exécute ; le receive reste bloquant, avec la justification qu'on ne devra effectuer un receive que juste avant l'utilisation des données transmises.
- Enfin, une hypothèse plus récente consiste à supposer les deux primitives non bloquantes ; un même processeur peut alors envoyer, recevoir et calculer simultanément, du moment bien sûr

que les données mises en jeu par les trois opérations ne sont pas partagées en écriture. On imagine très bien trois processus ou *threads* de contrôle, un pour le calcul, et deux dédiés aux communications asynchrones. Sur la plupart des machines, on implémente une communication asynchrone à l'aide de deux appels, un pour initier la communication (envoi ou réception), et l'autre pour vérifier qu'elle est bien terminée.

Il est difficile de modéliser le coût d'une communication sur les machines parallèles actuelles (voir le Chapitre 6). Nous nous contenterons ici de la formule la plus simple : le coût pour envoyer/recevoir un message de longueur L est $\beta + L\tau$, où β et τ sont deux constantes architecturales : β est le coût d'initialisation (ou *startup*), dû à la configuration du matériel et à la mise en oeuvre logicielle de la communication ; τ est l'inverse de la bande passante, et mesure la vitesse du lien de communication en régime permanent. Si un message de longueur L est envoyé, disons, du processeur 0 au processeur q, le coût sera proportionnel au nombre d'étapes de communication entre voisins, i.e. $q(\beta + L\tau)$.

Nous reviendrons sur l'impact des différentes hypothèses sur les modes de communication au fil de l'écriture de nos premiers programmes.

5.2.1 Diffusion

Soit k un processeur fixé. On veut écrire un programme qui réalise l'envoi par k d'un même message de longueur L à tous les autres processeurs : c'est une diffusion, ou *broadcast*. Cette primitive de communication est fondamentale : par exemple, le processeur émetteur peut jouer le rôle d'un processeur maître qui diffuse l'information (taille du problème, données, etc) à tous les autres processeurs.

Au début du programme, le message est stocké à l'adresse adr du processeur émetteur k. A la fin du programme, le message sera stocké à l'adresse adr de chaque processeur. Tous les processeurs vont exécuter la procédure

broadcast(k, adr, L)

On va faire circuler le message sur l'anneau, du processeur k au processeur k+1, puis du processeur k+1 au processeur k+2, etc (les indices sont pris modulo p). Il n'y a pas de parallélisme possible : les send et les receive de chaque processeur ne sont pas indépendants. On obtient le programme du processeur numéro q :

```
broadcast(k, adr, L) {
q ← my_num() ;
p ← tot_proc_num() ;
if q = k then
   send(adr, L) ;
else
   if q = k − 1 mod p then
     receive(adr, L) ;
   else
     receive(adr, L) ;
     send(adr, L) ;
}
```

Tout d'abord, il est important de noter que le prédécesseur $k-1$ modulo p du processeur émetteur k ne doit pas envoyer le message : non pas tant parce que k a déjà l'information (c'est lui l'émetteur), mais surtout parce qu'il n'est pas prévu de receive dans son programme !

Pour que le programme soit correct, il faut évidemment supposer le receive bloquant, puisqu'on transmet directement le message dès réception. La sémantique du send n'a pas d'importance ici.

Comme on a $p-1$ communications successives, le temps de la diffusion d'un message de longueur L est

$$(p-1)(\beta + L\tau).$$

5.2.2 Diffusion personnalisée

On étudie maintenant la diffusion personnalisée, ou *scatter*. Le processeur k envoie un message différent à chaque autre processeur. Pour simplifier l'analyse, on suppose que tous les messages envoyés ont la même longueur L. Une diffusion personnalisée est utile, par exemple, pour distribuer des données différentes, comme des blocs de matrices ou d'images partitionnées entre les différents processeurs.

Au début du programme, le processeur k contient le message pour le processeur q à l'adresse adr[q]. A la fin du programme, chaque processeur a son message à l'adresse adr. Par uniformité, il y a aussi un message adr[k] pour le processeur k.

L'idée de l'algorithme est d'envoyer les messages en pipeline, en commençant par le message destiné au processeur le plus éloigné $k-1$ modulo p : pendant que ce message avancera sur l'anneau, d'autres messages destinés à des processeurs plus proches circuleront également :

```
scatter(k, adr, L) {
q ← my_num();
p ← tot_proc_num();
if q = k then
   adr ← adr[k];
else
   receive(adr, L);
   for i = 1 to k − 1 − q mod p
     {{ send(adr, L) // receive(temp, L) }};
     adr ← temp;
}
```

Dans la procédure, chaque processeur utilise une adresse tampon temp pour permettre la parallélisation de l'émission d'un message et de la réception du suivant. Cette parallélisation est symbolisée par la notation $\{\{\ldots // \ldots\}\}$. On a supposé le send non bloquant et le receive bloquant : si ce dernier ne l'était pas, il faudrait simplement ajouter un point de synchronisation à l'entrée de chaque pas de la boucle (y compris avant celle-ci), pour être assuré de ne pas transmettre des données non encore reçues. Notons enfin que l'instruction adr ← temp doit s'interpréter comme une mise à jour de pointeur, et non pas comme une copie du message.

Le temps total est le même que pour la diffusion, à savoir $(p-1)(\beta + L\tau)$. Si nous avons pu faire circuler $p-1$ messages au lieu d'un seul dans le même temps, c'est bien grâce au pipeline qui permet l'utilisation simultanée de plusieurs liens de communication de l'anneau.

5.2.3 Echange total

Notre troisième (et dernière) procédure est l'échange total, ou *all-to-all*. Chaque processeur k veut envoyer un message à tous les autres : il s'agit donc de p diffusions simultanées. Supposons encore tous les messages de même longueur L. Au départ, chaque processeur dispose d'un message stocké à l'adresse my_adr. A la fin, chaque processeur aura en mémoire le même tableau, contenant à la case k le message diffusé par le processeur numéro k. L'algorithme est évident : on fait tourner les messages en $p-1$ étapes de communication. On a la procédure :

```
all-to-all(my_adr, adr, L) {
q ← my_num() ;
p ← tot_proc_num() ;
adr[q] ← my_adr ;
for i = 1 to p − 1
   {{ send(adr[q-i+1 mod p], L) // receive(adr[q-i mod p], L) }} ;
}
```

Les commentaires au sujet de la sémantique sont les mêmes que pour la diffusion personnalisée. Le temps d'exécution est encore $(p-1)(\beta + L\tau)$, et cette fois tous les liens de communication sont utilisés à plein régime.

Nous ne détaillerons pas la dernière procédure classique, celle de l'échange total personnalisé : chaque processeur k a un message $m[k,q]$ différent pour tous les processeurs q. Notons simplement l'humour américain, qui appelle cette procédure *gossiping*, ou commérage. No comment !

5.2.4 Diffusion pipelinée

Peut-on diminuer le temps de la diffusion ? Nous avons proposé une procédure qui nécessite un temps $(p-1)(\beta + L\tau)$ pour diffuser un message de longueur L. L'algorithme est simple, mais guère efficace pour des messages de grande longueur.

Pour améliorer les choses, on peut penser à découper le message en r morceaux (de taille égale, en supposant L divisible par r). Le processeur émetteur envoie successivement les r morceaux, qui circulent simultanément sur l'anneau : c'est ce mode pipeline qui va diminuer le temps d'exécution. Au début du programme, les r morceaux du message sont stockés aux adresses adr[1],. . .,adr[r] du processeur k. A la fin du programme, le message est stocké dans tous les processeurs aux mêmes adresses. Pendant qu'un processeur reçoit une portion du message, il envoie la portion précédente au processeur suivant, qui lui-même envoie la portion précédente au processeur suivant, etc :

```
broadcast(k, adr, L) {
q ← my_num() ;
p ← tot_proc_num() ;
if q = k then
   for i = 1 to r send(adr[i], L/r) ;
else
   if q = k − 1 mod p then
     for i = 1 to r receive(adr[i], L/r) ;
   else
     receive(adr[1], L/r) ;
     for i = 1 to r − 1 {{ send(adr[i], L/r) // receive(adr[i+1], L/r) }} ;
     send(adr[r], L/r) ;
}
```

Pour évaluer le temps d'exécution, regardons à quel moment l'ensemble du message sera arrivé au dernier processeur $k-1 \bmod p$. Il faut d'abord $p-1$ étapes de communication pour qu'arrive le premier morceau, soit $(p-1)(\beta + \frac{L}{r}\tau)$; puis les autres $r-1$ morceaux arrivent immédiatement les uns derrière les autres, en $(r-1)(\beta + \frac{L}{r}\tau)$. Donc en tout un temps égal à

$$(p-2+r)(\beta + \frac{L}{r}\tau)$$

On cherche la valeur de r qui minimise cette expression. C'est un problème de la forme $(c + ar)(d + b/r)$ avec quatre constantes a, b, c et d. En enlevant les termes constants, on se ramène à minimiser $ad.r + cb/r$, d'où la valeur

$$r_{opt} = \sqrt{\frac{cb}{ad}}.$$

En effet, la somme de deux termes dont le produit est constant est minimale quand ceux-ci sont égaux, c'est le fameux théorème de la chèvre dont le périmètre de l'enclos, à surface fixée, est minimale si c'est un carré[1]. Et la valeur minimale obtenue avec r_{opt} est

$$\begin{aligned} ((c+ar)(d+b/r))[\sqrt{cb/ad}] &= ((c/d)(d+\sqrt{dba/c}))(d+\sqrt{adb/c}) \\ &= (\sqrt{c/d}(d+\sqrt{dba/c}))^2 \\ &= (\sqrt{cd}+\sqrt{ba})^2 \ . \end{aligned}$$

Donc, ici, $r_{opt} = \sqrt{\frac{L(p-2)\tau}{\beta}}$ et le temps optimal est

$$(\sqrt{(p-2))\beta} + \sqrt{L\tau})^2 \ .$$

Dans cette expression, p, β et τ sont fixés. Donc, pour de longs messages, l'expression devient asymptotiquement équivalente à $L\tau$: le facteur p a disparu. Magique !

5.3 Produit matrice-vecteur

Programmons maintenant notre premier algorithme parallèle, à savoir la multiplication $y = Ax$ d'une matrice A de dimension $n \times n$ par un vecteur x à n composantes (on numérotera les indices de 0 à $n-1$). On utilise toujours un anneau orienté de p processeurs. On peut voir le produit matrice-vecteur comme un ensemble de n produits scalaires :

```
for i = 1 to n do
  for j = 1 to n do
    y[i] = y[i] + a[i,j] * x[j] ;
```

Chaque instance de la boucle externe i calcule un produit scalaire, celui de la i-ème ligne de A par le vecteur x. Pour paralléliser le produit matrice-vecteur, on va donc distribuer le calcul des produits scalaires aux processeurs. Supposons n, la taille de la matrice, divisible par p, le nombre de processeurs, et posons $r = n/p$. Chaque processeur va stocker un bloc de r lignes de la matrice, et calculer les composantes du résultat y correspondantes. Toutes les lignes de la matrice étant de même taille, on peut par exemple allouer les r premières lignes au premier processeur, les r suivantes au deuxième processeur, etc.

Si on suppose le vecteur x dupliqué sur tous les processeurs, les calculs sont totalement indépendants. Mais comme chaque processeur ne calcule qu'une fraction du résultat y, il est plus modulaire de supposer qu'à l'appel de la procédure, x est distribué comme A et y : les r premières composantes au premier processeur, etc. Cette hypothèse permettra, par exemple, d'enchaîner un autre produit $z = By$ en appelant la même procédure, pour peu que B soit distribuée comme A.

Chaque processeur a donc en mémoire r lignes de la matrice A, rangées dans une matrice a de dimension $r \times n$. Plus précisément, le processeur q contient les lignes qr à $(q+1)r-1$ de la matrice, et les composantes de même rang des vecteurs x et y. On aurait donc la déclaration de variable :

```
var a : array[0..r-1,0..n-1] of real ;
var x, y : array[0..r-1] of real ;
```

[1]Le lecteur méfiant vis à vis des proverbes agricoles peut toujours dériver la fonction :-)

Cette déclaration permet d'enfoncer le clou : l'élément $a[0,0]$ du processeur 0 correspond à l'élément $A[0,0]$ de la matrice de départ, mais l'élément $a[0,0]$ du processeur 1 correspond à l'élément $A[r,0]$ de cette même matrice. Au passage, notons que l'intérêt du parallélisme n'est pas seulement de résoudre un problème plus vite : c'est aussi de résoudre des problèmes plus gros, puisqu'on distribue ici une matrice de taille n^2 dans p mémoires locales, au lieu d'une seule mémoire dans le cas séquentiel.

$$\begin{array}{c|c c}
P_0 & \begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} & A_{05} & A_{06} & A_{07} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} & A_{15} & A_{16} & A_{17} \end{pmatrix} & \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \\ \hline
P_1 & \begin{pmatrix} A_{20} & A_{21} & A_{22} & A_{23} & A_{24} & A_{25} & A_{26} & A_{27} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} & A_{35} & A_{36} & A_{37} \end{pmatrix} & \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \\ \hline
P_2 & \begin{pmatrix} A_{40} & A_{41} & A_{42} & A_{43} & A_{44} & A_{45} & A_{46} & A_{47} \\ A_{50} & A_{51} & A_{52} & A_{53} & A_{54} & A_{55} & A_{56} & A_{57} \end{pmatrix} & \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \\ \hline
P_3 & \begin{pmatrix} A_{60} & A_{61} & A_{62} & A_{63} & A_{64} & A_{65} & A_{66} & A_{67} \\ A_{70} & A_{71} & A_{72} & A_{73} & A_{74} & A_{75} & A_{76} & A_{77} \end{pmatrix} & \begin{pmatrix} x_6 \\ x_7 \end{pmatrix}
\end{array}$$

(a) Distribution initiale des données

FIG. 5.2 – Produit matrice-vecteur : $n = 8$, $p = 4$, $r = 2$; partie 1/5.

$$\begin{array}{c|c c c}
P_0 & \begin{pmatrix} A_{00} & A_{01} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ A_{10} & A_{11} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} & \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \\ \hline
P_1 & \begin{pmatrix} \bullet & \bullet & A_{22} & A_{23} & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & A_{32} & A_{33} & \bullet & \bullet & \bullet & \bullet \end{pmatrix} & \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \\ \hline
P_2 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & A_{44} & A_{45} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & A_{54} & A_{55} & \bullet & \bullet \end{pmatrix} & \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \\ \hline
P_3 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{66} & A_{67} \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{76} & A_{77} \end{pmatrix} & \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} & \text{temp} \leftarrow \begin{pmatrix} x_4 \\ x_5 \end{pmatrix}
\end{array}$$

(b) Première étape

FIG. 5.3 – Produit matrice-vecteur : $n = 8$, $p = 4$, $r = 2$; partie 2/5

Le principe de l'algorithme est illustré aux Figures 5.2 à 5.6. Il y a p étapes. A chaque étape, les processeurs calculent un produit partiel, celui d'une matrice $r \times r$ par un vecteur de taille r. Au départ, à l'étape 0, le processeur q dispose de la tranche q du vecteur x, et peut donc calculer un produit partiel qui correspond aux blocs diagonaux de la matrice A. En d'autres mots, en utilisant une notation par blocs (x_q est le q-ème bloc de taille r de x, etc), le processeur q peut effectuer le calcul $y_q = A_{q,q}x_q$. Pendant ce calcul, on fait tourner le vecteur x, i.e. on décale circulairement les composantes par blocs. A l'étape 1, le processeur q, qui vient de recevoir $x_{q-1 \bmod p}$, peut effectuer le calcul suivant, à savoir $y_q = y_q + A_{q,q-1 \bmod p}x_{q-1 \bmod p}$, tout en faisant à nouveau tourner x. A la fin du programme, i.e. à la fin des p étapes, le vecteur y du processeur q contient les r composantes désirées $y_{qr}, \ldots, y_{qr+r-1}$ du résultat :

$$\begin{array}{c|c}
P_0 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{06} & A_{07} \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{16} & A_{17} \end{pmatrix} \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \\
\hline
P_1 & \begin{pmatrix} A_{20} & A_{21} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ A_{30} & A_{31} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \\
\hline
P_2 & \begin{pmatrix} \bullet & \bullet & A_{42} & A_{43} & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & A_{52} & A_{53} & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \\
\hline
P_3 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & A_{64} & A_{65} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & A_{74} & A_{75} & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_2 \\ x_3 \end{pmatrix}
\end{array}$$

(c) Deuxième étape

FIG. 5.4 – Produit matrice-vecteur : $n = 8$, $p = 4$, $r = 2$; partie 3/5

$$\begin{array}{c|c}
P_0 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & A_{04} & A_{05} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & A_{14} & A_{15} & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \\
\hline
P_1 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{26} & A_{27} \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{36} & A_{37} \end{pmatrix} \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \\
\hline
P_2 & \begin{pmatrix} A_{40} & A_{41} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ A_{50} & A_{51} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \\
\hline
P_3 & \begin{pmatrix} \bullet & \bullet & A_{62} & A_{63} & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & A_{72} & A_{73} & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}
\end{array}$$

(c) Troisième étape

FIG. 5.5 – Produit matrice-vecteur : $n = 8$, $p = 4$, $r = 2$; partie 4/5

```
matrice-vecteur(A, x, y) {
q ← my_num() ;
p ← tot_proc_num() ;
for step = 0 to p − 1 do
   {{ send(x, r) // receive(temp, r) //
     for i=0 to r-1 do
       for j=0 to r-1 do
         y[i] ← y[i]+ a[i,(q - step mod p)r+j]*x[j] }} ;
   x ← temp ;
}
```

La rotation du vecteur x à la dernière étape est en principe inutile, puisque chaque processeur reçoit sa donnée initiale. Mais nous avons écrasé celle-ci au fil de la procédure, et les communications ont lieu en parallèle avec les calculs, donc ce n'est pas bien grave. A ce propos, quel est le temps d'exécution $T(p)$ de la procédure ? Facile : il y a p étapes identiques, chacune régie par le temps du plus long des trois processus mis en oeuvre :

$$T(p) = (p - 1) \max\{r^2\tau_a, \beta + r\tau_c)$$

Nous avons noté τ_a le temps de calcul élémentaire, et τ_c le temps de communication élémentaire. Comme $r = n/p$, pour un nombre de processeurs p fixé, le terme dominant à chaque étape est le terme de calcul dès que n est assez grand ($\frac{n^2}{p^2}\tau_a \geq \beta + \frac{n}{p}\tau_c$). Voila alors une jolie parallélisation d'efficacité égale à 1.

$$
\begin{array}{c|c}
P_0 & \begin{pmatrix} \bullet & \bullet & A_{02} & A_{03} & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & A_{12} & A_{13} & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \\
\hline
P_1 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & A_{24} & A_{25} & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & A_{34} & A_{35} & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_2 \\ x_3 \end{pmatrix} \\
\hline
P_2 & \begin{pmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{46} & A_{47} \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & A_{56} & A_{57} \end{pmatrix} \begin{pmatrix} x_6 \\ x_7 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_4 \\ x_5 \end{pmatrix} \\
\hline
P_3 & \begin{pmatrix} A_{60} & A_{61} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ A_{70} & A_{71} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \quad \text{temp} \leftarrow \begin{pmatrix} x_6 \\ x_7 \end{pmatrix}
\end{array}
$$

(c) Quatrième étape

FIG. 5.6 – Produit matrice-vecteur : $n = 8$, $p = 4$, $r = 2$; partie 5/5

Un lecteur grincheux voudra peut-être gâcher notre joie en soulignant la relative vacuité de nos efforts : un échange total au début de la procédure pour dupliquer x, et voila une procédure triviale sans communication pour le prix d'un léger accroissement des exigences en mémoire, et un temps d'exécution asympotiquement équivalent dès que n est grand ... d'accord, mais quelle insensibilité à la beauté algorithmique !

5.4 Algorithmes de balayage d'image

Nous présentons dans ce paragraphe une étude de cas complète : la parallélisation d'un algorithme générique pour le traitement d'images. La version séquentielle de l'algorithme est basée sur des balayages successifs au cours desquels chaque pixel est mis à jour selon un masque appliqué à son voisinage. Deux applications importantes sont le calcul de la distance au contour et la détermination de la trajectoire optimale. Le caractère récurrent de l'algorithme (qu'on peut voir comme une instance particulière de la programmation dynamique) rend difficile sa parallélisation efficace.

5.4.1 Algorithme séquentiel

Nous décrivons ici l'algorithme séquentiel générique, et nous détaillons deux applications importantes : le calcul de la distance au contour, et la détermination d'une trajectoire optimale. Soit P une grille de $n \times n$ points, où n est un entier positif donné. Les huit voisins d'un point p sont référencés par les points cardinaux correspondant à la direction du compas : voir la Figure 5.7, où nous avons conservé les initiales anglaises.

$$
\begin{array}{ccc}
NW & N & NE \\
W & \boxed{\mathrm{p}} & E \\
SW & S & SE
\end{array}
$$

FIG. 5.7 – Identification des huit voisins d'un point p.

L'algorithme générique que nous considérons consiste à balayer la grille un certain nombre de fois, en alternant les orientations des passes : la grille est parcourue de haut en bas et de gauche à droite (passe avant, ou PAV), puis de bas en haut et de droite à gauche (passe arrière, ou PAR). Au cours d'une passe avant, les points de la grille sont mis à jour suivant le masque décrit Figure 5.8(a).

La nouvelle valeur d'un point p est obtenue comme fonction de la valeur courante et des valeurs courantes des voisins NW, N, NE et W de p :

(PAV) p := mise_à_jour_avant(p, W, NW, N, NE)

De même pour la passe arrière, les valeurs des points sont mises à jour en fonction du masque de la Figure 5.8(b) :

(PAR) p := mise_à_jour_arrière(p, E, SE, S, SW)

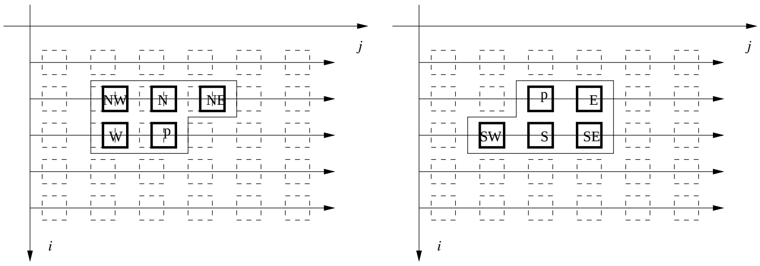


FIG. 5.8 – Les masques associés aux balayages avant et arrière.

Distance au contour

Soit P une image binaire composée d'une figure $F = \{+\infty\}$, entièrement entourée par son complémentaire $P \setminus F = \{0\}$. L'image obtenue par calcul de la distance (CD) au contour est une copie de F, dans laquelle on associe à chaque pixel sa distance (selon une certaine métrique) à $P \setminus F$. Dans l'image CD, chaque pixel peut être associé à un disque centré en ce pixel : le rayon du disque dépend de la valeur du pixel, tandis que la forme du disque dépend de la métrique choisie. Un disque est dit maximal si aucun autre disque ne le recouvre totalement. Comme l'union des disques maximaux coïncide avec F, l'axe médian, défini comme le lieu des centres des disques maximaux (les points de CD dont la valeur est un maximum local), joue un rôle important pour diminuer l'occupation mémoire et pour divers algorithmes d'analyse et description des formes (voir par exemple [61]).

Pour ce calcul de distance, un algorithme de programmation dynamique bien connu consiste à balayer l'image deux fois. On effectue d'abord une passe avant :

(PAV) $p := \min(p, W + t_1, NW + t_2, N + t_1, NE + t_2)$

Le principe de mise à jour est très simple : la distance de p au contour est le minimum de sa valeur actuelle et des valeurs des voisins du masque incrémentées de leur distance à p. Les coefficients t_1 et t_2 sont les poids choisis pour les voisins horizontaux/verticaux et pour les voisins diagonaux respectivement. Pour $t_1 = 1$ et $t_2 = +\infty$ on a la distance de Manhattan d_4. Pour $t_1 = t_2 = 1$ on retrouve la distance de l'échiquier d_8. Pour $t_1 = 3$ et $t_2 = 4$ on a une distance qui est une bonne approximation de la distance Euclidienne [8].

Le balayage arrière est tout à fait similaire. En utilisant le masque de la Figure 5.8(b) cette fois :

(PAR) $p := \min(p, E + t_1, SE + t_2, S + t_1, SW + t_2)$

Après deux passes, chaque point est remplacé par sa distance à $P \setminus F$. Plusieurs applications du calcul CD pour déterminer des propriétés géométriques comme la surface, le contour et le périmètre sont décrits dans [64].

Calcul de la trajectoire optimale

Le calcul de la trajectoire optimale s'effectue également à l'aide de passes successives sur une carte dont les points sont caractérisés par un coût de traversabilité [15]. Il s'agit de déterminer un chemin de coût minimal d'un point donné de la carte (la source) à tous les autres points.

Sur la carte P de taille $n \times n$, on associe à chaque point p un réel positif ou nul $tc(p)$ correspondant au coût de la traversabilité en p. Etant donné un point p et un voisin q de p, l'arête (p,q) est pondérée par le coût

$$c(p,q) = \begin{cases} \frac{tc(p)+tc(q)}{2} & \text{si } q \in \{N, S, W, E\} \\ \frac{tc(p)+tc(q)}{\sqrt{2}} & \text{si } q \in \{NW, NE, SW, SE\} \end{cases}$$

Le coefficient $\sqrt{2}$ reflète la distance supplémentaire due à la connexion diagonale. Etant donné un point particulier appelé la source, on veut calculer le plus court chemin (ou chemin de poids minimal) de la source à tous les autres points de la carte.

Bitz et Kung [15] proposent l'algorithme suivant : initialement, le meilleur coût connu $f(p)$ pour tout point p est initialisé à la valeur 0 à la source et $+\infty$ en tous les autres points. L'algorithme effectue une succession de passes avant et arrière. Pour la valeur courante du point p, on met à jour le meilleur coût connu $f(p)$ s'il existe un meilleur chemin passant par l'un des voisins sélectionnés par le masque de balayage. Par exemple lors d'une passe avant, si le meilleur coût connu $f(W)$ du voisin Ouest de p plus le coût $c(W,p)$ de l'arête qui relie W à p est inférieur à $f(p)$, alors on met à jour $f(p)$, i.e. $f(p) := f(W) + c(W,p)$. Dans le cas général, la formule de mise à jour est la suivante :

$$\text{(PAV)} \qquad f(p) \leftarrow \min\left(f(p), f(W) + c(W,p), f(NW) + c(NW,p),\right.$$
$$\left. f(N) + c(N,p), f(NE) + c(NE,p)\right)$$

$$\text{(PAR)} \qquad f(p) \leftarrow \min\left(f(p), f(E) + c(E,p), f(SE) + c(SE,p),\right.$$
$$\left. f(S) + c(S,p), f(SW) + c(SW,p)\right)$$

Dans ces expressions, on utilise la définition de $c(p,q)$ pour calculer les poids des arêtes. Etant données les valeurs initiales spécifiées plus haut, les passes avant et arrière sont effectuées successivement, jusqu'à ce qu'aucune valeur de $f(p)$ ne change au cours d'une passe. Colorions les arêtes d'un chemin suivant leur direction : les arêtes pointant dans les directions W, NW, N et NE sont coloriées en bleu, et celles pointant dans les directions E, SE, S et SW sont coloriées en rouge. Bitz et Kung montrent que le nombre de passes nécessaire pour que chaque point reçoive sa valeur finale est C ou $C+1$, où C est le nombre maximal de changements de couleur sur un plus court chemin de la source à un autre point quelconque. En conséquence, dans le pire cas, le nombre de passes nécessaires pourra être quadratique, i.e. en $O(n^2)$. Cependant, en pratique, ce nombre sera largement inférieur à n [15].

Dans toute la suite, on étudie la parallélisation d'une seule passe (une passe avant) sur un anneau orienté de processeurs.

5.4.2 Implémentation parallèle

On considère un anneau de p processeurs numérotés de 0 à $p-1$. Nous allouons des lignes de l'image aux processeurs, tout le problème étant de trouver une distribution qui équilibre la charge de calculs entre les processeurs, sans dégrader les performances par des temps de communication trop importants.

Algorithme glouton

Une première idée est d'utiliser un schéma de calcul "glouton" où les processeurs transmettent au plus tôt à leurs voisins les pixels calculés : cet algorithme minimise le temps de mise en route et conduit très rapidement à des calculs bien équilibrés entre les processeurs.

Supposons tout d'abord que le nombre de processeurs p, est égal a la taille du problème n. Dans ce cas, la ligne i de l'image, est allouée au processeur i, avec $0 \leq i < n$. Pour le balayage avant, dès que le processeur i a calculé la valeur d'un pixel, il la transmet au processeur $i+1$, et calcule le prochain pixel de sa ligne. Notons bien que pour commencer une ligne, un processeur a besoin de deux valeurs de la ligne précédente, et ne peut donc commencer le calcul de sa ligne que deux étapes après son voisin. Nous résumons en Figure 5.9 les étapes durant lesquelles chaque pixel est calculé.

$$
\begin{array}{l}
P_1 \ \boxed{0}\,\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}\,\boxed{5}\,\boxed{6}\,\boxed{7}\,\boxed{8}\,\boxed{9}\,\cdots \\
P_2 \ \boxed{2}\,\boxed{3}\,\boxed{4}\,\boxed{5}\,\boxed{6}\,\boxed{7}\,\boxed{8}\,\boxed{9}\,\cdots \\
P_3 \ \boxed{4}\,\boxed{5}\,\boxed{6}\,\boxed{7}\,\boxed{8}\,\boxed{9}\,\cdots \\
P_4 \ \boxed{6}\,\boxed{7}\,\boxed{8}\,\boxed{9}\,\cdots \\
P_5 \ \boxed{8}\,\boxed{9}\,\cdots \\
\cdots\ \cdots
\end{array}
$$

FIG. 5.9 – Etapes de calcul de l'algorithme glouton.

A l'instant $2i+j$, le processeur i effectue donc les opérations suivantes (partout où les indices ont un sens) :

- Il reçoit le pixel $(i-1, j+1)$ du processeur $i-1$
- Il calcule le pixel (i, j)
- Il transmet (i, j) au processeur $i+1$

Quand p est inférieur à n, des techniques de partitionnement doivent être mises en oeuvre. Supposons pour la clarté de l'exposé, et sans perte de généralité, que p divise n. Afin de permettre aux processeurs de débuter leurs calculs le plus tôt possible, une solution consiste à allouer les lignes de l'image aux processeurs de manière entrelacée : la ligne j est allouée au processeur j modulo p. La répartition entrelacée est une technique classique permettant de bien répartir la charge entre les processeurs.

Notons P_i le processeur i. Avec cette répartition, P_0 a besoin de valeurs calculées par P_{p-1}. Notons que P_0 reçoit le pixel $(p-1, 0)$ au temps $2p-1$; au temps $2p$, il reçoit le deuxième pixel $(p-1, 1)$ de la ligne $p-1$, et calcule le pixel $(p, 0)$. Donc, il ne faut pas que P_0 ait fini de calculer sa première ligne avant l'instant $2p$, sinon, il serait inactif, en attente des valeurs transmises par P_{p-1}. Cela entraîne que $n \geq 2p$. Lorsque $n > 2p$, P_0 stockera simplement les pixels calculés par P_{p-1}, en finissant sa première ligne, puis utilisera les valeurs stockées pour calculer sa deuxième ligne.

Nous voyons que le délai entre deux processeurs voisins est petit (deux unités de temps). L'inconvénient majeur de l'algorithme est qu'il nécessite de nombreuses communications de petite taille. On a vu que le temps de transfert de L mots entre deux processeurs voisins, peut être modélisé par $\beta + L\tau$. Or β, le temps d'initialisation de la communication, indépendant de sa longueur, est souvent un ordre de grandeur supérieur à τ, le temps de communication élémentaire, ce qui rend prohibitif le coût des messages courts.

Dans la suite, nous expliquons comment modifier l'algorithme glouton de manière à diminuer le coût des communications. Nous commençons par une description informelle du nouvel algorithme, et reportons l'analyse de sa complexité au paragraphe suivant.

Augmenter la granularité de l'algorithme

La première technique utilisée pour diminuer le coût global des communications, est d'utiliser de plus longs messages. Nous utilisons la même allocation des données que précédemment, mais nous calculons k pixels consécutifs à chaque étape. L'algorithme est illustré Figure 5.10. Notons que k n'est pas forcément un diviseur de n. Dans la Figure 5.10 nous désignons par l_0 le nombre de pixels calculés par P_0 à l'étape 0 : on peut par exemple choisir $l_0 = k - 1$, comme si P_0 venait de recevoir k valeurs fictives avant de commencer, mais l_0 peut être choisi différemment (comme c'est le cas dans la Figure 5.10). A chaque étape, sauf la première et éventuellement la dernière, chaque processeur calcule un segment de k pixels consécutifs. Lorsque le segment déborde de la frontière d'une ligne, on termine la ligne courante et on débute la ligne suivante au cours de la même étape. Considérons l'exemple de la Figure 5.10 : à la quatrième étape, P_0 calcule les $k-1$ derniers pixels de sa première ligne, et le premier de sa deuxième ligne (en fait, la cinquième ligne de l'image). Notons que P_0 a pu débuter le calcul de sa deuxième ligne à cette étape grâce au fait que P_3 avait envoyé à P_0 à la fin de l'étape précédente, les deux premiers pixels de sa première ligne. La condition pour que P_0 n'achève pas le calcul de sa première ligne avant de recevoir des données de P_{p-1} sera établie plus loin : on obtient $n \geq (k+1)p$.

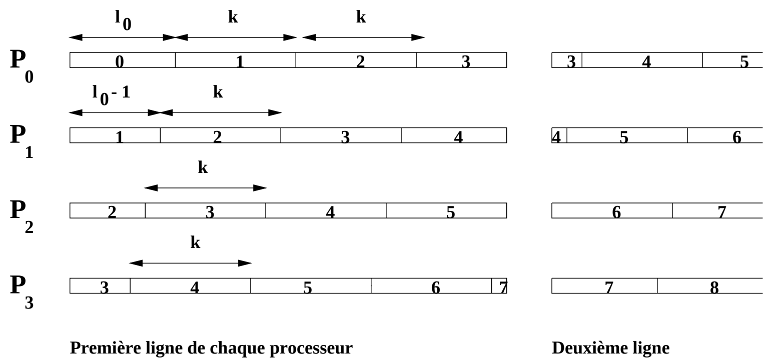


FIG. 5.10 – Mise a jour d'un segment de longueur k avec $p = 4$ processeurs.

Le nombre de pixels communiqués entre deux voisins est exactement le même que précédemment, mais plus k est grand, moins les communications sont coûteuses. D'un autre côté, plus k est grand, plus le délai d'attente entre deux processeurs adjacents est grand. Il nous faut donc trouver un compromis entre deux exigences contradictoires, à savoir un délai d'initialisation minimal (petit k), et des communications peu coûteuses (grand k).

D'autres stratégies d'allocation

Un autre moyen de diminuer le coût des communications est de diminuer leur volume total, en échangeant moins d'informations entre processeurs voisins. On considère des fonctions d'allocation plus générales que l'allocation entrelacée des lignes, en attribuant à un processeur, des blocs de r lignes consécutives, et en entrelaçant les blocs autour de l'anneau. Par exemple, avec $r = 3$, $n = 44$ et $p = 4$, on a la répartition suivante des lignes :

P_0	P_1	P_2	P_3
$0, 1, 2$	$3, 4, 5$	$6, 7, 8$	$9, 10, 11$
$12, 13, 14$	$15, 16, 17$	$18, 19, 20$	$21, 22, 23$
$24, 25, 26$	$27, 28, 29$	$30, 31, 32$	$33, 34, 35$
$36, 37, 38$	$39, 40, 41$	$42, 43, 44$	

De manière analytique, le processeur i contient les lignes j telles que $i = \lfloor \frac{j}{r} \rfloor$ modulo p, $0 \leq j \leq n-1$. On reviendra sur cette technique d'allocation cyclique par blocs au Paragraphe 6.5.

Les étapes de l'algorithme sont illustrées Figure 5.11. A chaque étape, exceptées la première et la dernière, chaque processeur calcule $r \times k$ pixels. De même que précédemment, avec $r = 1$, on calcule le début d'un bloc à la même étape que la fin du bloc précédent. La condition que doivent vérifier k et r pour que les processeurs restent actifs au cours de l'algorithme s'exprime maintenant par : $n \geq p(r + k)$ (voir paragraphe suivant). Nous montrons un exemple à la Figure 5.12 où cette condition n'est pas vérifiée : on voit que P_0 est inactif à l'étape 4.

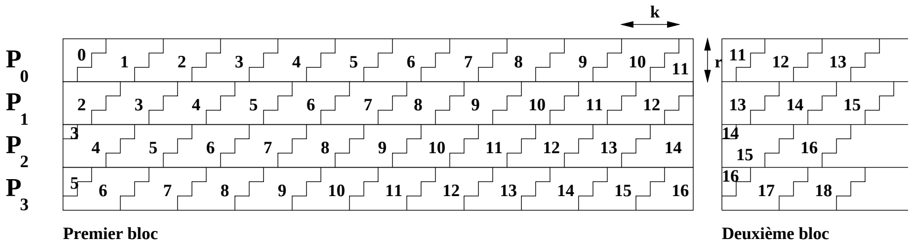


FIG. 5.11 – L'algorithme parallèle, avec $n = 44$, $p = 4$, $r = 3$ et $k = 4$.

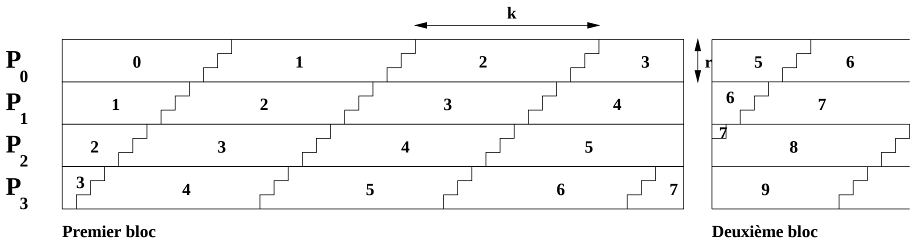


FIG. 5.12 – L'algorithme parallèle, avec $n = 44$, $p = 4$, $r = 3$ et $k = 13$.

Avec cette répartition des données, le volume total des communications entre deux processeurs voisins est r fois plus petit qu'avec la stratégie précédente, car les processeurs n'ont besoin d'échanger que les informations relatives aux lignes frontières entre blocs. Les segments appartenant aux lignes internes d'un bloc ne requièrent pas de communications inter-processeurs. Le mécanisme de communication entre deux processeurs voisins est illustré Figure 5.13.

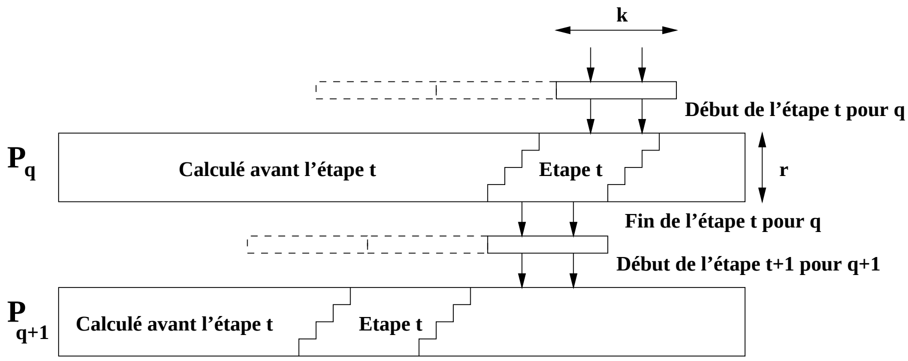


FIG. 5.13 – Communications entre deux processeurs voisins.

Le prix à payer pour cette importante réduction du volume de communication, est encore l'augmentation du délai d'initialisation entre deux processeurs voisins. A nouveau, la meilleure valeur de r résultera d'un compromis, tout comme la meilleure valeur de k.

5.4.3 Analyse de complexité

Nous analysons la complexité de l'algorithme parallèle décrit ci-dessus : nous déterminons de manière analytique en fonction de n et p, les valeurs de k et de r qui minimisent le temps d'exécution. Sans perte de généralité, on supposera n divisible par le produit $p \times r$. Soit τ_a le temps arithmétique élémentaire nécessaire au calcul d'un pixel, par une application de la formule (PAV) ou (PAR). Puisqu'il y a n^2 pixels à calculer, le temps séquentiel d'un balayage est $T_{seq} = n^2\tau_a$.

En ce qui concerne la taille mémoire : la place mémoire nécessaire à l'exécution de l'algorithme est proportionnelle à la taille de l'image, c'est à dire n^2 pixels. Pour chacun d'eux, il faut stocker un mot pour sa valeur : un entier pour la transformation de distance ou un réel pour la trajectoire optimale. Dans ce dernier cas, il faut aussi mémoriser la carte de traversabilité, ce qui représente n^2 réels. La place mémoire nécessaire est donc de l'ordre de n^2. Etant donné un processeur de taille mémoire M (mesurée en nombre d'informations stockées par pixel de l'image), le problème de taille maximale pouvant être traité est $n_{max,1} = \sqrt{M}$. Considérons maintenant un anneau de p processeurs : on dispose de p mémoires de taille M, on peut donc traiter un problème de taille $n_{max,p} = \sqrt{pM}$. Remarquons que nous avons négligé dans cette analyse, tout stockage supplémentaire nécessité par l'implémentation parallèle, tels que les tampons de communication. En fait, la valeur de $n_{max,p}$ est une borne supérieure. Comme décrit plus haut, nous considérons une allocation par blocs de r lignes consécutives, de manière entrelacée, avec $1 \leq r \leq n/p$. Pour la simplicité de l'exposé, nous supposerons que $p \times r$ divise n, pour que chaque processeur contienne le même nombre de lignes dans sa mémoire.

Bien que l'implémentation soit asynchrone, on peut voir l'algorithme parallèle comme une succession de phases de calcul, séparées par des communications : à chaque étape, un processeur reçoit un segment de longueur k de son prédécesseur dans l'anneau, calcule r segments de k pixels, puis transmet le dernier d'entre eux à son successeur. Notons qu'on suppose la réception d'un message est bloquante, alors que son émission ne l'est pas. L'émission d'une étape se fait donc en même temps que la réception de l'étape suivante. Le coût des communications au cours d'une étape est de $\beta + k\tau_c$. En conséquence, le temps d'exécution d'une étape est $\tau_{step} = \beta + k\tau_c + rk\tau_a$.

Pour évaluer le nombre total d'étapes dans l'algorithme, nous calculons tout d'abord le numéro d'étape t_q à laquelle le processeur P_q, $0 \leq q \leq p-1$, effectue son premier calcul. A l'instant $t_0 = 0$, le processeur P_0 calcule l_0 pixels de la première ligne de l'image. On voit que P_1 calcule $l_1 = (l_0 - r)$ modulo k pixels au temps $t_1 = 1 + \lfloor \frac{r-l_0}{k} \rfloor$, et que plus généralement, P_q calcule les l_q premiers pixels de sa ligne à l'étape t_q avec

$$l_q = (l_0 - q * r) \mathrm{mod} k$$

$$t_q = q + \lfloor \frac{qr - l_0}{k} \rfloor$$

Il est maintenant facile de dénombrer le nombre total d'étapes T_p de l'algorithme, puisque P_{p-1} est le processeur qui achève ses calculs après tous les autres. Après le calcul de son premier "parallélogramme", il lui reste $\lfloor (n^2/(p * r) - l_q + r - 1)/k \rfloor$ parallélogrammes à calculer, de telle sorte que

$$Tp = t_{p-1} + \lfloor (n^2/(p * r) - l_q + r - 1)/k \rfloor$$

Le temps d'exécution de l'algorithme est donc $T_{//} = \tau_{step} T_p$.

Cette évaluation n'est pas valide si les processeurs sont mis en attente par des données n'arrivant pas à temps de leur prédécesseur. Comme expliqué plus haut, cette condition revient à s'assurer que P_0 n'a pas fini de calculer son premier bloc de r lignes avant de recevoir de P_{p-1} les données nécessaires au calcul de son deuxième bloc. P_0 effectue sa première réception au temps t_p. A cet instant, il a déjà calculé $l_0 + k(t_p - 1)$ pixels dans la première ligne de l'image (l_0 au top 0 et k par étape aux tops suivants). La condition cherchée est que la somme du nombre de pixels qui lui restent à calculer dans cette ligne plus ceux qu'il peut calculer dans la première ligne de son deuxième bloc, à l'aide des informations transmises par P_{p-1}, soit supérieure ou égale à k. Ainsi, P_0 pourra calculer un parallélogramme complet à l'étape t_p. Cette condition s'exprime par $n - (l_0 + k(t_p - 1)) + l_p \geq k$. Ce qui se réécrit :

$$n \geq p(r + k)$$

On retrouve la condition illustrée aux Figures 5.11 et 5.12.

En négligeant les termes du second ordre, et les parties entières, on obtient l'expression du temps d'exécution $T_{//}$:

Proposition 12 *Etant donnés un problème de taille n, et un anneau de p processeurs, le temps d'exécution parallèle d'un balayage de l'image avec une allocation des lignes aux processeurs par blocs de taille r $1 \leq r \leq \frac{n}{p}$, et utilisant des segments de taille k, $1 \leq k \leq \frac{n}{p} - r$), est :*

$$T_{//} = (\beta + k\tau_c + rk\tau_a)\left((p-1)(1 + \frac{r}{k}) + \frac{n^2}{prk}\right).$$

Etant donnés n, p et r, il est aisé de déterminer la valeur $k_{opt}(r)$ de k, qui minimise le temps d'exécution $T_{//}$. On obtient la valeur $k_{opt}(r) = \min(k_{max}(r), k_{//}(r))$, où $k_{max}(r) = \frac{n}{p} - r$ et $k_{//}$ est la valeur optimale obtenue à partir de l'expression de $T_{//}$:

$$k//(r) = \sqrt{\frac{\beta}{\tau_c + r\tau_a}\left(\frac{n^2}{p(p-1)r} + r\right)}.$$

Finalement, étant donnés n, p et les valeurs numériques des paramètres β, τ_c et τ_a, on calcule $k_{opt}(r)$ que l'on reporte dans l'expression de $T_{//}$ pour déterminer la meilleure valeur de r. Le lecteur intéressé trouvera des résultats expérimentaux dans [52, 53].

5.5 Factorisation LU

Nous terminons ce chapitre par la résolution d'un système linéaire dense $Ax = b$ par la méthode de triangularisation de Gauss. Nous faisons deux hypothèses restrictives. D'une part nous décrivons l'algorithme de décomposition LU de A sans pivotage, ce qui n'est pas très important : puisque les colonnes de A vont être distribuées aux processeurs, introduire le pivotage partiel n'induit aucune communication supplémentaire. D'autre part nous présentons l'algorithme classique où les colonnes de A sont éliminées tour à tour, l'une après l'autre. Il se trouve que l'implémentation de cet algorithme point-à-point n'est absolument par réaliste sur une machine moderne! La bonne utilisation de la hiérarchie mémoire de chaque processeur (réutilisation des registres et du cache) impose une version de l'algorithme par blocs. Il n'y a pas de différence conceptuelle à opérer sur des blocs de colonnes que sur de simples colonnes, et le schéma algorithmique global reste le même. Mais question détails de bas niveau et programmation concrète, c'est une autre histoire : on passe d'un algorithme d'une dizaine de lignes à un programme de plusieurs pages, et ce même sans parallélisme : voir par exemple la version séquentielle de l'algorithme de Gauss qui figure dans la bibliothèque (du domaine public) LAPACK [7].

5.5.1 Première version

Voila la version de base de l'algorithme séquentiel, pour une matrice de taille $n \times n$ indicée de 0 à $n-1$:

```
for k = 0 to n − 2 do
  /* Tâche T_kk : préparation de la colonne k
  prep(k) : for i = k + 1 to n − 1 do a_ik = −a_ik/a_kk
  for j = k + 1 to n − 1 do
    /* Tâche T_kj : mise à jour de la colonne j
    update(k,j) : for i = k + 1 to n − 1 do a_ij = a_ij + a_ik ∗ a_kj
```

On le voit, rajouter le pivotage dans les procédures prep et update ne serait pas difficile. Comment paralléliser cet algorithme sur un anneau de p processeurs ? puisque l'algorithme opère par colonnes, il est naturel de distribuer celles-ci aux processeurs. Supposons pour l'instant qu' une fonction d'allocation alloc des colonnes aux processeurs est donnée. A l'étape k, le processeur qui possède la colonne k la diffuse à tous les autres, qui peuvent alors mettre à jour leurs propres colonnes. En utilisant la routine broadcast du Paragraphe 5.2, on obtient :

```
q ← my_num() ;
p ← tot_proc_num() ;
for k = 0 to n − 2 do
  if alloc(k) = q then
    prep(k) : for i = k + 1 to n − 1 do buffer_{i−k−1} = −a_ik/a_kk
  broadcast(alloc(k), buffer, n-k)    for j = k + 1 to n − 1 do
    update(k,j) : for i = k + 1 to n − 1 do a_ij = a_ij + buffer_{i−k} ∗ a_kj
```

Reste à gérer convenablement les indices de la matrice A, dont les colonnes sont distribuées aux processeurs : si $r = n/p$, chaque processeur va stocker r colonnes de A dans sa mémoire locale. Au fil de l'élimination, certaines colonnes ne vont plus être accédées : après l'étape k, seules les colonnes d'indice global supérieur à k vont être mises à jour. L'idée est d'utiliser un indice local l qui va pointer sur la première colonne à accéder : au départ, tous les processeurs ont $l = 1$. A l'étape $k = 1$, le processeur $alloc(1)$ va préparer la colonne 1 par l'appel à la routine prep(1). Son indice local l devra alors passer à deux, pour la mise à jour, tandis que pour les autres processeurs l sera inchangé. A l'étape $k = 2$, le processeur $alloc(2)$ incrémentera l après avoir appelé la routine prep(2). On obtient ainsi la routine complète :

```
var a : array[0..n-1,0..r-1] of real
...
q ← my_num() ;
p ← tot_proc_num() ;
l = 0
for k = 0 to n − 2 do
  if alloc(k) = q then
    for i = k + 1 to n − 1 do buffer[i − k − 1] = −a[i, l]/a[k, l]
    l = l + 1
  broadcast(alloc(k), buffer, n-k)    for j = l to r − 1 do
    update(k,j) : for i = k + 1 to n do a[i, j] = a[i, j] + buffer_{i−k} ∗ a[k, j]
```

Cette fois nous avons utilisé des vrais tableaux indicés, en incluant la déclaration de A en local, pour bien montrer que les données sont distribuées. La seule chose qui reste à spécifier est précisément cette distribution. Le schéma de l'algorithme est plus complexe que celui du produit matrice-vecteur ou de la transformée en distance, et ceci pour deux raisons :

- D'une part, le nombre de données à traiter varie au cours de l'exécution. Le coeur de l'algorithme est la mise à jour des colonnes $k+1$ à j à chaque étape k. On le voit, il y a de moins en moins de colonnes au fil de l'exécution, quand k grandit.
- D'autre part, le volume des calculs n'est plus proportionnel au nombre de données. En effet la dernière colonne est mise à jour $n-1$ fois, et donc représente pour le processeur à qui elle est allouée un nombre total d'opérations arithmétiques à effectuer bien plus important que la colonne, disons, $n/2$, laquelle représente à son tour un volume de calculs bien plus important que pour la colonne 1.

Il faut donc trouver une allocation qui équilibre à la fois l'occupation mémoire (chaque processeur a le même nombre de colonnes) et la charge de travail. Pour ce dernier point, c'est plus subtil : il faut équilibrer la charge à chaque étape de l'algorithme, et non pas simplement la charge globale.

Pas de mystère : une allocation cyclique va faire l'affaire. Le processeur $p-1$ aura bien un peu plus de travail que le processeur 0, mais on peut vérifier que la différence est asymptotiquement négligeable. Supposons en effet la colonne j allouée au processeur j modulo p : celle-ci est mise à jour pour toutes les étapes allant de $k=0$ à $k=j-1$, pour un coût $n-k-1$ à l'étape k (éléments en position $k+1$ à $n-1$). On obtient pour la somme des mises à jour de la colonne j la valeur

$$\sum_{k=0}^{j-1}(n-k-1)\tau_a.$$

Pour obtenir une procédure complètement déterminée, il suffira donc de remplacer le test if $\mathsf{alloc(k) = q}$ par if $\mathsf{k = q}$ modulo p. Notons $p(k, P_q)$ et $u(k, j, P_q)$ l'exécution de la procédure PREP(K) et de la procédure UPDATE(K,J) sur le processeur P_q. Le temps d'exécution de la procédure est celui du chemin critique, à savoir

$$p(0, P_0) \rightarrow u(0, 1, P_1), p(1, P_1) \rightarrow u(1, 2, P_2), p(2, P_2) \rightarrow \dots$$

Ici, chaque flèche $\rightarrow$ symbolise une communication entre processeurs voisins. En décomposant les calculs, on trouve

- un terme en $n\beta + \frac{n^2}{2}\tau_c + o(1)$ pour les $n-1$ communications
- un terme en $\frac{n^2}{2}\tau_a + o(1)$ pour les préparations de pivot
- et le terme dominant en $\frac{n^3}{3p}\tau_a + O(n^2)$ pour les mises à jour

Si p reste petit devant n, la parallélisation est relativement efficace.

5.5.2 Pipeline sur l'anneau

La solution précédente fait appel à une procédure de diffusion abstraite. Rien n'indique en fait que l'algorithme s'exécute sur un anneau : il peut s'exécuter sur n'importe quelle topologie où l'on dispose d'une routine de diffusion, que ce soit un hypercube ou une ferme de processeurs ; cette portabilité a un prix : les $n-1$ étapes de communication (les $n-1$ diffusions) ne sont pas recouvertes par les calculs. Au contraire, sur un anneau et avec une allocation cyclique, on peut entrelacer les diffusions et les calculs : il suffit presque d'insérer le texte source de la procédure de diffusion au milieu de l'algorithme :

```
var a : array[0..n-1,0..r-1] of real
...
q ← my_num() ;
p ← tot_proc_num() ;
l = 0
for k = 0 to n − 2 do
  if k = q modulo p then
    for i = k + 1 to n − 1 do buffer[i − k − 1] = −a[i, l]/a[k, l]
    l = l + 1
    send(buffer, n-k)
  else
    receive(buffer, n-k)
    if (q ≠ k-1 modulo p) then send(buffer, n-k)
  for j = l to r − 1 do
    update(k,j) : for i = k + 1 to n do a[i, j] = a[i, j] + buffer_{i−k} ∗ a[k, j]
```

Dans cette version, dès qu'un processeur reçoit la colonne pivot à l'étape k (i.e. le buffer de communication), il la transmet à son successeur. Il fait cette communication avant de commencer ses mises à jour. La Figure 5.14 illustre le mécanisme avec l'hypothèse pessimiste, celle où aucune communication ne peut être recouverte avec des calculs indépendants sur un même processeur. Même avec cette hypothèse restrictive, on voit que globalement certaines communications ont lieu en parallèle aux calculs : plus précisément, la colonne pivot circule le long de l'anneau, et les premiers processeurs commencent leurs calculs alors que les derniers reçoivent encore le pivot. Le processeur q est donc en avance sur le processeur $q + 1$, ce qui permet de pipeliner les étapes de manière emboîtée, comme le montre la figure. Bien sûr, toutes les p étapes, une bulle d'inactivité se crée quand c'est au tour du processeur le plus en retard de devenir pivot.

5.5.3 Algorithme *look-ahead*

Comment améliorer l'algorithme précédent ? en entrelaçant les étapes, et ce d'une manière très astucieuse. Pour voir comment, reprenons le fil de l'algorithme et observons le processeur P_1 :

1. A l'étape $k = 0$, le processeur P_1 reçoit la colonne pivot 0 de P_0
2. Il se dépêche de la faire passer à son successeur
3. Puis il met à jour ses colonnes avec les appels UPDATE(0,J) pour tous les j en sa possession : $j = 1$ modulo p
4. Il passe alors à l'étape $k = 1$ et prépare le pivot en appelant PREP(1)
5. Il envoie la colonne pivot 1 à P_2
6. Il met à jour ses colonnes pour l'étape avec les UPDATE(1,J)

Où est l'erreur de timing ? pourquoi P_1, alors qu'il va être le pivot à la prochaine étape, exécute-t-il tous les appels UPDATE(0,J) au lieu d'exécuter le seul appel UPDATE(0,1), puis PREP(1), puis l'envoi, avant de revenir terminer tranquillement les UPDATE(0,J) pour $j = p+1, 2p+1, 3p+1, \ldots$. Ces deux stratégies sont équivalentes pour le monde extérieur, à un détail près : le pivot de la prochaine étape circule plus tôt, ce qui évite les bulles d'inactivité.

Imaginons une machine moderne dotée d'un système exécutif à base de processus légers, ou *threads*. Chaque processeur exécute deux processus légers, un de calcul, et un de communication qui fait circuler, de manière totalement asynchrone et non bloquante, la colonne de l'étape suivante

P_0	P_1	P_2	P_3
p(0)			
send(0)	receive(0)		
u(0,4)	send(0)	receive(0)	
u(0,8)	u(0,1)	send(0)	receive(0)
u(0,12)	u(0,5)	u(0,2)	u(0,3)
	u(0,9)	u(0,6)	u(0,7)
	u(0,13)	u(0,10)	u(0,11)
	p(1)	u(0,14)	u(0,15)
	send(1)	receive(1)	
	u(1,5)	send(1)	receive(1)
receive(1)	u(1,9)	u(1,2)	send(1)
u(1,4)	u(1,13)	u(1,6)	u(1,3)
u(1,8)		u(1,10)	u(1,7)
u(1,12)		u(1,14)	u(1,11)
		p(2)	u(1,15)
		send(2)	receive(2)
receive(2)		u(2,6)	send(2)
send(2)	receive(2)	u(2,10)	u(2,3)
u(2,4)	u(2,5)	u(2,14)	u(2,7)
u(2,8)	u(2,9)		u(2,11)
u(2,12)	u(2,13)		u(2,15)
			p(3)
receive(3)			send(3)
send(3)	receive(3)		u(3,7)
u(3,4)	send(3)	receive(3)	u(3,11)
u(3,8)	u(3,5)	u(3,6)	u(3,15)
u(3,12)	u(3,9)	u(3,10)	
	u(3,13)	u(3,14)	

FIG. 5.14 – Les quatre premières étapes de l'algorithme LU pipeline sur l'anneau.

pendant les mises à jour de l'étape courante. Hormis le calcul du premier pivot et sa diffusion, l'algorithme est alors totalement parallèle, et son efficacité tend vers 1.

Nous reprenons la Figure 5.14 avec cette fois l'hypothèse optimiste que les processeurs peuvent simultanément communiquer et calculer sur des données indépendantes, en utilisant des processus légers distincts. Nous obtenons la Figure 5.15, où il subsiste des périodes d'inactivité. Par contraste, la Figure 5.16 montre bien l'efficacité de la version lookahead. Quelle beauté algorithmique !

Voici enfin l'algorithme lookahead schématisé :

P_0	P_1	P_2	P_3
p(0)			
send(0) // u(0,4)	recv(0)		
u(0,8)	send(0) // u(0,1)	recv(0)	
u(0,12)	u(0,5)	send(0) // u(0,2)	recv(0)
	u(0,9)	u(0,6)	u(0,3)
	u(0,13)	u(0,10)	u(0,7)
	p(1)	u(0,14)	u(0,11)
	send(1) // u(1,5)	recv(1)	u(0,15)
	u(1,9)	send(1) // u(1,2)	recv(1)
recv(1)	u(1,13)	u(1,6)	send(1) // u(1,3)
u(1,4)		u(1,10)	u(1,7)
u(1,8)		u(1,14)	u(1,11)
u(1,12)		p(2)	u(1,15)
		send(2) // u(2,6)	recv(2)
recv(2)		u(2,10)	send(2) // u(2,3)
send(2) // u(2,4)	recv(2)	u(2,14)	u(2,7)
u(2,8)	u(2,5)		u(2,11)
u(2,12)	u(2,9)		u(2,15)
			p(3)
recv(3)	u(2,13)		send(3) // u(3,7)
send(3) // u(3,4)	recv(3)		u(3,11)
u(3,8)	send(3) // u(3,5)	recv(3)	u(3,15)
u(3,12)	u(3,9)	u(3,6)	
p(4)	u(3,13)	u(3,10)	
send(4) // u(4,8)	recv(4)	u(3,14)	

FIG. 5.15 – LU pipeline, avec processus légers pour le recouvrement calcul/communication.

```
q ← my_num() ;
p ← tot_proc_num() ;
for k = 0 to n − 2 do
  if k = q modulo p then
    prep(k)
    send(buffer, n-k)
    /* finir la mise à jour de l'étape précédente
    for all j=k modulo p, j¿k, do apply(k-1,j)
    /* mettre à jour pour l'étape courante
    for all j=k modulo p, j¿k, do apply(k,j)
  else
    receive(buffer, n-k)
    if (q ≠ k-1 modulo p) then send(buffer, n-k)
    if k+1 = q modulo p then
      /* pivot pour la prochaine étape, mise à jour d'une seule colonne
      update(k,k+1)
    else
      /* mise à jour standard pour l'étape
      for all j=k modulo p, j¿k, do apply(k,j)
```

P_0	P_1	P_2	P_3
p(0)			
send(0) // u(0,4)	recv(0)		
u(0,8)	send(0) // u(0,1)	recv(0)	
u(0,12)	p(1)	send(0) // u(0,2)	recv(0)
	send(1) // u(0,5)	recv(1) // u(0,6)	u(0,3)
	u(0,9)	send(1) // u(0,10)	recv(1) // u(0,7)
recv(1)	u(0,13)	u(0,14)	send(1) // u(0,11)
u(1,4)	u(1,5)	u(1,2)	u(0,15)
u(1,8)	u(1,9)	p(2)	u(1,3)
u(1,12)	u(1,13)	send(2) // u(1,6)	recv(2) // u(1,7)
recv(2)		u(1,10)	send(2) // u(1,11)
send(2) // u(2,4)	recv(2)	u(1,14)	u(1,15)
u(2,8)	u(2,5)	u(2,6)	u(2,3)
u(2,12)	u(2,9)	u(2,10)	p(3)
recv(3)	u(2,13)	u(2,14)	send(3) // u(2,7)
send(3) // u(3,4)	recv(3)		u(2,11)
p(4)	send(3) // u(3,5)	recv(3)	u(2,15)
send(4) // u(3,8)	recv(4) // u(3,9)	u(3,6)	u(3,7)
u(3,12)	send(4) // u(3,13)	recv(4) // u(3,10)	u(3,11)
u(4,8)	u(4,5)	send(4) // u(3,14)	recv(4) // u(3,15)

FIG. 5.16 – LU lookahead, avec processus légers pour le recouvrement calcul/communication.

Notes bibliographiques

Une référence incontournable, le livre de Kumar et ses étudiants [46]. Ceci dit, les paragraphe sur les macro-communications, et le produit matrice-vecteur, font partie du folklore. L'étude de cas de la transformée de distance s'inspire directement des articles de Miguet et Robert [52, 53]. L'étude de l'algorithme de Gauss est classique, voir par exemple le livre [60] et les références citées.

Chapitre 6

Communications et routage

6.1 Introduction

Ce chapitre se veut un approfondissement du Chapitre 5 qui était consacré aux anneaux de processeurs. Il débute (Paragraphe 6.2) par la description de quelques réseaux d'interconnexion classiques, et des mécanismes de routage les plus usuels (Paragraphe 6.3). Nous ne traitons pas le sujet dans toute sa généralité (voir le livre de Culler et Singh [27] pour cela). Au contraire, nous détaillons deux études de cas :

1. L'analyse des propriétés topologiques de l'hypercube : au Paragraphe 6.4, nous décrivons les méthodes de routage statique et dynamique qui peuvent être mises en oeuvre sur cette architecture bien plus riche que celle de l'anneau ; nous expliquons également les techniques de plongement d'une topologie dans une autre.
2. L'analyse précise du coût de plusieurs algorithmes pour la multiplication de deux matrices carrées (Paragraphe 6.5). Ces algorithmes sont destinés à s'exécuter sur des grilles bidimensionnelles toriques de processeurs.

6.2 Réseaux d'interconnexion

Ce paragraphe décrit brièvement quelques réseaux d'interconnexion classiques pour machines à mémoire distribuée. Il faut savoir qu'il existe également des machines multi-processeur à mémoire partagée ! Ces dernières sont plus simples à programmer, puisque les différents processeurs peuvent échanger des informations au travers de leurs lectures et écritures dans la mémoire commune. A contrario, sur une machine à mémoire distribuée, les processeurs doivent recourir à des échanges de messages pour communiquer, puisque la mémoire locale d'un processeur donné n'est pas directement accessible (adressable) par les autres.

Une petite mise en garde cependant : le terme “à mémoire distribuée” n'est pas défini formellement ici. On peut très bien construire un système de mémoire partagée virtuelle au dessus d'un ensemble de mémoires locales physiquement distribuées, et implanter ainsi l'accès global (virtuel) à toute les mémoires. En d'autres termes, une mémoire partagée peut être simulée à partir de mémoires physiquement distribuées. Réciproquement, pour des raisons d'efficacité et de performances, les mémoires partagées sont souvent implantées avec plusieurs niveaux de hiérarchie : ainsi elles peuvent être divisées en bancs dont l'accès privilégié est réservé à certains processeurs ; elles s'accompagnent également de mémoires caches privatives à chaque processeur (et dont le système gère la cohérence avec la mémoire partagée centrale). On le voit bien, l'opposition mémoire partagée–mémoire distribuée réside davantage dans le style de programmation : en règle générale, les machines à mémoire distribuée se programment en mode SPMD par échange de messages, tandis que celles à mémoire partagée se programment à l'aide d'un langage impératif séquentiel (à la Fortran) enrichi de primitives de parallélisation (boucles parallèles, sections critiques, etc).

6.2.1 Topologies

Les processeurs d'une machine parallèle à mémoire distribuée sont reliés entre eux par l'intermédiaire d'un réseau d'interconnexion, statique ou dynamique. La plupart des machines actuelles possèdent des processeurs dédiés aux communications : leur rôle est de permettre l'acheminement des messages vers les processeurs destination, et de ranger les données reçues dans les mémoires locales. Les noeuds des topologies présentées par la suite sont donc constitués d'un ou plusieurs processeurs de calcul, de mémoire, et d'un processeur de communication.

Les possibilités de connexion des processeurs entre eux sont nombreuses et un grand nombre de projets ont été initiés depuis les années 1970, tant dans la recherche que dans l'industrie, afin d'obtenir le meilleur moyen d'interconnecter les processeurs pour communiquer ensuite à moindre coût. On peut distinguer :

- **Les topologies statiques :** le réseau d'interconnexion est fixé par le constructeur et ne peut être modifié. C'est le cas de la plupart des machines parallèles de la décennie précédente, comme l'Intel Paragon.
- **Les topologies dynamiques :** la topologie peut être modifiée en cours d'exécution et un ou plusieurs processeurs peuvent demander l'établissement d'un lien pour une communication à un gestionnaire de connexions ; il s'agit de configurer un *switch*. Citons la série SP d'IBM et les cartes Myrinet comme exemples.

6.2.2 Quelques topologies statiques

Les topologies statiques les plus classiques sont schématisées à la Figure 6.1 : (a) le le réseau complet ; (b) l'anneau ; (c) la grille ; (d) le tore et (e) l'hypercube. On a également représenté en (f) une topologie dynamique en *fat-tree*, sur laquelle nous reviendrons dans un moment.

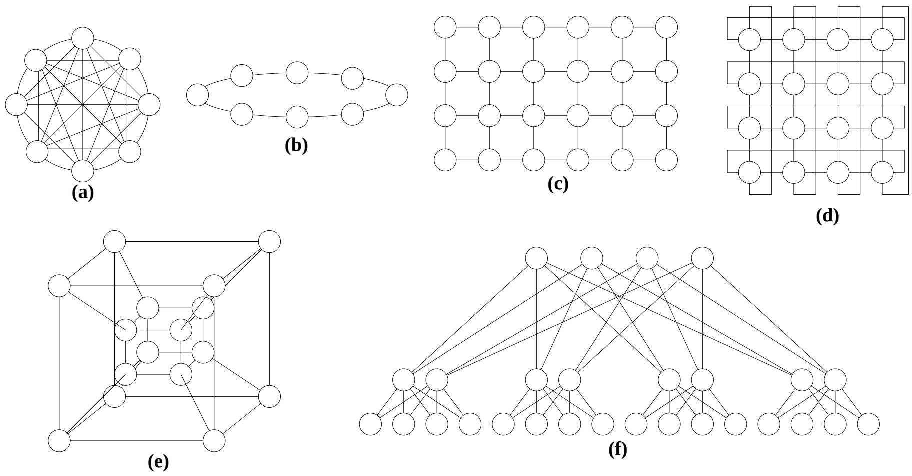


FIG. 6.1 – Différentes topologies de processeurs.

Les topologies statiques peuvent être décrites en terme de graphes où les sommets sont des processeurs et les arêtes les liens de communication. Les caractéristiques des réseaux sont données par :

Nombre de noeuds (processeurs p. C'est la caractéristique élémentaire d'une machine, car elle borne le degré de parallélisme utilisable.

Degré k. Nombre d'arêtes partant d'un noeud. Pour les topologies non-régulières, on peut distinguer le degré minimum δ et le degré maximum Δ. Mais par la suite on posera $k = \Delta$, ce qui correspond au nombre de liens du processeur de communication.

Diamètre D. La distance étant la longueur du plus court chemin entre deux noeuds, le diamètre est le maximum des distances.

Nombre de liens N_l. C'est le nombre de liens total de la topologie.

Largeur de bissection L_B. Minimum du nombre de liens nécessaires pour relier deux moitiés égales d'une topologie entre elles.

Notons que le nombre de processeurs p d'une topologie statique de degré Δ est bornée par

$$p = 1 + \Delta + \Delta(\Delta - 1) + \Delta(\Delta - 1)^2 + \ldots + \Delta(\Delta - 1)^{D-1}$$

C'est la borne de Moore, qui s'obtient en comptant les voisins d'un processeur, puis les voisins des voisins qui n'ont pas déjà été comptés, jusqu'aux processeurs les plus éloignés à distance D.

Voici une petite discussion sur les topologies de la Figure 6.1. Un résumé de leurs principales caractéristiques est donné dans la Table 6.1.

- **Le réseau complet** (Figure 6.1 (a)). C'est le réseau idéal. Tous les processeurs sont à une distance de 1. Par contre, le nombre k de liens par processeur est $p - 1$, soit un total de $p(p - 1)/2$. Ceci n'est réalisable que pour un nombre de processeurs très petit. Grâce aux techniques de routage, un réseau d'interconnexion quelconque peut toujours être vu comme un réseau complet mais le prix à payer est la présence de contentions lorsque deux messages veulent utiliser un même lien.
- **L'anneau** (Figure 6.1 (b)). C'est l'une des topologies les plus simples, et de nombreux algorithmes parallèles l'utilisent pour cette raison : voir le Chapitre 5.
- **La grille 2D** (Figure 6.1 (c)). Le degré maximum des processeurs est égal à 4. Un des défauts de la grille est son manque de symétrie. En effet, les processeurs sur les bords de la grille ont des caractéristiques différentes des processeurs centraux et nécessitent, de ce fait, des programmes particuliers. La grille 2D est parfaitement adaptée à des problèmes d'imagerie où les calculs se font voisins-à-voisins. Un autre avantage concerne sa faisabilité matérielle avec beaucoup de processeurs (scalabilité) : il suffit de rajouter des processeurs sur les bords.
- **Le tore 2D** (Figure 6.1 (d)). Facilement dérivé de la grille 2D en reliant les processeurs sur les bords entre eux, son diamètre est réduit de moitié. On peut encore augmenter la connectivité des processeurs en réalisant un tore 3D, comme pour le Cray T3D.
- **L'hypercube** (Figure 6.1 (e)). Il a été le roi de la décennie précédente, à cause de sa construction récursive qui permet de concevoir des algorithmes qui travaillent dimension par dimension. Autre avantage, son faible diamètre (logarithmique). Par contre, son nombre de liens croît avec le nombre de processeurs, ce qui ne lui permet pas d'être un bon candidat pour les machines massivement parallèles. On caractérise un hypercube par sa dimension d, où $p = 2^d$. Nous reviendrons sur l'hypercube au Paragraphe 6.4.

6.2.3 Un mot sur les topologies dynamiques

La topologie dynamique en *fat-tree* de la Figure 6.1 (e) est la topologie choisie par Thinking Machine Corporation pour équiper la CM-5 [48]. Il s'agit d'un arbre binaire dont la section (bande passante) augmente au fur et à mesure que l'on arrive vers la racine. Ce type d'arbre assure qu'il

Topologie	**Nbre de proc.** p	**Degré** k	**Diamètre** D	**Nbre de liens** N_l	**Larg. Bisec.** L_B
Réseau complet	p	$p-1$	1	$p(p-1)/2$	$(p/2)^2$
Anneau	p	2	$\lfloor p/2 \rfloor$	p	2
Grille 2D	$\sqrt{p}\sqrt{p}$	$2 \to 4$	$2(\sqrt{p}-1)$	$2p-2\sqrt{p}$	$\sqrt{p}$
Tore 2D	$\sqrt{p}\sqrt{p}$	4	$2\lfloor \sqrt{p}/2 \rfloor$	$2p$	$2\sqrt{p}$
Hypercube	$p=2^d$	$d=\log(p)$	d	$p\log(p)/2$	$p/2$

TAB. 6.1 – Principales caractéristiques de quelques topologies.

n'y aura pas de dégradation de la bande passante et que le nombre de processeurs de la machine pourra être augmenté sans que les performances des communications s'en trouvent diminuées. Au niveau des feuilles de l'arbre sont placés les processeurs eux-mêmes. Le réseau est redondant par nature et donc tolérant aux fautes.

Plus actuels, les réseaux de commutation, de type Benes ou Oméga, sont construits à l'aide de modules permettant de connecter des entrées à des sorties. Les différences résident principalement dans le nombre de modules et de fils nécessaires pour réaliser n'importe quelle topologie. Les réseaux construits à l'aide de tels modules sont appelés réseaux multi-étages. Leur coût est moins important que celui des crossbars : ceux-ci sont réseaux complets, mono-étage, mais difficiles à réaliser (voir la Figure 6.2) car ils nécessitent p^2 interrupteurs pour p processeurs.

En revanche le contrôle nécessaire au choix des chemins dans les réseaux multi-étage est plus difficile à mettre en oeuvre. La Figure 6.3 représente un réseau de Benes : pour connecter les p entrées aux p sorties, il utilise $2(\log_2 p - 1)$ étages composés chacun de $p/2$ crossbars de taille 2×2.

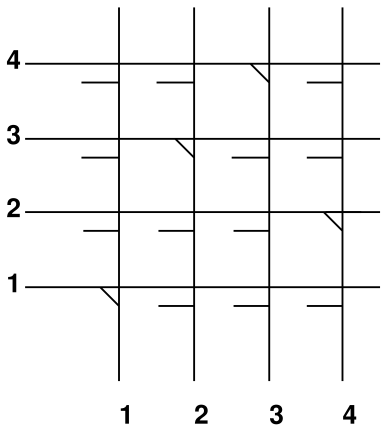


FIG. 6.2 – Réseau crossbar pour $p = 4$.

Les topologies dynamiques sont actuellement les plus utilisées, et de loin. Il y a dix ans, c'était le contraire. Pour une discussion très intéressante et prospective, on pourra consulter le livre de Culler et Singh [27].

6.3 Routage

Dans ce paragraphe, nous présentons différents modèles de routage pour topologies statiques. Nous supposons que deux processeurs x et y, non voisins, souhaitent s'échanger un message M de

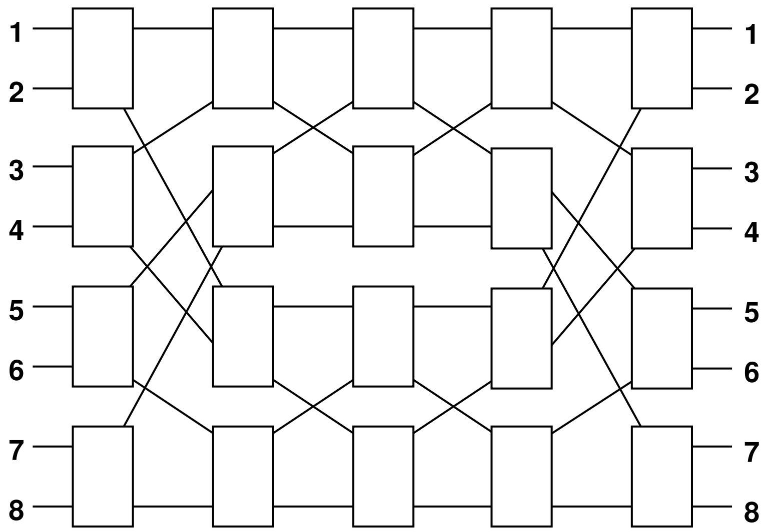


FIG. 6.3 – Réseau de Benes pour $p = 10$.

taille L.

6.3.1 Modèle "store-and-forward" (SF)

Dans ce protocole, chaque processeur intermédiaire sur le chemin de la communication reçoit et stocke le message M avant de le ré-émettre en direction du processeur destination. Ce modèle est également appelé modèle à commutation de message. Il correspond à la première génération de machines, sans co-processeur de communication. Les processeurs intermédiaires sont interrompus dans leurs tâches pour gérer (router) des communications qui ne les concernent pas.

Si un message de taille L est transmis d'un processeur x vers un processeur y distants de $d(x, y)$, alors la modélisation de ce protocole est donné par $d(x, y)(\beta + L\tau)$, où β est le temps d'initialisation (ou *startup time*) et τ le temps de propagation d'un octet. Notons que β et τ sont sujets à des variations plus ou moins importantes suivant que plusieurs liens sont utilisés en parallèle (modèle k-port si k liens sont utilisés en parallèle) ou bien que le lien est utilisé en mode unidirectionnel (modèle half-duplex) ou en mode bidirectionnel (modèle full-duplex).

Comme sur l'anneau, on peut utiliser la technique du pipeline étudiée au Paragraphe 5.2.4. Pour réduire le coût de la communication, on divise le message en r paquets de taille $\frac{L}{r}$. Les paquets sont envoyés les uns à la suite des autres à partir du processeur x vers le processeur y. Le premier paquet atteint le processeur y après $d(x, y)$ étapes de coût $\beta + \frac{L}{r}\tau$; puis les autres $r - 1$ morceaux arrivent immédiatement les uns derrière les autres, en $(r-1)(\beta + \frac{L}{r}\tau)$. Donc en tout un temps égal à

$$(d(x, y) - 1 + r)(\beta + \frac{L}{r}\tau)$$

On optimise la valeur de r pour obtenir $r_{opt} = \sqrt{\frac{L(d(x,y)-1)\tau}{\beta}}$ et le temps optimal est

$$(\sqrt{(d(x, y) - 1))\beta} + \sqrt{L\tau})^2 \; .$$

Le coût total est donc en $L\tau + o(\sqrt{L})$.

6.3.2 Modèle "cut-through" (CT)

Dans ce modèle, pour lequel on peut distinguer deux protocoles, le message acheminé n'a pas besoin d'arriver entièrement sur un noeud pour être renvoyé vers une autre destination. Si le nombre de processeurs sur le chemin entre la source et la destination est Q, alors le modèle communément utilisé est $\beta+(Q-1)\delta+L\tau$, où δ est le surcoût dû au calcul du chemin utilisé. En général $\delta \ll \beta$, car le chemin est calculé par le matériel, alors que β comprend une partie logicielle. Les deux protocoles sont les suivants :

- **Circuit-switching (CC)** Un chemin est créé avant émission des premiers octets constituant le message. Après création de ce chemin, le message est acheminé directement entre la source et la destination. Notons que les processeurs sur le chemin sont immobilisés au niveau des communications pendant toute la durée de l'échange. Ce modèle était utilisé pour l'iPSC d'Intel.
- **Wormhole (WH)** Pour ce mode d'acheminement, l'adresse du destinataire est placée dans l'en-tête du message. Le routage se fait sur chaque processeur, la sortie d'un processeur étant calculée par celui-ci à la lecture de l'adresse destination. Le message est découpé en petits paquets appelés *flits*. En cas de blocage, les flits sont stockés dans les registres internes des routeurs des processeurs intermédiaires. De nombreux algorithmes de routage existent, statiques ou dynamiques : voir le Paragraphe 6.4. Le modèle WH est utilisé pour le Paragon d'Intel.

6.3.3 Comparaison des différents modèles

A cause de la technologie et des connaissances en matière de routage de l'époque, le modèle à commutation de messages *store and forward* fut le premier utilisé sur les machines parallèles de type à "passage de messages" (*message passing*). Les modèles cut-through (CC et WH) sont plus efficaces dans ce sens qu'ils masquent la distance entre les processeurs communiquant de manière matérielle sans utilisation de buffers trop importants. Entre le CC *circuit switching* et le WH *wormhole*, l'avantage revient au WH qui construit sa route tandis que le message avance dans le réseau, au contraire du CC qui construit son chemin (et l'acquitte) avant l'envoi des premiers paquets de données.

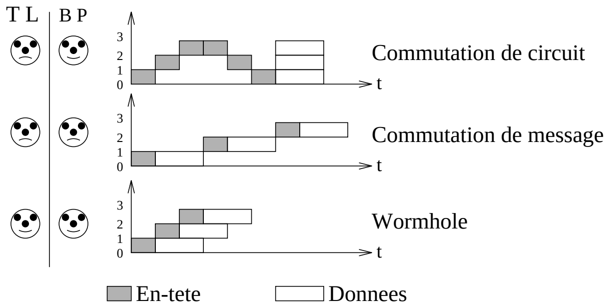


FIG. 6.4 – Différents modèles de communication.

La Figure 6.4 montre une comparaison des différents modèles (CM, CC et WH) sur un exemple illustré par un diagramme de Gantt pour la communication d'un message entre deux noeuds distants de trois unités. La partie grisée représente le coût d'initialisation (en-tête du message) et la partie blanche les données à envoyer. En abscisse, nous avons le temps et en ordonnée les processeurs. Sur la gauche de la figure, des visages souriants ou tristes donnent les qualités et les défauts du modèle en termes de temps de latence (TL) ou de bande passante (BP).

6.4 Une étude de cas : l'hypercube

L'hypercube a été brièvement mentionné plus haut. Nous détaillons ici quelques une de ses propriétés.

6.4.1 Numérotation des sommets

Les hypercubes se construisent par duplication. Etant donnés deux m-cubes identiques (i.e. avec 2^m sommets), on construit un $m+1$-cube en reliant deux à deux les sommets correspondants. Si on ajoute qu'un 0-cube consiste en un sommet, on a une définition récursive des hypercubes. Celle-ci est illustrée sur la Figure 6.5, où on a relié deux 3-cubes pour construire un 4-cube.

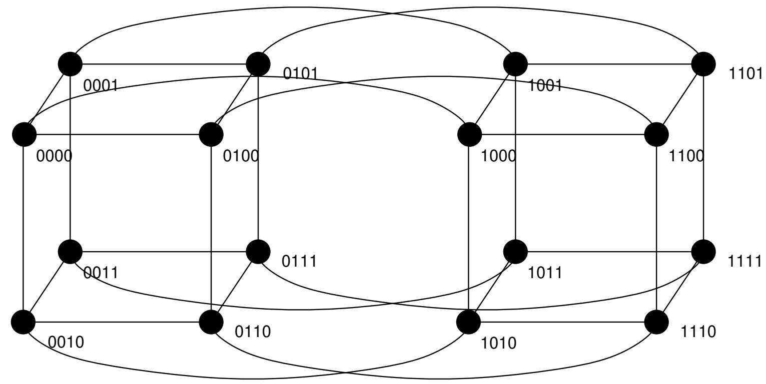


FIG. 6.5 – Hypercube de dimension 4.

Une autre définition (équivalente) est la suivante : un m-cube est composé de 2^m sommets numérotés de 0 à $2^m - 1$, interconnectés de telle sorte qu'il existe un lien entre deux sommets ssi leur écriture binaire ne diffère que par un seul bit. Avec la construction récursive on obtient facilement un étiquetage adéquat des sommets : étant donnés deux m-cubes, on renumérote les sommets du premier cube en $0a_i$ et ceux du second cube en $1a_i$, où a_i est l'écriture binaire représentant les deux sommets correspondants du m-cube. Cette numérotation est utilisée à la Figure 6.5.

La construction montre clairement qu'un hypercube de dimension m, ou m-cube, est de diamètre $D = m$ et de degré $\Delta = m$, puisque ceux-ci sont incrémentés d'une unité à chaque étape de la construction récursive.

6.4.2 Chemins et routage dans l'hypercube

Chemins

Cet étiquetage des sommets facilite la construction des chemins de communication entre deux noeuds quelconques. Soient A et B deux sommets du m-cube. *Router* un message de A à B consiste à trouver un chemin dans le m-cube qui mène de A à B et qui soit, objectif naturel, de longueur minimale.

Soit $H(A, B)$ la distance de Hamming entre A et B, définie comme le nombre de bits qui diffèrent dans l'écriture binaire de A et B. Pour voir que H est bien une distance, on écrit

$$H = \sum_{q=0}^{m-1} h_q$$

où $h_q(A, B) = 1$ si les q-èmes bits de A et B sont distincts, et 0 sinon. On voit facilement que h_q est une distance, d'où le résultat pour H.

Supposons $H(A, B) = i$. Comme la distance de Hamming entre deux sommets adjacents est égale à 1, tout chemin de A à B sera de longueur au moins égale à i. Il est facile de construire un chemin de longueur exactement égale à i. Soit $A = a_{m-1} \dots a_1 a_0$ et $B = b_{m-1} \dots b_1 b_0$. Sans perte de généralité, supposons que les i bits qui diffèrent dans ces deux écritures soient les i premiers, ceux de poids faibles. Alors

$$\begin{aligned} A &= a_{m-1}a_{m-2}\dots a_{i+1}a_i a_{i-1}a_{i-2}\dots a_2 a_1 a_0 \\ B &= a_{m-1}a_{m-2}\dots a_{i+1}a_i \overline{a_{i-1}a_{i-2}}\dots \overline{a_2 a_1 a_0} \end{aligned}$$

où $\overline{x} = 1 - x$ est le complément du bit x. Voici un chemin de A à B qui corrigent les bits en commençant par les poids faibles :

$$\begin{array}{lll} A = & \text{sommet } 0 & = a_{m-1}a_{m-2}\dots a_{i+1}a_i a_{i-1}a_{i-2}\dots a_2 a_1 a_0 \\ & \text{sommet } 1 & = a_{m-1}a_{m-2}\dots a_{i+1}a_i a_{i-1}a_{i-2}\dots a_2 a_1 \overline{a_0} \\ & \text{sommet } 2 & = a_{m-1}a_{m-2}\dots a_{i+1}a_i a_{i-1}a_{i-2}\dots a_2 \overline{a_1 a_0} \\ & \text{sommet } 3 & = a_{m-1}a_{m-2}\dots a_{i+1}a_i a_{i-1}a_{i-2}\dots \overline{a_2 a_1 a_0} \\ & \dots & \\ B = & \text{sommet i} & = a_{m-1}a_{m-2}\dots a_{i+1}a_i \overline{a_{i-1}a_{i-2}}\dots \overline{a_2 a_1 a_0} \end{array}$$

Bien sûr rien n'oblige à commencer par les poids faibles ; il existe d'autres chemins de longueur i entre A et B. La bonne question est la suivante : combien y-a-t-il de chemins indépendants de longueur minimum i entre A et B ? Deux chemins sont dits indépendants s'ils n'ont aucun sommet en commun à l'exception de A et de B. L'intérêt des chemins indépendants est de permettre l'acheminement simultané de plusieurs messages différents de A à B (par exemple le message original scindé en parties).

Il existe clairement $i!$ chemins de longueur i de A à B, obtenus en corrigeant les i bits distincts dans un ordre arbitraire. Mais combien de chemins indépendants ? La construction précédente permet d'en construire i : pour le j-ème chemin ch_j, $0 \leq j < i$, on corrige d'abord le bit en position j, puis celui en position $j + 1$, jusqu'au bit en position $i - 1$, à partir duquel on corrige les bits 0, 1, ..., $j - 1$ dans cet ordre. On voit que ch_0 est bien le chemin déjà construit. Pour vérifier que tous les chemins ch_j sont indépendants, considérons ch_{j_0} et ch_{j_1}, et imaginons qu'ils partagent un sommet X. Ces deux chemins en peuvent pas atteindre X en le même nombre de pas, puisqu'ils ne corrigent pas les bits dans le même ordre ; et ils ne peuvent pas atteindre X en des nombres de pas différents, car tous les noeuds intermédiaires sont sur un plus court chemin qui mènent d'eux-mêmes à B. Enfin, on ne peut pas ajouter un $(i + 1)$-ème chemin indépendant à notre collection : son premier sommet intermédiaire, nécessairement obtenu par correction d'un bit de A, disons le j-ème, coïnciderait avec celui du chemin ch_j.

Notons qu'avec les i chemins indépendants on ne visite que $i(i-1)+2$ sommets, et qu'on utilise toute la bande passante issue du sommet de départ que pour $i = m$.

Routage

De la construction des chemins qui précède, on peut déduire plusieurs algorithmes de routage. Le plus simple est le suivant : pour acheminer un message du processeur A au processeur B, on utilise systématiquement le premier chemin, celui qui corrige les bits à partir des poids faibles. Pour implémenter le processus sur le routeur matériel, on calcule A XOR B, où l'opérateur XOR opère bit à bit. Le premier bit non nul de A XOR B est le numéro du lien de communication sur lequel il faut envoyer le message. Le message va circuler avec une en-tête égale à A XOR B au départ, puis modifiée au cours de l'acheminement. Le routeur de chaque processeur va examiner l'en-tête : si elle est nulle, le processeur est le destinataire final du message ; sinon, il va modifier le premier bit non nul de l'en-tête et faire suivre le message sur le lien correspondant à la position de ce bit. Un exemple sur le 4-cube, avec $A = 1011$ et $B = 1101$. A XOR $B = 0110$, le processeur A envoie le message sur le lien numéro 1 (les numéros vont de 0 à 3), avec l'en-tête corrigée 0100. Le routeur du processeur destinataire (il s'agit du processeur 1001), au vu de l'en-tête 0100, sait que le message n'est pas à ranger dans la mémoire mais à acheminer sur le lien 2 ; le processeur 1001 ne sera pas interrompu dans son programme. Enfin, à son tour, le routeur du processeur B, au vu de l'en-tête nulle, stockera le message dans sa mémoire.

De par sa simplicité, l'algorithme de routage précédent est parfaitement adapté à une implantation matérielle, au sein d'un protocole *wormhole* ou *cut-through*. Il a deux défauts :

– il ne fait pas appel à plusieurs chemins indépendants
– il est purement statique

Le premier défaut n'est pas si grave : les machines capables de communiquer efficacement en parallèle sur plusieurs liens ne sont pa si nombreuses ; quitte à utiliser un seul lien, autant pipeliner le message en acheminant des paquets sur le chemin. Le deuxième défaut est plus génant : si une autre communication mettant en jeu une autre paire de processeurs a déjà réservé l'un des liens utilisés par notre chemin, notre message restera bloqué jusqu'à la fin de l'autre communication. C'est dommage ca nous avons plusieurs autres chemins à notre disposition.

Une variante dynamique de l'algorithme précédent consiste à corriger le premier bit correspondant à un lien disponible : au moment d'acheminer le message, le routeur d'un processeur décide dynamiquement sur quel lien envoyer le message, au vu de sa table de réservation des liens. Reprenons notre exemple sur le 4-cube, avec $A = 1011$ et $B = 1101$. A XOR $B = 0110$, et le routage statique commande d'utiliser le lien numéro 1. Mais si celui-ci est réservé par une autre communication, on envoie le message sur le lien 2, au processeur 1001, dont le routeur recevra l'en-tête 0010, provoquant l'acheminement sur le lien 1. Bien sûr si un routeur détermine que tous les liens candidats (ceux dont bits sont à 1 dans l'en-tête) sont occupés, il n'a d'autre choix que d'attendre la libération de l'un d'entre eux. Cet algorithme est séduisant mais plus difficile à implanter : en effet il faut que le protocole de routage soit équitable : il faut gérer équitablement les communications en attente pour ne pas que certaines attendent indéfiniment sur un réseau très actif. Il faut aussi savoir reconstruire un message à l'arrivée, au cas où les paquets qui le constituent aient suivi des routes différentes et arrivent dans le désordre. Enfin, il faut garantir que tous les messages arrivent bien, ce qui est facile avec notre algorithme mais deviendrait un vrai casse-tête si on avait autorisé les messages à s'écarter de leur route pour prendre des voies détournées en cas de bouchon ; dans l'exemple avec $A = 1011$ et $B = 1101$, si les liens 1 et 2 du processeur A ne sont pas libres, pourquoi ne pas essayer de passer par le lien 0, quitte à allonger la route ?

Un dernier mot : l'algorithme statique a été implanté par Intel sur l'iPSC2. L'algorithme dynamique a été implanté également, mais sur la grille du Paragon, pas sur l'hypercube. L'idée est la même : si le processeur de coordonnées (x, y) veut envoyer un message au processeur (x', y'), l'en-tête du message sera $x' - x, y' - y$, en supposant $x' > x$ et $y' > y$ (sinon, la grille étant orientée, on prend les différences modulo la taille de la grille dans chaque dimension). Un routage statique consiste à faire avancer le message horizontalement $x' - x$ fois, puis verticalement $y' - y$ fois. Mais dynamiquement, tous les chemins de Manhattan sont possibles, et le message peut circuler verticalement chaque fois que le lien horizontal est occupé. La restriction est de ne jamais sortir du rectangle déterminé par les sommets de départ et d'arrivée, pour garantir facilement l'absence de deadlock.

6.4.3 Plongements d'anneaux et de grilles dans l'hypercube

L'attractivité des hypercubes vient bien sûr de leur forte connectivité, mais également du fait que les topologies classiques en anneau, grille 2D, grille 3D, etc, peuvent y être plongées en préservant la proximité des voisins : deux sommets voisins sur l'anneau ou sur la grille seront envoyés sur deux sommets voisins dans l'hypercube.

Ces techniques de plongement sont fondamentales pour l'algorithmicien, qui conçoit des algorithmes sur l'anneau et la grille, et utilise pleinement la connectivité de l'hypercube pour les opérations de communications globales comme les diffusions.

Les plongements qui préservent la proximité ne sont pas toujours possibles. On cherche alors des plongements qui minimisent la distance entre les processeurs image de processeurs voisins sur l'anneau ou sur la grille.

Nous nous limitons ici aux plongements d'anneaux et de grilles dont les dimensions sont des puissances de 2 et qui utilisent tous les sommets de l'hypercube. L'outil adéquat : les codes de Gray. Le code de Gray de dimension m, noté G_m, est défini récursivement pour $m \geq 2$ comme la concaténation

$$G_m = 0G_{m-1}|1G_{m-1}^{rev}$$

où

- xG énumère les éléments de G en rajoutant un x en tête de leur écriture binaire.
- G^{rev} énumère les éléments de G dans l'ordre renversé (en partant de la fin).
- l'opérateur $|$ désigne la concaténation

On initie la construction avec $G_1 = \{0, 1\}$. Ainsi :

$$G_2 = \{00, 01, 10, 11\}$$
$$G_3 = \{000, 001, 010, 011, 111, 110, 101, 100\}$$
$$G_4 = \{0000, 0001, 0010, 0011, 0111, 0110, 0101, 0100, 1100, 1101, 1110, 1111, 1011, 1010, 1001, 1000\}$$

Le code de Gray G_m permet de définir un anneau de 2^m processeurs dans le m-cube. Supposons la propriété vraie pour $m-1$, avec $m \geq 2$ (c'est évident pour $m = 1$). Alors par construction, comme on a renversé la deuxième copie de G_{m-1}, les éléments en position $2^{m-1}-1$ et 2^{m-1} sont égaux au dernier élément de G_{m-1} précédé respectivement de 0 et de 1. De même, les éléments en position 0 et $2^m - 1$ sont égaux au premier élément de G_{m-1} précédé respectivement de 0 et de 1. Dans tous les autres cas, deux éléments consécutifs ne diffèrent que par un seul bit, par hypothèse de récurrence.

Pour définir une grille torique de taille $2^r \times 2^s$ dans un m-cube, avec $r + s = m$, on utilise le produit cartésien $G_r \times G_s$. Le processeur (i, j) sur la grille a pour image le processeur $f(i, j) = (g_i^{(r)}, g_j^{(s)})$ dans le m-cube. Les voisins horizontaux $(i \pm 1, j)$ ont pour image $f(i \pm 1, j) = (g_{i\pm1}^{(r)}, g_j^{(s)})$, où tous les premiers indices sont pris modulo 2^r : ceux-ci sont bien voisins de $f(i, j)$ sur le m-cube. On procède de même pour les voisins verticaux. On généralise facilement à un tore 3D de taille $2^r \times 2^s \times 2^t$, où $r + s + t = m$.

6.4.4 Macro-communications dans l'hypercube

Le but de ce paragraphe est de faire toucher du doigt la difficulté des macro-communications dans les réseaux d'interconnexion, pour peu qu'ils soient plus riches que l'anneau du Chapitre 5.

Nous nous contentons de la primitive la plus simple, la diffusion. Supposons que le processeur 0 veuille diffuser un message à tous les processeurs. L'algorithme naïf est le suivant : le processeur 0 envoie le message à tous ses voisins ; puis chacun des voisins de 0 envoie à son tour le message à tous ses propres voisins ; etc. Cet algorithme est redondant : un même processeur va recevoir le message plusieurs fois, par exemple 0 de tous ses voisins : attention aux send qui ne font pas face à des receive ! Nous allons chercher une stratégie où chaque processeur ne reçoit l'information qu'une fois, et où le nombre d'étapes est minimal. Pour préciser la notion d'étape, il y a plusieurs variantes, selon qu'on autorise un processeur à communiquer sur plusieurs lien à la fois ou non. L'idée est de construire un ou plusieurs arbres couvrants de l'hypercube.

Commençons avec un seul arbre couvrant, et prenons $m = 4$ pour fixer les idées. Il faut d'abord ajouter un argument aux primitives send et receive, à savoir la dimension du lien sur lequel a lieu la communication (sur l'anneau orienté, le problème ne se posait pas). Posons donc :

```
send(cube-link, send-adr, L)
receive(cube-link, recv-adr, L)
```

On décrit à la fois l'arbre couvrant et l'algorithme de diffusion, qui opère en m phases, numérotées de $m-1$ à 0. L'idée est que les processeurs vont recevoir le message sur le lien correspondant à leur premier 1 (à partir des poids faibles), et vont le propager sur les liens qui précèdent ce premier 1. A la phase i, tous les processeurs qui ont leur premier bit à 1 en position strictement supérieure à i propagent le message sur le lien i. Le processeur 0 émetteur est supposé avoir un 1 fictif en position m. Pour $m = 4$, l'algorithme débute ainsi :

- à la phase 3, seul 0000 a un premier bit à 1 en position quatre, il envoie le message sur le lien 3 à 1000,
- à la phase 2, 0000 et 1000 ont un premier bit à 1 en position au moins trois, ils envoient le message sur le lien 2, à 0100 et 1100 respectivement,
- ainsi de suite jusqu'à la phase 0 où tous les processeurs de numéro pair envoient le message sur le lien 0

L'algorithme, et partant l'arbre couvrant, sont représentés à la Figure 6.6.

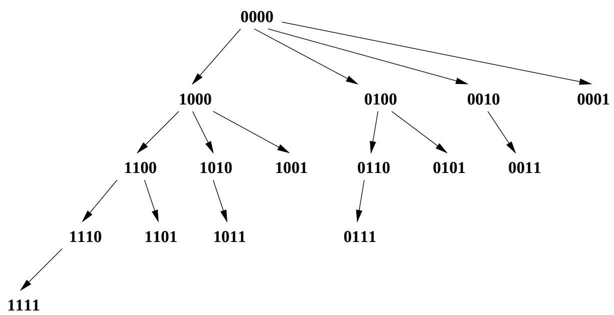


FIG. 6.6 – Diffusion avec l'arbre couvrant pour $m = 4$.

En notant bit(A,b) la valeur du bit en position b du processeur A, l'algorithme de diffusion par le processeur k d'un message de longueur L se détaille comme suit :

```
broadcast(k, adr, L) {
q ← my_num();
pos = q XOR k;
/* calcul du premier bit à 1 en syntaxe inspirée du langage C
first1 ← 0; while ((bit(pos,first1) == 0) && (first1 < m)) first1++
/* succession des phases de l'algorithme
for phase = m − 1 downto 0 step −1 do
  if (phase = first1) then
    receive(phase, adr, L)
  else
    if (phase < fist1) send(phase, adr, L)
}
```

Pour se ramener au cas de l'émetteur 0, on a pris le XOR bit à bit entre le numéro absolu q et le numéro de l'émetteur k.

L'algorithme comprend m étapes, son coût en modèle *store and forward* est donc $m(\beta + L\tau)$ (avec les notations du Paragraphe 6.3). Notons qu'à chaque étape les processeurs ne communiquent que sur un seul lien, l'algorithme est valide en mode 1-port. On peut pipeliner le message en le découpant en paquets ; à condition de disposer d'un modèle m-port (parce que en régime permanent 0, par exemple, va émettre des paquets sur tous ses liens simultanément), le nombre de paquets optimal se calcule comme expliqué précédemment.

Pour améliorer l'algorithme pipeliné en mode m-port, il faudrait disposer de m arbres couvrants à arêtes indépendantes. Les m parties du message pourraient alors être toutes pipelinées en paquets sur les m arbres de manière parallèle. La construction est possible, à condition de recourir à des arbres couvrants de hauteur m au lieu de $m - 1$.

Voici comment construire ces m arbres couvrants à arêtes indépendantes :

1. Prendre l'arbre couvrant précédent, enraciné en 0, noté A_0 sur la Figure 6.7.
2. Effectuer une rotation (à gauche) des écritures binaires des sommets de A_0 pour construire les autres arbres : A_i est l'image de A_0 par l'opérateur de rotation $R^{(i)}$. Si $s = s_{m-1}s_{m-2}\dots s_1s_0$ est l'écriture binaire du sommet s, on définit $R(s) = s_{m-2}s_{m-3}\dots s_0s_{m-1}$. On pose $R^{(1)} = R$ et $R^{(i)} = R \circ R^{(i-1)}$ pour $2 \le i \le m - 1$.
3. Les m arbres A_i ne sont pas à arêtes disjointes (par exemple les voisins de 0 sur l'arbre restent globalement les mêmes dans tous les arbres). On inverse le i-ème bit (modulo m) des sommets de A_i, comme indiqué sur la Figure 6.7.
4. Après cette opération XOR, le noeud 0 est une feuille dans chacun des m arbres A_i. On fusionne ceux-ci en un seul arbre de racine 0, et de hauteur m, comme indiqué Figure 6.8.

Notons qu'on suppose les liens pleinement bidirectionnels : par exemple sur la Figure 6.8, 101 envoie à 100 dans le sous-arbre de gauche, mais 100 envoie à 101 dans le sous-arbre de droite.

Cette construction nous fait passer d'un temps d'exécution

$$T = \left(\sqrt{L\tau} + \sqrt{(m-1)\beta}\right)^2$$

obtenu en mode pipeline optimal sur un seul arbre à un temps d'exécution

$$T = \left(\sqrt{\frac{L}{m}\tau} + \sqrt{m\beta}\right)^2$$

en pipelinant m messages de longueur L/m sur les m arbres. Ce temps d'exécution est à un facteur deux de la borne inférieure universelle pour diffuser un message de longueur L dans un m-cube en

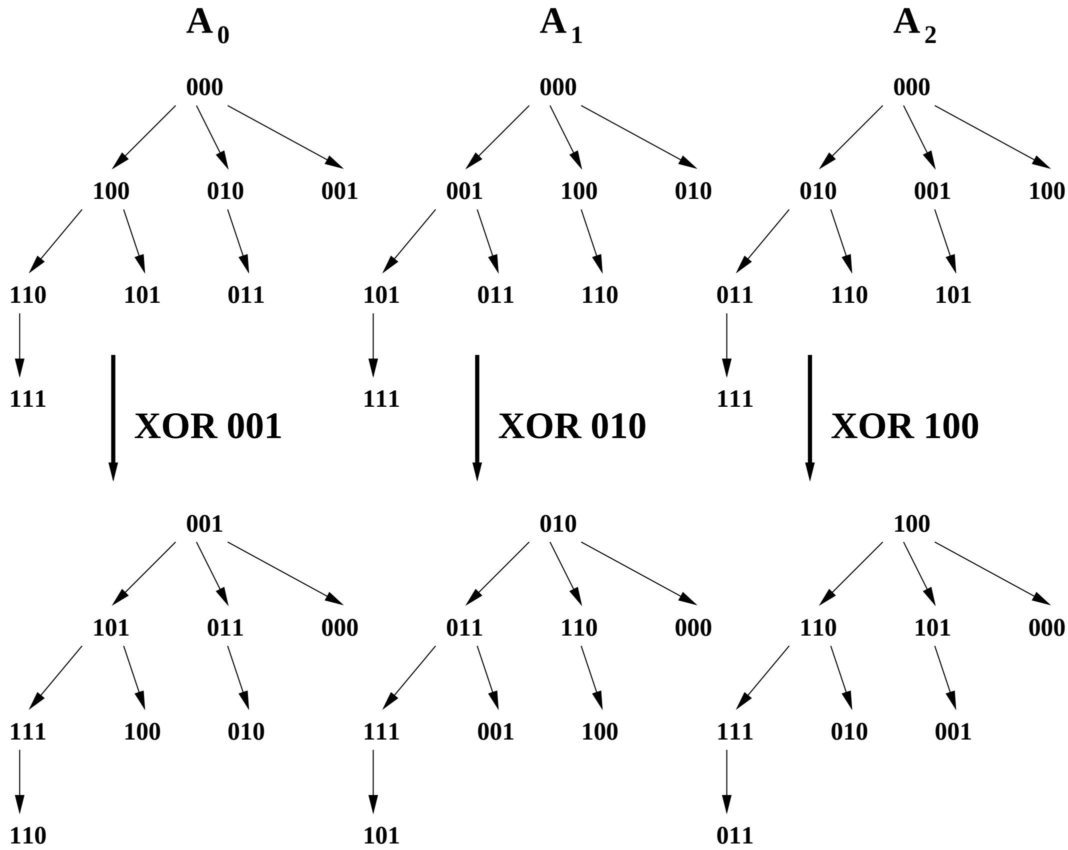


FIG. 6.7 – Rotations et opérations XOR sur les arbres couvrants, $m = 3$.

modèle m-port : la borne est $m\beta + (\frac{L}{m} + m - 1)\tau$ et s'obtient ainsi : il faut un temps au moins $m(\beta + \tau)$ pour que le premier mot du message atteigne le processeur le plus éloigné de l'émetteur ; à cet instant, ce processeur éloigné n'a pu recevoir que m mots du message, un sur chacun de ses liens. Il lui reste $L - m$ mots à recevoir sur m liens, ce qui nécessite un temps au moins $(\frac{L}{m} - 1)\tau$.

Bien sûr la borne ne peut pas être atteinte : si on n'a envoyé qu'un seul mot au départ, comment le reste du message se trouverait-il par magie déjà chez les processeur voisins du processeur éloigné. En fait, établir la complexité des opérations de macro-communications est une activité qui a fait couler beaucoup d'encre dans les années 80 : toute la difficulté provient du terme en β qui rend le modèle non linéaire, et la théorie des flots inapplicable.

6.5 Produit de matrices sur grille 2D

Ce paragraphe est consacré au produit de deux matrices carrées sur une grille torique à deux dimensions. Un algorithme simple qui révèle bien des surprises !

6.5.1 Quelles sont les contraintes ?

Allocation des données sur les processeurs

Nous nous intéresserons plus particulièrement aux rangements de données propres à l'algèbre linéaire dense. Supposons que nous ayons à distribuer un vecteur de M données sur p processeurs. Nous pouvons décrire cette allocation comme le *mapping* d'un indice global d'une donnée m sur

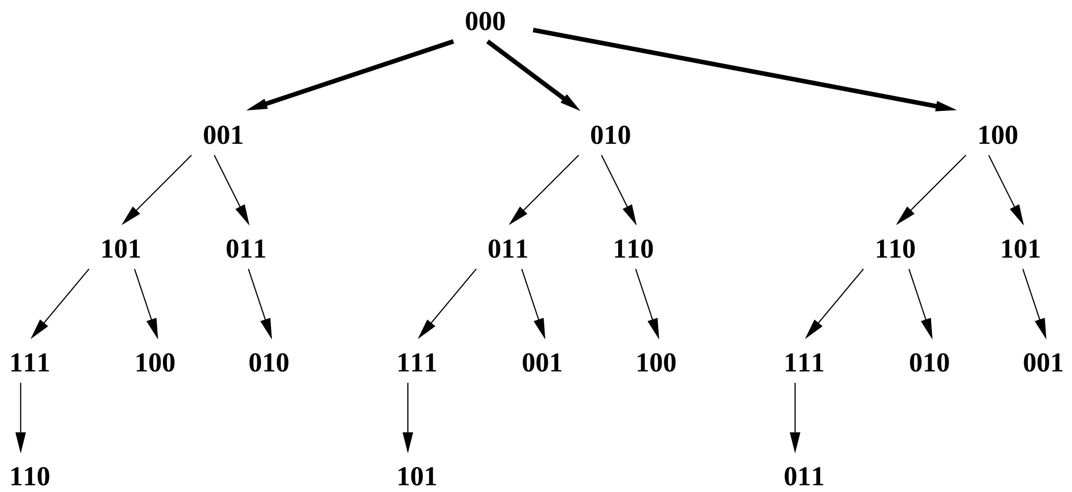


FIG. 6.8 – Diffusion avec $m = 3$ arbres couvrants à arêtes indépendantes.

une paire d'indices (q,i) où q est le processeur sur lequel la donnée est déposée et i, l'indice local dans ce processeur. Nous avons $0 \leq m < M$ et $0 \leq q < p$.

Pour déterminer une stratégie d'allocation, on observe que

- Les algorithmes par blocs sont indispensables en algèbre linéaire pour augmenter le grain de calcul et améliorer l'utilisation des mémoires hiérarchiques.
- Pour équilibrer la charge, on distribue les données de manière cyclique aux processeurs. Cela permet, par exemple, de prendre en compte les algorithmes de type factorisation LU où la taille de la matrice à traiter diminue à chaque étape.

Ces deux considérations conduisent à adopter une distribution cyclique par blocs, qui est donnée par un triplet (q, b, i) où q est le numéro de processeur, b le numéro de bloc et i l'index dans le bloc. Cette distribution est donnée par

$$m \mapsto \left(\lfloor \frac{m \mod T}{r} \rfloor,\ \lfloor \frac{m}{T} \rfloor,\ m \mod r \right)$$

où r est la taille de bloc et $T = rp$. Notons que cette distribution suppose que l'élément 0 est placé sur le processeur 0, ce qui est souvent le cas.

Supposons à présent, que nous ayons à décomposer une matrice de taille $M \times N$ sur une grille de $P \times Q$ (P lignes et Q colonnes de processeurs). Le nombre de processeurs est donc de $N_p = PQ$. Les décompositions précédentes unidimensionnelles s'appliquent également aux matrices, de manière indépendante dans chaque dimension ; toutes les variations de tailles de blocs sont possibles. La décomposition d'une matrice donne un triplet de paires d'indices $(m,n) \mapsto ((p,q),(b,d),(i,j))$. La Figure 6.9 donne différents exemples de décomposition d'une matrice 10×10 (les numéros dans la matrice sont les indices des processeurs où sont rangés les éléments). Sur la gauche de la figure, nous donnons les numérotations des processeurs et les topologies virtuelles utilisées. Les deux décompositions supérieures concernent un réseau linéaire de $P = 4$ processeurs, les deux décompositions inférieures s'effectuent sur une grille 4×4. La taille verticale des blocs est donnée par le paramètre r et la taille horizontale des blocs par le paramètre s.

Les distributions cycliques par bloc constituent le standard, adopté à la fois par la bibliothèque ScaLAPACK [16] et par le langage High Performance Fortran (HPF) [45], pour l'allocation des tableaux sur machine à mémoire distribuée.

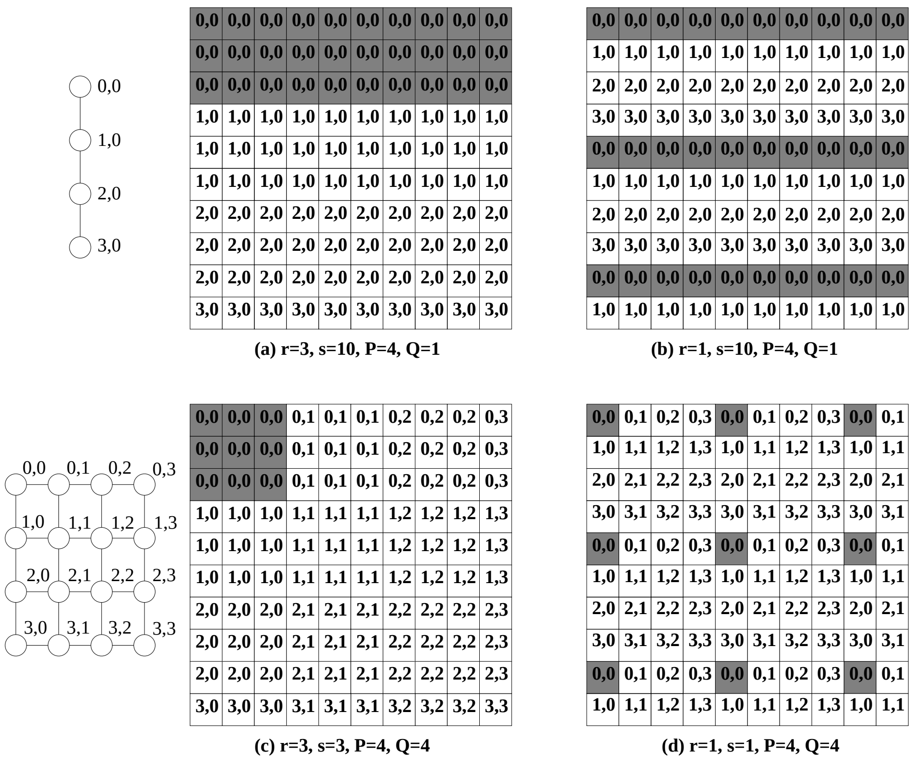


FIG. 6.9 – Différentes allocations d'une matrice sur un réseau linéaire ((a) et (b)), et sur une grille ((c) et (d)).

Contraintes liées à l'implémentation dans une bibliothèque

De nouveaux problèmes apparaissent avec l'utilisation de machines parallèles à mémoire distribuée pour l'implémentation de bibliothèques d'algèbre linéaire. Ceux-ci n'existent pas avec les machines à mémoire partagée car ils sont liés au fait que les données sont distribuées dans le réseau. L'interface avec l'extérieur a une extrême importance. Deux solutions sont à considérer :

1. **Version centralisée** Les routines sont appelées en supposant que les données (les matrices initiales) résident sur la machine hôte. De même, les routines renvoient des résultats (matrices finales) qui seront stockés sur la machine hôte. Ceci nécessite un chargement sur le réseau avec distribution adéquate des matrices, une exécution sur les différents processeurs, et enfin un déchargement sur l'hôte.
2. **Version distribuée** Les routines prennent en argument des données déjà distribuées sur les processeurs, et renvoient des résultats distribués également.

La version centralisée minimise le nombre de routines à écrire. Elle permet en outre de choisir la distribution des données la plus adaptée au noyau de calcul considéré. Par contre, son coût s'avère prohibitif si on enchaîne plusieurs appels, par exemple un produit de matrices $C = A \times B$ suivi d'une factorisation LU de la matrice C, suivi encore de la résolution de plusieurs systèmes linéaires $(LU)x_i = b_i$: les données vont être chargées depuis l'hôte avant chaque appel, puis déchargées sur l'hôte après chaque calcul.

La version distribuée doit pouvoir fonctionner sur une large classe de distributions de matrices. Par exemple pour un produit de matrices $C = A \times B$, il n'est pas raisonnable de définir une routine qui ne fonctionne que pour une distribution bien précise des matrices A et B. Il semble naturel de demander que A et B soient distribuées de la même manière, et que la matrice C résultat soit disponible en sortie avec la même distribution que A et B. Cependant, certaines distributions (certaines tailles de blocs) peuvent être mieux adaptées que d'autres à certains noyaux de calcul : il faut alors envisager une redistribution des données entre deux appels. Il y a donc des compromis à trouver entre le *coût de redistribution + coût de l'algorithme avec rangement adapté* et le *coût de l'algorithme avec rangement non adapté*.

6.5.2 Trois produits de matrices sur un grille 2D

Dans ce paragraphe, nous présentons trois algorithmes classiques pour le produit de matrices sur machines à mémoire distribuée. Les trois algorithmes utilisent comme réseau sous-jacent un tore à deux dimensions.

Nous supposons avoir trois matrices A, B et C de taille $N \times N$. Nous avons à calculer le produit $C = C + AB$. Par souci de simplicité, nous supposons que le nombre p de processeurs est un carré parfait : $p = q^2$, la grille cible est de taille $q \times q$.

Nous partons d'une distribution des matrices par blocs de telle manière que les sous-matrices A_{ij}, B_{ij} et C_{ij} soient disposées sur le processeur dont les coordonnées dans la grille sont (i, j). Après le calcul du produit, les matrices doivent se retrouver à leur place de départ afin de pouvoir être utilisées pour d'autres calculs.

Algorithme de Cannon

L'algorithme de Cannon [19] nécessite une redistribution de deux des matrices avant le début des calculs et le rangement inverse ensuite. Les sous-matrices C_{ij} sont rangées de manière naturelle sur le tore (sous-matrice C_{ij} rangée sur le noeud (i, j)). Mais pour les matrices A et B, de manière à avoir des communications voisins/voisins, la matrice A est "tournée" horizontalement de telle sorte que sa diagonale soit rangée sur la première colonne de processeurs du tore, la diagonale de B étant distribuée sur la première ligne (Figure 6.10). Pour obtenir ce rangement, les sous-matrices sont déplacées sur les lignes et les colonnes de processeurs suivant leur indice de ligne ou de colonne (A_{ij} sur le processeur $(i, (i + j) \mod q)$), B_{ij} sur le processeur $((i + j) \mod q, j)$. Ces rotations sont souvent appelées *preskewing* et *postskewing* dans la littérature.

$$
\begin{array}{cccc}
\mathrm{C00} & \mathrm{C01} & \mathrm{C02} & \mathrm{C03} \\
\mathrm{C10} & \mathrm{C11} & \mathrm{C12} & \mathrm{C13} \\
\mathrm{C20} & \mathrm{C21} & \mathrm{C22} & \mathrm{C23} \\
\mathrm{C30} & \mathrm{C31} & \mathrm{C32} & \mathrm{C33}
\end{array}
=
\begin{array}{cccc}
\mathrm{A00} & \mathrm{A01} & \mathrm{A02} & \mathrm{A03} \\
\mathrm{A11} & \mathrm{A12} & \mathrm{A13} & \mathrm{A10} \\
\mathrm{A22} & \mathrm{A23} & \mathrm{A20} & \mathrm{A21} \\
\mathrm{A33} & \mathrm{A30} & \mathrm{A31} & \mathrm{A32}
\end{array}
\mathrm{x}
\begin{array}{cccc}
\mathrm{B00} & \mathrm{B11} & \mathrm{B22} & \mathrm{B33} \\
\mathrm{B10} & \mathrm{B21} & \mathrm{B32} & \mathrm{B03} \\
\mathrm{B20} & \mathrm{B31} & \mathrm{B02} & \mathrm{B13} \\
\mathrm{B30} & \mathrm{B01} & \mathrm{B12} & \mathrm{B23}
\end{array}
$$

FIG. 6.10 – Rangement des sous-matrices pour l'algorithme de Cannon.

Cet algorithme ne requiert que des communications voisins/voisins. Pendant la phase de calculs, les sous-matrices A_{ij} se déplacent de droite à gauche sur les lignes de processeurs (rotation horizontale) tandis que les sous-matrices B_{ij} se déplacent de haut en bas (rotation verticale), les matrices C_{ij} restant sur place. De plus, si une bufférisation est possible, communications et calculs de deux étapes différentes peuvent être aisément recouverts (Table 6.2).

```
/* mouvements de données avant les calculs */
Rotations de A et B

/* calcul du produit de matrice */
pour k = 1 à √P en // faire
   LocalProdMat(A, B, C)
   Rotation verticale de B
   Rotation horizontale de A
finpour

/* mouvements de données après les calculs */
Rotations de A et B
```

TAB. 6.2 – Algorithme de Cannon.

Algorithme de Fox

Cet algorithme, décrit par Fox dans [33], a été originellement développé pour l'hypercube de CalTech, même si le réseau sous-jacent est le tore à deux dimensions. Cet algorithme nécessite la diffusion des sous-matrices A_{ij} durant son exécution. Il est d'ailleurs souvent appelé dans la littérature algorithme *broadcast-multiply-roll*.

Les données sont réparties de manière naturelle sur le tore, c'est-à-dire que les sous-matrices $[A, B, C]_{ij}$ sont rangées sur les processeurs dont les coordonnées sont (i, j). Cet algorithme nécessite des diffusions horizontales des diagonales de A. Les communications des sous-matrices B sont des rotations verticales (Table 6.3). Les deux premiers pas de l'algorithme sont donnés sur la Figure 6.11.

$$\begin{matrix} C00 & C01 & C02 & C03 \\ C10 & C11 & C12 & C13 \\ C20 & C21 & C22 & C23 \\ C30 & C31 & C32 & C33 \end{matrix} \;+=\; \begin{matrix} A00 & A00 & A00 & A00 \\ A11 & A11 & A11 & A11 \\ A22 & A22 & A22 & A22 \\ A33 & A33 & A33 & A33 \end{matrix} \;\mathrm{x}\; \begin{matrix} B00 & B01 & B02 & B03 \\ B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \end{matrix}$$

$$\begin{matrix} C00 & C01 & C02 & C03 \\ C10 & C11 & C12 & C13 \\ C20 & C21 & C22 & C23 \\ C30 & C31 & C32 & C33 \end{matrix} \;+=\; \begin{matrix} A01 & A01 & A01 & A01 \\ A12 & A12 & A12 & A12 \\ A23 & A23 & A23 & A23 \\ A30 & A30 & A30 & A30 \end{matrix} \;\mathrm{x}\; \begin{matrix} B10 & B11 & B12 & B13 \\ B20 & B21 & B22 & B23 \\ B30 & B31 & B32 & B33 \\ B00 & B01 & B02 & B03 \end{matrix}$$

FIG. 6.11 – Les deux premiers pas de l'algorithme de Fox.

Algorithme de Snyder

Cet algorithme utilise également une distribution des matrices par blocs sur un tore à deux dimensions. Le schéma de communications est différent car il utilise une transposition préalable de la matrice B avant exécution de l'algorithme ainsi que des sommes globales sur les lignes de processeurs des produits calculés à chaque étape [50]. Les matrices A_{ij} et C_{ij} sont rangées sur les

```
/* pas de mouvements de données avant les calculs */

/* calcul du produit de matrices */
Diffuser la première diagonale de A
pour k = 1 à √P − 1 en // faire
   LocalProdMat(A, B, C)
   Rotation verticale de B
   Diffuser la diagonale suivante de A
finpour
en // faire
   LocalProdMat(A, B, C)
   Rotation verticale de B
fin

/* pas de mouvements de données après les calculs */
```

TAB. 6.3 – Algorithme de Fox.

processeurs de manière naturelle. Pour la matrice B, il s'agit d'une transposition par blocs (B_{ij} sur le processeur (j, i)).

Les deux premiers pas de cet algorithme sont donnés sur la Figure 6.12. Nous avons mis en gras sur cette figure le processeur où s'effectue l'accumulation des sous-matrices C. Remarquons qu'il correspond aux différentes diagonales du tore, dans le même ordre que pour les diffusions de Fox (Table 6.4).

$$\begin{array}{cccc} \mathbf{C00} & C01 & C02 & C03 \\ C10 & \mathbf{C11} & C12 & C13 \\ C20 & C21 & \mathbf{C22} & C23 \\ C30 & C31 & C32 & \mathbf{C33} \end{array} = \begin{array}{cccc} A00 & A01 & A02 & A03 \\ A10 & A11 & C12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{array} \text{x} \begin{array}{cccc} B00 & B10 & B20 & B30 \\ B01 & B11 & B21 & B31 \\ B02 & B12 & B22 & B32 \\ B03 & B13 & B23 & B33 \end{array}$$

$$\begin{array}{cccc} C00 & \mathbf{C01} & C02 & C03 \\ C10 & C11 & \mathbf{C12} & C13 \\ C20 & C21 & C22 & \mathbf{C23} \\ \mathbf{C30} & C31 & C32 & C33 \end{array} = \begin{array}{cccc} A00 & A01 & A02 & A03 \\ A10 & A11 & C12 & A13 \\ A20 & A21 & A22 & A23 \\ A30 & A31 & A32 & A33 \end{array} \text{x} \begin{array}{cccc} B01 & B11 & B21 & B31 \\ B02 & B12 & B22 & B32 \\ B03 & B13 & B23 & B33 \\ B00 & B10 & B20 & B30 \end{array}$$

FIG. 6.12 – Les deux premiers pas de l'algorithme de Snyder.

6.5.3 Analyse des trois algorithmes

Dans ce paragraphe, nous présentons une analyse de complexité pour nos trois algorithmes. Une fois n'est pas coutume, nous détaillons cette analyse pour deux protocoles de communication différents :

SF4P (Store and Forward 4 Ports) On utilise un protocole de communication de type à commuta-

```
/* mouvements de données avant les calculs */
transposer la matrice B

/* calcul du produit de matrices */
en // faire
   LocalProdMat(A, B, C)
   Rotation verticale de B
finpour
pour k = 1 à √P − 1 en // faire
   Somme globale sur les lignes pour C_{i,(i+k−1)mod√P}
   LocalProdMat(A, B, C)
   Rotation verticale de B
finpour
Somme globale sur les lignes pour C_{i,(i+√P−1)mod√P}

/* mouvements de données après les calculs */
transposer la matrice B
```

Tab. 6.4 – Algorithme de Snyder pour le noeud (i, j).

Mode de comm.	**Paramètres**	**Signification**
SF4P	$\beta_{sf} + L\tau_a u_{sf}$	coût d'un msg de taille L entre 2 voisins
WH1P	$\beta_{wh} + L\tau_a u_{wh}$	coût d'un msg de taille L entre 2 proc. quelconques
/	τ_a	coût d'une multiplication suivie d'une addition

Tab. 6.5 – Paramètres des machines cibles pour le produit de matrices.

tion de messages 4-port, i.e. des échanges peuvent s'effectuer entre 4 processeurs voisins pour le coût d'une communication.

WH1P (WormHole 1 port) On utilise un protocole de communication de type wormhole 1-port, i.e. une seule communication (ou un échange) peut être traitée à la fois.

Dans les deux cas, les liens de communication sont supposés bi-directionnels (*full-duplex*). Les paramètres des machines cibles sont résumés dans la Table 6.5.

Pour chaque algorithme, nous donnons les coûts des pré- et post-traitements ($T_{algo}^{red\ modele}$) et des produits parallèles eux-mêmes ($T_{algo}^{comp\ modele}$). La somme de ces deux coûts donne le coût total de l'algorithme. Les caractéristiques des trois algorithmes sont résumées dans la Table 6.6. Rappelons que la grille de processeurs est de taille $p = q \times q$.

Algorithme de Cannon

Dans la Table 6.7, nous donnons les résultats de complexité pour une étape de l'algorithme de Cannon, pour chacun des deux modèles de communication. Le coût de calcul correspond aux trois boucles du produit de matrices de taille m, soit $m^3\tau_a$.

Nous allons analyser en premier le modèle SF4P. En ce qui concerne les pré et post-traitements pour les matrices A et B, une solution consiste à limiter le nombre de rotations jusqu'à $\lfloor q/2 \rfloor$. En effet, si l'on prend comme exemple la dernière ligne de processeurs, pour les rotations de A, au lieu

Algorithme	**Cannon**	**Fox**	**Snyder**
Pré(post)-traitements	rotations de A et B	aucun	transpo. de B
Calculs des produits	sur place	sur place	sommes globales sur les lignes
Mouvements de A	rotations hor.	diffusions hor.	aucun
Mouvements de B	rotations vert.	rotations vert.	rotations vert.

TAB. 6.6 – Résumé des caractéristiques des trois algorithmes de produit de matrices par blocs.

Etape	**Modèle**	
	SF4P	**WH1P**
Pré(post)trait.	$\lfloor \frac{q}{2} \rfloor (\beta_{sf} + m^2\tau_a u_{sf})$	$2 \lfloor \frac{q}{2} \rfloor (\beta_{wh} + m^2\tau_a u_{wh})$
Prod. de ss-mat	$2m^3\tau_a$	$2m^3\tau_a$
Com. de A	$\beta_{sf} + m^2\tau_a u_{sf}$	$2(\beta_{wh} + m^2\tau_a u_{wh})$
Com. de B	$\beta_{sf} + m^2\tau_a u_{sf}$	$2(\beta_{wh} + m^2\tau_a u_{wh})$

TAB. 6.7 – Coûts pour un pas de l'algorithme de Cannon, avec les deux modèles.

de tourner q fois vers la gauche, il suffit de tourner une fois vers la droite. Le plus grand nombre de rotations se trouve alors au milieu du tore ou l'on doit tourner $\lfloor q/2 \rfloor$ fois. Remarquons que, grâce aux communications simultanées sur tous les liens, les coûts de pré(post)-traitements pour A et B ne s'additionnent pas car ils sont exécutés en parallèle. Les rotations de A et de B pendant le calcul se font également en parallèle.

Pour le modèle SF4P, en doublant l'espace mémoire pour le stockage des matrices A et B, on peut recouvrir les communications et les calculs, ce qui donne le résultat (sans les pré et post-traitements)

$$T_{Cannon}^{comp\ SF4P} = q \max\left(2m^3\tau_a, \beta_{sf} + m^2\tau_a u_{sf}\right),$$

auquel il faut ajouter le coût des pré et post-traitements

$$T_{Cannon}^{red\ SF4P} = 2\lfloor q/2 \rfloor (\beta_{sf} + m^2\tau_a u_{sf}).$$

Pour la machine utilisant le modèle WH1P et à cause du port unique utilisable, les coûts de communication s'additionnent. Malgré tout, les communications peuvent être recouvertes par les calculs.

Les pré et post-traitements nécessitent une attention toute particulière. En effet, un algorithme naïf utilisera le même algorithme que pour le modèle précédent, avec un temps $4\lfloor q/2 \rfloor (\beta_{wh} + m^2\tau_a u_{wh})$. Mais grâce au modèle de communication wormhole, les processeurs peuvent envoyer leurs données directement au processeur concerné sans passer par tous les processeurs intermédiaires, à condition d'éviter au maximum les problèmes de contention, La Figure 6.13 montre des exécutions de pré- (post-) rotations avec le modèle WH1P pour un anneau à 8 processeurs et pour des rotations de taille 1, 2, 3 et 4. On constate alors que l'on peut réduire le nombre de pas en tenant compte des contentions. Le temps total est compris entre $2(\beta_{wh} + m^2\tau_a u_{wh})$ et $4\lfloor q/2 \rfloor (\beta_{wh} + m^2\tau_a u_{wh})$. Et plus précisément, si l'on considère que l'on peut faire en moyenne deux communications par pas, il est approximativement égal à $2\lfloor q/2 \rfloor (\beta_{wh} + m^2\tau_a u_{wh})$; le facteur 2 est dû à l'addition du temps pour le pré-traitement de la matrice A et pour celui de la matrice B.

Le coût nécessaire au calcul est donné par

$$T_{Cannon}^{comp\ WH1P} = q \max\left(2m^3\tau_a, 4(\beta_{wh} + m^2\tau_a u_{wh})\right),$$

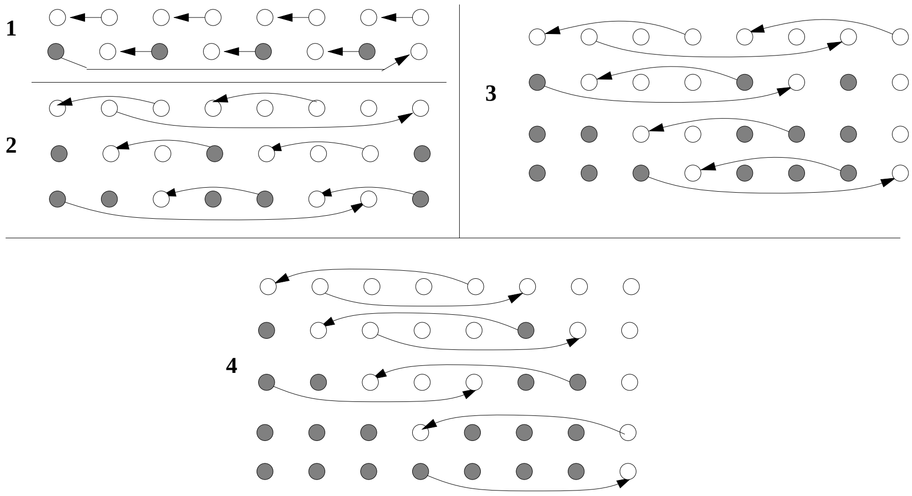


FIG. 6.13 – Pré- (et post-) rotations WH1P de la matrice A pour l'algorithme de Cannon.

Etape	**Modèle**	
	SF4P	**WH1P**
Pré(post)trait.	0	0
Prod. de ss-mat	$2m^3\tau_a$	$2m^3\tau_a$
Com. de A (ss. pipeline)	$\lfloor \frac{q}{2} \rfloor (\beta_{sf} + m^2\tau_a u_{sf})$	$\log(q)(\beta_{wh} + m^2\tau_a u_{wh})$
Com. de A (av. pipeline)	$\left(\sqrt{\beta_{sf}\left(\lfloor \frac{q}{2} \rfloor - 1\right)} + \sqrt{m^2\tau_a u_{sf}}\right)^2$	/
Com. de B	$\beta_{sf} + m^2\tau_a u_{sf}$	$2(\beta_{wh} + m^2\tau_a u_{wh})$

TAB. 6.8 – Coûts pour un pas de l'algorithme de Fox, avec les deux modèles.

auquel il faut ajouter le coût des pré et post-traitements

$$T_{Cannon}^{red\ WH1P} = 2\lfloor q/2 \rfloor (\beta_{wh} + m^2\tau_a u_{wh}).$$

Algorithme de Fox

Comme nous avons q étapes pour les calculs, nous avons uniquement à multiplier le coût total d'un pas par le nombre de pas. Le surcoût le plus important de l'algorithme de Fox est lié aux diffusions des sous-matrices A. A la différence de l'algorithme précédent, suivant le modèle de communication, des algorithmes différents sont utilisés. Le problème est la diffusion dans un anneau de processeurs. Pour le modèle SF4P, on peut utiliser ou non le pipeline, suivant la taille du message à diffuser. Si le pipeline est utilisé, comme l'anneau est bidirectionnel, la taille optimale des paquets est donnée par $\sqrt{\frac{m^2\beta_{sf}}{\lfloor \frac{q}{2} \rfloor \tau_a u_{sf}}}$. Remarquons que, avec le modèle SF4P, les rotations de B sont complètement recouvertes par les diffusions de A, elles n'apparaissent donc pas dans le calcul du temps total.

Sans utiliser de pipeline pour les diffusions, le coût nécessaire au calcul est donné par

$$\begin{aligned} T_{Fox}^{compSP\ SF4P} = \ & \lfloor q/2 \rfloor (\beta_{sf} + m^2\tau_a u_{sf}) + (q-1)\max\left(2m^3\tau_a, \lfloor q/2 \rfloor (\beta_{sf} + m^2\tau_a u_{sf})\right) + \\ & \max\left(2m^3\tau_a, \beta_{sf} + m^2\tau_a u_{sf}\right) \end{aligned}$$

Etape	**Modèle**	
	SF4P	**WH1P**
Pré(post)trait.	$\lfloor q/2 \rfloor \left(\sqrt{\beta_{sf}} + \sqrt{\frac{N^2}{2P}\tau_a u_{sf}} \right)^2$	$log_2(q)\beta_{wh} + \left(\frac{q}{2}+1\right) m^2\tau_a u_{wh}$
Prod. de ss-mat	$2m^3\tau_a$	$2m^3\tau_a$
Acc. des sous-mat. C **sans pipeline**	$\lfloor (q-2)/2 \rfloor \left(\beta_{sf} + m^2\tau_a u_{sf}\right)$ $+\lfloor (q+1)/2) \rfloor m^2\tau_a$	$\log(q)(\beta_{wh} + m^2\tau_a u_{wh} + m^2\tau_a)$
Acc. des sous-mat. C **avec pipeline**	$(q-3)\beta_{sf} + m^2\tau_a u_{sf}$ $+2m\sqrt{\beta_{sf}((q-3)\tau_a u_{sf} + 2\tau_a)}$	/
Com. de A	0	0
Com. de B	$\beta_{sf} + m^2\tau_a u_{sf}$	$2(\beta_{wh} + m^2\tau_a u_{wh})$

TAB. 6.9 – Coûts pour un pas de l'algorithme de Snyder, avec les deux modèles.

Si on utilise un pipeline pour les diffusions, le coût nécessaire au calcul est donné par

$$\begin{aligned} T_{Fox}^{compAP\ SF4P} \;=\; & \left(\sqrt{\beta_{sf}\left(\lfloor q/2 \rfloor - 1\right)} + \sqrt{m^2\tau_a u_{sf}}\right)^2 + \\ & (q-1)\max\left(2m^3\tau_a, \left(\sqrt{\beta_{sf}\left(\lfloor q/2 \rceil - 1\right)} + \sqrt{m^2\tau_a u_{sf}}\right)^2\right) + \\ & \max\left(2m^3\tau_a, \beta_{sf} + m^2\tau_a u_{sf}\right). \end{aligned}$$

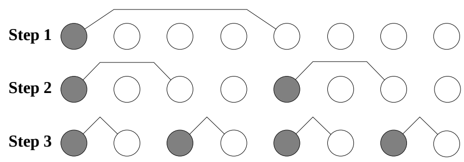


FIG. 6.14 – Méthode de diffusion sur un anneau avec modèle WH1P.

Avec le modèle WH1P, on utilise pour la diffusion l'analogue de l'algorithme sur l'hypercube avec un arbre couvrant (Paragraphe 6.4.4). Le coût nécessaire au calcul est donné par

$$\begin{aligned} T_{Fox}^{comp\ WH1P} \;=\; & \log(q)(\beta_{wh} + m^2\tau_a u_{wh}) + \\ & (q-1)\max\left(2m^3\tau_a, (\log(q)+2)(\beta_{wh} + m^2\tau_a u_{wh})\right) + \\ & \max\left(2m^3\tau_a, 2(\beta_{wh} + m^2\tau_a u_{wh})\right). \end{aligned}$$

Algorithme de Snyder

Les différences majeures entre l'algorithme de Snyder et les précédents sont l'utilisation d'une pré- et post-transposition de la matrice B d'une part, et les sommes globales pour accumuler les sous-matrices C_{ij} d'autre part. Remarquons que dans la Table 6.9 résumant les coûts, nous avons séparé le calcul du produit AB lui-même de l'accumulation des matrices C qui nécessite ici des communications sur les lignes de processeurs (effectuées en pipeline ou non). Pour la transposition, nous renvoyons à la littérature : sur un tore 2D en SF4P, on peut utiliser une méthode pipeline sur deux chemins disjoints ; our le modèle wormhole, on peut utiliser un algorithme récursif : voir [26] dans les deux cas. Les complexités sont données pour q impair.

Le coût total pour le modèle SF4P s'obtient en additionnant le coût nécessaire au calcul du produit au coût des pré- et post-transpositions (nous ne considérons que la version pipeline pour les accumulations) :

$$\begin{aligned} T_{Snyder}^{comp\ SF4P} = & \max\left(2m^3\tau_a, \beta_{sf} + m^2\tau_a u_{sf}\right) + \\ & (q-1)\max\left(2m^3\tau_a, (q-3)\beta_{sf} + m^2\tau_a u + 2m\sqrt{\beta_{sf}((q-3)\tau_a u_{sf} + 2\tau_a)}\right) + \\ & (q-3)\beta_{sf} + m^2\tau_a u_{sf} + 2m\sqrt{\beta_{sf}((q-3)\tau_a u_{sf} + 2\tau_a)}, \end{aligned}$$

$$T_{Snyder}^{red\ SF4P} = \lfloor q/2 \rfloor \left(\sqrt{\beta_{sf}} + \sqrt{\frac{N^2}{2P}\tau_a u_{sf}}\right)^2.$$

Pour le WH1P, les accumulations sont effectuées avec une méthode proche de celle utilisée auparavant pour la diffusion des sous-matrices A_{ij} dans l'algorithme de Fox. L'algorithme s'exécute de manière inverse de la diffusion, et après chaque réception, les processeurs additionnent les sous-matrices C reçues avec la sous-matrice C calculée à l'étape courante et renvoie le résultat si nécessaire. Le nombre d'étapes est bien entendu le même que pour la diffusion. Le coût total est

$$\begin{aligned} T_{Snyder}^{comp\ WH1P} = & \max\left(2m^3\tau_a, 2(\beta_{wh} + m^2\tau_a u_{wh})\right) + \\ & (q-1)\max\left(2m^3\tau_a u_a, \log(q)(\beta_{wh} + m^2\tau_a u_{wh} + m^2\tau_a) + 2(\beta_{wh} + m^2\tau_a u_{wh})\right) + \\ & \log(q)(\beta_{wh} + m^2\tau_a u_{wh} + m^2\tau_a), \end{aligned}$$

$$T_{Snyder}^{red\ WH1P} = log_2(q)\beta_{wh} + \left(\frac{q}{2} + 1\right) m^2\tau_a u_{wh}.$$

Ouf! La comparaison est achevée. Elle a eu le mérite de nous montrer tous les détails qu'il restait à régler après avoir compris le principe des algorithmes ...

Notes bibliographiques

Ce chapitre doit beaucoup à la thèse de Desprez [29] et au livre de Gengler, Ubéda et Desprez [35]. Pour d'autres exemples d'algorithmes parallèles sur grille ou hyeprcube, on pourra consulter le livre de Kumar et al [46]. Pour plus d'information sur les réseaux d'interconnexion dynamiques, et sur les architectures à mémoire distribuée en général, on pourra consulter l'excellent livre de Culler et Singh [27].

Chapitre 7

Algorithmique sur ressources hétérogènes

7.1 Introduction

Ce chapitre présente quelques algorithmes parallèles destinés à s'exécuter sur un réseau de stations hétérogène, en anglais *Heterogeneous Network Of Workstations*, ou *HNOW*. Un réseau hétérogène est la machine parallèle "du pauvre". L'idée est d'utiliser conjointement les machines les plus lentes et les machines les plus récentes.

Dans un premier temps (Paragraphe 7.2), nous nous restreignons à un problème simple d'équilibrage de charge : étant données M tâches à gros grains, indépendantes, et p ressources de puissance de calcul respectivement $\frac{1}{t_1}, \frac{1}{t_2}, \ldots, \frac{1}{t_p}$, déterminer combien de tâches allouer à chaque processeur afin que le temps d'exécution global soit le plus petit possible ? Nous utiliserons ces résultats pour paralléliser de l'implémentation sur réseau hétérogène *unidimensionnel* des deux algorithmes de balayage d'image et de factorisation LU étudiés aux Paragraphe 5.4 et 5.5 respectivement.

Autant l'équilibrage de charge s'avère simple pour une architecture 1D, autant il s'avère difficile (NP-complet) pour une architecture 2D, comme le montre le Paragraphe 7.3 où nous retrouvons notre produit de matrices sur grille bi-dimensionnelle.

Heureusement, les grilles de processeur ne sont plus à la mode ! Il y en a eu beaucoup entre l'Illiac64 et le Paragon, mais la tendance est maintenant aux réseaux d'interconnexion reconfigurables pour les machines dédiées, et au simple Ethernet pour les réseaux de stations. Le chapitre se conclut (Paragraphe 7.4) avec l'étude du produit de matrices sur un réseau hétérogène : l'équilibrage de charge redevient facile, puisqu'on peut toujours configurer le réseau en un anneau virtuel, mais l'impact des communications doit être pris en compte. NP-complétude et heuristiques sont encore au menu.

7.2 Equilibrage de charge 1D

L'utilisation efficace de ressources hétérogènes pour l'exécution d'une application, a fait l'objet d'une attention importante ces dernières années. Lorsque l'ensemble des ressources n'est pas dédié, que la charge de chaque processeur n'est pas prévisible, ou même qu'une panne de l'une des ressources est envisageable, une allocation dynamique des tâches semble incontournable. Les stratégies dynamiques vont du simple paradigme maître-esclave (où chaque processeur choisit dans une file d'attente une nouvelle tâche à exécuter dès que son travail courant est terminé) à des stratégies plus sophistiquées qui, le plus souvent, décident d'une nouvelle distribution des tâches en fonction du travail effectué par chaque processeur durant un passé proche. Pour plus de détails, nous référons le lecteur à l'article de synthèse de Berman [13].

Dans le cas de ressources dédiées, la flexibilité de *l'équilibrage de charge dynamique* rend cette approche attrayante : la charge de chaque machine peut s'auto-réguler et donc s'auto-équilibrer malgré l'hétérogénéité des ressources de calcul. Mais, comme allons le voir dans ce chapitre, la présence de dépendances de données, risque de réduire l'activité de *tous* les processeurs à celle du processeur le plus lent. Ceci motive le recours à une approche statique.

7.2.1 Allocation statique de tâches

Dans le cas d'une allocation statique de noyaux de calcul numérique, la vitesse d'un processeur est évaluée grâce à une mesure préalable du temps d'exécution d'un noyau. Ce noyau devant être, bien entendu, représentatif des calculs effectués par l'application. Notons $t_1, t_2, \ldots, t_p$, les temps mesurés sur chaque processeur, que l'on nomme "*temps de cycle des processeurs*".

Supposons que les tâches à distribuer sur les processeurs correspondent à une quantité de travail identique (*homogénéité des tâches*) et qu'elles soient *atomiques* et indépendantes. Nous nommons désormais cette entité de base *un atome*. Si le nombre d'atomes est petit, l'équilibrage de charge n'est pas trivial : intuitivement, le nombre de tâches à distribuer par processeur (notons le $c_1, c_2, \ldots, c_p$) doit être inversement proportionnel à t_i. Formellement, il faut que

$$c_i \times t_i = Constante = K$$

En d'autres termes, s'il y a B atomes à distribuer,

$$c_i = \frac{\frac{1}{t_i}}{\sum_{k=1}^{p} \frac{1}{t_k}} \times B$$

Bien sûr, si M est un multiple de $C = \operatorname{ppcm}(t_1, t_2, \ldots, t_p) \sum_{i=1}^{p} \frac{1}{t_i}$, l'équilibrage de charge est parfait. En contrepartie, si B est petit, et que B n'est pas multiple de C, on voit bien que la formule ci-dessus ne fournit pas un renseignement assez précis (car c_i doit être un entier). En particulier, il n'est pas clair, pour un processeur P_i tel que $c_i < 1$, que l'on doive lui allouer un atome ou non. En fait, l'allocation optimale est obtenue à l'aide de l'algorithme de la Figure 7.1. Le principe de cet algorithme a déjà été utilisé dans la littérature notamment pour résoudre le problème dual d'allocation de tâches hétérogènes sur un ensemble homogène de processeurs [51].

Distribue$(B, t_1, t_2, \ldots, t_p)$
{ Initialisation : calcule c_i tels que $c_i \times t_i \approx Constante$ et $c_1 + c_2 + \ldots + c_p \leq B$ }
do $i = 1, p$
  $c_i = \left\lfloor \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times B \right\rfloor$
{ Incrémenter itérativement les c_i qui minimisent le temps d'exécution tant que $\sum_{k=1}^{p} c_i < B$}
while $\sum_{i=1}^{p} c_i < B$ do
  trouve $k \in \{1, \ldots, p\}$ tel que $t_k \times (c_k + 1) = \min\{t_i \times (c_i + 1)\}$
  $c_k = c_k + 1$
Retourne$(c_1, c_2, \ldots, c_k)$

FIG. 7.1 – Allocation statique optimale de B atomes indépendants sur p processeurs de temps de cycle respectifs $t_1, t_2, \ldots, t_p$.

Proposition 13 *L'algorithme de la Figure 7.1 fournit l'allocation optimale.*

Preuve Considérons une allocation optimale notée $o_1, o_2, \ldots, o_p$. Soit j tel que $\forall i \in \{1, \ldots, p\}$, on a $T_{exe} = o_j t_j \geq o_i t_i$. Afin de prouver que l'algorithme est correct, nous prouvons l'invariant

$$(C) : \forall i \in \{1, \ldots, p\},\, c_i t_i \leq o_j t_j$$

Après l'étape d'initialisation, $c_i \leq \frac{\frac{1}{t_k}}{\sum_{k=1}^{p} \frac{1}{t_k}} \times B$. Comme, $B = \sum_{k=1}^{p} o_k \leq o_j t_j \times \sum_{k=1}^{p} \frac{1}{t_k}$, on a $c_i t_i \leq \frac{B}{\sum_{k=1}^{p} \frac{1}{t_k}} \leq o_j t_j$, et la condition (C) est vérifiée.

Le fait que (C) soit vérifiée *après chaque incrémentation* est prouvé par récurrence : supposons pour cela, qu'à une étape donnée, c_k soit incrémenté. Avant cette étape, $\sum_{i=1}^{p} c_i < B$, ainsi il existe $k' \in \{1, \ldots, p\}$ tel que $c_{k'} < o_{k'}$. On a $t_{k'}(c_{k'} + 1) \leq t_{k'} o_{k'} \leq t_j o_j$, et le choix de k implique que $t_k(c_k + 1) \leq t_{k'}(c_{k'} + 1)$. La condition (C) est donc vérifiée après cette étape. Finalement, l'allocation ainsi obtenue $(c_1, c_2, \ldots, c_p)$ est optimale car $\max\{c_i \times t_i\} \leq o_j t_j = \max(o_i t_i)$. ■

Comme après la phase d'initialisation $c_1 + c_2 + \ldots + c_p \geq B - p$, l'algorithme est composé d'au plus p étapes, sa complexité est donc $O(p^2)$, et peut être réduite en utilisant une structure de données *ad-hoc*.

7.2.2 Algorithme incrémental

Si nous souhaitons allouer B atomes indépendants, nous pouvons utiliser l'algorithme précédent. Mais si nous voulons connaître l'allocation optimale pour tout nombre d'atomes compris entre 1 et B, il est plus efficace d'utiliser un autre algorithme, dont le principe est basé sur la programmation dynamique.

Cet algorithme, dit incrémental, retourne en effet la solution optimale pour tout nombre d'atomes de 1 à B, ainsi que la distribution associée. Afin d'illustrer son principe, considérons un exemple avec 3 processeurs de temps de cycle respectifs $t_1 = 3$, $t_2 = 5$ et $t_3 = 8$. La Table 7.1 résume l'allocation fournie par l'algorithme jusqu'à $B = 10$. Dans ce tableau, l'intitulé "processeur sélectionné" représente l'indice du processeur choisi pour détenir l'atome suivant. A chaque étape, le processeur sélectionné est choisi afin que le coût de l'allocation globale obtenue soit minimal. Notons c_i le nombre d'atomes alloués au processeur i. Dans ce cas, le coût de l'allocation $\mathcal{C} = (c_1, c_2, \ldots, c_p)$ vaut $\max_{1 \leq i \leq p} c_i t_i$. Ainsi le coût moyen d'exécution d'un atome vaut $t_{para} = \frac{\max_{1 \leq i \leq p} c_i t_i}{\sum_{i=1}^{p} c_i}$.

Par exemple, à la quatrième étape, c'est à dire lors de l'allocation du quatrième atome, l'algorithme se base sur l'allocation obtenue pour 3 atomes, c'est à dire $(c_1, c_2, c_3) = (2, 1, 0)$. A quel processeur P_i doit être assigné le quatrième atome, en d'autres termes, quel c_i doit être incrémenté ? Il y a trois possibilités $(c_1 + 1, c_2, c_3) = (3, 1, 0)$, $(c_1, c_2 + 1, c_3) = (2, 2, 0)$ et $(c_1, c_2, c_3 + 1) = (2, 1, 1)$, de coûts respectifs $\frac{9}{4}$ (P_1 termine en dernier), $\frac{10}{4}$ (P_2 termine en dernier), et $\frac{8}{4}$ (P_3 termine en dernier). Ainsi, la troisième solution est sélectionnée, correspondant au résultat $(c_1, c_2, c_3) = (2, 1, 1)$.

De manière plus formelle, l'algorithme est décrit à la Figure 7.2. Sa complexité est $O(pB)$ où p est le nombre de processeurs et B le nombre d'atomes à distribuer.

Proposition 14 *L'algorithme incrémental retourne la distribution optimale pour tout sous-ensemble d'atomes* $[1, m]$ *où* $m \leq B$.

Preuve La preuve peut être calquée sur celle de la proposition précédente. Il suffit de modifier légèrement l'invariant à prouver :

$$(C_m) : \forall i \in \{1, \ldots, p\},\, c_i t_i \leq o_{j_m}^{(m)} t_{j_m} = \max_{1 \leq j \leq m} o_j^{(m)} t_j$$

Nombre d'atomes	c_1	c_2	c_3	Coût	Proc. sélectionné	Allocation σ
0	0	0	0		1	
1	1	0	0	3	2	$\sigma[1] = 1$
2	1	1	0	2.5	1	$\sigma[2] = 2$
3	2	1	0	2	3	$\sigma[3] = 1$
4	2	1	1	2	1	$\sigma[4] = 3$
5	3	1	1	1.8	2	$\sigma[5] = 1$
6	3	2	1	1.67	1	$\sigma[6] = 2$
7	4	2	1	1.71	1	$\sigma[7] = 1$
8	5	2	1	1.87	2	$\sigma[8] = 1$
9	5	3	1	1.67	3	$\sigma[9] = 2$
10	5	3	2	1.6		$\sigma[10] = 3$

TAB. 7.1 – Différentes étapes de l'algorithme incrémental avec 3 processeurs de temps de cycle respectifs $t_1 = 3$, $t_2 = 5$, et $t_3 = 8$.

```
Distribue(B, t_1, t_2, ..., t_p)
{ Initialisation : aucune tâche à distribuer. m = 0 }
do i = 1, p
   c_i = 0
{ Construit itérativement l'allocation σ}
do m = 1, B
   trouve k ∈ {1, ..., p} tel que t_k × (c_k + 1) = min{t_i × (c_i + 1)}
   c_k = c_k + 1
   σ[m] = k
Retourne(σ, c_1, c_2, ..., c_k)
```

FIG. 7.2 – Distribution incrémentale de B atomes sur p processeurs de temps de cycle respectifs $t_1, t_2, \ldots t_p$, optimale pour tout sous-ensemble d'atomes $[1, m]$, $m \leq B$.

où $\mathcal{O}^{(m)} = (o_1^{(m)}, o_2^{(m)}, \ldots, o_p^{(m)})$ est une solution optimale au problème avec m atomes, et j_m l'indice du processeur qui, pour une telle allocation, termine son travail en dernier. Par récurrence, à l'étape m, on a (C_m) et comme trivialement (le temps total d'exécution ne peut qu'augmenter si l'on rajoute un atome à exécuter) $o_{j_m}^{(m)} t_{j_m} \leq o_{j_{m+1}}^{(m+1)} t_{j_{m+1}}$, avant l'étape $m+1$ on a donc (C_{m+1}). L'étape $m+1$ étant identique à toute étape de l'algorithme de la Figure 7.1, la preuve de la proposition précédente montre que (C_{m+1}) est maintenu après l'étape $m+1$. ■

7.2.3 Application au balayage d'image

Nous reprenons ici l'algorithme de balayage d'image présenté au Paragraphe 5.4. Plus généralement, nous cherchons à paralléliser un calcul du type représenté à la Figure 7.3. Le domaine de calcul est un rectangle, et les dépendances entre tuiles de calcul (les atomes) sont données par les vecteurs de dépendance

$$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}$$

Sans anticiper sur le Chapitre 8, cela signifie que le calcul d'une tuile ne peut pas débuter avant la fin de l'exécution de ses voisins de gauche et du bas.

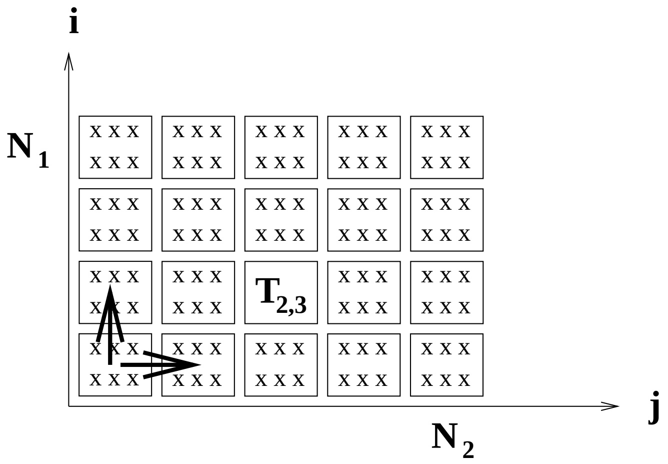


FIG. 7.3 – Allocation de tuiles de calcul.

Nous cherchons à allouer les colonnes de tuile à p processeurs de temps de cycle $t_1, t_2, \ldots, t_p$. Dans le cas homogène, une allocation cyclique est adaptée pour équilibrer la charge. Notons qu'une stratégie dynamique gloutonne conduirait au même résultat dans ce cas : si tous les processeurs opèrent à la même vitesse, ils se verront allouer une nouvelle colonne à tour de rôle.

Dans le cas hétérogène, discutons l'allocation de N colonnes par tranches de B colonnes consécutives, en prenant à nouveau l'exemple à 3 processeurs avec $t_1 = 3$, $t_2 = 5$, et $t_3 = 8$. Le temps séquentiel est obtenu en confiant toutes les colonnes au processeur P_1, donc $T_{seq} \approx 3N$. Une allocation gloutonne conduit à une solution cyclique de période B, et est illustrée à la Figure 7.4 pour $B = 10$. En régime permanent, les calculs progressent au rythme du processeur le plus lent, P_3, et donc $T \approx \frac{8}{3}N$. Notons que toute solution dynamique gloutonne est condamnée à opérer au rythme du processeur le plus lent.

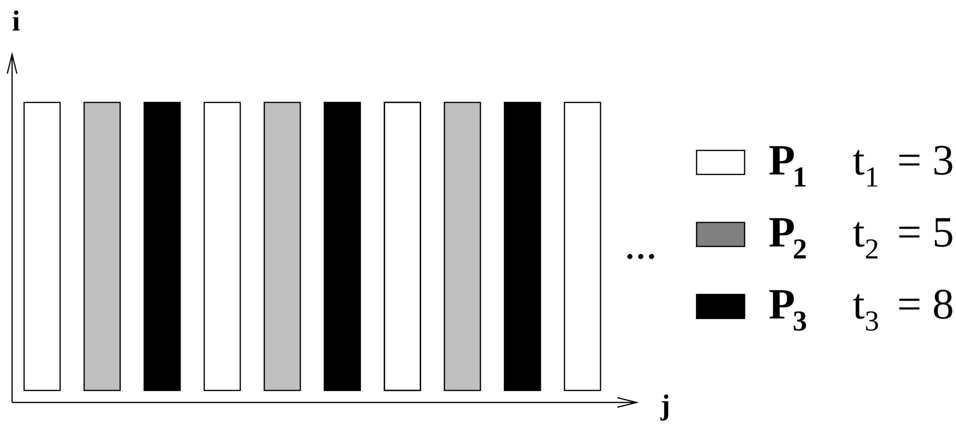


FIG. 7.4 – Allocation cyclique (gloutonne) des tuiles de calcul.

L'algorithme de la Figure 7.1 conduit à une solution statique bien meilleure. En effet pour $B = 10$, disons, nous obtenons $c_1 = 5$, $c_2 = 3$ et $c_3 = 2$. L'idée est de trier les quantités $c_i t_i$ et d'allouer, dans cet ordre, c_i colonnes consécutives au processeur P_i. Chaque processeur va progresser ligne par ligne plutôt que directement en colonne : P_i effectuera c_i tâches sur la même ligne avant d'attaquer la ligne supérieure. Grâce au tri des $c_i t_i$, le temps d'exécution d'une ligne de c_i tâches est croissant avec i, et aucun temps d'attente n'est crée en régime permanent. La solution est décrite à la Figure 7.4. On obtient $T \approx 1.6N$. Notons que la meilleure répartition asymptotique de la charge conduit à $T_{opt} \approx \frac{120}{79} N = 1.52N$. En effet si $B = 79$, on trouve $c_1 = 40$, $c_2 = 24$ et $c_3 = 15$: pour ces valeurs, $c_1 t_1 = c_2 t_2 = c_3 t_3$, et l'équilibrage de charge est parfait.

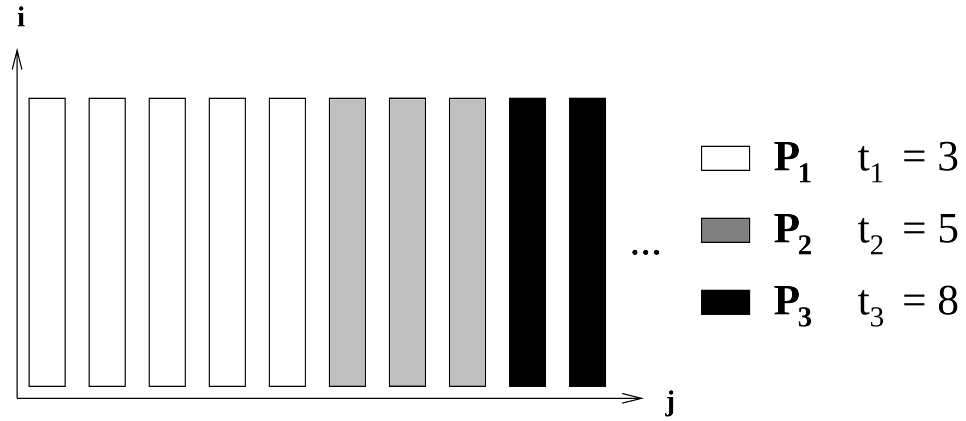


FIG. 7.5 – Allocation statique périodique des tuiles de calcul.

Comme indiqué au début de ce paragraphe, cette approche s'applique bien à l'algorithme de balayage d'image du Chapitre 5. Distribuant les lignes à la place des colonnes, on obtient le schéma d'exécution de la Figure 7.6, avec 3 processeurs de temps de cycle $t_1 = 1$, $t_2 = 2$ and $t_3 = 4$, et une période $B = 7$ calculée pour un équilibrage de charge optimal.

P_1	0	1	3	6	10	14	18	22	26	30	...
P_1	2	4	7	11	15	19	23	27	31	...	
P_1	5	8	12	16	20	24	28	32	...		
P_1	9	13	17	21	25	29	33	...			
P_2	14	18	22	26	30	34	...				
P_2	20	24	28	32	36	...					
P_3	26	30	34	38	...						
...	...										

FIG. 7.6 – Allocation statique périodique pour le balayage d'image, $t_1 = 1$, $t_2 = 2$, $t_4 = 4$.

7.2.4 Application à la décomposition LU

L'algorithme de décomposition LU fonctionne schématiquement de la manière suivante : à chaque étape, le processeur qui possède le bloc *pivot*, le *factorise* et le diffuse à tous les autres processeurs qui *mettent* alors *à jour* les colonnes restantes. A l'étape suivante, c'est le bloc des b colonnes suivantes qui devient à son tour le pivot, et le calcul progresse ainsi (voir le Paragraphe 5.5.

Comme la plus grande partie du temps de calcul corresponde à la mise à jour, l'idée est de trouver une allocation statique qui équilibre au mieux les tâches lors de chaque étape de mise à jour. Considérons la première étape. Après la factorisation du premier bloc, chacune des mises à jour d'un bloc, effectuée par un processeur, est un noyau de calcul indépendant : si la matrice est de taille $(nb) \times (nb)$, il y a $B = (n-1)$ noyaux de calcul à allouer. Nous pourrions utiliser l'algorithme de la Figure 7.1 pour équilibrer les charges, mais la taille du domaine de travail diminue au fur et à mesure que la décomposition progresse. A la seconde étape, le nombre de blocs à mettre à jour est $(n-2)$. Si l'on souhaite à nouveau équilibrer ces charges, les données doivent être redistribuées, ce qui implique ainsi, entre chaque étape, un nombre important de communications. De plus, dans l'optique du développement d'une librairie d'algèbre linéaire distribuée hétérogène, l'allocation doit a priori être identique au début et à la fin de la décomposition.

Nous cherchons donc une distribution statique des données fournissant un équilibrage des charges de mise à jour, commun à toutes les étapes. En d'autres termes, nous cherchons une distribution de B tâches sur p processeurs de temps de cycle respectifs $t_1, t_2, \dots t_p$ telle que pour tout $i \in \{2, \dots, B\}$ le nombre de blocs de $\{i, \dots, B\}$ que possède chaque processeur P_j, soit (approximativement) inversement proportionnel à son temps de cycle t_j. L'algorithme incrémental permet d'obtenir une solution optimale à ce problème. L'idée est d'allouer périodiquement, sous forme d'un motif de largeur B, les blocs de colonnes aux processeurs, comme indiqué Figure 7.7. B est un paramètre pouvant valoir n, pour une matrice de taille $(nb) \times (nb)$ découpée en blocs de taille $b \times b$, mais pouvant aussi être plus petit si l'on souhaite l'allocation plus régulière. En effet, le recouvrement des communications avec les calculs n'est possible que grâce à un processus de pipeline entre les colonnes, processus brisé lorsque l'allocation est trop irrégulière. Supposons ici, sans perte de généralité, que B vaille n, c'est à dire que le motif ait la même largeur que la matrice et ne soit donc pas répété.

Pour définir le motif, nous utilisons le résultat de l'algorithme incrémental *en sens inverse*, c'est à dire que le bloc de colonnes d'indice $1 \leq k \leq B$ sera alloué au processeur $\sigma(B-k+1)$. En effet, l'algorithme incrémental retourne l'allocation optimale pour tout sous-ensemble $[1,i]$ de $[1,B]$ où $i \in [1,B]$, alors que la distribution recherchée doit équilibrer tout sous-ensemble $[i,B]$ de $[1,B]$ où $i \in [1,B]$. La Table 7.1 lu de bas en haut donne l'allocation obtenue par notre algorithme pour $B = 10$ avec 3 processeurs de temps de cycle $t_1 = 3$, $t_2 = 5$ et $t_3 = 8$. Ainsi, le motif obtenu vaut $(P_3P_2P_1P_1P_2P_1P_3P_1P_2P_1)$.

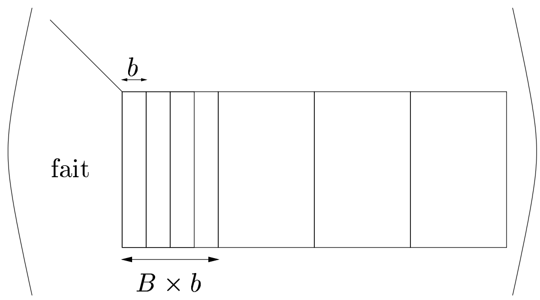


FIG. 7.7 – Allocation périodique utilisant un motif de largeur B.

7.3 Equilibrage de charge 2D

Dans ce paragraphe, nous montrons que l'équilibrage de charge est bien plus difficile en deux dimensions que sur une seule. Nous utilisons à cet effet l'algorithme du produit de matrices sur grille torique, algorithme qui dont diverses variantes ont été décortiquées au Paragraphe 6.5. Nous commençons par en résumer les caractéristiques.

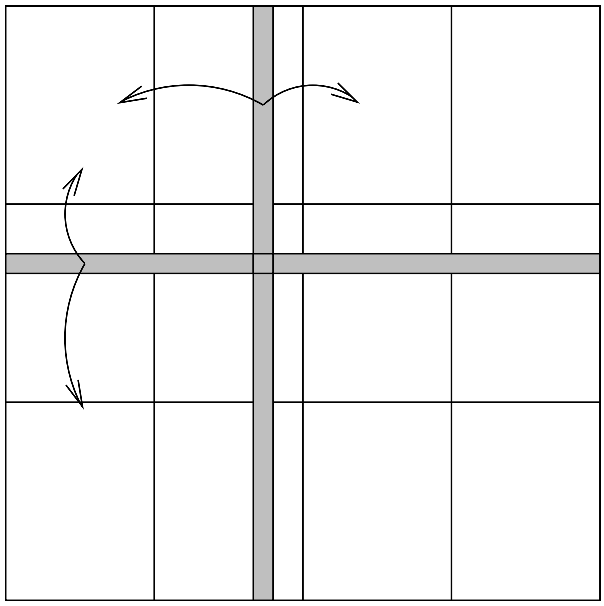


FIG. 7.8 – Produit de matrices sur une grille homogène de taille 3×4.

7.3.1 Multiplication de matrices sur une grille homogène

Considérons le produit de matrices carrées $C = A \times B$ de taille $n \times n$, destiné à s'exécuter sur une grille de processeurs de taille $p \times q$. L'algorithme utilisé dans ScaLAPACK [16] est plus simple que les variantes décrites au Paragraphe 6.5 : il fait appel à une double diffusion horizontale et verticale. Le principe est le suivant : supposons dans un premier temps que $p = q = n$. Dans ce cas, les trois matrices partagent le même schéma d'allocation sur la grille 2D : le processeur $P_{i,j}$ stocke les scalaires $a_{i,j}$, $b_{i,j}$ et $c_{i,j}$. Ainsi à chaque étape k,

- chaque processeur $P_{i,k}$ (pour tout $i \in \{1,..,p\}$) diffuse horizontalement $a_{i,k}$ aux processeurs $P_{i,1:q}$.
- chaque processeur $P_{k,j}$ (pour tout $j \in \{1,..,q\}$) diffuse verticalement $b_{k,j}$ aux processeurs $P_{1:p,j}$.
- de telle manière que chaque processeur $P_{i,j}$ peut alors indépendamment exécuter $c_{i,j} = a_{i,k} \times b_{k,j} + c_{i,j}$.

Cet algorithme, appelé *outer product algorithm* dans [1, 33, 46], est utilisé dans ScaLAPACK parce qu'il s'adapte très facilement au cas de matrices et grilles rectangulaires, et parce qu'il ne nécessite aucune redistribution initiale des données. ScaLAPACK utilise en fait une version pavée de cet algorithme : chaque scalaire dans la description ci-dessus doit être remplacé par un bloc de taille $b \times b$. Une matrice de taille $nb \times nb$ sera donc à partir de maintenant considérée virtuellement comme une matrice de taille $n \times n$ où un élément atomique est un bloc de taille $b \times b$. De plus, généralement, le nombre de blocs est bien supérieur à pq, et ceux-ci sont distribués cycliquement sur la grille de processeurs, de telle manière qu'à chaque étape un processeur soit responsable de la mise à jour de plusieurs blocs. En d'autres termes, ScaLAPACK utilise une distribution $CYCLIC(b)$ dans chacune des directions, identique pour chacune des trois matrices. L'algorithme est illustré à la Figure 7.8.

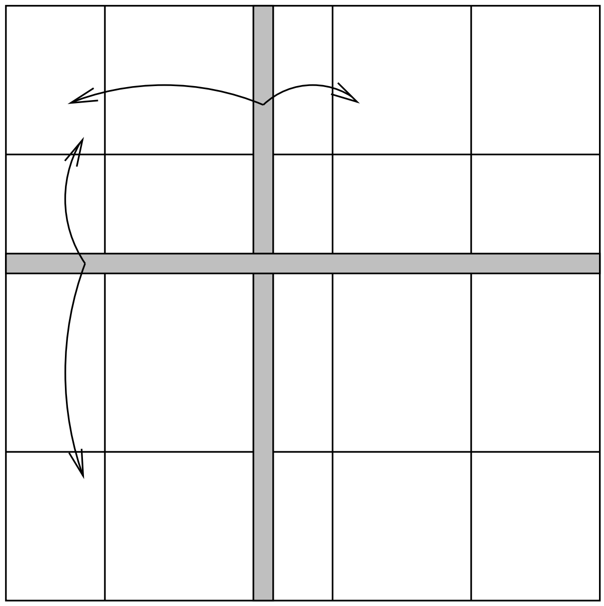


FIG. 7.9 – Produit de matrices sur une grille hétérogène de taille 3×4.

7.3.2 Multiplication de matrices sur une grille hétérogène

Il est facile d'adapter l'algorithme précédent à une grille hétérogène. Il suffit pour cela d'allouer des rectangles de tailles différentes aux processeurs, en fonction de leur vitesse relative. Le principe de l'algorithme reste inchangé : à chaque étape, deux diffusions ont lieu, une horizontale pour A et une verticale pour B, et chaque processeur est responsable de la mise à jour des blocs de C au sein du rectangle qui lui est alloué. Comme dans le cas 1D, l'intuition est d'allouer à chaque processeur un rectangle de taille proportionnelle à sa vitesse relative. L'algorithme est illustré à la Figure 7.9.

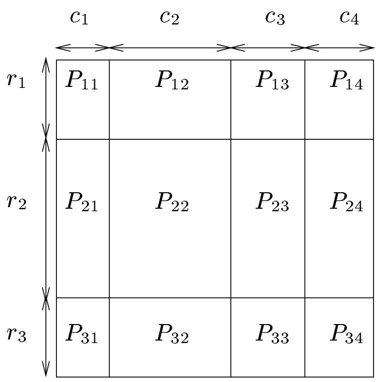


FIG. 7.10 – Allocation des matrices sur une grille de processeurs de taille 3×4.

Formellement, considérons $p \times q$ processeurs $P_{i,j}$ de temps de cycle $t_{i,j}$ où $1 \leq i \leq p$ et $1 \leq j \leq q$. Supposons que soit assigné au processeur P_{ij} un rectangle de taille $r_i \times c_j$, comme indiqué à la Figure 7.10.

Avant de se lancer dans l'évaluation analytique des propriétés d'une allocation, soulignons qu'il n'est pas toujours possible de réaliser un équilibrage de charge parfait, à cause de la contrainte géométrique de la grille. La Figure 7.11 montre sur un petit exemple que pour avoir un équilibrage parfait, il faut et il suffit que la matrice $T = (t_{ij}$ soit de rang 1. C'est facile à voir : si la matrice T est de rang 1, on pose $r_1 = 1$, $c_1 = 1/t_{11}$, puis $r_i = 1/(t_{i1}c_1)$ $(i \geq 2)$ et $c_j = 1/t_{1j}$ $(j \geq 2)$. On vérifie alors que $r_i t_{ij} c_j = 1$ pour $i, j \geq 2$ car le déterminant $\begin{vmatrix} t_{11} & t_{1j} \\ t_{i1} & t_{ij} \end{vmatrix}$ est nul. Réciproquement, si $r_i t_{ij} c_j = 1$ pour $1 \leq i \leq p, 1 \leq j \leq q$, alors la matrice T est de rang 1 (faire la même construction à l'envers).

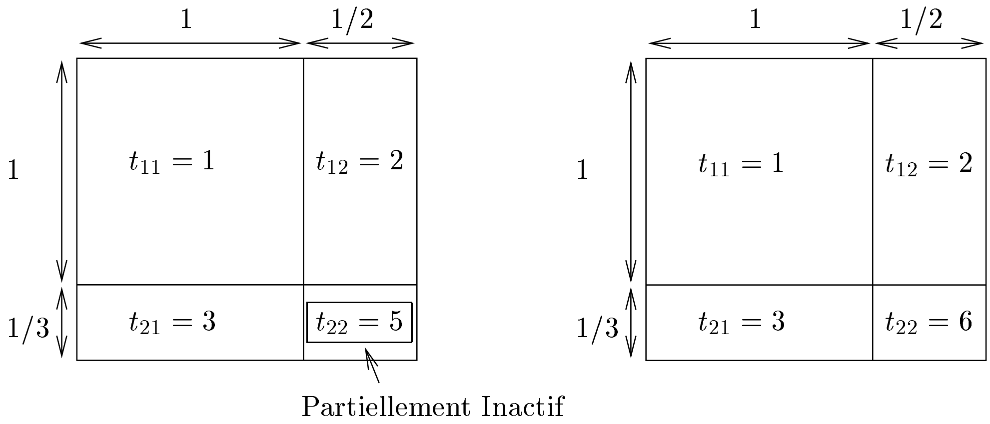


FIG. 7.11 – L'équilibrage parfait n'est possible que pour des matrices de rang 1.

Il y a alors plusieurs manières équivalentes d'évaluer la valeur de cette allocation :

– Chaque processeur $P_{i,j}$ travaillant durant $r_i \times c_j \times t_{ij}$ unités de temps, le temps total d'exécution du rectangle de taille $\sum_{i=1}^{p} r_i \times \sum_{j=1}^{q} c_j$ vaut donc
$$t_{exe} = \max_{i,j}\{r_i \times t_{ij} \times c_j\}.$$
Cette fonction de $r = (r_1, \ldots, r_p)$ et $c = (c_1, \ldots, c_q)$ étant bilinéaire, le coût d'une allocation peut être normalisé à un motif de taille 1×1. On obtient le problème d'optimisation suivant :
$$\textbf{Objectif } Obj_1 : \min_{(\sum_i r_i = 1; \sum_j c_j = 1)} \max_{i,j}\{r_i \times t_{ij} \times c_j\}$$
A partir d'une solution au problème Obj_1, on déduit l'allocation recherchée en multipliant les r_i et les c_j par n et en ajustant à des entiers proches.

– L'expression "duale" de ce problème d'optimisation est la suivante : quelle est la taille de motif maximum pouvant être traité en une unité de temps par l'ensemble des processeurs ? On aboutit au problème d'optimisation équivalent suivant :
$$\textbf{Objectif } Obj_2 : \max_{r_i \times t_{ij} \times c_j \leq 1} \{(\sum_i r_i) \times (\sum_j c_j)\}$$

– Dans cette expression il y a $p + q$ variables r_i et c_j alors qu'il n'y a que $p + q - 1$ degrés de liberté : si on multiplie les r_i par un facteur λ et que l'on divise les c_j par ce même facteur, on obtient une solution équivalente. Ainsi, on peut imposer, sans perte de généralité, $r_1 = 1$.

Le problème qui nous concerne est en fait **bien plus complexe** que l'optimisation des problèmes Obj_1 ou Obj_2. En effet, il faut minimiser l'une de ces expressions *pour tout agencement* de pq processeurs en une grille de taille $p \times q$. La position des processeurs dans la grille n'est pas une donnée du problème : toutes les permutations sont possibles, et il faut chercher la meilleure, i.e. celle pour laquelle la solution optimale de Obj_2 est maximale. Ce problème est NP-complet, comme on va le voir au paragraphe suivant.

7.3.3 Difficulté de l'équilibrage de charge 2D

Considérons $P = p^2$ processeurs $P_1, P_2, \ldots, P_P$ de temps de cycle $t_1, t_2, \ldots, t_P$. Le problème est d'agencer ces processeurs sur une grille bi-dimensionnelle de taille $p \times p$, de telle manière que l'allocation ainsi construite permette une exécution du produit de matrices en un temps minimal. Si on oublie notre motivation, celle du produit de matrices, on a le problème abstrait suivant :

Définition 13 *MAX-GRID(s) : soient p^2 nombres réels positifs $s_1, \ldots, s_{p^2}$, trouver*

$$(r_1, \ldots, r_p, c_1, \ldots, c_p) \text{ et un arrangement (bijection) } \sigma \text{ de } [1,p] \times [1,p] \text{ vers } [1,p^2]$$

tels que

$$\forall (i,j) \in [1,p] \times [1,p], \quad r_i c_j \leq s_{\sigma(i,j)} \text{ et } (\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j) \text{ soit maximal.}$$

On a simplement remplacé les temps de cycle par les vitesses : $s_i = \frac{1}{t_i}$ est la vitesse relative du processeur P_i ;. σ représente l'arrangement des p^2 processeurs sur la grille de taille $p \times p$, et r_i et c_j sont les variables utilisées dans Obj_2. La condition $r_i t_{\sigma(i,j)} c_j \leq 1$ équivaut bien à $r_i c_j \leq s_{\sigma(i,j)}$

Le problème MAX-GRID permet de modéliser l'équilibrage de charge pour une large classe d'algorithmes sur la grille, tous ceux pour lesquels une matrice ou une image est partitionnée entre les divers processeurs, chacun étant responsable de la mise à jour du sous-rectangle qui lui a été alloué. Le problème de décision associé au problème d'optimisation MAX-GRID est le suivant :

Définition 14 *MAX-GRID(s,K) : soient p^2 nombres réels positifs $s_1, \ldots, s_{p^2}$ et un nombre réel positif K, existe-t-il*

$$(r_1, \ldots, r_p, c_1, \ldots, c_p) \text{ ainsi qu'un arrangement (bijection) } \sigma \text{ de } [1,p] \times [1,p] \text{ vers } [1,p^2]$$

tels que

$$\forall (i,j) \in [1,p] \times [1,p], \quad r_i c_j \leq s_{\sigma(i,j)} \text{ et } (\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j) \geq K \text{ ?}$$

Théorème 10 *MAX-GRID(s,K) est NP-complet.*

Proof La preuve est longue et assez technique. Elle nécessite plusieurs lemmes. L'idée est de réduire le problème à celui de 2-Partition :

Lemme 13

$$2P\text{-}eq \leq_P MAX\text{-}GRID,$$

où 2P-eq est défini comme suit :

Définition 15 *2-Partition-Equal (2P-eq)*
Soit un ensemble de m entiers $\mathcal{A} = \{a_1, \ldots, a_m\}$, existe-t-il une partition de $\{1, \ldots, m\}$ en deux ensembles $\mathcal{A}_1$ et $\mathcal{A}_2$ tels que

$$\sum_{i \in \mathcal{A}_1} a_i = \sum_{i \in \mathcal{A}_2} a_i \text{ et } card(\mathcal{A}_1) = card(\mathcal{A}_2) \text{ ?}$$

Comme 2P-eq est NP-complet [34], le Lemme 13 prouve le Théorème 10.

(a) Réduction : 2P-eq $\leq_P$ MAX-GRID(s,K) Considérons une instance arbitraire du problème de 2-Partition-Equal, i.e. la donnée d'un ensemble $\mathcal{A} = \{a_1, \ldots, a_{2n}\}$ composé de $2n$ entiers. Il nous faut transformer polynômialement cette instance en une instance du problème MAX-GRID ayant une solution si et seulement si l'instance initiale de 2-Partition-Equal admet une solution.

Définissons pour cela $\{b_1, \ldots, b_{2n}\}$ par $\forall i, b_i = (a_i + 2n \max_k a_k)$. Ainsi, $2n \max a_i \leq b_i \leq (2n+1) \max a_i$. On considère alors l'instance du problème MAX-GRID (notée abusivement MAX-GRID$(b_1, \ldots, b_{2n}, K)$) où

$$\left\{\begin{array}{rcl} K &=& (4n \max a_i)^2 + \sum_{i=1}^{2n} b_i + \frac{(\sum_{i=1}^{2n} b_i)^2}{4(4n \max a_i)^2}, \\ s_1 &=& (4n \max a_i)^2, \\ s_{i+1} &=& b_i, \quad \forall i,\ 1 \leq i \leq 2n, \\ s_i &=& 1, \quad \forall i,\ 2n+2 \leq i \leq (n+1)^2. \end{array}\right.$$

Par la suite, nous allons montrer qu'une solution à ce problème est nécessairement de la forme schématisée dans la Figure 7.12, où σ, restreint à l'ensemble $([2, n+1], 1) \bigcup (1, [2, n+1])$, définit une bijection avec $[2, 2n+1]$ et

$$\left\{\begin{array}{rrcl} & r_1 c_1 &=& (4n \max a_i)^2, \\ \forall i,\ \ 2 \leq i \leq n+1, & r_i &=& \frac{b_{\sigma(i,1)}}{c_1}, \\ \forall j,\ \ 2 \leq j \leq n+1, & c_j &=& \frac{b_{\sigma(1,j)}}{r_1}, \end{array}\right.$$

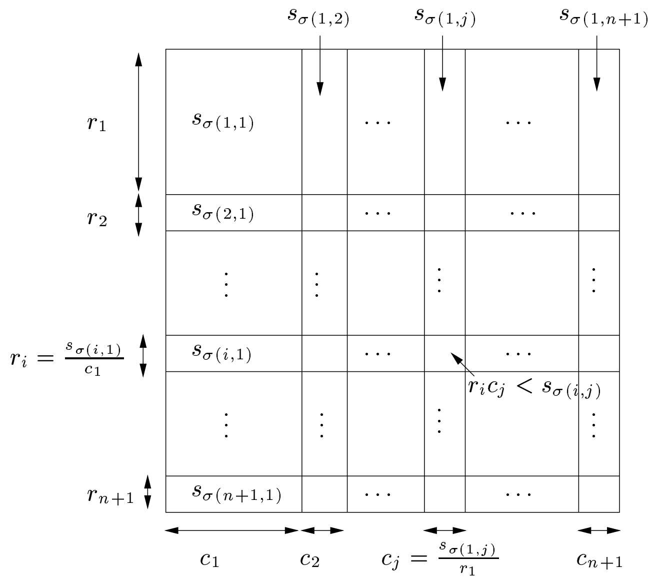


FIG. 7.12 – Solution de MAX-GRID$(b_1, b_2, \ldots, b_{2n}, K)$.

Ainsi, nous pouvons vérifier que

$$(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j) \geq K \iff \sum_{2}^{n+1} b_{\sigma(i,1)} = \sum_{2}^{n+1} b_{\sigma(1,j)}$$

De manière intuitive, comme s_1 est largement plus grand que les autres aires et que $b_i \gg 1$, il faut (à une permutation près sur les lignes et sur les colonnes) positionner s_1 dans le coin en haut à gauche et remplir la première ligne et la première colonne avec les b_i. Il est donc possible de saturer chacun de ces processeurs en prenant $r_1 c_1 = s_1$, $r_i = \frac{s_{\sigma(i,1)}}{c_1}$ et $c_j = \frac{s_{\sigma(1,j)}}{r_1}$. Ensuite, l'aire de la grille est maximale lorsqu'elle est équilibrée, c'est à dire si les b_i vérifient 2P-eq. Ces différents points constituent les étapes successives de la preuve ci-dessous.

Lemme 14 *Si* $(\sum_{i=1}^{n+1} r_i)(\sum_{j=1}^{n+1} c_j) \geq K$, *alors l'aire assignée au processeur de vitesse* s_1 *a pour valeur minimum* $11n^2 \max {a_i}^2$.

Preuve A une permutation près, supposons que $\sigma^{-1}(1) = (1,1)$, c'est à dire que l'aire assignée au processeur de vitesse s_1 vaille $r_1 c_1$. Comme $\sum_{i \geq 2} s_i \leq 5n^2 \max {a_i}^2$, et que

$$(\sum_{i=1}^{p} r_i)(\sum_{j=1}^{p} c_j) \leq r_1 c_1 + \sum_{i \geq 2} s_i,$$

on a

$$r_1 c_1 \geq 11n^2 \max {a_i}^2.$$

■

Lemme 15 *Soit* σ *une bijection de* $[1, n+1] \times [1, n+1]$ *vers* $[1, (n+1)^2]$ *telle que* $\sigma(1,1) = 1$. *Supposons que* $r_1 c_1 \geq 11n^2 \max {a_i}^2$ *(Lemme 14), alors* $(\sum_{i=1}^{n+1} r_i)(\sum_{j=1}^{n+1} c_j)$ *est maximal si et seulement si*

$$\begin{cases} \forall i \geq 2, & r_i c_1 = s_{\sigma(i,1)}, \\ \forall j \geq 2, & c_j r_1 = s_{\sigma(1,j)}. \end{cases}$$

Preuve Par définition,

$$\begin{cases} \forall i \geq 2, & r_i c_1 \leq s_{\sigma(i,1)} \\ \forall j \geq 2, & c_j r_1 \leq s_{\sigma(1,j)}, \end{cases}$$

et $(\sum_{i=1}^{n+1} r_i)(\sum_{j=1}^{n+1} c_j)$ est maximal lorsque chaque r_i et chaque c_j est maximal.

De plus, si

$$\begin{cases} \forall i \geq 2, & r_i c_1 = s_{\sigma(i,1)} \\ \forall j \geq 2, & c_j r_1 = s_{\sigma(1,j)} \end{cases}$$

alors toutes les autres conditions $r_i c_j \leq s_{\sigma(i,j)}$ sont automatiquement vérifiées. En effet,

$$\forall (i,j) \neq (1,1), \quad 1 \leq s_{\sigma(i,j)} \leq (2n+1) \max a_i$$

d'où

$$\begin{aligned} r_i c_j &= \frac{s_{\sigma(i,1)} s_{\sigma(1,j)}}{r_1 c_1}, \\ &\leq \frac{(2n+1)^2 \max {a_i}^2}{11n^2 \max {a_i}^2} \\ &\leq 1 \\ &\leq s_{\sigma(i,j)}. \end{aligned}$$

■

Pour un σ donné, la valeur maximale de $(\sum_{i=1}^{n+1} r_i)(\sum_{j=1}^{n+1} c_j)$ vaut

$$S(\sigma) = (r_1 + \frac{\sum_2^{n+1} s_{\sigma(i,1)}}{c_1})(c_1 + \frac{\sum_2^{n+1} s_{\sigma(1,j)}}{r_1}).$$

Notons

$$s = r_1 c_1, \quad S_1 = \sum_2^{n+1} s_{\sigma(i,1)} \text{ et } S_2 = \sum_2^{n+1} s_{\sigma(1,j)}.$$

Alors,

$$S(\sigma) = s + (S_1 + S_2) + \frac{S_1 S_2}{s}.$$

Lemme 16

$$S(\sigma) \geq K \iff (s = (4n \max a_i)^2) \text{ et } (S_1 = S_2 = \frac{\sum b_i}{2})$$

et alors $S(\sigma) \geq K$ si et seulement si il existe une solution au problème de 2P-eq défini ci-dessus.

Preuve Par construction, $S_1 + S_2 \leq \sum b_i$ et donc $S_1 S_2 \leq \frac{(\sum b_i)^2}{4}$. De plus, $S_1 S_2 = \frac{(\sum b_i)^2}{4}$ si et seulement si $S_1 = S_2 = \frac{\sum b_i}{2}$. Ainsi

$$S(\sigma) \leq s + \sum b_i + \frac{(\sum b_i)^2}{4s}.$$

Considérons la fonction g définie par

$$g(s) = s + \frac{(\sum b_i)^2}{4s}.$$

Cette fonction est décroissante puis croissante et admet un minimum en $\frac{\sum b_i}{2}$. Comme

$$\frac{\sum b_i}{2} \ll 11n^2 \max {a_i}^2 \leq s \leq (4n \max a_i)^2,$$

g est maximal quand $s = (4n \max a_i)^2$. En d'autres termes,

$$\begin{cases} S(\sigma) \leq (4n \max a_i)^2 + \sum b_i + \frac{(\sum b_i)^2}{4(4n \max a_i)^2} = K \\ S(\sigma) = K \text{ si et seulement si } (s = (4n \max a_i)^2) \text{ et } (S_1 = S_2 = \frac{\sum b_i}{2}). \end{cases}$$

■

D'où le résultat : MAX-GRID$(b_1, \ldots, b_{2n}, K)$ admet une solution ssi 2P-eq$(b_1, \ldots, b_{2n})$ admet une solution, et donc ssi l'instance initiale de 2P-eq$(a_1, \ldots, a_{2n})$ admet une solution.

(b) Consistance de la transformation effectuée La dernière partie de la preuve consiste à vérifier que l'instance du problème de MAX-GRID s'exprime à l'aide d'un énoncé de taille polynômiale en la taille de l'énoncé de l'instance initiale du problème de 2P-eq.

Lemme 17 *Notons $MAX = \max_k a_k$. Représentons le codage des données a et b par $c(a)$ et $c(b)$. Alors,*

$$\mathit{Taille}(c(b)) = O(\mathit{Taille}(c(a))^2).$$

Preuve

$$\text{Taille}(c(a)) = \sum_k \log(a_k) \geq \log(\text{MAX}) + (n-1)\log(\min_k a_k) \geq (n-1)\log 2 + \log \text{MAX}.$$

$$\text{Taille}(c(b)) = \sum_k \log(b_k) = \sum_k \log(2n\ \text{MAX}(1 + \frac{a_k}{n\text{MAX}})) \leq 1 + n(\log n + \log 2) + n \log \text{MAX}.$$

D'où,

$$\text{Taille}(c(b)) = O(\text{Taille}(c(a))^2).$$

■

Ceci achève la preuve de NP-complétude du problème MAX-GRID. ■

On trouvera dans la thèse de Rastello [58] une heuristique polynômiale, hélas non garantie, pour la résolution de MAX-GRID. L'idée est d'obtenir un placement dont la matrice T est "proche" d'une matrice de rang 1, via une décomposition et un raffinement itératif. Bien compliqué !

7.4 Partitionnement libre sur réseau hétérogène

7.4.1 Contexte

Revenons à notre produit de matrices sur grille homogène, décrit à la Figure 7.8. La généralisation naturelle pour un réseau de stations hétérogène ne devrait pas se limiter aux grilles virtuelles : pourquoi s'imposer cette contrainte architecturale ? On représente Figure 7.13 un schéma d'exécution possible avec 13 processeurs de vitesses différentes. Il n'y a pas d'autre contrainte que de partitionner le carré unité en 13 rectangles de surfaces prescrites, proportionnelles à la vitesse relatives des processeurs. A chaque étape, il y a diffusion horizontale d'un bloc colonne de A et diffusion verticale d'un bloc ligne de B. Chaque processeur (hormis ceux qui ont les données diffusées dans leur mémoire locale) reçoit deux messages, de longueur proportionnelles aux côtés du rectangle qui lui est alloué.

Etant donnés p processeurs, de vitesses respectives $s_1, s_2, \ldots, s_p$, normalisées pour que $\sum_{i=1}^{p} s_i = 1$, il est toujours possible de partitionner le carré unité en p rectangles de surfaces $s_1, s_2, \ldots, s_p$, par exemple en dessinant p bandes horizontales (ou verticales) de largeur idoine : un équilibrage des charges parfait est toujours possible avec une telle solution de type 1D. Quelle est la fonction objective ? On va s'attacher à minimiser le coût des communications effectuées à chaque étape du produit de matrice. On l'a vu, pour chaque processeur P_i, le volume de communication effectué est proportionnel au demi-périmètre (hauteur + largeur) du rectangle de surface s_i qui lui est alloué. En d'autres termes, la surface des rectangles est imposée par les vitesses relatives des processeurs, tandis qu'on peut jouer sur leur forme pour minimiser les communications.

Selon le type de réseau interconnectant les différentes ressources de calcul, l'ensemble des communications pourront être séquentielles (réseau Ethernet), ou parallèles (réseau Myrinet). Si les communications s'effectuent en parallèle, on cherchera à partitionner la matrice en minimisant le *maximum* des demi-périmètres des rectangles. Si les communications sont séquentielles, on cherchera à minimiser *la somme* des demi-périmètres. L'interprétation géométrique de la somme des demi-périmètres est la suivante : cela correspond à la somme des longueurs des lignes tracées pour effectuer la partition, augmentée de 2 (correspondant à une bordure horizontale et à une bordure verticale du carré unitaire).

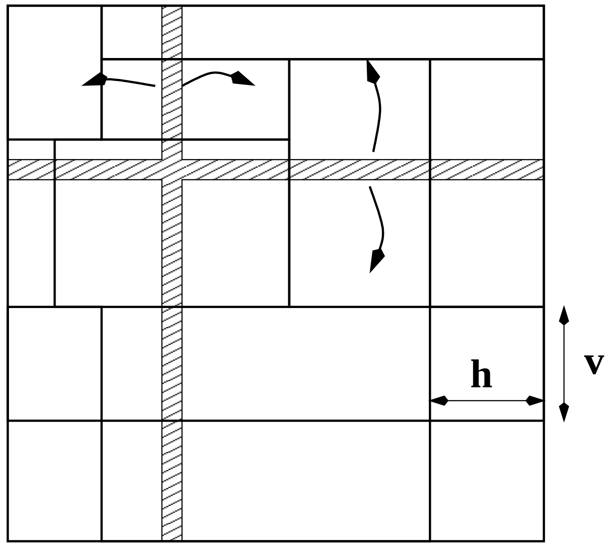


FIG. 7.13 – Produit de matrices avec 13 processeurs.

A la lumière de ces considérations, on retrouve le fait (bien connu) que le produit de matrices a vocation à s'exécuter sur une grille 2D plutôt que sur un anneau quand on dispose de pq processeurs homogènes. Sur l'anneau, qui correspond à une partition en bande, la somme des demi-périmètres vaut pq, alors que sur la grille de taille $p \times q$, cette somme vaut $p + q$. Bien sûr si le nombre de processeurs vaut 13 il n'y a pas beaucoup de grilles possibles ! Par contre si la grille est carrée, avec $p \times p$ processeurs homogènes, la somme des demi-périmètres vaut $2p$, ce qui est optimal car chaque rectangle alloué est un carré. Hélas, partitionner le carré unité en p carrés, d'aires égales (cas homogène) ou non (cas hétérogène) n'est pas toujours possible.

De manière purement géométrique, les deux problèmes d'optimisation précédents s'expriment de la manière suivante : comment partitionner un carré unitaire en p rectangles d'aires fixées $s_1, s_2, \ldots, s_p$ (tels que $\sum_{i=1}^{p} s_i = 1$) afin de minimiser

– soit la somme des demi-périmètres des rectangles dans le cas de communications séquentielles,
– soit le plus grand des demi-périmètres des rectangles dans le cas de communications parallèles.

Voici un exemple avec $p = 5$ rectangles $R_1, \ldots, R_5$ d'aires $s_1 = 0.36$, $s_2 = 0.25$, $s_3 = s_4 = s_5 = 0.13$. Une partition possible est schématisée Figure 7.14. La taille de chaque rectangle est la suivante : $0.61 \times \frac{36}{61}$ pour R_1, $0.61 \times \frac{25}{61}$ pour R_2, et $0.39 \times \frac{1}{3}$ pour R_3, R_4, et R_5. Le demi-périmètre maximal est celui de R_1, approximativement 1.2002, très proche de la borne inférieure absolue $2\sqrt{0.36} = 1.2$ atteinte lorsque le plus grand rectangle est un carré (ce qui n'est pas possible dans cet exemple). En ce qui concerne la seconde fonction objective, la somme des demi-périmètres vaut 4.39, alors que la borne absolue inférieure vaut $\sum_{i=1}^{p} 2\sqrt{s_i} \approx 4.36$ (atteinte lorsque tous les rectangles sont des carrés, ce qui n'est encore pas possible dans cet exemple). Ainsi, la partition s'avère très satisfaisante pour chacune des deux fonctions objectives.

Dans ce qui suit, on se limite à la première fonction objective, celle de la somme des demi-périmètres : on établit la NP-complétude du problème et présente une heuristique garantie. Tout ce que vous voulez savoir sur la deuxième fonction objective, celle du maximum des demi-périmètres, se trouve dans la thèse de Rastello [58].

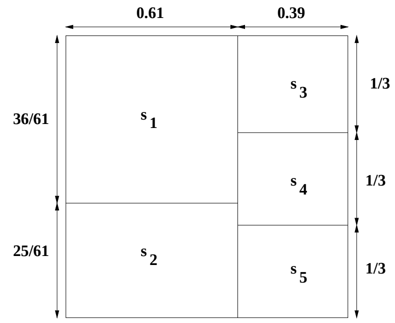


FIG. 7.14 – Un partitionnement du carré unitaire à l'aide de 5 rectangles.

7.5 NP-complétude

Dans ce paragraphe, nous décrivons formellement le problème d'optimisation dont la fonction objective est la somme des demi-périmètres des rectangles, et prouvons la NP-complétude du problème de décision associé.

Il faut paver le carré unitaire à l'aide de p rectangles R_i, d'aire s_i $(1 \leq i \leq p)$ où $\sum_{i=1}^{p} s_i = 1$. La forme de chaque R_i correspond aux degrés de liberté.

Définition 16 *PERI-SUM(s) : Soient p nombres réels positifs $s_1, \ldots, s_p$ tels que $\sum_{i=1}^{p} s_i = 1$, trouver une partition du carré unitaire en p rectangles R_i d'aire s_i et de forme $h_i \times v_i$, tels que $\hat{C} = \sum_{i=1}^{p}(h_i + v_i)$ soit minimisé.*

Il existe une borne inférieure triviale au problème PERI-SUM(s) :

Lemme 18 *Pour toute solution de PERI-SUM(s), $\hat{C} \geq 2\sum_{i=1}^{p} \sqrt{s_i}$.*

Preuve Le demi-périmètre de chaque rectangle R_i sera toujours plus grand que lorsque c'est un carré, c'est à dire $2\sqrt{s_i}$. Bien sûr, paver un carré en p carrés d'aires s_i n'est pas nécessairement possible, et donc, la borne inférieure pour PERI-SUM(s) n'est pas nécessairement atteignable. ■

Le problème de décision associé au problème d'optimisation PERI-SUM est le suivant :

Définition 17 *PERI-SUM(s,K) : soient p nombres réels positifs $s_1, \ldots, s_p$ tels que $\sum_{i=1}^{p} s_i = 1$ ainsi qu'une borne réelle positive K, existe-t-il une partition du carré unitaire en p rectangles R_i, d'aire s_i et de forme $h_i \times v_i$, telle que $\sum_{i=1}^{p}(h_i + v_i) \leq K$?*

Notre premier résultat établit la difficulté intrinsèque du problème d'optimisation PERI-SUM :

Théorème 11 *PERI-SUM(s,K) est NP-complet.*

Preuve Les principales idées de la preuve sont les suivantes :
Réduction PERI-SUM à ASP

1. Dans un premier temps, nous effectuons la réduction de PERI-SUM(s,K) à un problème géométrique (ASP) dont le problème est de vérifier l'existence d'une partition du carré unitaire en carrés d'aires fixées.
2. Ensuite, nous prouvons la NP-complétude de ASP à l'aide d'une réduction polynômiale au problème de 2-Partition-Equal qui lui même est NP-Complet [34]. Cette réduction est fondée sur les idées suivantes :

Réduction ASP à 2-Partition-Equal

1. Nous démarrons d'une instance arbitraire du problème de 2-Partition-Equal (Définition 15), c'est à dire d'un ensemble $\mathcal{A} = \{a_1, \ldots, a_n\}$ de n entiers, qu'il faut partitionner en deux sous-ensembles disjoints *de même cardinal* et de même somme.

2. L'idée est de construire une instance équivalente à ce problème à l'aide d'un ensemble $\mathcal{B} = \{b_1, \ldots, b_n\}$ de n entiers vérifiant $b > \frac{2}{3} \max b_i$. On définie simplement et polynômialement pour cela $b_i = 2(a_i + 2nmax_k a_k)$. Sous certaines considérations techniques, nous montrons qu'il existe une solution au problème de 2-Partition-Equal initial si et seulement si $\mathcal{B}$ peut être partitionné en deux sous-ensembles de même somme, mais *pas nécessairement de même cardinal.*

3. Finalement, nous construisons à partir de $\mathcal{B}$ une instance de ASP à l'aide de trois types de carrés : de larges carrés dénotés par $A_{i,j}$ Figure 7.15, n carrés de taille $b_i \times b_i$ dénotés par A_{b_i} dans la Figure 7.16 et un nombre polynômial de carrés de remplissage dénotés par $A_{\overline{b_i},j}$ dans la Figure 7.16.

4. Nous montrons alors que la seule configuration possible est celle décrite Figure 7.15. Dans cette configuration, il y a deux zones rectangulaires d'aire $m \times S$ où $M = \frac{4}{3} \max b_i$ et $S = \sum_i \frac{b_i}{2}$, partitionnées comme schématisé Figure 7.15. Grâce à la condition $b_i > \frac{M}{2}$, nécessairement les carrés A_{b_i} d'aire $b_i \times b_i$ ne peuvent pas être superposés et doivent agencés les uns à coté des autres dans la direction S. Ainsi, pour chacune des deux zones, la somme des b_i qu'elle contient vaut S.

5. Intuitivement, les rectangles $A_{i,j}$ sont introduit afin de créer ces deux zones non adjacentes d'aire $M \times S$; les n carrés A_{b_i} doivent alors être alignés à l'intérieur de ces deux zones, et les carrés restant $A_{\overline{b_i},j}$ sont introduit afin de remplir les zones restantes.

Clairement, PERI-SUM(s,K) $\in$ NP. Nous utilisons la réduction suivante :

Lemme 19

$$\textit{2P-eq} \leq_P \textit{ASP} \leq_P \textit{PERI-SUM},$$

où la Définition 15 définit le problème 2P-eq, et où ASP est défini comme suit :

Définition 18 *All-Squares-Partition (ASP)*
Soit un ensemble $\mathcal{L} = \{l_1, \ldots, l_p\}$ de p nombres réels positifs tels que $\sum_{i=1}^{p} l_i^2 = 1$, existe-t-il une partition du carré unitaire en p carrés S_i de largeur l_i ?

Comme 2P-eq est NP-complet, le Lemme 19 suffit à démontrer le Théorème 11.

7.5.1 Réduction : ASP $\leq_P$ PERI-SUM(s,K)

Nous commençons par la partie simple de la preuve du Lemme 19, c'est à dire ASP $\leq_P$ PERI-SUM(s,K). Considérons un ensemble $\mathcal{L} = \{l_1, \ldots, l_p\}$ constitué de p nombres réels positifs tels que $\sum_{i=1}^{p} l_i^2 = 1$. Résoudre ASP est équivalent à résoudre PERI-SUM(s,K) avec

$$\begin{cases} K = 2\sum_{i=1}^{p} l_i \\ \forall i,\ s_i = l_i^2 \end{cases}$$

et ainsi,

$$\text{ASP} \leq_P \text{PERI-SUM}.$$

7.5.2 Réduction : 2P-eq $\leq_P$ ASP

Dans ce paragraphe, nous considérons une instance arbitraire du problème de 2-Partition-Equal, c'est à dire la donnée d'un ensemble $\mathcal{A} = \{a_1, \dots, a_n\}$ de n nombres entiers. Nous supposons, sans perte de généralité, que $n \geq 400$. Il nous faut transformer polynômialement cette instance en une instance du problème ASP ayant une solution si et seulement si l'instance originale du problème de 2-Partition-Equal a une solution.

Définissons pour cela $\{b_1, \dots, b_n\}$ par $\forall i, b_i = 2(a_i + 2n \max_k a_k)$. Soit $N = \max_k b_k$ Ainsi, $b_i \geq \frac{2N}{3}$, et b_i est pair. De plus, si on note $M = \frac{4N}{3}$ et $S = \frac{\sum_i b_i}{2}$, alors $S \geq 100M$. On a aussi, $\frac{M}{2} < b_i < \frac{3M}{4}$ pour tout i. La raison pour laquelle nous introduisons M est de pouvoir paver les n rectangles de taille $b_i \times (M - b_i)$ en un nombre minimal de carrés $KS(i)$, à l'aide de la procédure de Kenyon [43]. Afin d'avoir un nombre polynômial de carrés $KS(i)$, le rectangle $\overline{R_i}$ ne doit pas être trop allongé, ce qui est assuré grâce à l'inégalité $M - b_i < b_i < 3(M - b_i)$. On obtient alors de [43] que $KS(i) \leq 3 + C \log b_i$, où C est une constante universelle. par la suite, nous dénotons par $w(\overline{b_i}, j)$ pour $1 \leq j \leq KS(i)$ la largeur des $KS(i)$ carrés obtenus par la procédure de [43] qui pave le rectangle $\overline{R_i}$ de taille $b_i \times (M - b_i)$.

Nous construisons ainsi l'instance du problème (mis à l'échelle d'un rapport $20S + 17M$) ASP suivante (noté abusivement ASP$(b_1, \dots, b_n)$) : existe-t-il une partition du carré de taille $(20S + 17M) \times (20S + 17M)$ en $14 + n + \sum_k KS(i)$ carrés de largeur

$$\left\{ \begin{array}{rcll} l_{13,11} & = & (13S + 11M) & (\times 1), \\ l_{7,6} & = & (7S + 6M) & (\times 3), \\ l_{4,3} & = & (4S + 3M) & (\times 2), \\ l_{3,3} & = & (3S + 3M) & (\times 2), \\ l_{3,2} & = & (3S + 2M) & (\times 2), \\ l_{2,2} & = & (2S + 2M) & (\times 4), \\ l_{bi} & = & b_i & (\forall i, 1 \leq i \leq n), \\ l_{\overline{b_i},j} & = & w(\overline{b_i}, j) & (\forall i,\ 1 \leq i \leq n, \forall j,\ 1 \leq j \leq KS(i))? \end{array} \right.$$

A partir de maintenant, $A_{x,y}$ représente un carré de largeur $l_{x,y} = (xS + yM)$, A_{b_i} un carré de largeur $l_{b_i} = b_i$ et $A_{\overline{b_i},j}$ représente le carré de Kenyon de largeur $l_{\overline{b_i},j} = w(\overline{b_i}, j)$.

Dans ce qui suit, nous allons montrer qu'une telle partition est nécessairement de la forme schématisée Figure 7.15, dans laquelle les deux petits rectangles $(M \times S)$ fléchés sont eux-même partitionnés en d'autres rectangles comme schématisé Figure 7.16. L'idée intuitive de la preuve est la suivante : les grands carrés sont utilisés afin de forcer les zones rectangles $M \times S$ à être séparées. Ainsi, ces deux surfaces doivent être remplies à l'aide des petits carrés restants d'aires b_i et des carrés de Kenyon. Ceci étant possible si et seulement si l'ensemble des b_i peut être partitionné en deux sous-ensembles de même somme, ce qui est possible si et seulement si l'ensemble des a_i peut être partitionné en deux sous-ensembles de même cardinal et de même somme. Les carrés de Kenyon sont introduit afin de remplir les trous dans les deux zones rectangles et d'obtenir ainsi un pavage complet du carré initial.

Position des quatre plus grands carrés La position générale des quatre plus grands carrés est donnée Figure 7.17a. Clairement, si on peut paver l'aire restante avec les carrés restants, il en est de même dans la configuration donnée Figure 7.17b. Ainsi, sans perte de généralité, nous considérons dès à présent que les quatre plus grands carrés sont positionnés comme montré Figure 7.17b.

Pavage de la surface restante Analysons maintenant le pavage de la surface restante (partie non grisée de la Figure 7.17b). Toutes les dimensions sont fournies Figure 7.18. Une étude exhaustive

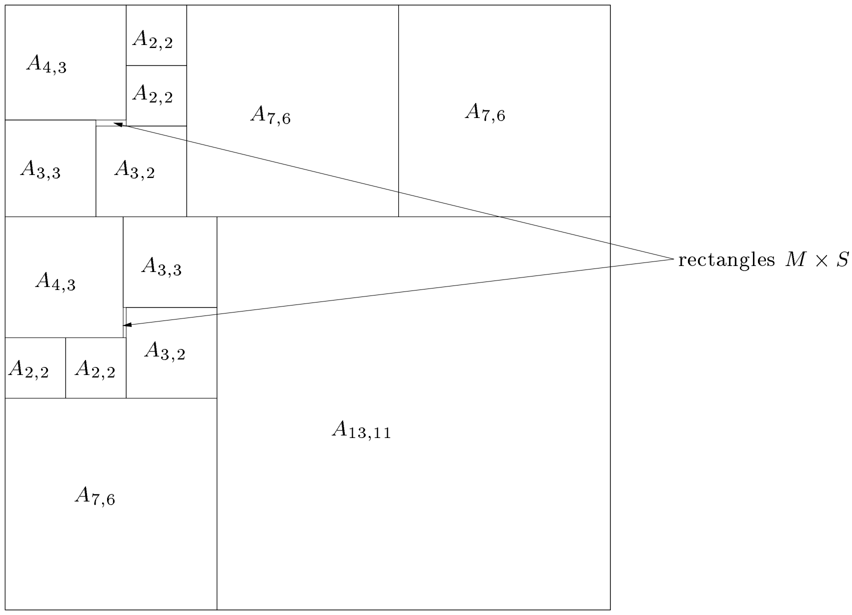


FIG. 7.15 – Positionnement des carrés.

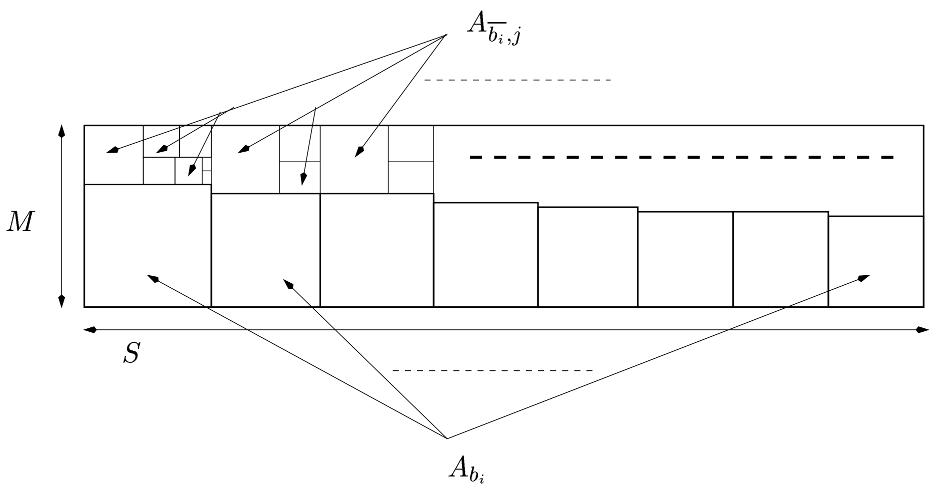


FIG. 7.16 – Description des rectangles $M \times S$.

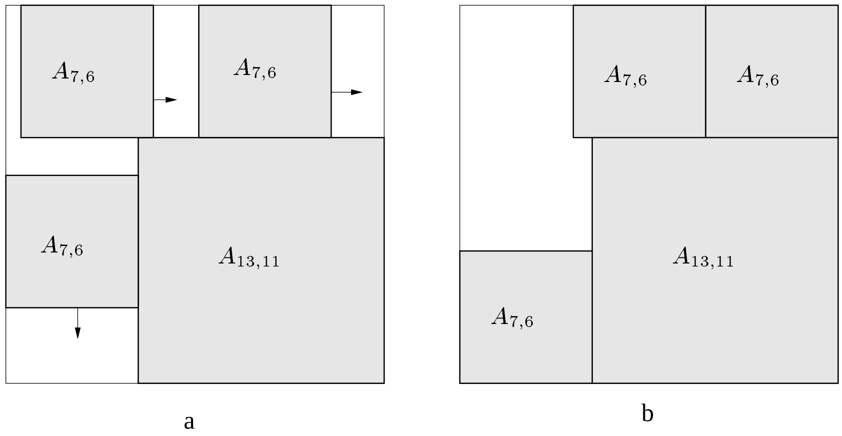


FIG. 7.17 – Position des quatre plus grands carrés.

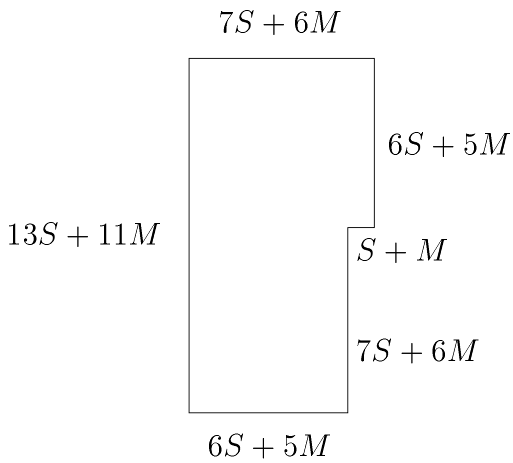


FIG. 7.18 – Surface restante.

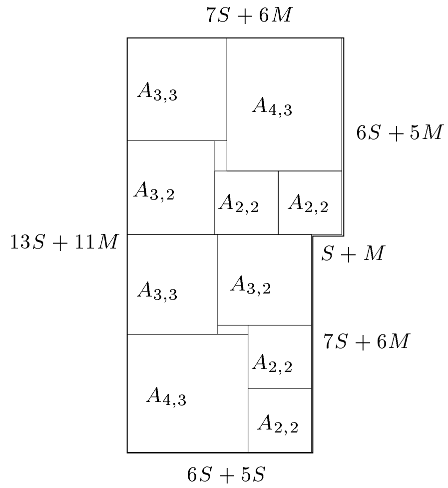


FIG. 7.19 – Pavage possible de la surface restante.

montre que la seule configuration possible (d'autres solutions équivalentes sont possibles) est celle décrite dans la Figure 7.19.

Conclusion partielle Ainsi, tout pavage de la surface restante (cf Figure 7.18) est similaire à celui représenté Figure 7.19 : après avoir placé tous les gros rectangles $A_{x,y}$, il reste deux rectangles non adjacents de forme $M \times S$ à paver. Ainsi, le problème ASP admet une solution si et seulement si il est possible de paver ces deux rectangles restants à l'aide de n carrés de largeur b_i $(1 \leq i \leq n)$, et $\sum_k KS(i)$ carrés de Kenyon de largeur $w(\overline{b_i}, j)$. Comme $\min_k b_k > \frac{M}{2}$ et $\sum_k b_k = 2S$, il n'est pas possible de superposer deux rectangles A_{b_i} et A_{b_j} pour $i \neq j$ l'un au dessus de l'autre. Comme $\sum_i b_i = 2S$ chaque zone rectangle $M \times S$ doit être pavée comme décrit Figure 7.16 et pour chacune des deux zones la somme des b_i vaut S. Ainsi, notre instance du problème ASP admet une solution si et seulement si il existe une partition de $\{b_1, \ldots, b_n\}$ en deux sous-ensembles de même somme S.

Réduction finale Pour compléter la réduction, il nous faut montrer qu'il existe une partition de l'ensemble $\{b_1, \ldots, b_n\}$ en deux sous-ensembles de même somme, si et seulement si l'instance originale du problème de 2P-eq admet une solution.

Supposons dans un premier temps que l'instance du problème de 2P-eq admet une solution, c'est à dire qu'il existe une partition de $\{1, \ldots, n\}$ en deux sous-ensembles $\mathcal{A}_1$ et $\mathcal{A}_2$ satisfaisant

$$\sum_{k \in \mathcal{A}_1} a_k = \sum_{k \in \mathcal{A}_2} a_k \text{ et } \mathrm{card}(\mathcal{A}_1) = \mathrm{card}(\mathcal{A}_2).$$

Rappelons que $b_k = 2(a_k + 2n \times \mathrm{MAX})$, où $\mathrm{MAX} = \max_k a_k$. Alors,

$$\begin{aligned}
\sum_{k \in \mathcal{A}_1} b_k &= \sum_{k \in \mathcal{A}_1} a_k + 2n \times \mathrm{MAX} \times \mathrm{card}(\mathcal{A}_1) \\
&= \sum_{k \in \mathcal{A}_2} a_k + 2n \times \mathrm{MAX} \times \mathrm{card}(\mathcal{A}_2) \\
&= \sum_{k \in \mathcal{A}_2} b_k
\end{aligned}$$

Il existe donc une partition acceptable de $\{b_1, \ldots, b_n\}$.

Réciproquement, supposons qu'il existe une partition de $\{1, \ldots, p\}$ en deux sous-ensembles $\mathcal{A}_1$ et $\mathcal{A}_2$ tels que

$$\sum_{k \in \mathcal{A}_1} b_k = \sum_{k \in \mathcal{A}_2} b_k.$$

Montrons que

$$\text{card}(\mathcal{A}_1) = \text{card}(\mathcal{A}_2)$$

On a,

$$\begin{aligned}
\sum_{k \in \mathcal{A}_1} a_k &= \sum_{k \in \mathcal{A}_1} b_k - 2n \times \text{MAX} \times \text{card}(\mathcal{A}_1) \\
\sum_{k \in \mathcal{A}_2} a_k &= \sum_{k \in \mathcal{A}_2} b_k - 2n \times \text{MAX} \times \text{card}(\mathcal{A}_2) \\
\sum_{k \in \mathcal{A}_1} a_k - \sum_{k \in \mathcal{A}_2} a_k &= 2n \text{ MAX card}(\mathcal{A}_2 - \mathcal{A}_1)
\end{aligned}$$

De plus, comme

$$\sum_{a_k \in \mathcal{A}_1} a_k - \sum_{a_k \in \mathcal{A}_2} a_k \leq n \text{ MAX},$$

on obtient

$$\text{card}(\mathcal{A}_1) = \text{card}(\mathcal{A}_2) \text{ et } \sum_{k \in \mathcal{A}_1} a_k = \sum_{k \in \mathcal{A}_2} a_k$$

En conséquence, l'instance d'origine du problème de 2P-eq admet une solution.

Consistance de la transformation Le dernier point de la démonstration est la vérification de la consistance de la transformation : il nous faut prouver que l'expression de notre instance du problème ASP a une taille polynômiale en la taille de l'expression de l'instance du problème de 2P-eq d'origine.

Lemme 20 *Notons MAX* $= \max_k a_k$, *et représentons par* $c(a)$ *et* $c(b)$ *le codage des données* a *et* b. *Alors,*

$$\textit{Taille}(c(\mathcal{L})) = O(\textit{Taille}(c(\mathcal{A}))^3).$$

Preuve Nous utilisons ici les notations classiques de comparaison asymptotique de fonctions O et Ω : on dit que $f(x) = O(g(x))$ s'il existe une constante c telle que $f(x) \leq cg(x)$ pour x suffisamment grand, et on dit que $f(x) = \Omega(g(x))$ s'il existe une constante c telle que $g(x) \leq cf(x)$ pour x suffisamment grand.

Pour l'encodage de l'instance initiale $\mathcal{A}$, on a Taille$(c(\mathcal{A})) = \Omega\left(\sum_i \log a_i\right)$. Comme

$$\sum_i \log a_i \geq \log \text{MAX} + (n-1)\log(\min_i a_i) \geq (n-1)\log 2 + \log \text{MAX},$$

on en déduit que

$$\text{Taille}(c(\mathcal{A})) = \Omega(n + \log \text{MAX}).$$

Pour l'encodage de l'instance ASP $\mathcal{L}$, on a

$$\begin{cases}
\log b_i \leq \log((4n+2)\text{MAX}) = O(\log n + \log \text{MAX}), \\
\log M = O(\log n + \log \text{MAX}), \\
\log S = O(n(\log n + \log \text{MAX}), \\
\sum_i KS(i) \log b_i \leq \sum_i (3 + C \log b_i) \log b_i = O(n(\log n + \log \text{MAX})^2),
\end{cases},$$

où C est la constante universelle donnée par Kenyon [43]. Ainsi,

$$\text{Taille}(c(\mathcal{L})) = O(\text{Taille}(c(\mathcal{A}))^3).$$

■

Ceci termine la preuve de NP-complétude du problème ASP, et achève donc la preuve de NP-complétude du problème PERI-SUM. ■

7.5.3 Heuristique garantie

Il s'avère que la restriction du problème PERI-SUM aux partitionnements par colonnes admet une solution polynômiale. Nous présentons la solution de ce problème, et la garantie offerte par rapport à la solution optimale de PERI-SUM.

Solution de COL-PERI-SUM

Comme PERI-SUM(s) est NP-complet, nous considérons le problème plus contraint COL-PERI-SUM(s) dans lequel nous imposons le fait que le pavage soit constitué de colonnes de processeurs, comme décrit Figure 7.20. En d'autres termes, COL-PERI-SUM(s) est la restriction de PERI-SUM(s) aux partitionnements par colonnes. Dans ce paragraphe, nous proposons une solution optimale en un temps polynômial ($O(p\sqrt{p}\log p)$) à ce problème.

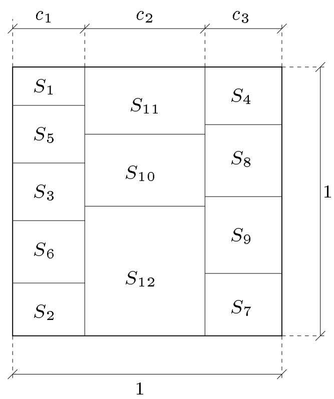


FIG. 7.20 – Partitionnement par colonnes du carré unitaire : le nombre de colonnes vaut $\mathcal{C} = 3$; la première colonne est composée de $k_1 = 5$ rectangles, la seconde de $k_2 = 3$ rectangles et la troisième de $k_3 = 4$ rectangles.

Donnons une description plus formelle du problème COL-PERI-SUM(s) : soit $s_1, \ldots, s_p$ p nombres réels positifs tels que $\sum_i s_i = 1$. Nous cherchons à paver le carré unitaire en $\mathcal{C}$ colonnes (où $\mathcal{C}$ est à déterminer) de largeur $c_1, \ldots, c_{\mathcal{C}}$. Chaque colonne C_i est elle-même partitionnée en k_i lignes (à déterminer aussi) d'aires $s_{\sigma(i,1)}, \ldots, s_{\sigma(i,k_i)}$. Bien entendu, le partitionnement final est composé de $\sum_{i=1}^{\mathcal{C}} k_i = p$ rectangles, et chaque aire $s_1, \ldots, s_p$ est représentée une fois et une seule. Le but est de construire un tel pavage qui minimise la somme des demi-périmètres des rectangles.

L'algorithme, fondé sur la technique de programmation dynamique, utilise les idées suivantes :

1. Renumérotation des variables $s_1, \ldots, s_p$ telle que $s_1 \leq s_2 \leq \ldots \leq s_p$.

2. Construction itérative des fonctions $f_{\mathcal{C}}^{perimeter}$, où $\mathcal{C}$ est incrémenté de 1 à la valeur désirée. Pour $q \in \{1, \ldots, p\}$, $f_{\mathcal{C}}^{perimeter}(q)$ représente le périmètre total du partitionnement optimal par $\mathcal{C}$ colonnes et q rectangles d'aires respectives $s_1, \ldots, s_q$, du rectangle de hauteur 1 et de largeur $(\sum_{i=1}^{q} s_i)$.

Afin de comprendre le principe, nous appliquons l'algorithme sur l'exemple suivant : soient $p = 8$ aires de valeurs (0.02, 0.04, 0.06, 0.08, 0.2, 0.2, 0.2, 0.2). Le résultat de l'algorithme est décrit dans la Table 7.2. Chaque colonne $\mathcal{C}_i$ contribue à la valeur de la somme des demi-périmètres comme suit : la ligne verticale compte 1, et l'ensemble des k_i lignes horizontales de largeur c_i compte $k_i \times c_i$.

	q=1	q=2	q=3	q=4	q=5	q=6	q=7	q=8
$\mathcal{C}=1$	1.02 | 0	1.12 | 0	1.36 | 0	**1.8 | 0**	3 | 0	4.6 | 0	6.6 | 0	9 | 0
$\mathcal{C}=2$		2.06 | 1	2.18 | 2	2.4 | 2	2.92 | 3	**3.6 | 4**	4.6 | 4	5.8 | 5
$\mathcal{C}=3$			3.12 | 2	3.26 | 3	3.6 | 4	4.12 | 5	4.72 | 5	**5.4 | 6**
$\mathcal{C}=4$				4.2 | 3	4.46 | 4	4.8 | 5	5.32 | 6	5.92 | 7
$\mathcal{C}=5$					5.4 | 4	5.66 | 5	6 | 6	6.52 | 7
$\mathcal{C}=6$						6.6 | 5	6.86 | 6	7.2 | 7
$\mathcal{C}=7$							7.8 | 6	8.06 | 7
$\mathcal{C}=8$								9 | 7

TAB. 7.2 – Tableau contenant les valeurs de $f_{\mathcal{C}}^{perimeter}(q)$ et a séparés par $|$. $f_{\mathcal{C}}^{perimeter}(q) = \min_{a \in [\mathcal{C}-1, q-1]} \Big(1 + \big(\sum_{a<i\leq q} s_i\big) \times (q-a) + f_{\mathcal{C}-1}^{perimeter}(a)\Big)$ et a correspond à la valeur minimisant l'expression ci-dessus. Les valeurs écrites en caractères gras correspondent à la solution optimale : pour $\mathcal{C} = 3$, $f_{\mathcal{C}}^{cut}(8) = 6$, ainsi les deux premières colonnes contiennent 6 éléments ; pour $\mathcal{C} = 2$, $f_{\mathcal{C}}^{cut}(6) = 4$, et la première colonne contient 4 éléments.

Dans cet exemple, le partitionnement optimal est obtenu avec 3 colonnes ($f_3(8) = 5.4$). La dernière colonne de largeur $c_3 = s_7 + s_8 = 0.4$ est composée de deux éléments. La seconde, de largeur $c_2 = s_5 + s_6 = 0.4$ est aussi composée de deux éléments. Finalement, la dernière, de largeur $c_1 = s_1 + s_2 + s_3 + s_4 = 0.2$ est composée des 4 plus petits éléments. Le partitionnement optimal est représenté Figure 7.21.

L'algorithme est donné à la Figure 7.22. Dans le pire des cas, sa complexité est en $O(p^3)$. Le partitionnement final correspondant à la fonction $f_{\mathcal{C}_{opt}}^{perimeter}(p) = \min_{1\leq \mathcal{C} \leq p} f_{\mathcal{C}}^{perimeter}(p)$ est trouvé à l'aide de l'algorithme décrit à la Figure 7.23.

Cet algorithme correspond à parcourir en sens inverse le Tableau 7.2 en passant par les valeurs représentées en caractères gras. Le carré unitaire est partitionné en $\mathcal{C}_{opt}$ colonnes. La i-ème colonne contient les rectangles $s_{d_i}, s_{d_i+1}, \ldots, s_{d_i+k_i}$ avec $d_i = k_1 + k_2 + \ldots + k_{i-1}$.

Les algorithmes des Figures 7.22 et 7.23 fournissent la solution optimale du problème COL-PERI-SUM. La seule difficulté consiste à montrer que l'on peut réduire la recherche à des séquences ordonnées $s_1 \leq s_2 \ldots \leq s_p$:

Définition 19 *Un partitionnement est dit bien-ordonné si pour toute paire de colonnes $\mathcal{C}_i$ et $\mathcal{C}_j$, soit tous les éléments de $\mathcal{C}_i$ sont plus petits ou égaux à tous les éléments de $\mathcal{C}_j$, soit l'inverse.*

La Figure 7.24 illustre cette définition. Considérons un partitionnement composé de $\mathcal{C}$ colonnes de tailles $k_1 \geq k_2 \geq \ldots k_{\mathcal{C}}$. Supposons les s_i indexés de telle manière que $s_1 \leq s_2 \leq \ldots \leq s_p$; soit τ une permutation de $\{1, 2, \ldots, p\}$ telle que la i-ème colonne du partitionnement contienne les rectangles $s_{\tau(d_i+1)}, \ldots, s_{\tau(d_i+k_i)}$ où $d_i = k_1 + k_2 + \ldots + k_{i-1}$. Maintenant, rappelons que le coût

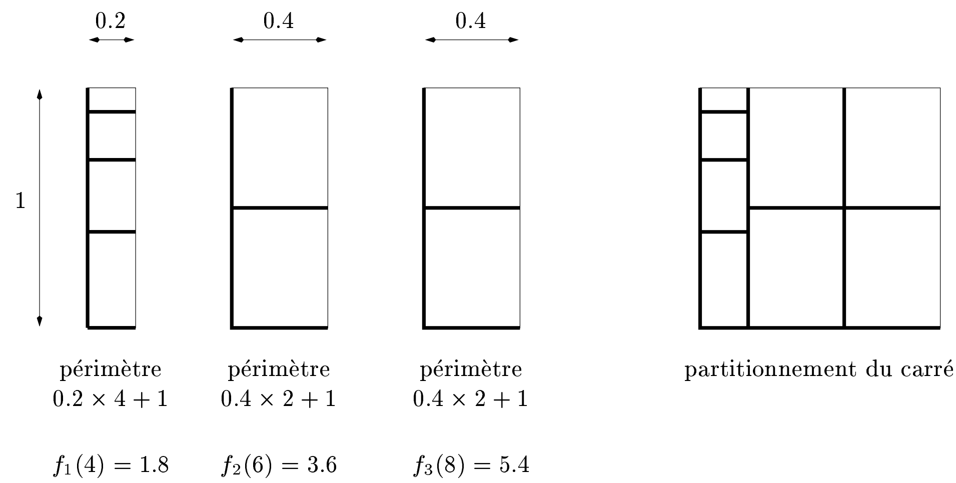


FIG. 7.21 – Partitionnement par colonnes optimal. Les lignes plus épaisses correspondent aux lignes comptabilisées dans la somme des demi-périmètres.

$S = 0$
for $q = 1, p$
  $S = S + s_q$
  $f_1^{perimeter}(q) = 1 + S \times q$
  $f_1^{cut}(q) = 0$
for $\mathcal{C} = 2, p$
  for $q = \mathcal{C}, p$
    $f_{\mathcal{C}}^{perimeter}(q) = \min_{a \in [\mathcal{C}-1, q-1]} \left(1 + \sum_{q-a<i\leq q} s_i(q-r) + f_{\mathcal{C}-1}^{perimeter}(a)\right)$
    $f_{\mathcal{C}}^{cut}(q) = q - a_{opt}$ {Où a_{opt} atteint le minimum dans l'expression ci-dessus}

FIG. 7.22 – Algorithme pour COl-PERI-SUM : construction des fonctions $f_{\mathcal{C}}^{perimeter}$ ($f_{\mathcal{C}}^{cut}(q)$ correspond au nombre de blocs utilisés dans les $\mathcal{C}-1$ premières colonnes, ce qui laisse $q - f_{\mathcal{C}}^{cut}(q)$ dans la colonne $\mathcal{C}$).

d'une colonne C_i est $1 + k_i \sum_{j=d_i+1}^{d_i+k_{i-1}} s_{\tau(j)}$. Ainsi, le demi-périmètre total vaut

$$\begin{array}{lll} \mathcal{C} & + & k_1 s_{\tau(1)} + k_1 s_{\tau(2)} + \ldots + k_1 s_{\tau(k_1)} \\ & + & k_2 s_{\tau(k_1+1)} + k_2 s_{\tau(k_1+2)} + \ldots + k_2 s_{\tau(k_1+k_2)} \\ & + & \ldots \\ & + & k_{\mathcal{C}} s_{\tau(k_1+\ldots+k_{\mathcal{C}-1}+1)} + k_{\mathcal{C}} s_{\tau(k_1+\ldots+k_{\mathcal{C}-1}+2)} + \ldots + k_{\mathcal{C}} s_{\tau(k_1+\ldots+k_{\mathcal{C}})} \end{array}$$

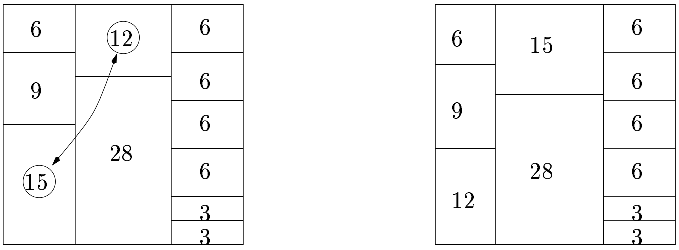
Comme $k_1 \geq k_2 \geq \ldots \geq k_{\mathcal{C}}$, cette expression est minimisée pour $\tau =$ Identité. Ce qui correspond à un partitionnement bien-ordonné. La preuve est obtenue par récurrence sur le nombre d'inversions dans la permutation τ. Ainsi, pour tout partitionnement, il existe un partitionnement bien-ordonné correspondant meilleur ou équivalent.

Garantie

Dans ce paragraphe, nous montrons que le partitionnement par colonnes fournit une bonne approximation, surtout lorsque le rapport entre la plus grande aire $\max s_i$ et la plus petite aire $\min s_i$ est faible.

$q = p$
for $\mathcal{C} = \mathcal{C}_{opt}, 2 : -1$
  $k_{\mathcal{C}} = q - f_{\mathcal{C}}^{cut}(q)$
  $q = f_{\mathcal{C}}^{cut}(q)$
$k_1 = q$

FIG. 7.23 – Reconstruction de la solution optimale à partir des fonctions $f_{\mathcal{C}}^{cut}$



pas bien-ordonné bien-ordonné

FIG. 7.24 – Deux partitionnements de la même instance. Celle de droite est bien ordonnée, celle de gauche ne l'est pas.

Théorème 12 *Soit* $r = \frac{\max s_i}{\min s_i}$. *Notons* $\hat{C}$ *la somme des demi-périmètres des rectangles du partitionnement par colonnes optimal, et* $LB = 2\sum_{i=1}^{p} \sqrt{s_i}$. *Alors,*

$$\frac{\hat{C}}{LB} \leq \sqrt{r}\left(1 + \frac{1}{\sqrt{p}}\right) = \sqrt{\frac{\max s_i}{\min s_i}}\left(1 + \frac{1}{\sqrt{p}}\right)$$

Si $r = 1$, c'est à dire si tous les processeurs ont la même vitesse, le partitionnement par colonnes est asymptotiquement optimal. En contrepartie, lorsque r est grand, la borne est très pessimiste.

Preuve Considérons le partitionnement par colonnes constitué de $\mathcal{C} = \lceil \sqrt{r} \sum \sqrt{s_i} \rceil$ colonnes. Les rectangles sont également distribués sur les colonnes de telle manière que le nombre de rectangles dans chaque colonne est soit $\lfloor \frac{p}{\mathcal{C}} \rfloor$ soit $\lceil \frac{p}{\mathcal{C}} \rceil$. Notons $\hat{C}^*$ la somme des demi-périmètres de ce partitionnement, on a :

$$\begin{aligned} \hat{C}^* &\leq \lceil \sqrt{r} \sum \sqrt{s_i} \rceil + \lceil \frac{p}{\lceil \sqrt{r} \sum \sqrt{s_i} \rceil} \rceil \\ &\leq 2 + \sqrt{r} \sum \sqrt{s_i} + \frac{p}{\sqrt{r} \sum \sqrt{s_i}} \end{aligned}$$

Ainsi,

$$\frac{\hat{C}^*}{2 \sum \sqrt{s_i}} \leq \frac{1}{\sum \sqrt{s_i}} + \frac{\sqrt{r}}{2} + \frac{p}{2\sqrt{r} \sum \sqrt{s_i}}.$$

De plus,

$$\begin{aligned} \sum s_i = 1 &\implies p \max s_i \geq 1 \\ &\implies \min s_i \geq \frac{1}{pr} \end{aligned}$$

et ainsi,

$$\sum \sqrt{s_i} \geq p\sqrt{\min s_i} \geq \sqrt{\frac{p}{r}}.$$

Soit finalement,

$$\begin{aligned} \frac{\hat{C}^*}{2\sum\sqrt{s_i}} &\leq \sqrt{\frac{r}{p}} + \frac{\sqrt{r}}{2} + \frac{\sqrt{r}}{2} \\ &\leq \sqrt{r}(1 + \frac{1}{\sqrt{p}}). \end{aligned}$$

Comme $\hat{C}$ correspond à la meilleure solution parmi toutes les solutions possibles par colonnes, $\hat{C}$ vérifie $\hat{C} \leq \hat{C}^*$, ce qui achève la preuve. ■

Notes bibliographiques

Ce chapitre doit beaucoup à la thèse de Rastello [58], qui contient de nombreux résultats complémentaires. Pour une introduction générale au *cluster computing*, on recommande les livre édités par Buyya [17, 18]. Pour une ouverture au *meta-computing*, on dit encore *grid computing*, on pourra consulter l'ouvrage édité par Foster et Kesselman [32].

Troisième partie

Techniques de compilation

Chapitre 8

Nids de boucles

8.1 Introduction

Les nids de boucles imbriquées constituent la meilleure source potentielle de parallélisme dans les programmes. Leur structure régulière facilite l'analyse du code qui va être exécuté un grand nombre de fois, et, partant, justifie le recours à des optimisations de compilation sophistiquées. En effet, diverses transformations de codes sont nécessaires pour extraire le parallélisme qui se cache (peut-être) au coeur de ces nids de boucles. Certaines transformations sont dépendantes de l'architecture-cible, car la granularité du code généré doit être adaptée à la hiérarchie mémoire présente : un parallélisme à grain fin sera efficace pour des machines vectorielles ou à long mot d'instruction (VLIW), tandis qu'on préférera un code à gros grain (obtenu par des techniques de partitionnement) pour réduire les communications inter-processeur dans une machine à mémoire distribuée. Par contre, la détection du parallélisme (c.à.d. l'identification de certaines boucles comme étant parallèles) et sa compréhension (c.à.d. la détection des dépendances qui sont responsables de la séquentialité intrinsèque de certaines parties du code) sont deux opérations indépendantes de l'architecture-cible : elles ne dépendent que de la structure du code séquentiel à paralléliser. D'où l'idée d'une approche en deux étapes : on étudie la parallélisation sur une machine idéale à nombre de processeurs illimité (une P-RAM !), puis on prend en compte des optimisations spécifiques pour la machine visée, avec des ressources données.

Ce chapitre comprend trois parties :

- Au Paragraphe 8.2, nous introduisons le vocabulaire et rappelons quelques concepts fondamentaux pour l'analyse des dépendances de données.
- Au Paragraphe 8.3, nous présentons le plus célèbre algorithme pour la détection de parallélisme : celui d'Allen et Kennedy. C'est le plus simple à comprendre et et à implémenter. Le parallélisme est révélé grâce à la technique de distribution de boucles.
- Au Paragraphe 8.4, nous présentons des transformations de boucles classiques, basés sur des matrices unimodulaires, comme la méthode de l'hyperplan de Lamport.

8.2 Analyse de dépendances

Nous commençons par quelques définitions et concepts simples autour des nids de boucles imbriquées. Puis nous illustrons, à l'aide d'exemples, les différents types de dépendance.

8.2.1 Nids de boucles et ordre séquentiel

Considérons le bout de code suivant :

```
DO i=1, N
   DO j=i, N+1
     DO k=j-i, N
       S1
       S2
     ENDDO
   ENDDO
   DO r=1, N
     S3
   ENDDO
ENDDO
```

Il y a 4 boucles et 3 instructions (*statement* en anglais, d'où la notation S). Les instructions S_1 et S_2 sont englobées par les boucles i, j, et k, tandis que S_3 est englobé par les boucles i et r. Les boucles ne sont pas *parfaitement imbriquées* car toutes les instructions ne sont pas englobées par les mêmes boucles. On dit qu'un nid de boucles est *parfait* quand toutes les boucles sont parfaitement imbriquées.

On supposera que tous les pas des boucles sont égaux à 1, i.e. que chaque compteur de boucle est incrémenté d'une unité à chaque instance. Les itérations de n boucles parfaitement imbriquées sont représentées par un vecteur à n composantes, appelé *vecteur d'itération*. L'ensemble de toutes les valeurs possibles du vecteur d'itération (définies par les bornes inférieure et supérieure de chaque boucle) est le *domaine d'itération*. Dans l'exemple, les boucles i et r définissent un vecteur d'itération bi-dimensionnel (i, r) dont le domaine d'itération est l'ensemble des points entiers du carré $1 \leq i, r \leq N$; les boucles i, j, et k définissent un vecteur d'itération tri-dimensionnel (i, j, k) dont le domaine d'itération est plus compliqué : c'est l'ensemble des points entiers tels que $1 \leq i \leq N$, $i \leq j \leq N+1$, et $j - i \leq k \leq N$. D'une manière générale, le domaine d'itération est constitué des points entiers à l'intérieur d'un certain polyèdre.

Prenons une instruction quelconque S : au cours de l'exécution du programme, une instance de S est exécutée pour chaque valeur I du vecteur d'itération défini par les boucles qui englobent S. On notera $S(I)$ l'instance de S exécutée à l'itération I. Ces instances sont exécutées suivant un ordre prédéfini, dit *ordre séquentiel*, et représenté par le symbole $<_{seq}$. Expliquons comment cet ordre séquentiel est défini :

- Les différentes itérations d'un nid parfait sont exécutées en respectant l'ordre lexicographique des vecteurs d'itération. Les instances d'une instruction (ou bloc d'instructions) S sont donc exécutées comme suit :

$$S(I) <_{seq} S(J) \Leftrightarrow I <_{lex} J$$

- En outre, les différentes instructions d'un bloc donné sont exécutées en suivant l'ordre textuel (donné par le texte du programme).
- Voici alors la définition de $<_{seq}$ dans le cas général de boucles non parfaitement imbriquées : prenons deux instances $S(I)$ and $T(J)$ correspondant à deux instructions S et T, englobées par $n_{S,T}$ boucles communes. Si S (resp. T) est englobée par $n_S \geq n_{S,T}$ (resp. $n_T \geq n_{S,T}$) boucles, alors I (resp. J) est un vecteur d'itération vecteur de taille n_S (resp. n_T). Soit $\tilde{I}$ (resp. $\tilde{J}$) le vecteur de taille $n_{S,T}$ dont les composantes sont les $n_{S,T}$ premières composantes de I (resp. J). Considérons le code C englobé par les $n_{S,T}$ boucles les plus externes. C contient les

définitions de S et T, et est associé à un vecteur d'itération de taille $n_{S,T}$. Il faut distinguer trois cas :

- Si $\tilde{I} <_{lex} \tilde{J}$, alors $C(\tilde{I}) <_{seq} C(\tilde{J})$, et donc $S(I) <_{seq} T(J)$.
- Si $\tilde{J} <_{lex} \tilde{I}$, alors $C(\tilde{J}) <_{seq} C(\tilde{I})$, et donc $T(J) <_{seq} S(I)$.
- Si $\tilde{J} = \tilde{I}$, alors $S(I)$ et $T(J)$ sont deux opérations effectuées durant une même itération des boucles englobant S et T. Ces opérations sont donc exécutées dans l'ordre dans lequel elles apparaissent dans le code C, i.e., $S(I) <_{seq} T(J)$ ssi $S <_{text} T$.

Pour résumer, l'ordre séquentiel est donné par l'ordre lexicographique défini sur le vecteur d'itérations correspondant aux boucles communes, plus l'ordre textuel en cas d'égalité sur l'ordre lexicographique : :

$$S(I) <_{seq} T(J) \Leftrightarrow (\tilde{I} <_{lex} \tilde{J}) \text{ ou } (\tilde{I} = \tilde{J} \text{ et } S <_{text} T)$$

L'ordre lexicographique est une disjonction d'égalités ou d'inégalités affines. Si I et J sont deux vecteurs de taille n, alors $I <_{lex} J$ ssi

$$\begin{array}{ll} & \{I_1 + 1 \leq J_1\} \\ \text{ou} & \{I_1 = J_1 \text{ et } I_2 + 1 \leq J_2\} \\ \text{ou} & \ldots \\ \text{ou} & \{I_1 = J_1 \text{ et } \ldots \text{ et } I_{n-1} = J_{n-1} \text{ et } I_n + 1 \leq J_n\} \end{array}$$

Dans l'exemple, $S_1(I)$ est exécuté avant $S_2(J)$ ssi $I <_{lex} J$ ou $I = J$ (parce que $S_1 <_{text} S_2$ est vrai), et $S_3(i,r) <_{seq} S_1(i',j,k)$ ssi $i < i'$ (parce que $S_3 <_{text} S_1$ est faux).

Dans la suite, on se restreindra aux nids de boucles qui peuvent être décrits comme précédemment : chaque instruction S englobée par n_S boucles imbriquées est identifiée par sa position au sein de ces boucles (ordre textuel $<_{text}$) et par un domaine d'itération n_S-dimensionnel, décrit par un polyèdre $\mathcal{D}_S$. Tout vecteur entier I du polyèdre $\mathcal{D}_S$ définit un vecteur d'itération valide qui correspond à une instance de S, notée $S(I)$. Toutes les différentes opérations $S(I)$ (où S est une instruction et $I \in \mathcal{D}_S$) sont ordonnées suivant l'ordre séquentiel $<_{seq}$. Des exemples de nids de boucles vérifiant ces hypothèses sont des boucles imbriquées à la Fortran, avec des pas unitaires et des bornes qui s'expriment comme des fonctions affines des paramètres et des indices englobants.

8.2.2 Dépendances de données : Flot, Anti, et Sortie

Bien sûr, les opérations d'un programme donné ne peuvent pas être exécutées dans n'importe quel ordre. Permuter l'ordre d'évaluation de deux instances d'instruction données peut changer le résultat final du programme ; par contre, certaines opérations peuvent être exécutées simultanément sans risque. Bernstein [14] a identifié un ordre partiel (noté $\Rightarrow$) qui est l'ordre minimal à respecter pour produire un code sémantiquement équivalent au code initial : tout ordre d'exécution (partiel ou non) qui est une extension de l'ordre partiel $\Rightarrow$ conduira au même résultat, celui de l'exécution séquentielle. En particulier, $<_{seq}$ est une extension de $\Rightarrow$. Deux instances d'instruction comparables par $\Rightarrow$ sont appelées *dépendantes*. L'ordre partiel $\Rightarrow$ est défini à partir de trois types de dépendances de données, appelées flot, anti, et output. Avant de définir précisément celles-ci, regardons l'exemple suivant :

```
DO i=1, N
  DO j=1, N
    a(i+j) = a(i+j-1) + 1
  ENDDO
ENDDO
```

Déroulons les boucles pour mieux comprendre les relations entre l'exécution des différentes instances d'instruction. Les différentes opérations sont données à la Figure 8.1 for $N = 4$. Les dépendances y sont représentées par des flèches.

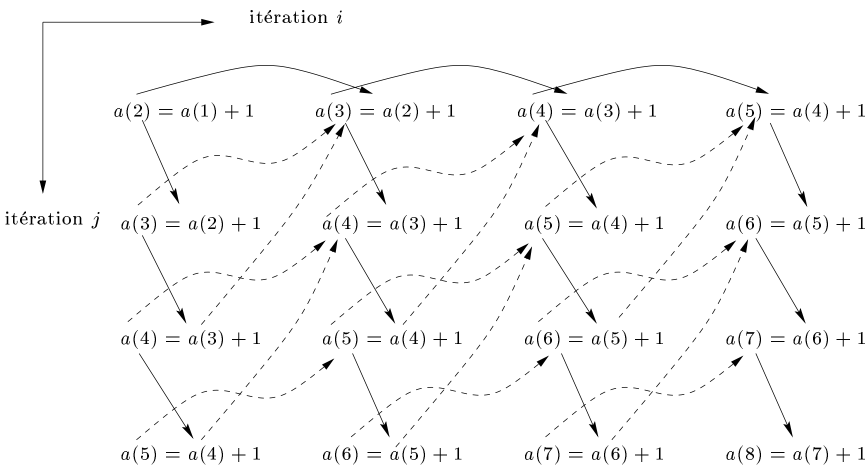


FIG. 8.1 – Déroulement des boucles du programme.

Considérons l'opération effectuée à l'itération $i = 2$, $j = 2$, i.e., représentée par le vecteur d'itération $(2, 2)$. On écrit à l'adresse mémoire $a(4)$ une valeur égale à un plus la valeur que contient l'adresse mémoire $a(3)$. Cette valeur est la valeur calculée à l'itération $(2, 1)$. On dit qu'il y a une *dépendance de flot* de l'itération $(2, 1)$ vers l'itération $(2, 2)$, et que la *distance de dépendance* est $(0, 1)$ pour exprimer le fait que la dépendance provient d'une instance exécutée à la même itération de la boucle i et à l'itération précédente de la boucle j.

Considérons l'opération effectuée à l'itération $i = 2$ et $j = 1$. $a(2)$ est lu, mais cette fois la valeur de $a(2)$ est la valeur calculée à l'itération $(1, 1)$, i.e. à la même itération de la boucle j, $j = 1$, mais à l'itération précédente de la boucle i. Il y a donc une dépendance de flot de l'itération $(1, 1)$ à l'itération $(2, 1)$, et la distance de dépendance est $(1, 0)$. Remarquons que cette dépendance n'est présente que pour des vecteurs d'itérations de la forme $(i, 1)$.

Il ne suffit pas de respecter les dépendances de flot pour garantir que le résultat du code sera inchangé. Les dépendances de flot imposent d'exécuter l'itération $(1, 3)$ avant l'itération $(1, 4)$ parce que la valeur de $a(4)$ dont on a besoin à l'itération $(1, 4)$ est celle calculée à l'itération $(1, 3)$; de même pour les itérations $(2, 2)$ et $(2, 3)$. Cependant, aucune dépendance de flot ne nous empêche d'exécuter l'itération $(2, 2)$ avant les itérations $(1, 3)$ et/ou $(1, 4)$. Supposons alors que les calculs soient effectués dans l'ordre : $(1, 3)$, $(2, 2)$, $(1, 4)$, et enfin $(2, 3)$. Le calcul à l'itération $(1, 4)$ a changé ; la valeur de $a(4)$ est incorrecte, car $a(4)$ devrait être lu à l'itération $(1, 4)$ avant d'être modifié à l'itération $(2, 2)$. On dit qu'il y a une *anti-dépendance* de l'itération $(1, 4)$ à l'itération $(2, 2)$. La distance de dépendance est ici $(1, -2)$.

Supposons maintenant que les calculs soient effectués dans l'ordre : $(2, 2)$, $(1, 3)$, $(1, 4)$, et enfin $(2, 3)$. Le calcul de l'itération $(1, 4)$ reste juste, mais celui de l'itération $(2, 3)$ n'est plus correct – la valeur de $a(4)$ n'est pas la bonne. Dans l'ordre séquentiel, $a(4)$ est écrit deux fois avant d'être lu à l'itération $(2, 3)$ – une fois à l'itération $(1, 3)$, et une à l'itération $(2, 2)$ – et la valeur cherchée est la deuxième. Dans ce cas, on dit qu'il y a une *dépendance de sortie* de l'itération $(1, 3)$ à l'itération $(2, 2)$. La distance de dépendance est ici $(1, -1)$.

En résumé, on distingue trois types de dépendances : flot, anti, et sortie. Ces dépendances sont des dépendances de données (en opposition aux dépendances de contrôle, non traitées dans ce polycopié) : elles sont définies par le fait que deux calculs accèdent au même emplacement mémoire, en lecture ou en écriture.

Formellement, il y a une *dépendance de données* entre $S(I)$ et $T(J)$ si elles accèdent le même emplacement mémoire, et que l'un au moins de ces accès est en écriture. La dépendance est dirigée selon l'ordre séquentiel, i.e. c'est une dépendance de $S(I)$ vers $T(J)$ si $S(I) <_{seq} T(J)$. En outre, cette dépendance est :

- **une dépendance de flot** si l'emplacement mémoire commun est une écriture pour $S(I)$ et une lecture pour $T(J)$, et si cet emplacement mémoire n'est pas accédé en écriture entre $S(I)$ et $T(J)$, au sens de l'ordre séquentiel
- **une anti-dépendance** si l'emplacement mémoire commun est une lecture pour $S(I)$ et une écriture pour $T(J)$, et si cet emplacement mémoire n'est pas accédé en écriture entre $S(I)$ et $T(J)$
- **une dépendance de sortie** si $S(I)$ et $T(J)$ sont deux accès en écriture consécutifs à un même emplacement mémoire.

Ces trois types de dépendances sont des dépendances directes, et tous les autres cas correspondent à des dépendances par transitivité. En résumé :

Flot : écriture puis lecture.
Anti : lecture puis écriture.
Sortie : écriture puis écriture.

Une dépendance de flot correspond à la définition d'une cellule mémoire et à une référence ultérieure à celle-ci ; une anti-dépendance correspond à une référence à une cellule mémoire, et à une re-définition ultérieure de cette cellule ; et une dépendance de sortie correspond à la définition d'une cellule mémoire et à la re-définition ultérieure de celle-ci.

Comme déjà dit, on note $S(I) \Rightarrow T(J)$ une dépendance de $S(I)$ vers $T(J)$. La relation $\Rightarrow$ définit un ordre partiel sur l'exécution des instances d'instruction, sous-ordre de l'ordre total $<_{seq}$. Tout ordre d'exécution qui respecte l'ordre partiel $\Rightarrow$ conduit à un code sémantiquement équivalent [1]. L'objet des techniques de détection du parallélisme est de déterminer de tels ordres d'exécution, et en particulier des ordres dont les plus longues chaînes (d'éléments en relation) soient les plus courtes possible, i.e. des ordres qui révèlent un parallélisme maximal.

8.2.3 Calcul (approché) des dépendances

De nombreux tests pour l'analyse de dépendance ont été proposés ces vingt dernières années. Ces tests utilisent diverses représentations (ou *abstractions*) des dépendances. Parmi ces abstractions, nous nous limiterons aux distances de dépendance, aux niveaux de dépendance, et aux vecteurs de direction de dépendance. S'il ne fallait citer qu'un seul logiciel du domaine public pour l'analyse de dépendance, ce serait Omega [57] : `http://www.cs.umd.edu/projects/omega/`.

Supposons que $S(I)$ et $T(J)$ accèdent au même tableau a, par exemple S pour une écriture et T pour une lecture :

$$\begin{array}{rccc} S(I): & a(f(I)) & = & \ldots \\ T(J): & \ldots & = & a(g(J)) \end{array}$$

L'accès au tableau a est commun si :

$$f(I) = g(J)$$

[1]Pour autant que les dépendances de données soient les seules dépendances à considérer.

condition qui peut être vérifiée en temps polynômial, quand f et g sont des fonctions affines à coefficients entiers (système d'équations Diophantiennes). Le fait que I et J sont deux vecteurs d'itération valide s'exprime par les relations

$$I \in \mathcal{D}_S \quad \text{et} \quad J \in \mathcal{D}_T$$

qui peuvent être vérifiées, si les domaines $\mathcal{D}_S$ et $\mathcal{D}_T$ sont les points entiers d'un polyèdre, par des techniques de programmation linéaire en nombres entiers (ou par des techniques plus directes si les domaines d'itération sont simples – rectangles, triangles, etc). Chercher une dépendance de flot (directe ou transitive) revient à vérifier si l'écriture se produit avant la lecture dans l'ordre séquentiel,

$$S(I) <_{seq} T(J)$$

condition qui, on l'a vu, s'exprime comme une disjonction d'égalités ou inégalités en les composantes des vecteurs d'itération I et J. La recherche d'anti-dépendances ou de dépendances de sorties est similaire. Avant de traiter un exemple, notons que la recherche de dépendances *directes* est plus coûteuse : ainsi pour une dépendance de flot dirigée vers $T(J)$, il faut trouver l'opération d'écriture (dans la case mémoire lue par $T(J)$) *la plus récente*. Cela peut se faire via des techniques de programmation linéaire capable de trouver les minima ou maxima lexicographiques dans des polyèdres.

Revenons sur l'exemple de la Figure 8.1. Cherchons à caractériser une dépendance due à l'accès en écriture sur $a(i+j)$ et en lecture sur $a(i+j-1)$. On cherche une dépendance entre une opération d'écriture $S(i',j')$ et une opération de lecture $S(i,j)$, où i et j sont donnés. L' équation $f(I) = g(J)$ s'écrit

$$i' + j' = i + j - 1$$

L'équation $S(i',j') <_{seq} S(i,j)$ s'écrit

$$(i' \leq i - 1) \text{ ou } (i = i' \text{ et } j' \leq j - 1)$$

Ainsi, pour trouver une dépendance de flot directe de $S(i',j')$ vers $S(i,j)$, il faut résoudre le problème lexicographique

$$\max_{<_{seq}}\{(i',j') \mid (i',j') <_{seq} (i,j),\ i' + j' = i + j - 1,\ 1 \leq i, i', j, j' \leq N\}$$

dont la solution est :
- $(i, j-1)$ si $j \geq 2$ et
- $(i-1, j)$ if $j = 1$.

De même, trouver une anti-dépendance directe de $S(i,j)$ vers $S(i',j')$ revient à résoudre le problème lexicographique

$$\min_{<_{seq}}\{(i',j') \mid (i,j) <_{seq} (i',j'),\ i' + j' = i + j - 1,\ 1 \leq i, i', j, j' \leq N\}$$

dont la solution est $(i+1, j-2)$ si $j \geq 3$ et $i \leq N-1$. Au total, on retrouve bien les deux dépendances de flot dont les vecteurs de distance sont $(1,0)$ et $(0,1)$, et l'anti-dépendance dont le vecteur de distance est $(1,-2)$.

8.2.4 Approximation des dépendances

Les dépendances entre les instances d'instructions définissent un ensemble de contraintes qui peuvent être représentées par un graphe orienté, appelé *graphe de dépendance étendu*, ou GDE. Les sommets du GDE sont toutes les instances $\{S_i(I) \mid 1 \leq i \leq s \text{ et } I \in \mathcal{D}_{S_i}\}$, où s est le nombre d'instructions dans le programme considéré. Dans le GDE, il y a une arête $S(I) \Rightarrow T(J))$ pour chaque dépendance. L'ensemble des paires de dépendance entre les instructions S et T est

$$\{(I,J) \mid S(I) \Rightarrow T(J)\} \ \subset \mathbb{Z}^{n_S} \times \mathbb{Z}^{n_T}$$

où n_S (resp. n_T) est le nombre de boucles englobant S (resp. T), et l'ensemble de distance de ces dépendances est

$$\{(\tilde{J} - \tilde{I}) \mid S(I) \Rightarrow T(J)\} \ \subset \mathbb{Z}^{n_{S,T}}$$

où $n_{S,T}$ est le nombre de boucles englobant à la fois S et T ; comme plus haut, $\tilde{I}$ (resp. $\tilde{J}$) est le vecteur formé des $n_{S,T}$ premières composantes de I (resp. J). Une dépendance est toujours orientée selon l'ordre séquentiel : si $S(I) \Rightarrow T(J)$, alors le vecteur de distance $(\tilde{J} - \tilde{I})$ est lexico-positif : $(\tilde{J} - \tilde{I}) \geq_{lex} \mathbf{0}$; de plus, si $S \geq_{text} T$, alors $(\tilde{J} - \tilde{I}) \neq \mathbf{0}$.

En général, on ne peut pas, à la compilation, calculer le GDE, i.e. l'ensemble des paires de dépendance et leurs distances : certains paramètres peuvent ne pas être instanciés, et le calcul exact des accès mémoire peut s'avérer impossible. De toute façon, on ne souhaite pas vraiment calculer le GDE : sa taille, proportionnelle aux nombre d'opérations du nid de boucles, devient rapidement prohibitive. On souhaite au contraire disposer d'un *graphe de dépendance réduit*, ou GDR, dont les sommets soient les instructions elle-mêmes, et non plus leurs instances. Ainsi dans l'exemple de la Figure 8.1 on passerait de N^2 sommets pour le GDE à un seul, puisqu'il n'y a qu'une instruction ! On définit le GDR comme suit : il y a une arête $e = (S,T) : S \to T$ dans le GDR s'il existe au moins une paire (I,J) telle que $S(I) \Rightarrow T(J)$. Chaque arête $e = (S,T)$ du GDR a une étiquette $w(e)$ qui décrit un sous-ensemble D_e de $\mathbb{Z}^{n_{S,T}}$. Ces étiquettes doivent spécifier une sur-approximation des ensembles de distance du GDE pour garantir l'équivalence sémantique du programme. En d'autres termes :

$$\begin{gathered}\text{Si } S(I) \Rightarrow T(J) \text{ (dans le GDE) alors} \\ \exists e = (S,T) \text{ (dans le GDR) telle que } (\tilde{J} - \tilde{I}) \in D_e\end{gathered}$$

Dans la suite, nous donnons deux représentations (ou abstractions) classiques des ensembles de distance : les *niveaux de dépendance*, et les *vecteurs de direction*.

Niveaux de dépendance Les dépendances dans les boucles sont traditionnellement classifiées en deux catégories. Une dépendance entre deux instances $S(I)$ et $T(J)$ est *boucle-indépendante* si elle a lieu au cours d'une même itération des boucles qui englobent à la fois S et T, i.e. si $\tilde{I} = \tilde{J}$. Sinon, la dépendance est *portée par la boucle* ; elle a lieu entre deux instances qui correspondent, pour au moins l'une des boucles englobantes, à des valeurs différentes du compteur de boucle.

Plus formellement, une arête $e = (S,T)$ du GDR est étiquetée par la valeur $l(e) \in [1 \ldots n_{S,T}] \cup \{\infty\}$ définie comme suit :

- $l(e) = \infty$ si $S(I) \Rightarrow T(J)$ avec $\tilde{J} - \tilde{I} = \mathbf{0}$.
- $l(e) \in [1 \ldots n_{S,T}]$ si $S(I) \Rightarrow T(J)$ et la première composante non nulle de $\tilde{J} - \tilde{I}$ est la $l(e)$-ème composante.

Quand $l(e) \leq n_{S,T}$, la dépendance est portée par la boucle de niveau $l(e)$, et quand $l(e) = \infty$, la dépendance est boucle-indépendante. On construit ainsi ce qu'on appelle le *graphe de dépendance réduit par niveaux*, ou GDRN.

Vecteurs de direction Dans l'exemple de la Figure 8.1, il y a trois paires de références au tableau a qui induisent une dépendance : la paire *lecture de* $a(i+j-1)$*-écriture de* $a(i+j)$, la paire *écriture de* $a(i+j)$*-lecture de* $a(i+j-1)$, et la paire *écriture de* $a(i+j)$*-écriture de* $a(i+j)$. Les ensembles de distance sont très simples ; celui associé à l'anti-dépendance est le singleton $\{(1,-2)\}$, celui associé à la dépendance de flot est la paire $\{(1,0),(0,1)\}$, et celui associé à la dépendance de sortie est le singleton $\{(1,-1)\}$.

Quand l'ensemble des vecteurs de distance est fini (comme dans cet exemple), ou plus précisément quand sa taille ne dépend pas (des paramètres) de la taille du domaine d'itération, on dit la dépendance est *uniforme*. Sinon, on peut quand même représenter un ensemble de distance entre S et T par un vecteur de dimension $n_{S,T}$, appelé *vecteur de direction*, dont les composantes sont prises dans $\mathbb{Z} \cup \{*,+,-\} \cup (\mathbb{Z} \times \{+,-\})$. La i-ème composante du vecteur de direction est une approximation des i-èmes composantes de tous les vecteurs de distance possibles : elle est égale à $z+$ (resp. $z-$) si toutes les i-èmes composantes sont plus grandes (resp. plus petites), au sens large, que $z \in \mathbb{Z}$; elle est égale à $*$ si les i-èmes composantes peuvent prendre n'importe quelle valeur ; enfin, elle est égale à z si la dépendance est uniforme dans cette dimension, avec pour unique valeur $z \in \mathbb{Z}$. En général, on écrit $+$ (resp. $-$) plutôt que $1+$ (resp. $(-1)-$).

Voici un exemple (nid de boucles de Petersen) :

```
DO i=2, N
   S1 : s(i) = 0
   DO j=1, i-1
      S2 : s(i) = s(i) + a(j,i) * b(j)
   ENDDO
   S3 : b(i) = b(i) - s(i)
ENDDO
```

Le GDRN (avec les niveaux de dépendance) est donné à la Figure 8.2. Le GDR étiqueté par les vecteurs de directions est donné à la Figure 8.3. Pour simplifier ces figures, seules les dépendances directes sont représentées, et pour les multi-arêtes on trace une seul arête multi-étiquetée.

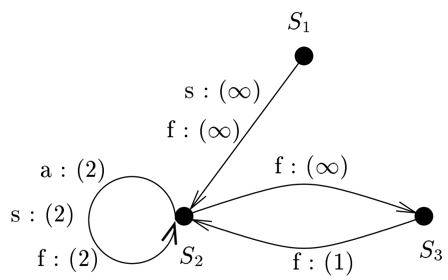


FIG. 8.2 – GDRN pour le nid de boucles de Petersen.

Considérons d'abord les accès au tableau s. Pour chaque itération de la boucle i, $s(i)$ est initialisé par S_1, et lu par S_2 à la première itération de la boucle j, donc il y a une dépendance de flot, boucle-indépendante, de S_1 à S_2 : c'est une dépendance de niveau ∞ et dont le vecteur de direction est $\mathbf{0}$. Il y a également une dépendance de sortie identique de S_1 vers S_2. Au cours des itérations suivantes de la boucle j, $s(i)$ est lu puis ré-écrit, d'où (d'un coup !) une dépendance de flot, une dépendance de sortie, et une anti-dépendance de S_2 vers S_2, de vecteur de direction $(0,1)$, de niveau 2. Enfin, $s(i)$ est lu par S_3 alors que sa mise à jour la plus récente a eu lieu à la dernière itération de la boucle j, d'où une dépendance de flot, boucle-indépendante, de S_2 vers S_3.

Pour ce qui concerne le tableau b : $b(i)$ est écrit une seule fois par $S_3(i)$ mais est lu par toutes les instances $S_2(i',i)$ telles que $1 \leq i \leq i'-1$. Comme $i' \geq i+1$, ces deux références au tableau b

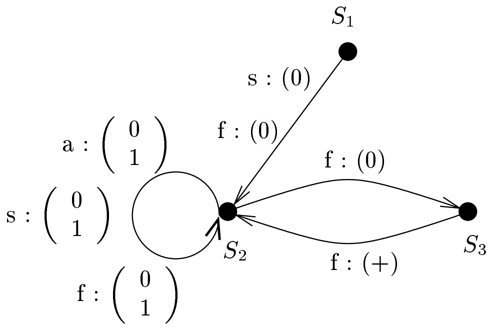


FIG. 8.3 – GDR avec vecteurs de direction pour le nid de boucles de Petersen.

n'induisent que des dépendances de flot, de distances $(i' - i) \geq 1$, d'où l'abstraction avec le vecteur de direction $(+)$ et le niveau 1.

8.2.5 Limitations des abstractions de dépendance

Considérons l'exemple simple du noyau SOR suivant :

```
DO i=1, N
   DO j=1, N
      a(i,j) = a(i,j-1) + a(i-1,j)
   ENDDO
ENDDO
```

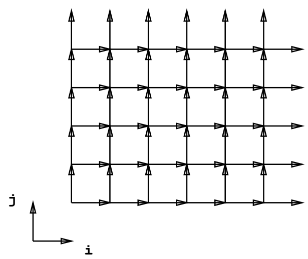


FIG. 8.4 – GDE du noyau SOR.

Le GDE est donné à la Figure 8.4. Le plus long chemin de dépendance dans le GDE est de longueur $2 * N$ (tout chemin du coin inférieur gauche au coin supérieur droit). Le GDR n'a qu'un seul sommet et deux arêtes, l'une de niveau 1 correspondant au vecteur de dépendance uniforme $(1, 0)$, et l'autre de niveau 2, correspondant au vecteur de dépendance uniforme $(0, 1)$, cf. Figure 8.5.

FIG. 8.5 – GDRN et GDR avec vecteurs de direction pour le noyau SOR.

Le GDRN est une bien pauvre approximation du GDE, car toutes les dépendances sont uniformes : les niveaux 1 et 2 sont équivalents aux vecteurs de direction $(+,*)$ et $(0,+)$. En fait, le nid de boucles suivant

```
DO i=1, N
  DO j=1, N
    a(i,j) = a(i,j-1) + a(i-1,N)
  ENDDO
ENDDO
```

a le même GDRN que le noyau SOR, alors que son GDE, représenté à la Figure 8.6, contient un chemin de dépendance de longueur N^2 : toutes les opérations doivent être exécutées séquentiellement. Il est important de noter que rien ne distingue plus ce dernier code du noyau SOR si l'on utilise les niveaux de dépendance comme approximation. Le parallélisme pourtant bien présent dans le noyau SOR ne pourra jamais être détecté à partir de son GDRN, quelle que soit la finesse de l'algorithme de parallélisation utilisé pour la manipulation du GDRN. En d'autres termes, rien ne sert d'utiliser une détection de parallélisme trop sophistiquée si on l'applique sur une approximation rudimentaire. Comme on le verra dans la suite, les niveaux de dépendance constituent l'abstraction adaptée à l'algorithme d'Allen et Kennedy, tandis que les vecteurs de direction sont le bon outil pour les transformations unimodulaires de Lamport.

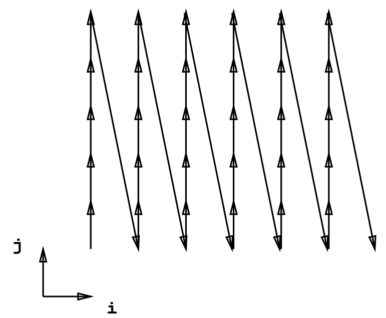


FIG. 8.6 – GDE du code séquentiel dont le GDRN est le même que celui du noyau SOR.

8.3 Algorithme d'Allen et Kennedy

L'algorithme développé par Allen et Kennedy [6], puis par by Allen, Callahan, et Kennedy [5], utilise les niveaux de dépendance comme abstraction des dépendances. Le code initial est un nid de boucles quelconque, parfait ou non. Les boucles qui englobent une instruction donnée ne sont pas changées, seule l'imbrication globale des boucles peut être modifiée par l'algorithme, qui a deux objectifs :

- pour chaque instruction, détecter le nombre maximal de boucles parallèles qui l'englobent, et
- une fois le premier objectif atteint, minimiser le nombre de points de synchronisation dans le code parallélisé.

Nous nous limitons ici au premier objectif. Les transformation à la base de l'algorithme sont la distribution de boucle, et la transformation inverse, la fusion de boucle, dont nous commençons par rappeler le principe.

8.3.1 Distribution de boucle

La distribution de boucle est basée sur la propriété suivante :

Proposition 15 *Toute instance d'une instruction (ou bloc d'instructions) S peut être exécutée avant toute instance d'une instruction T si on n'a jamais $T(J) \Rightarrow S(I)$, i.e. par définition de $T \rightarrow S$, s'il n'y a pas de dépendance de T vers S.*

(a)	(b)	(c)	(d)

DO i=..., ...
  S_1
  S_2
ENDDO
	DO i=..., ...
	  S_1
	ENDDO
	DO i=..., ...
	  S_2
	ENDDO
		DO i=..., ...
		  S_2
		ENDDO
		DO i=..., ...
		  S_1
		ENDDO
			DO i=..., ...
			  S_2
			  S_1
			ENDDO

FIG. 8.7 – Distribution de boucle : les divers cas.

La Figure 8.7 illustre le cas le plus simple de la distribution de boucle : deux instructions S_1 et S_2 au sien d'une unique boucle (Figure 8.7(a)), où S_1 précède textuellement S_2. Il y a quatre cas à considérer :

- $S_1 \rightarrow S_2$ mais il n'y a pas de dépendance de S_2 vers S_1. La distribution de boucle conduit au code de la Figure 8.7(b).
- $S_2 \rightarrow S_1$ mais il n'y a pas de dépendance de S_1 vers S_2. La distribution de boucle conduit au code de la Figure 8.7(c).
- Il n'y a pas de dépendance entre S_1 et S_2. Les quatre codes de la Figure 8.7 sont valides.
- $S_1 \rightarrow S_2$ et $S_2 \rightarrow S_1$. La distribution de boucle n'est pas valide. Le code initial de la Figure 8.7(a) est valide, bien sûr, de même que le code de la Figure 8.7(d) s'il n'y a pas de dépendance boucle-indépendante, i.e. si toutes les dépendances sont portées par la boucle.

La distribution de boucle maximale s'obtient ainsi :

1. Calcul des composantes fortement connexes (CFC) du GDR.
2. Tri topologique[2] des CFC : numérotation des CFC de telle sorte que pour toute arête $S \rightarrow T$, S appartient à une CFC C_i et T à une CFC C_j avec $i \leq j$.
3. Distribution des boucles autour des CFC : on duplique les structures de boucles autour de chaque CFC. Les instructions d'une même CFC sont conservées dans le même ordre textuel, et les codes correspondant aux différentes CFC sont ordonnés selon l'ordre topologique.

8.3.2 Algorithme générique

On présente une version simplifiée de l'algorithme d'Allen et Kennedy qui marque "parallèles" autant de boucles que possible pour chaque instruction.

Définition 20 *Une boucle est parallèle si toutes les itérations de la boucle peuvent être exécutées simultanément et dans n'importe quel ordre, i.e. s'il n'y a pas de dépendance portée par la boucle. En d'autres termes, une boucle de profondeur k est parallèle s'il n'y a pas de dépendance de niveau k dans le GDRN du code englobé par la boucle.*

En plus de la notation **DOPAR** pour les boucles parallèles, on notera **DOSEQ** une boucle séquentielle au sens classique, pour faire la distinction entre une boucle **DO** boucle qui peut être parallèle et une boucle **DO** que l'algorithme d'Allen et Kennedy a reconnu comme n'étant pas une

[2] Ceci est toujours possible car le graphe des CFC est acyclique par définition.

boucle **DOPAR**. Notons que la boucle **DOPAR** n'est pas la boucle FORALL de High Performance Fortran (HPF) [45] décrite au Paragraphe 2.2, mais la boucle INDEPENDENT en HPF (ou son équivalent dans les dialectes similaires comme OPENMP, SUNMP, ou les directives CRAY).

L'idée à la base de l'algorithme d'Allen et Kennedy est d'utiliser la distribution de boucle pour diminuer le nombre d'instructions à l'intérieur d'une boucle, et partant réduire le nombre de dépendances potentielles. Après distribution, la détection du parallélisme sera menée indépendamment dans chaque CFC du GDRN : la boucle la plus externe sera marquée **DOPAR** dans certaines composantes et **DOSEQ** dans d'autres. L'algorithme se poursuit alors récursivement.

Rappelons que n_S est le nombre de boucles englobant une instruction S. Pour un GDRN G, on note $l_{\min}(G)$ le niveau minimal d'une arête de G :

$$l_{\min}(G) = \min\{l(e) \mid e \in G\}$$

Voici un formulation générique de l'algorithme ; pour l'appel initial, $k = 1$, et G est le GDRN du code à paralléliser :

ALLEN-KENNEDY(G, k)

1. Supprimer dans G toutes les arêtes de niveau $< k$.
2. Calculer les composantes fortement connexes (CFC) G.
3. **Pour chaque** CFC C dans l'ordre topologique **faire**

   Si C est réduit à une seule instruction S, sans arête,

   Alors

   Générer des boucles **DOPAR** dans toutes les dimensions restantes, i.e. du niveau k au niveau n_S, et générer le code pour S.

   Sinon

   (a) Soit $l = l_{\min}(C)$.

   (b) Générer des boucles **DOPAR** boucles du niveau k au niveau $l - 1$, et une boucle **DOSEQ** pour le niveau l.

   (c) Appeler ALLEN-KENNEDY(C, $l + 1$).

Illustrons le fonctionnement de l'algorithme sur l'exemple de la Figure 8.8. Le pseudo-code est donné Figure 8.8(a), et le GDRN correspondant est supposé être le graphe de la Figure 8.8(b).

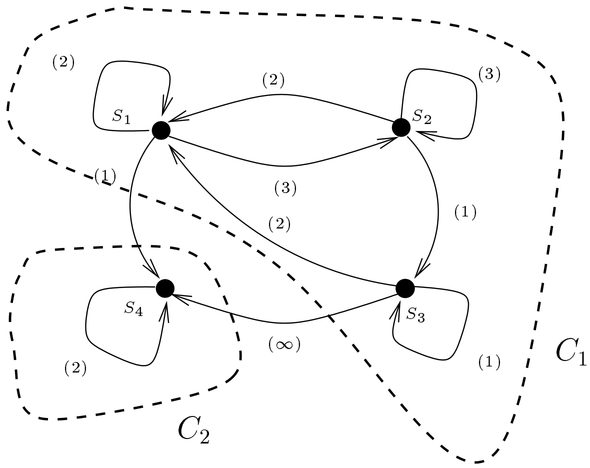
Le GDRN a deux CFC : C_1 qui comprend les instructions S_1, S_2, S_3, et C_2 avec la seule instruction S_4. C_1 a au moins une arête de niveau 1, donc la boucle la plus externe qui englobe les instructions de C_1 est marquée séquentielle. C_2 n'a pas d'arête de niveau 1 mais par contre a une arête de niveau 2, donc la boucle la plus externe est marquée parallèle, tandis que la seconde boucle est marquée séquentielle. Ainsi, après le premier appel, le code est celui de la Figure 8.9(a).

La récursion continue. Dans C_1 on supprime toutes les arêtes de niveau 1, ce qui signifie qu'on considère désormais le code pour une itération donnée de la boucle la plus externe. De même pour for C_2, on supprime toutes les arêtes de niveau < 3. On obtient le GDRNs de la Figure 8.9(b). S_1 et S_2 appartiennent à la même CFC C_3, qui contient une arête de niveau 2. Une boucle séquentielle est générée, et la récursion se poursuit. S_3 et S_4 forment chacune une CFC sans arête. On obtient alors le code de la Figure 8.10(a). Remarquer comme S_3 a été remontée avant les itérations de S_1 and S_2.

Finalement, après trois appels récursifs, seules les instructions S_1 et S_2 restent encore dans le graphe. Les arêtes de niveau 2 sont supprimées et on obtient le GDRN de la Figure 8.10(b), avec deux CFC C_4 and C_5. Les deux dernières phases sont représentées sur la Figure 8.11(a,b).

```
DO i=..., ...
  DO j=..., ...
    DO k=..., ...
      S1
      S2
    ENDDO
    S3
    S4
  ENDDO
ENDDO
```

(a)

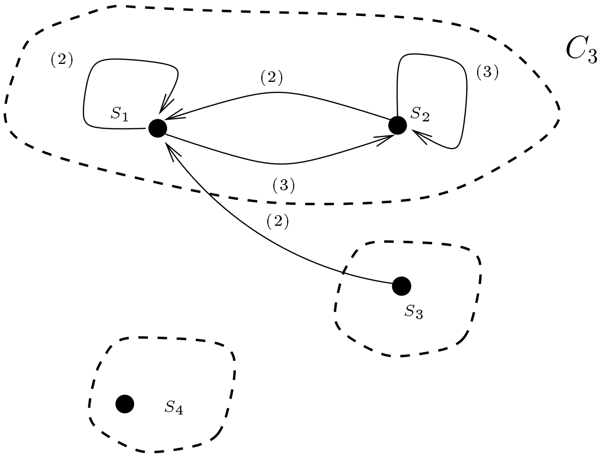


(b)

FIG. 8.8 – Pseudo-code et GDRN correspondant.

```
DOSEQ i=..., ...

  ALLEN-KENNEDY(C1, 2)

ENDDOSEQ
DOPAR i=..., ...
  DOSEQ j=..., ...

    ALLEN-KENNEDY(C2, 3)

  ENDDOSEQ
ENDDOPAR
```

(a)



(b)

FIG. 8.9 – Code et GDRN avant le deuxième appel récursif.

8.4 Transformations unimodulaires

On se restreint dans ce paragraphe au cas de nids parfaits (toutes les instructions sont englobées par n boucles), et dont toutes les dépendances sont uniformes

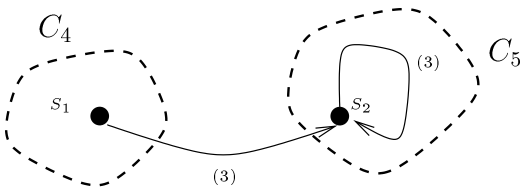
8.4.1 Définition et validité

Une transformation unimodulaire change l'ordre dans lequel le nid de boucles énumère les points du domaine d'itération : celui-ci est scanné par de nouveaux compteurs de boucle, i.e. par un nouveau vecteur d'itération I', défini à partir du vecteur d'itération I par une matrice unimodulaire $T : I' = TI$. Une matrice unimodulaire est une matrice entière carrée de déterminant égal à 1 or -1 (donc son inverse est à coefficients entiers également).

Une transformation unimodulaire ne change que l'ordre d'énumération : tout se passe comme si le corps de boucles n'était composé que d'une seule instruction. En d'autres termes, on considère que le corps du nid de boucles (supposé parfait) est atomique, et que le GDR est réduit à un seul

```
DOSEQ i=..., ...
  DOPAR j=..., ...
    S3
  ENDDOPAR
  DOSEQ j=..., ...

    ALLEN-KENNEDY(C3, 3)

  ENDDOSEQ
ENDDOSEQ
DOPAR i=..., ...
  DOSEQ j=..., ...
    S4
  ENDDOSEQ
ENDDOPAR
```

(a)



(b)

FIG. 8.10 – Code et GDRN avant le troisième appel récursif.

sommet. Ainsi, les dépendances boucle-indépendantes seront toujours préservées. Pour les autres dépendances, i.e. celles portées par certaines boucles, il y aura des conditions de validité que nous allons préciser.

Auparavant, discutons brièvement des matrices unimodulaires. Soit I_n la matrice identité de taille n, et soit $E_{i,j}$ la matrice carrée de taille n dont tous les coefficients sont nuls sauf celui de la i-ème ligne et j-ème colonne qui vaut 1. Par exemple on a $I_n = \sum_{i=1}^{n} E_{i,i}$. On définit trois types de matrices unimodulaires dites *élémentaires* :

1. $R_i = I_n - 2E_{i,i}$ est entière et égale à son inverse : $(R_i)^{-1} = R_i$.
2. $P_{i,j} = I_n - E_{i,i} - E_{j,j} + E_{i,j} + E_{j,i}$ est entière et égale à son inverse : $(P_{i,j})^{-1} = P_{i,j}$.
3. Pour $i \neq j$, $S_{i,j}(k) = I_n + kE_{i,j}$ est entière, et son inverse est $S_{i,j}(-k)$.

Toute matrice unimodulaire peut être décomposée en produit de matrices unimodulaires élémentaires [54]–On peut même imposer que les matrices $S_{i,j}(k)$ qui apparaissent dans le produit vérifient $k = 1$ et $i \geq j$. Aux matrices unimodulaires élémentaires correspondent des transformations de boucles classiques :

- *Permutation de boucle.* Avant transformation le domaine d'itération est scanné par le vecteur d'itération $(I_1, \dots, I_n)$; après transformation par la permutation σ, il est scanné par le vecteur d'itération $(I_{\sigma(1)}, \dots, I_{\sigma(n)})$. L'échange de la i-ème et de la j-éme boucle correspond à la matrice $P_{i,j}$.
- *Inversion de boucle.* Inverser la i-ème boucle revient à la parcourir en sens inverse. Le nouveau vecteur d'itération est $(I_1, \dots, I_{i-1}, -I_i, I_{i+1}, \dots, I_n)$. La transformation correspondante est R_i.
- *Torsion de boucle.* Tordre la boucle I_j d'un facteur (entier) k à l'aide la boucle I_i (avec $i < j$) revient à scanner le domaine d'itération avec le nouveau vecteur d'itération $(I_1, \dots, I_{j-1}, I_j + kI_i, I_{j+1}, \dots, I_n)$. La transformation correspondante est $S_{i,j}(k)$.

Reprenons l'exemple du noyau SOR (Figure 8.4). L'algorithme d'ALLEN-KENNEDY ne détecte aucun parallélisme dans ce code parce que l'abstraction des dépendances par niveau n'est pas assez précise. Pourtant, une simple torsion de boucle suivie d'un échange des deux boucles permet de mettre une boucle parallèle en évidence, comme le montre la Figure 8.12.

Comme le montre cet exemple, l'un des intérêts des transformations unimodulaires est la prise

```
DOSEQ i=..., ...
  DOPAR j=..., ...
   S3
  ENDDOPAR
  DOSEQ j=..., ...
    DOPAR k=..., ...
     S1
    ENDDOPAR
    DOSEQ k=..., ...
     ALLEN-KENNEDY(C4, 4)
    ENDDOSEQ
  ENDDOSEQ
ENDDOSEQ
DOPAR i=..., ...
  DOSEQ j=..., ...
   S4
  ENDDOSEQ
ENDDOPAR
```

(a)

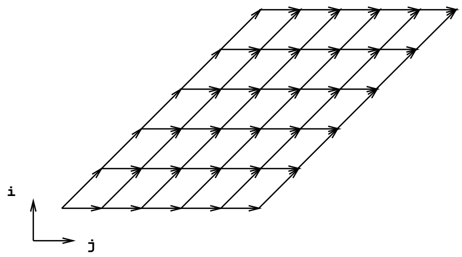
```
DOSEQ i=..., ...
  DOPAR j=..., ...
   S3
  ENDDOPAR
  DOSEQ j=..., ...
    DOPAR k=..., ...
     S1
    ENDDOPAR
    DOSEQ k=..., ...
     S2
    ENDDOSEQ
  ENDDOSEQ
ENDDOSEQ
DOPAR i=..., ...
  DOSEQ j=..., ...
   S4
  ENDDOSEQ
ENDDOPAR
```

(b)

FIG. 8.11 – Code avant et après le dernier appel récursif.

```
DO j=2, 2N
  DOPAR i=max(1,j-N), min(N,j-1)
   a(i,j-i) = a(i-1,j-i) + a(i,j-i-1)
  ENDDOPAR
ENDDO
```

(a)



(b)

FIG. 8.12 – Code parallèle et GDE (avec $N = 6$) pour le noyau SOR.

en compte d'information multi-dimensionnelle sur les vecteurs de distance. Par ailleurs, leur mise en oeuvre est aisée, tant pour ré-écrire le nid de boucles après transformation que pour tester la validité de celles-ci.

Commençons par la ré-écriture du nid. Appliquer une transformation unimodulaire T à un nid parfait revient à calculer l'instance $S(I)$ à l'itération $I' = T(I)$ plutôt qu'à l'itération I, i.e. à transformer le code de la Figure 8.13(a) en le code de la Figure 8.13(b). Ce dernier peut être lui-même ré-écrit en le code de la Figure 8.13(c) : en effet, si T est unimodulaire, T établit une bijection entre les points entiers du domaine d'itération $\mathcal{D}$ et les points entiers de $T(\mathcal{D})$.

```
DO I∈D, I entier
  S(I)
ENDDO
```

(a)

```
DO I'=T(I), I∈D, I entier
  S(I)
ENDDO
```

(b)

```
DO I'∈T(D), I' entier
  S(T^-1(I'))
ENDDO
```

(c)

FIG. 8.13 – Code avant et après une transformation unimodulaire T.

Toute transformation unimodulaire n'est pas valide : les dépendances entre les instances d'instructions doivent être préservées par la transformation. Si $S_i(I) \Rightarrow S_j(J)$ dans le code initial, on doit garantir que $S_i(I) <_{seq} S_j(J)$ dans le code transformé. Pour celui-ci, $S_i(I)$ est calculée à l'itération $T(I)$et $S_j(J)$ à l'itération $T(J)$. Les dépendances boucle-indépendantes ($I = J$ and $S_i <_{text} S_j$) sont préservées automatiquement, l'ordre textuel n'étant pas modifié. Mais pour une dépendance de vecteur de distance non nul, on doit garantir $T(I) <_{lex} T(J)$, i.e. $T(J - I) >_{lex} \mathbf{0}$ car T est linéaire. D'où la condition de validité :

Proposition 16 *Une transformation unimodulaire T est valide si et seulement si l'image $T(d)$ de tout vecteur de distance non nul d est lexicographiquement strictement positif : $d \neq \mathbf{0} \Rightarrow T(d) >_{lex} \mathbf{0}$.*

La condition de validité de la Proposition 16 est particulièrement facile à tester quand le nombre de vecteurs de distance différents est borné (fini et non paramétré). C'est le cas pour les nids de boucles uniformes, ce qui explique que nous nous restreignions à ceux-ci.

8.4.2 La méthode de l'hyperplan

Considérons un nid de boucles à dépendances uniformes : soit D l'ensemble des m vecteurs de dépendance. Chacun d'entre eux est lexicographiquement positif (par définition de l'orientation des dépendances) et non nul (on ne considère pas les dépendances boucles-indépendantes) : si $d \in D$, $T(d) >_{lex} \mathbf{0}$. Pour simplifier les notations, on identifie transformation et matrice T, et on écrit $T(d)$ comme un produit matrice-vecteur Td. Pour s'assurer que $Td >_{lex} \mathbf{0}$, i.e. que la première composante non nulle de Td soit positive, l'idée naturelle est d'appliquer une stratégie gloutonne : transformer tous les vecteurs de dépendance de manière à ce que leur *première* composante soit positive. Si une telle matrice T existe, toutes les dépendances auront été transformées en dépendances de niveau 1, et seront portées par la boucle la plus externe. Le code après transformation sera composé de $n - 1$ boucles parallèles englobées par une boucle séquentielle. Cette technique est connue sous le nom de *méthode de l'hyperplan* de Lamport [47], et ce pour la raison suivante : soit s la première ligne de T ; transformer toutes les dépendances en dépendances de niveau 1 est équivalent à obtenir $sd \geq 1$ pour tout vecteur de dépendance d. Pour toute constante c, toutes les instances d'instruction $S(I)$ telles que $sI = c$ peuvent être exécutées en parallèle. Ces opérations appartiennent toutes à un hyperplan affine (ou front d'onde) normal à s. Le flot des calculs progresse selon la direction s. Le vecteur s est souvent appelé *vecteur de temps.*

Théorème 13 *Etant donné un ensemble D de m vecteurs lexicographiquement (strictement) positifs, il existe un vecteur entier s tel que $sd \geq 1$ pour tout $d \in D$.*

Preuve La preuve est constructive. Soit D_i l'ensemble des vecteurs de D dont la première composante non nulle est la i-ème composante. Comme tous les vecteurs de D sont lexicographiquement positifs, pour tout $d \in D_i$, on a $d(1) = \ldots = d(i-1) = 0$ et $d(i) > 0$. On construit les composantes de s par récurrence, en commençant par la dernière. Supposons que $s(i+1), \ldots, s(n)$ aient déjà été construits de telle sorte que, pour tout $k > i$ et pour tout $d \in D_k$, $sd \geq 1$. Si D_i est vide, on pose $s(i) = 0$, sinon on pose

$$s(i) = \max \left\{ \left\lceil \frac{1 - \sum_{k=i+1}^{n} s(k)d(k)}{d(i)} \right\rceil \middle| \ d \in D_i \right\}.$$

Alors, pour tous les vecteurs d précédents, i.e. pour tout $k > i$ et pour tout $d \in D_k$, la valeur de sd est inchangée car $d(i) = 0$. De plus, pour tout $d \in D_i$, $sd = s(i)d(i) + \sum_{k=i+1}^{n} s(k)d(k)$ est, par construction, supérieur ou égal à 1. ■

Revenons à l'exemple de la Figure 8.1, qui a quatre vecteurs de dépendance uniformes : $D_1 = \{(1,-1),(1,-2),(1,0)\}$ et $D_2 = \{(0,1)\}$. En suivant la construction de la preuve, on obtient $s(2) = 1$, puis :

$$s(1) = \max\{(1-(-1))/1,(1-(-2))/1,(1-0)/1\} = 3$$

D'où $s = (3,1)$.

Une fois qu'on a le vecteur de temps s, reste à construire la matrice unimodulaire T. Notons d'abord qu'on peut toujours choisir les composantes de s premières entre elles : si $s = ps'$,

$$sd \geq 1 \Rightarrow sd > 0 \Rightarrow ps'd > 0 \Rightarrow s'd > 0 \Rightarrow s'd \geq 1$$

car s' et d sont des vecteurs entiers. Supposons donc s entier et de composantes premières entre elles. Pour construire une matrice unimodulaire T dont la première ligne est s, on peut utiliser la forme normale de Hermite [62, p. 45] : $s = [1\ 0 \ldots 0]U$, avec U unimodulaire .

Poursuivant l'exemple, la forme normale de Hermite de s est

$$s = (3,1) = (1,0)\begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix} \text{ et } T = \begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 3 & 1 \end{pmatrix}$$

La transformation T correspond à une torsion de boucle de facteur 3, suivi d'un échange de boucle. Le code après transformation est le suivant :

```
DO j=4, 4N
  DOPAR i= max(1, ⌈(j-N)/3⌉), min(N, ⌊(j-1)/3⌋)
   a(j-2i) = a(j-2i-1)+1
  ENDDOPAR
ENDDO
```

On peut vérifier directement la validité de la transformation. La boucle séquentielle s'interprète comme une boucle de temps, et les opérations de la boucle parallèle peuvent être confiés à des processeurs différents : la deuxième ligne de T s'interprète comme la fonction d'allocation des calculs aux processeurs.

8.4.3 Recherche du vecteur de temps optimal

Considérons un nid de boucles parfait à dépendances uniformes et de profondeur n. La Proposition 13 fournit une méthode pour trouver un vecteur de temps. Mais comment trouver l'optimal, défini comme celui qui minimise le temps d'exécution total ? On suppose une infinité de ressources : le temps d'exécution est alors le nombre de pas de la boucle (séquentielle) la plus externe, puisque les $n-1$ boucles internes sont parallèles.

Formalisons un peu. A l'intérieur des n boucles du nid parfait on a une seule instruction S (ou un seul bloc d'instruction considéré comme atomique). On a m vecteurs de dépendance uniformes (entiers et lexicographiquement strictement positifs), que l'on regroupe dans une matrice $D = (d_1, \ldots, d_m)$ à n lignes et m colonnes ; D est appelée *matrice de dépendance*. Le domaine d'itération est l'ensemble des points entiers d'un polyèdre :

$$J = J(A,b) = \{x \in \mathbb{Z}^n \mid Ax \leq b\}$$

A est une matrice de taille $l \times n$, et b est un vecteur à l composantes ; on suppose que A et b définissent bien un domaine d'itération fini.

Pour des raisons techniques, on généralise un peu ce qui précède en utilisant des vecteurs de temps à composantes rationnelles, et non plus entières. Un ordonnancement linéaire de vecteur de temps $s \in \mathbb{Q}^n$ est défini comme la fonction de J dans $\mathbb{N}$:

$$T_s(p) = \lfloor sp \rfloor + c_s \text{ où } c_s = -\min_{p \in J} \lfloor sp \rfloor$$

Le rôle de la constante c_s est simplement de faire débuter l'ordonnancement au temps 0 : pour tout $p \in J,\ .\ T_s(p) \geq 0$. On montre alors que l'ordonnancement est valide (respecte les dépendances) pour tout vecteur $s \in S(D)$, où

$$S(D) = \{s \in \mathbb{Q}^n \mid sD \geq \mathbf{1}\}$$

(on simplifie les notations en considérant s comme un vecteur ligne : on a $sd_i \geq 1$ pour tout i). En effet, soient $p, q \in J$ deux points du domaine d'itération correspondant à des instances d'instruction en dépendance directe : $S(p) \Rightarrow S(q)$. On a $q = p + d_k$ pour un certain k, $1 \leq k \leq m$. Alors, $sq = s(p + d_k) = sp + sd_k \geq sp + 1 \geq \lfloor sp \rfloor + 1$. Donc $\lfloor sq \rfloor \geq \lfloor sp \rfloor + 1$ et $T_s(q) \geq T_s(p) + 1$. T_s est bien valide.

Ce petit détour par les rationnels va nous simplifier la vie dans un instant. Le temps total d'exécution d'un ordonnancement linéaire, ou sa latence, est

$$\begin{aligned} L_s &= 1 + \max\{T_s(p) \mid p \in J(A,b)\} - \min\{T_s(p) \mid p \in J(A,b)\} \\ L_s &= 1 + \max\{\lfloor sp \rfloor \mid p \in J(A,b)\} - \min\{\lfloor sq \rfloor \mid q \in J(A,b)\} \\ L_s &= 1 + \max\{\lfloor sp \rfloor - \lfloor sq \rfloor \mid p, q \in J(A,b)\} \end{aligned}$$

On cherche le vecteur de temps valide de latence minimale $L_{\min} = \min\{L_s \mid s \in S(D)\}$.

Pour formuler le problème comme un problème de programmation linéaire (en rationnels), on supprime les parties entières et on définit

$$L_s^* = \max\{s(p-q) \mid p, q \in J(A,b)\} \text{ et } L_{\min}^* = \min\{L_s^* \mid s \in S(D)\}$$

L'approximation reste bonne, car on a $L_{\min} \leq L_{\min}^* + 2$. Pour le voir, notons $d(p,q) = \lfloor sp \rfloor - \lfloor sq \rfloor$: $d(p,q) = \lfloor (sq - \lfloor sq \rfloor) + s(p-q) \rfloor$. Comme $0 \leq sq - \lfloor sq \rfloor < 1$, on a $s(p-q) \leq (sq - \lfloor sq \rfloor) + s(p-q) < s(p-q) + 1$, d'où $s(p-q) - 1 < d(p,q) < s(p-q) + 1$. Soit alors $s \in S(D)$. Comme $J(A,b)$ est fini, la latence L_s de T_s est atteinte en un couple de points donnés p et q : $L_s = 1 + d(p,q)$. Comme $d(p,q) < s(p-q) + 1$, on a $L_s < s(p-q) + 2 \leq L_s^* + 2$. De plus, $L_s \geq L_{\min}$. Ceci prouve que pour tout $s \in S(D)$, $L_{\min} < L_s^* + 2$, d'où enfin $L_{\min} \leq L_{\min}^* + 2$.

La solution est maintenant à portée de main ! on cherche un vecteur $s \in S(D)$ qui minimise L_s^*, avec

$$L_s^* = \max\{s(p-q) \mid Ap \leq b,\ Aq \leq b\}$$

$L_{\min}^*$ est la solution d'un problème de min-max :

$$L_{\min}^* = \min\{\max\{s(p-q) \mid Ap \leq b,\ Aq \leq b\} \mid sD \geq \mathbf{1}\}$$

Mais par le théorème de dualité de la programmation linéaire (Corollaire 7.1g de [62]),

$$\begin{aligned} L_s^* &= \max\{s(p-q) \mid Ap \leq b,\ Aq \leq b\} \\ L_s^* &= \min\{(s_1 + s_2)b \mid s_1 \geq \mathbf{0},\ s_2 \geq \mathbf{0},\ s_1 A = s,\ s_2 A = -s\} \end{aligned}$$

Par suite :

$$L^*_{\min} = \min\{(s_1+s_2)b \mid s_1 \geq \mathbf{0},\ s_2 \geq \mathbf{0},\ s_1A = s,\ s_2A = -s,\ sD \geq \mathbf{1}\}$$

Ce problème d'optimisation est linéaire en b : la recherche du vecteur de temps s optimal pour la famille de domaines d'itération $J(A, kb)$ (où $k > 0$ est un paramètre de taille) se réduit à la recherche sur le domaine $J(A, b)$, parce que minimiser $(s_1 + s_2)kb$ équivaut à minimiser $(s_1 + s_2)b$ si $k > 0$.

Illustrons la recherche du meilleur vecteur de temps sur un exemple classique dans la littérature, du à Peir et Cytron [55] :

```
DO i = 0, n
  DO j = 0, n
    a(i,j) = b(i,j-6) + d(i-1,j+3)
    b(i+1,j-1) = c(i+2,j+5)
    c(i+3,j-1) = a(i,j-2)
    d(i,j-1) = a(i,j-1)
  ENDDO
ENDDO
```

On voit bien l'intérêt des transformations unimodulaires, et plus généralement des techniques de compilation, sur cet exemple un peu plus compliqué que d'habitude. Aucune des deux boucles ne peut être marquée comme parallèle, mais il y a du parallélisme ...bien caché dans ce code. Le domaine d'itération est un carré dont le côté est paramétré par N : $J = J(A, Nb)$ avec

$$A = \begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} \quad b = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

La matrice de dépendance est

$$D == \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 2 & -6 & 5 & 1 & -4 \end{pmatrix}.$$

Soit $s = (a, b)$, $s_1 = (a_1, b_1, c_1, d_1)$ et $s_2 = (a_2, b_2, c_2, d_2)$. On doit résoudre le problème d'optimisation suivant :

$$\begin{cases} sD \geq \mathbf{1} : 2b \geq 1, a - 6b \geq 1, a + 5b \geq 1, b \geq 1, a - 4b \geq 1 \\ s_1A = s : -a_1 + b_1 = a, -c_1 + d_1 = b \\ s_2A = -s : -a_2 + b_2 = -a, -c_2 + d_2 = -b \\ s_1 \geq 0 : a_1 \geq 0, b_1 \geq 0, c_1 \geq 0, d_1 \geq 0 \\ s_2 \geq 0 : a_2 \geq 0, b_2 \geq 0, c_2 \geq 0, d_2 \geq 0 \\ \min{(s_1 + s_2)b} : b_1 + b_2 + d_1 + d_2 \end{cases}$$

On peut utiliser la méthode du simplexe fournie dans des logiciels comme Maple [20]. On obtient la solution $s = (7, 1)$, $s_1 = (0, 7, 0, 1)$ et $s_2 = (7, 0, 1, 0)$. La latence optimale pour le domaine $J(A, Nb)$ est $N \times (7 + 0 + 1 + 0) = 8N$.

8.4.4 Une généralisation aux vecteurs de direction

Ce paragraphe technique a pour but d'illustrer l'utilisation des transformations unimodulaires quand les dépendances sont représentées par des vecteurs de direction. Nous ne supposons plus que le nid de boucles considéré n'a que des dépendances uniformes.

Soit $D = \{d_1, \dots, d_m\}$ l'ensemble de tous les vecteurs de direction. Rappelons que leurs composantes sont de la forme z, $z+$, $z-$ ou $*$, où $z \in \mathbb{Z}$. Soit $\mathcal{J} = \{j_1, \dots, j_p\}$ l'ensemble des indices des composantes qui ne sont pas des constantes pour tous les vecteurs, auquel on ajoute l'indice "1" s'il n'est pas déjà présent. En d'autres termes, l'indice k est dans $\mathcal{J}$ si $k = 1$ ou s'il existe $d \in D$ dont la k-ème composante $d(k)$ n'est pas un entier. L'idée de Lamport [47] est que chaque $V_k = D_{j_k} \cup \dots \cup D_{j_{k+1}-1}$ peut être traité comme un ensemble de dépendances uniforme ; il construit un vecteur de temps pour chacun d'entre eux.

Théorème 14 *Par transformation unimodulaire, le nid peut être ré-écrit avec p boucles séquentielles et $n-p$ boucles parallèles, où $p = |\mathcal{J}|$ (défini plus haut).*

Preuve Soit D_i l'ensemble des vecteurs de D dont la première composante non nulle est la i-ème. La matrice unimodulaire cherchée T est construite de telle sorte que sa k-ème ligne, pour tout $k \in [1, p]$, est un hyperplan qui satisfait les dépendances de l'ensemble $V_k = D_{j_k} \cup \dots \cup D_{j_{k+1}-1}$ (en notant $j_{m+1} = n + 1$), i.e. des vecteurs de direction dont les niveaux de dépendance sont compris entre j_k et $j_{k+1} - 1$. Cette propriété assure la validité de la transformation T.

Pour prouver l'existence d'un tel hyperplan, ne considérons que les composantes dans les dimensions entre j_k et $j_{k+1} - 1$. Par projection, on conduit un nouvel ensemble V'_k à partir de V_k. Pour tout vecteur d de V_k, sa projection d' reste lexicographiquement (strictement) positive. Par définition de $\mathcal{J}$ et V_k, d' est un vecteur à composantes entières, sauf peut-être la première, qui peut avoir la forme $z+$, où z est un entier positif ou nul ; si tel est le cas, on remplace $z+$ par z dans la première composante $d'(1)$ de d'. On applique alors le Théorème 13, qui assure l'existence d'un vecteur de temps s' pour V'_k. On peut supposer le vecteur s entier, à composantes positives ou nulles (dans la preuve du Théorème 13 on peut ajouter les conditions $s(i) \geq 0$), et premières entre elles. Soit s le vecteur de $\mathbb{Z}^n$ dont la projection est s', et les autres composantes nulles. Alors pour tout vecteur $d \in V_k$, $sd = \sum_{i=1}^{n} s(i)d(i) = \sum_{i=j_k}^{j_{k+1}-1} s(i)d(i) = s'(1)d(j_k) + \sum_{i=2}^{j_{k+1}-j_k} s'(i)d'(i) \geq s'd'$, car $s'(1) \geq 0$ et $d(j_k) \geq d'(1)$. Donc $sd \geq 1$.

Nous avons donc montré l'existence des p hyperplans. Reste à construire une matrice unimodulaire dont les p premières lignes soient ces p vecteurs. Pour $k \in [1, p]$, soit s_k le k-ème vecteur, celui construit pour V_k ; s_k est le seul vecteur dans $s_1, \dots, s_p$ qui puisse avoir des composantes non nulles entre les indices j_k et $j_{k+1} - 1$. On complète la projection de s_k sur les indices j_k à $j_{k+1} - 1$ en une matrice unimodulaire de taille $j_{k+1} - j_k$ à l'aide de la forme de Hermite, dont la première ligne est (la projection de) s_k. On rajoute des zéros pour obtenir une matrice de taille $(j_{k+1} - j_k) \times n$. Ne reste qu'a réordonner les lignes pour faire apparaître les vecteurs s dans les p premières positions. ■

```
DO i=1, N
  DO j=1, N
    DO k=1, N
      a(i,j,k) = a(i-1,i,k-1) + a(i,j-1,k+3)
                 + a(i,j-1,k+1)
    ENDDO
  ENDDO
ENDDO
```

Vecteurs de direction

$$\begin{pmatrix} 1 \\ * \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ -3 \end{pmatrix}, \text{ et } \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}$$

FIG. 8.14 – Un exemple de code et les vecteurs de direction associés.

Nous illustrons la preuve sur l'exemple de la Figure 8.14. On a $\mathcal{J} = \{1, 2\}$, $V_1 = \{(1, *, 1)\}$, and $V_2 = \{(0, 1, -3), (0, 1, -1)\}$. Pour construire s_1, on considère les composantes d'indice entre 1 et 1 dans l'ensemble V_1. On arrive ainsi à l'ensemble $\{(1)\}$ et on trouve $s_1 = (1, 0, 0)$. Pour construire s_2, on considère les composantes d'indice entre 2 et 3 dans l'ensemble V_2. On arrive ainsi à l'ensemble $\{(1, -3), (1, -1)\}$ et on trouve $s_2 = (0, 1, 0)$. T est la matrice identité et le code parallélisé est identique au code initial, mais la boucle la plus interne a été identifiée comme parallèle.

La méthode de Lamport ne tire pas profit de toute l'information disponible dans les vecteurs de direction. Dans l'exemple, on peut vérifier que le produit scalaire du vecteur $(2, 0, -1)$ avec chacun des vecteurs de direction est positif. A l'aide de la matrice unimodulaire

$$T = \begin{bmatrix} 2 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

on transforme alors le code initial en :

```
DOSEQ i = -n+2, 2n-1
   DOPAR j = 1, n
     DOPAR k = max(⌊(i+2)/2⌋,1), min(⌊(i+n)/2⌋,n)
       a(k, j, 2k-i) = a(k-1,k,2k-i-1) + a(k,j-1,2k-i+3) + a(k,j-1,2k-i+1)
     ENDDOPAR
   ENDDOPAR
ENDDOSEQ
```

Les deux boucles les plus internes sont maintenant parallèles. Le lecteur intéressé trouvera moult compléments dans [28].

Notes bibliographiques

Ce chapitre s'inspire honteusement de l'excellent livre de Darte, Robert et Vivien [28] : en fait, il s'agit d'un condensé, traduit en français, des deux derniers chapitres de cet ouvrage immortel, qui contient tout ce que vous voulez savoir, et même plus, sur les techniques de parallélisation automatique. Les livres de Zima et Chapman [65] et de Wolfe [63], ainsi que les articles de synthèse de Bacon, Graham, et Sharp [9], et Banerjee, Eigenmann, Nicolau, et Padua [10], sont de bons points de départ pour un approfondissement des notions développées ici.

Chapitre 9

Pipeline logiciel

9.1 Introduction

Ce chapitre traite d'un cas particulier du problème connu sous le nom de pipeline logiciel, ou *software pipelining*. Il s'agit d'ordonnancer un ensemble de tâches génériques qui vont être exécutées un grand nombre de fois : elle figurent, par exemple, dans le coeur d'une boucle externe. Il s'agit de les ré-ordonner de manière à utiliser au mieux les ressources existantes, d'où le nom de pipeline logiciel. Ce chapitre utilise à la fois les résultats du Chapitre 4 sur l'ordonnancement, et du Chapitre 8 sur l'analyse de dépendances. Pour faciliter la lecture, nous rappelons dans le texte les principales définitions et notions utilisées, mais le lecteur est bien sûr invité à se reporter aux chapitres en question pour plus ample information.

Au Paragraphe 9.2, nous définissons le problème d'ordonnancement cyclique auquel nous nous restreignons. Au Paragraphe 9.3, nous donnons une solution polynômiale pour la résolution de ce problème sans contraintes de ressources. Enfin, au Paragraphe 9.4, nous montrons que la version avec limitation de ressources est NP-complète, et nous proposons une heuristique garantie pour sa résolution.

9.2 Formulation du problème

Nous commençons par un exemple que nous allons ré-utiliser tout au long du chapitre.

9.2.1 Un exemple

La boucle de la Figure 9.1 a $n_i = 6$ instructions (A, B, C, D, E, et F) et N itérations ; chaque instruction est exécutée N fois. N est un paramètre de valeur inconnue, peut-être très grande. Une instruction est appelée une *tâche générique* parce qu'elle donne lieu à un grand nombre d'instances de calcul similaires, une à chaque pas de la boucle, i.e. pour chaque valeur du compteur de boucle k. On note (v, k) l'instance de la tâche générique v à l'itération k. Toutes les instances d'une même tâche générique v sont supposées avoir le même temps d'exécution (ou délai) $d(v)$.

Il y a des contraintes de précédence entre les instances des instructions. Ainsi, le calcul de A à l'itération k écrit $a(k)$, donc doit précéder le calcul de B à l'itération $k + 2$, qui lit cette valeur. Comme expliqué au Chapitre 8, on dit qu'il y a une dépendance de A vers B de distance 2.

Si on déroule complètement la boucle, on peut ordonnancer le graphe de dépendance obtenu, qui est un DAG comme au Chapitre 4. Ce graphe a une structure très particulière, comme nous l'avons vu Chapitre 8 : les sommets sont des copies des instructions de la boucle, c'est pourquoi ils

```
DO k=0, N-1
    (A) : a(k) = c(k-1)
    (B) : b(k) = a(k-2) * d(k-1)
    (C) : c(k) = b(k) + 1
    (D) : d(k) = f(k-1)/3
    (E) : e(k) = sin(f(k-2))
    (F) : f(k) = log(b(k) + e(k))
ENDDO
```

FIG. 9.1 – L'exemple de référence.

ne sont pas numérotés de 1 à $n_i \times N$ (le nombre de calculs total) ; on les étiquette plutôt par une paire d'indices : $\{(v,k) \mid v \in V,\ 0 \leq k < N\}$, où V est l'ensemble des tâche génériques, i.e., des instructions de la boucle, $V = \{A, B, C, D, E, F\}$ dans notre exemple.

Les dépendances ont également une structure régulière. Par exemple, pour chaque value de k, le calcul (B,k) écrit $b(k)$, donc doit précéder le calcul (C,k) qui lit cette valeur. Il y a une arête de dépendance $(B,k) \to (C,k)$ pour $0 \leq k < N$. Cette dépendance est boucle-indépendante, ou de niveau ∞, avec la terminologie du Paragraphe 8.2. Bien sûr, il peut y avoir des dépendances portées par la boucle, donc de niveau 1 puisqu'on a une seule boucle. Par exemple, le calcul (A,k) écrit $a(k)$, donc donc doit précéder le calcul $(B, k+2)$ qui lit cette valeur, et ce pour tout $0 \leq k < N-2$. Une représentation partielle du graphe de tâches est donnée à la Figure 9.2 (les traits en pointillé représentent les dépendances entre différentes itérations, de niveau 1).

Comme au Chapitre 8, on définit un graphe de dépendance réduit, ou GDR, dont les sommets sont les tâches génériques, et dont les arêtes sont étiquetées par les distances de dépendance. On note $G = (V, E, d, w)$ le GDR, où $d : V \to \mathbb{N}^*$ donne le délai de chaque sommet (i.e., tâche générique), et $w : E \to \mathbb{N}$ donne la distance de dépendance de chaque arête. Cette distance de dépendance signifie pour toute arête $e = (u,v) \in E$, et pour tout k, $0 \leq k < N - w(e)$, que le calcul $(v, k+w(e))$ ne peut commencer avant la fin du calcul (u,k). Le GDR de notre exemple est représenté à la Figure 9.3. Les poids des arêtes sont montrés près de leur extrémité. On vérifie qu'il y a bien une arête du sommet A vers le sommet B d'étiquette 2. Les délais sont montrés à l'intérieur de boîtes carrées : ainsi on suppose que la durée de la tâche E est 10 fois supérieure à celle de la tâche A.

9.2.2 Temps de cycle moyen

Comme au Chapitre 4, notre but est de déterminer un ordonnancement pour tous les calculs (v,k), $v \in V$, $0 \leq k < N$, de la boucle. Mais nous cherchons un ordonnancement valide pour toutes les valeurs possibles de N. Ainsi, un ordonnancement est une fonction $\sigma : V \times \mathbb{N} \to \mathbb{N}$ qui respecte les contraintes de dépendance :

$$\forall e = (u,v) \in E,\ \forall k \geq 0,\ \sigma(v, k + w(e)) \geq \sigma(u,k) + d(u) \tag{9.1}$$

On évalue la performance d'un ordonnancement σ par son *temps de cycle moyen* λ défini par :

$$\lambda = \liminf_{N \to \infty} \frac{\max\{\sigma(v,k) + d(v) \mid v \in V,\ 0 \leq k < N\}}{N}$$

Parmi tous les ordonnancements, on ne considère que des ordonnancements qui ont une structure régulière, à l'instar du graphe de tâches, à savoir les ordonnancements cycliques. Un *ordonnancement cyclique* σ est un ordonnancement tel que $\sigma(v,k) = c_v + \lambda k$ pour des valeurs $c_v \in \mathbb{N}$ et $\lambda \in \mathbb{N}$.

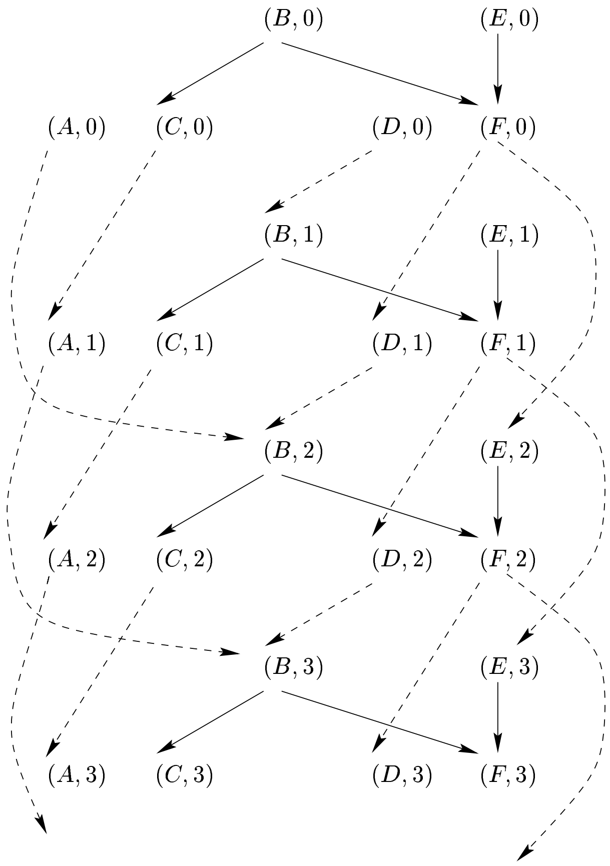


FIG. 9.2 – Graphe de tâches étendu (vue partielle) pour l'exemple de référence.

L'ordonnancement σ est un ordonnancement périodique, de période λ ; le même schéma de calcul se déroule toutes les λ unités de temps. En d'autres termes, on ordonnance des "tranches" de longueur λ. A l'intérieur de chaque tranche, une et une seule instance de chaque tâche générique est initiée ; ce qui explique pourquoi λ est appelé *initiation interval* dans la littérature.

Contrairement au Chapitre 8, nous ne traitons pas des nids de boucle généraux : nous nous restreignons à une seule boucle englobant des instructions que nous voulons ré-ordonnancer. Ces instructions sont liées par des contraintes de dépendance uniformes. Toutes ces hypothèses restrictives vont nous permettre de donner des résultats très précis, même dans le cas de ressources limitées (situation que nous n'avons pas abordée du tout au Chapitre 8).

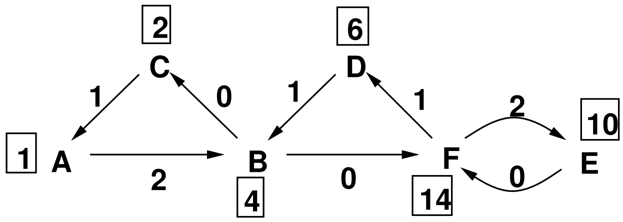


FIG. 9.3 – Graphe de dépendance réduit pour l'exemple de référence.

9.2.3 En jouant avec l'exemple de référence

Considérons le GDR de la Figure 9.3 et détruisons toutes les arêtes de poids non nul. On obtient un graphe acyclique $A(G)$ qui caractérise toutes les dépendances de niveau ∞ (voir Figure 9.4(a)). Pourquoi $A(G)$ est-il acyclique ? Par qu'un cycle d'arêtes de poids nul impliquerait un cycle de dépendance entre instances de tâches, i.e. un cycle dans le DAG de la Figure Figure 9.2 : on aurait une tâche qui dépend d'elle même ! Pour le dire différemment, une arête de poids nul va toujours d'une instruction à une autre qui apparaît textuellement plus tard dans le corps de la boucle, donc le sous-graphe des arêtes de poids nul définit un ordre partiel entre les instructions, et $A(G)$ est acyclique.

Peut-on ordonnancer $A(G)$ comme un DAG ? Aucun problème ; appliquons des techniques d'ordonnancement de liste vues au Paragraphe 4.4.2. Supposons disposer de deux processeurs (identiques). On obtient l'ordonnancement de liste basé sur le chemin critique σ_a donné à la Figure 9.4(b). On a $\sigma_a(B) = \sigma_a(E) = 0$, $\sigma_a(D) = 4$, $\sigma_a(C) = \sigma_a(F) = 10$, et $\sigma_a(A) = 12$.

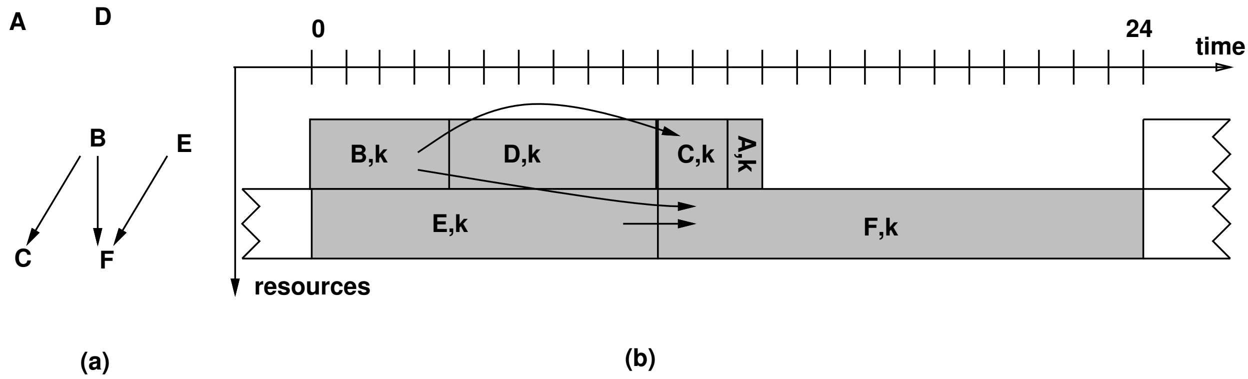


FIG. 9.4 – (a) Le graphe acyclique $A(G)$.
(b) Une allocation de liste : $\lambda = 24$.

Obtenir un ordonnancement cyclique σ à partir de l'ordonnancement de liste est facile. On choisit la période λ égale à la latence MS de l'ordonnancement de liste σ_a ($\lambda = 24$ ici), et on pose $\sigma(v, k) = \sigma_a(v) + \lambda k$ pour tout $v \in \{A, B, C, D, E, F\}$. Ce choix garantit que toutes les contraintes de dépendance sont satisfaites. En effet :

- c'est le cas des dépendances à l'intérieur d'une même itération, de niveau ∞, qui sont imposées par σ_a ;
- c'est le cas aussi pour les dépendances inter-itération, de niveau 1, parce que le choix de λ assure que chaque instance de tâche de l'itération suivante $k+1$ ne peut pas commencer avant la fin de l'exécution de toutes les instances de tâche de l'itération k.

Bien sûr la valeur de λ que nous avons obtenue n'est sûrement pas la plus petite possible. Considérons l'ordonnancement suivant : $\sigma(v, k) = c_v + \lambda k$, où $\lambda = 20$ et les constantes c_v sont données dans la Table 9.1.

Tâche générique v	A	B	C	D	E	F
Constante c_v	16	14	18	20	6	20

TAB. 9.1 – Constantes c_v pour l'ordonnancement de période $\lambda = 20$.

Nous verrons que $\lambda = 20$ n'est pas non plus la plus petite valeur possible. Comment nous avons trouvé cet ordonnancement sera expliqué au Paragraphe 9.4.4. Voyons simplement dans quel ordre les calculs sont effectués. On ré-écrit les constantes c_v sous la forme $c_v = s_v + \lambda r_v$ où $s_v = c_v \bmod \lambda$ et $r_v = c_v \div \lambda$ (division Euclidienne). On a a lors $\sigma(v, k) = s_v + \lambda(k + r_v)$. Tous les r_v sont donc égaux à 0 sauf $r_D = r_F = 1$. Ceci signifie que des calculs correspondant à des itérations différentes sont exécutés au sein de la même tranche. Précisément, la tranche k correspond à l'intervalle de temps $[\lambda k, \lambda(k+1) - 1]$. Alors le calcul (v, k) est initié dans la tranche $k - r_v$. Au sein d'une tranche (en régime permanent), on a l'exécution représentée à la Figure 9.5.

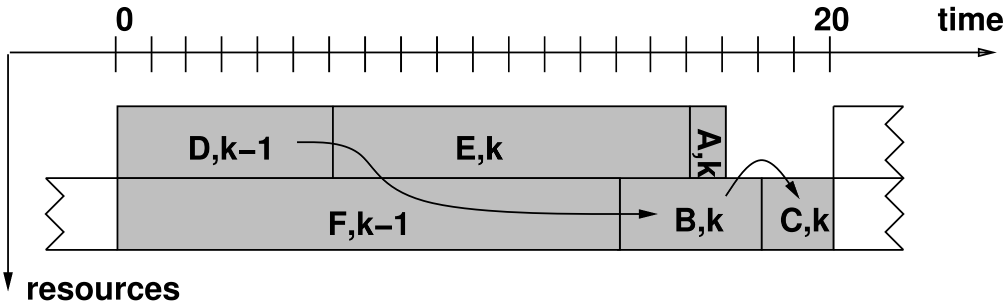


FIG. 9.5 – Une tranche de l'ordonnancement avec $\lambda = 20$.

Pour vérifier la validité de cet ordonnancement, deux solutions : entreprendre une vérification directe pour chaque arête de dépendance dans le GDR G (8 vérifications), ou attendre le Paragraphe 9.4.3.

9.2.4 Résumé

Les instances du problème d'ordonnancement cyclique de base *basic cyclic scheduling problem, ou BCS* sont représentées par un graphe $G = (V, E, d, w)$ où :

- L'ensemble V des sommets du graphe modélise les tâches génériques ; chaque $v \in V$ représente un ensemble infini de calculs $\{(v, k) \mid k \in \mathbb{N}\}$.
- Chaque tâche générique v a un délai, ou durée, $d(v) \in \mathbb{N}$; les délais d peuvent être des rationnels, mais comme le graphe est fini on peut toujours se ramener à des entiers.
- L'ensemble E des arêtes du graphe modélise les contraintes de dépendance. Soit $e = (u, v) \in E$ une arête de G de poids (positif ou nul) $w(e)$; l'instance k de la tâche générique u doit être terminée avant le début de l'instance $k + w(e)$ de la tâche générique v.

Le graphe G est le *graphe de dépendance réduit*, ou GDR. Le problème d'ordonnancement cyclique de base est de trouver un ordonnancement σ de temps de cycle moyen λ_σ, minimal, où

$$\lambda_\sigma = \liminf_{N \to \infty} \frac{\max\{\sigma(v, k) + d(v) \mid v \in V,\ 0 \leq k < N\}}{N}$$

avec les contraintes suivantes :

- **Contraintes de dépendance** Les dépendances entre opérations doivent être satisfaites, i.e. :

$$\forall e = (u, v) \in E,\ \forall k \leq N, \quad \sigma(u, k) + d(u) \leq \sigma(v, k + w(e)) \tag{9.2}$$

- **Contraintes de ressource** Chaque ressource ne peut exécuter qu'une opération à la fois ; une opération, une fois initiée, ne peut être interrompue. Si p ressources sont disponibles :

$$\forall t \in \mathbb{N},\ |\{(v, k) \mid t - d(v) < \sigma(v, k) \leq t\}| \leq p \tag{9.3}$$

Soulignons que l'expression des contraintes de ressource correspond au cas le plus simple de p ressources identiques, et où les opérations sont atomiques. On peut imaginer avoir des ressources de différents types (des additionneurs et des multiplieurs par exemple), ou encore des ressources pipelinées (une opération peut être initiée à chaque top).

Définition 21 *Etant donné un graphe $G = (V, E, d, w)$:*
- *BCS(∞) est le problème de trouver le temps de cycle moyen le plus petit possible λ_∞ pour tout ordonnancement utilisant des ressources illimitées.*
- *BCS(p) est le problème de trouver le temps de cycle moyen le plus petit possible λ_p pour tout ordonnancement utilisant p processeurs (identiques et non-pipelinés).*

9.2.5 Bornes inférieures pour λ

Au Chapitre 4, nous avons donné deux bornes inférieures pour la latence M de tout ordonnancement :
- Indépendamment du nombre de ressources, $M \geq w(\mathcal{P})$ pour tout chemin $\mathcal{P}$ dans le graphe (Proposition 7).
- Si on dispose de p ressources, $M \geq \frac{\text{Seq}}{p}$, where Seq est le temps séquentiel, i.e. la somme des durées de toutes les tâches tâche durations (Proposition 8).

La première borne exprime les contraintes de dépendance et la seconde, les contraintes de ressource. Ici, nous pouvons aussi trouver des bornes pour λ. Pour un cycle $\mathcal{C}$ du graphe, on note $\rho(\mathcal{C})$ le rapport durée sur distance $\frac{d(\mathcal{C})}{w(\mathcal{C})}$. On appelle $\rho(\mathcal{C})$ le *rapport* de $\mathcal{C}$.

Théorème 15 *Si un ordonnancement existe pour le graphe $G = (V, E, d, w)$, alors tous les cycles $\mathcal{C}$ de G ont un poids $w(\mathcal{C})$ strictement positif. En outre, pour tout ordonnancement σ de temps de cycle moyen λ_σ :*

$$\textit{pour tout cycle } \mathcal{C},\ \lambda_\sigma \geq \rho(\mathcal{C}) \tag{9.4}$$

$$\textit{si } p \textit{ ressources sont disponibles, } p\lambda_\sigma \geq \sum_{v \in V} d(v) \tag{9.5}$$

Preuve Soit σ un ordonnancement et $\mathcal{C}$ un cycle de G : $\mathcal{C} = (v_0, \ldots, v_{r-1}, v_r = v_0)$. Soit e_i l'arête (v_{i-1}, v_i) dans ce cycle. Toutes les contraintes (9.2) sont satisfaites par σ le long du cycle. Pour tout $k \in \mathbb{N}$:

$$\begin{array}{rcl}
\sigma(v_0, k) + d(v_0) & \leq & \sigma(v_1, k + w(e_1)) \\
\sigma(v_1, k + w(e_1)) + d(v_1) & \leq & \sigma(v_2, k + w(e_1) + w(e_2)) \\
 & \ldots & \\
\sigma(v_i, k + \sum_{j=1}^{i} w(e_j)) + d(v_i) & \leq & \sigma(v_{i+1}, k + \sum_{j=1}^{i+1} w(e_j))
\end{array}$$

Sommant les r inégalités pour les r arêtes de $\mathcal{C}$, on obtient

$$\sigma(v_0, k) + d(\mathcal{C}) \leq \sigma(v_0, k + w(\mathcal{C})) \tag{9.6}$$

Cette inégalité montre que $w(\mathcal{C}) = 0$ est impossible, puisque $d(\mathcal{C}) > 0$. Donc, une condition nécessaire pour l'existence d'un ordonnancement est que tous les cycles du graphe ont un poids strictement positif (on verra plus loin que cette condition est aussi suffisante).

Considérons donc un ordonnancement σ, et supposons que $w(\mathcal{C}) > 0$. Pour tout $m \in \mathbb{N}$, l'inégalité (9.6) est vraie pour $k = m.w(\mathcal{C})$. Par récurrence sur m, on obtient $\sigma(v_0, 0) + m.d(\mathcal{C}) \leq \sigma(v_0, m.w(\mathcal{C}))$, et comme $\sigma(v_0, 0) \geq 0$, on a $\sigma(v_0, m.w(\mathcal{C})) \geq md(\mathcal{C})$.

Soit $D(\sigma, n) = \max\{\sigma(v,k) + d(v) \mid v \in V,\ 0 \leq k < n\}$. $D(\sigma, n)$ est la latence de σ restreint aux n premières itérations. Par définition, $\lambda_\sigma = \lim_{N \to \infty} \inf\{\frac{D(\sigma,n)}{n} \mid n \geq N\}$. Supposons $N > 1$, et pour tout $n \geq N$, soit $m = \lfloor \frac{n}{w(\mathcal{C})} \rfloor$. Comme $D(\sigma, n) \geq D(\sigma, m.w(\mathcal{C})) \geq \sigma(v_0, m.w(\mathcal{C}))$, on a :

$$\frac{D(\sigma,n)}{n} \geq \frac{m.d(\mathcal{C})}{n} \geq \left(1 - \frac{w(\mathcal{C})}{n}\right)\rho(\mathcal{C}) \geq \left(1 - \frac{w(\mathcal{C})}{N}\right)\rho(\mathcal{C})$$

Cette inégalité est vraie pour $\inf\{\frac{D(\sigma,n)}{n} \mid n \geq N\}$; finalement, quand N tend vers l'infini, on obtient la première borne inférieure (inégalité (9.4)) : $\lambda_\sigma \geq \rho(\mathcal{C})$.

Pour établir la deuxième borne inférieure, on utilise la Proposition 8. Si p ressources sont disponibles, alors $pD(\sigma, n)$ est plus grand que la durée totale des opérations dans l'ensemble $\{(v,k) \mid v \in V,\ 0 \leq k < n\}$. Donc, pour tout $n \in \mathbb{N}$, $pD(\sigma, n) \geq n \sum_{v \in V} d(v)$, et enfin, $p\lambda_\sigma \geq \sum_{v \in V} d(v)$. ■

Dans la suite, on supposera que le graphe $G = (V, E, d, w)$ est tel que $d(v) > 0$ pour tout $v \in V$, et $w(\mathcal{C}) > 0$ pour tout cycle $\mathcal{C}$. Alors $\rho(\mathcal{C})$ est bien défini, et $\rho(\mathcal{C}) > 0$ pour tout cycle $\mathcal{C}$.

9.3 Résolution de BCS(∞)

Pour résoudre le problème à ressources illimitées, nous avons besoin de deux outils : les graphes à potentiel, et l'algorithme de Bellman-Ford.

9.3.1 Ordonnancement des graphes à potentiel

Les *graphes à potentiel* sont des graphes de tâches dont les arêtes sont pondérées, et qui peuvent avoir des cycles : deux différences avec le modèle du Paragraphe 4.2, dont les arêtes (modélisant les dépendances) ne sont pas pondérées, et qui sont acycliques (un cycle dans le graphe impliquerait qu'une tâche dépend d'elle même).

Considérons un système de tâches où chaque tâche v a plusieurs attributs : une durée (ou poids) $p(v)$, une date d'initiation (ou *ready time*) $r(v)$, et une date de fin (ou *deadline*) $d(v)$. On ajoute une tâche spéciale s, dite tâche source. Soit $\sigma(v)$ la date à laquelle débute l'exécution de v. On suppose que la tâche source débute au top 0 : $\sigma(s) = 0$. Toutes les contraintes d'ordonnancement peuvent être représentées par un graphe à arêtes pondérées : chaque sommet représente une tâche, et chaque arête $e = (u, v)$ signifie qu'il faut satisfaire la contrainte $\sigma(u) + w(e) \leq \sigma(v)$. La fonction de poids des arêtes $w : E \longrightarrow \mathbb{Z}$ est définie comme suit (noter que des valeurs négatives pour w sont autorisées) :

- **Dépendances.** Si la tâche v dépend de la tâche u, on introduit l'arête $e = (u, v)$ et on pose $w(e) = p(u)$. On retrouve l'inégalité classique de précédence $\sigma(u) + p(u) \leq \sigma(v)$.
- **Dates d'initiation.** Pour exprimer le fait que l'exécution de v ne peut pas débuter avant la date $r(v)$, on introduit l'arête $e = (s, v)$ de poids $w(e) = r(v)$. La contrainte correspondante est $\sigma(s) + r(v) \leq \sigma(v)$, ce qui est la condition cherchée, puisque $\sigma(s) = 0$.
- **Dates de fin.** Pour exprimer le fait que l'exécution de v doit être terminée à la date $d(v)$, on introduit l'arête $e = (v, s)$ de poids $w(e) = p(v) - d(v)$. On obtient $\sigma(v) + p(v) - d(v) \leq \sigma(s) = 0$, donc v termine avant la date $d(v)$.

On aboutit ainsi à un graphe contenant des cycles (dès qu'une tâche a une date d'initiation et une date de fin), et dont les arêtes ont des poids éventuellement négatifs.

Définition 22 *Un graphe à potentiel $G = (V, E, w)$ est un graphe de tâches où $w : E \longrightarrow \mathbb{Z}$ définit les poids des arêtes. Un ordonnancement σ pour G est une fonction $\sigma : V \longrightarrow \mathbb{N}$ telle que toutes les inégalités "de potentiel" sont satisfaites :*

$$\forall e = (u, v) \in E,\ \sigma(u) + w(e) \leq \sigma(v)$$

Théorème 16 *Il existe un ordonnancement pour un graphe à potentiel $G = (V, E, w)$ si et seulement si tous les circuits élémentaires de G ont un poids négatif ou nul.*

La preuve est constructive : si tous les circuits élémentaires de G ont un poids négatif ou nul, on construit un ordonnancement σ vérifiant toutes les inégalités "de potentiel".

Preuve Supposons d'abord qu'il existe un circuit élémentaire $\mathcal{C}$: $\mathcal{C} = v_0 \to v_1 \to \ldots \to v_{p-1} \to v_p = v_0$. Soit e_i l'arête (v_i, v_{i+1}) de ce circuit. Si un ordonnancement σ existe, alors $\sigma(v_j) + w(e_j) \leq \sigma(v_{j+1})$ pour $0 \leq j \leq p - 1$. En sommant, on obtient $\sigma(v_0) + \sum_{j=0}^{p-1} w(e_j) \leq \sigma(v_0)$; le poids $w(\mathcal{C}) = \sum_{j=0}^{p-1} w(e_j)$ du circuit $\mathcal{C}$ est bien négatif ou nul.

Supposons maintenant que tout circuit élémentaire de G a un poids négatif ou nul. On ajoute un sommet source s à G, ainsi qu'une arête $e = (s, v)$ de poids nul de s à chaque sommet v de G. Alors on peut poser $\sigma(v) = t(s, v)$ pour chaque sommet $v \in V$, où $t(s, v)$ est la longueur du plus long chemin de s à v dans G.

En effet, soit $v \in V$. On montre d'abord que le plus long chemin de s à v est bien défini. L'ensemble de tous les chemins de s à v n'est pas vide, puisqu'il y a une arête de s à v. Soit $\mathcal{P}$ un chemin quelconque de s à v. Si $\mathcal{P}$ n'est pas élémentaire, il contient un circuit élémentaire, dont la longueur est négative ou nulle par hypothèse. Si on supprime ce circuit de $\mathcal{P}$, on obtient un autre chemin dont la longueur est au moins égale à celle de $\mathcal{P}$. En supprimant tous les circuits on obtiendra un chemin élémentaire de longueur au moins égale à celle du chemin de départ $\mathcal{P}$. Donc les plus longs chemins de s à v peuvent être recherchés parmi les chemins élémentaires de s à v, dont le nombre est fini. On peut alors poser

$$t(s, v) = \max\{w(\mathcal{P}) \mid \mathcal{P} \text{ est un chemin élémentaire de } s \text{ à } v\}$$

Posons donc $\sigma(v) = t(s, v)$ pour chaque sommet $v \in V$. Pour vérifier que toutes les inégalités "de potentiel" sont satisfaites, considérons une arête $e = (u, v)$. La longueur du plus long chemin de s à v est au moins égale à la longueur du plus long chemin de s à u plus le poids de l'arête e : voir la Figure 9.6. En d'autres termes, t vérifie l'inégalité triangulaire : $t(s, u) + w(e) \leq t(s, v)$ pour chaque $e = (u, v)$, ce qui se ré-écrit en $\sigma(u) + w(e) \leq \sigma(v)$. ∎

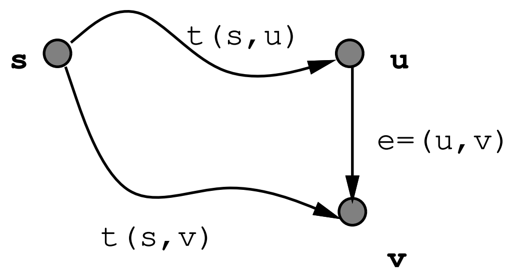


FIG. 9.6 – Inégalité triangulaire pour les plus longs chemins.

9.3.2 Algorithme de Bellman-Ford

L'algorithme de Bellman-Ford permet de déterminer les plus courts chemins (ou les plus longs chemins) dans les graphes avec cycles. Nous le présentons ici pour le calcul des plus longs chemins, par souci de cohérence avec ce qui précède. Il suffit de changer le signe de toutes les arêtes (la condition d'existence devenant que les cyles élémentaires ont un poids positif ou nul) pour obtenir, comme dans le livre de Cormen, Leiserson et Rivest [25], un algorithme de plus courts chemins.

Soit $G = (V, E, w)$ un graphe dont les arêtes sont pondérées par la fonction $w : E \longrightarrow \mathbb{Z}$, et supposons que tous les cycles ont un poids négatif ou nul. L'objectif est de calculer les plus longs chemins d'un sommet source $s \in V$ à tous les sommets de G. Comme on l'a montré dans la preuve du Théorème 16, on peutse limiter à des chemin élémentaires. L'algorithme de Bellman-Ford algorithm est basé sur la "relaxation" des arêtes. Pour chaque sommet $v \in V$, on met à jour deux variables :

1. $d(v)$, qui est la longueur du plus long chemin courant, noté $LP(v)$, de s à v ;
2. $\Pi(v)$, qui est le prédécesseur immédiat de v dans le plus long chemin courant : $LP(v) : s \rightsquigarrow \Pi(v) \to v$.

Ces variables sont initialisées avec la procédure :

Initialization

$$\begin{array}{l} d(s) \leftarrow 0 \\ \forall v \in V \\ \quad d(v) \leftarrow -\infty \\ \quad \Pi(v) \leftarrow \mathsf{NIL} \end{array}$$

Pour "relaxer" l'arête $e = (u, v)$, on utilise :

Relaxation(e=(u,v))

$$\begin{array}{l} \text{si } d(v) < d(u) + w(e) \text{ alors} \\ \quad d(v) \leftarrow d(u) + w(e) \\ \quad \Pi(v) \leftarrow u \end{array}$$

Bellman-Ford Algorithm

```
Initialisation
pour i = 1 à |V| − 1 faire
  pour chaque arête e ∈ E faire Relaxation(e)
Vérification de terminaison
pour chaque arête e = (u,v) ∈ E faire
  si d(v) < d(u) + w(e) alors
    renvoyer "cycle de poids strictement positif atteint à partir de s"
  sinon renvoyer "OK"
```

Pour chaque $v \in V$, soit $\delta(s, v)$ la longueur du plus long chemin de de s à v ($\delta(s, v) = -\infty$ si aucun chemin n'existe).

Lemme 21 *Soit $v \in V$. Tout au long de l'algorithme, $d(v) \leq \delta(s, v)$. Dès que $d(v) = \delta(s, v)$, $d(v)$ reste inchangé jusqu'à la fin.*

Preuve La propriété est vraie à l'initialisation : $d(s) = 0 = \delta(s, s)$, et $d(v) = -\infty$ pour chaque $v \neq s$.

Supposons par l'absurde qu'à une étape donnée de l'algorithme, il existe v tel que $d(v) > \delta(s, v)$. Plus précisément, soit v le premier sommet pour lequel $d(v)$ devient strictement supérieur à $\delta(s, v)$; $d(v)$ vient d'être mis à jour par la relaxation d'une arête $e = (u, v)$, pour un certain $u \in V$ (pendant l'appel **Relaxation(e)**). Alors $d(v) = d(u) + w(e)$ et $d(v) > \delta(s, v)$. A cause de l'inégalité triangulaire, $\delta(s, v) \geq \delta(s, u) + w(e)$. Donc $d(u) > \delta(s, u)$, ce qui contredit le fait que v est le premier sommet pour lequel $d(v) > \delta(s, v)$.

Enfin, $d(v)$ ne diminue jamais lors d'une mise à jour, d'où la deuxième partie de la propriété. ■

Nous revenons à la preuve de l'algorithme de Bellman-Ford, en montrant comment il peut être utilisé pour la construction d'un cycle de poids strictement positif, si un tel cycle existe :

Preuve S'il n'y a pas de cycle de poids strictement positif dans G, alors à la fin de l'algorithme, $d(v) = \delta(s, v), \forall v \in V$. En effet, soit $v \in V$; v peut être atteint à partir de s par le plus long chemin élémentaire $\mathcal{P}\ : v_0 = s \to v_1 \to \ldots \to v_k = v$. Le chemin est élémentaire, donc $k \leq |V| - 1$. Par récurrence sur le pas i de la boucle la plus externe, on a $d(v_i) = \delta(s, v_i)$, et donc $d_v = \delta(s, v)$ en au plus $|V| - 1$ étapes. De plus, $d(v) = \delta(s, v) \geq \delta(s, u) + w(e) = d(u) + w(e)$ à la fin de l'algorithme, et le test de terminaisons renvoie OK.

Supposons maintenant qu'il existe un cycle $\mathcal{C} = (v_0, v_1, \ldots, v_{k-1}, v_k = v_0)$ de poids strictement positif. Si l'algorithme avait renvoyé OK, on aurait $d(v_i) \geq d(v_{i-1}) + w(e_i)$ pour $1 \leq i \leq k$, où e_i est l'arête (v_{i-1}, v_i) de $\mathcal{C}$. En sommant, on aurait $w(C) \leq 0$, contradiction. Il existe donc une arête $e = (u, v)$ telle que $d(v) < d(u) + w(e)$.

Soit $e_0 = (v_1, v_0)$ une telle arête. Construisons à rebours un chemin *élémentaire* $\mathcal{P}$, aussi long que possible, en suivant la relation de parenté Π calculée par **Relaxation** : $v_{k+1} = \Pi(v_k)$ pour $k \geq 1$. Cette construction pourrait s'arrêter parce que $\Pi(v_k) = \text{NIL}$, ou parce que $\Pi(v_k)$ est déjà dans $\mathcal{P}$. Montrons que seul le second cas peut se produire : en effet, pour un sommet v, soit $I(v)$ la dernière itération i à laquelle $d(v)$ a été modifié, et posons $I(v) = 0$ si $d(v)$ n'est jamais modifié, i.e. si $\Pi(v) = \text{NIL}$. Pour tout sommet v, $I(\Pi(v)) \geq I(v) - 1$, et comme $d(v_0) < d(v_1) + w(e)$, $d(v_1)$ a été modifié à la dernière itération : $I(v_1) \geq |V| - 1$. Maintenant, si le premier sommet de $\mathcal{P}$ est v_k (le chemin a $k + 1$ sommets), on obtient en sommant $I(v_k) \geq I(v_1) - (k - 1) \geq |V| - k$. Si le premier cas se produit, $\Pi(v_k) = (NIL)$, donc $I(v_k) = 0$, et $k \geq |V|$. Le chemin $\mathcal{P}\ :\ v_k \rightsquigarrow v_0$ n'est pas élémentaire, contradiction. Finalement, $\Pi(v_k) = v_j$ pour un j entre 0 et $k - 1$, et on obtient un cycle $\mathcal{C}\ :\ v_j \to v_{k-1} \rightsquigarrow v_j$.

Rappelons que juste après la relaxation d'une arête $e = (u, v)$, on a $d(v) = d(u) + w(e)$, et $\Pi(v) = u$. Donc, $d(u)$ peut seulement croître. Donc pour chaque arête de $\mathcal{C}$, $d(v) \leq d(\Pi(v)) + w(e)$. En sommant ces inégalités le long du cycle, on a $w(\mathcal{C}) \geq 0$. En outre, l'une au moins de ces inégalités est stricte, par exemple pour l'arête (v_r, v_{r-1}) où v_r est le dernier sommet de $\mathcal{C}$ à avoir été modifié. D'où $w(\mathcal{C}) > 0$. ■

La complexité de l'algorithme de Bellman-Ford est $O(|V||E|) \leq O(|V|^3)$.

9.3.3 Ordonnancement optimal à ressources illimitées

Nous voila prêts à résoudre BCS(∞), le problème avec infinité de ressources ($p = \infty$). Etant donné le graphe $G = (V, E, d, w)$ (d définit les poids des sommets, w ceux des arêtes), on cherche un ordonnancement σ dont le temps de cycle moyen λ_σ est aussi petit que possible. Le Théorème 15 nous dit que $\lambda_\sigma \geq \rho(\mathcal{C})$ pour tous les cycles $\mathcal{C}$ de G. Nous allons montrer que cette borne peut être atteinte : il existe un ordonnancement dont le temps de cycle moyen est égal au cycle de rapport minimal dans le graphe.

On suppose que tous les cycles de G ont un poids strictement positif : sinon aucun ordonnancement n'existe (Théorème 15). Nous allons construire un ordonnancement de la forme $\sigma(v,k) = c_v + \lambda k$ où $c_v \in \mathbb{Q}$, et $\lambda \in \mathbb{Q}$. A vrai dire avec cette formule, σ n'est pas vraiment un ordonnancement comme défini au Paragraphe 9.2.2, parce qu'il ne prend pas toujours des valeurs entières. Mais on pourra toujours utiliser $\lfloor\sigma\rfloor$, défini par $\lfloor\sigma\rfloor(v,k) = \lfloor c_v + \lambda k\rfloor$; c'est aussi un ordonnancement (si σ satisfait l'Equation (9.1) alors $\lfloor\sigma\rfloor$ la satisfait aussi), avec le même temps de cycle moyen. Cet ordonnancement n'est pas cyclique mais K-périodique (voir [40] pour en savoir plus sur les ordonnancements K-périodiques). Pour simplifier, nous continuerons à dire que σ est un ordonnancement cyclique, même si λ n'est pas un entier.

Pour un ordonnancement cyclique, on peut simplifier les contraintes de dépendance. En effet, les contraintes (9.1)

$$\forall e = (u,v) \in E,\ \forall k \geq 0,\ \sigma(u,k) + d(u) \leq \sigma(v, k + w(e))$$

se réduisent à

$$\forall e = (u,v) \in E,\ c_u + d(u) \leq c_v + \lambda w(e) \tag{9.7}$$

On peut caractériser simplement toutes les valeurs possibles pour λ :

Définition 23 *Soit $\lambda \in \mathbb{Q}$. On définit le graphe $G'_\lambda = (V', E', w'_\lambda)$ à partir de G, comme suit :*
- *Sommets de G'_λ : ajouter à V un nouveau sommet s : $V' = V \cup \{s\}$.*
- *Arêtes de G'_λ : pour chaque sommet $v \in V$ ajouter à E une arête de s à v : $E' = E \cup (\{s\} \times V)$.*
- *Poids des arêtes de G'_λ : on pose $w'_\lambda(e) = 0$ si $e = (u,v) \in E' \setminus E$, et $w'_\lambda(e) = d(u) - \lambda w(e)$ sinon.*

Lemme 22 *λ est valide (i.e.il existe un ordonnancement de temps de cycle moyen égal à λ) si et seulement si G'_λ n'a pas de cycle de poids strictement positif.*

Preuve G'_λ et G ont les mêmes cycles. Et pour a cycle $\mathcal{C}$, $w'_\lambda(\mathcal{C}) = d(\mathcal{C}) - \lambda w(\mathcal{C})$. Donc $w'_\lambda(\mathcal{C}) > 0$ si et seulement si $\lambda < \rho(\mathcal{C})$.

Si λ est valide, alors $\lambda \geq \rho(\mathcal{C})$ pour tout cycle $\mathcal{C}$ (Théorème 15). Donc G'_λ n'a pas de cycle de poids strictement positif.

Réciproquement, si G'_λ n'a pas de cycle de poids strictement positif, on peut définir pour tout $v \in V$, le plus chemin (dans G'_λ) de s à v, que l'on note $t(s,v)$. Par définition, $t(s,v)$ vérifie l'inégalité triangulaire :

$$\forall e = (u,v) \in E,\ t(s,v) \geq t(s,u) + w'_\lambda(e)$$

i.e.,

$$\forall e = (u,v) \in E,\ t(s,u) + d(u) \leq t(s,v) + \lambda w(e)$$

Ceci prouve que $\sigma(v,k) = t(s,v) + \lambda k$ est un ordonnancement valide (ou $\lfloor\sigma\rfloor$ si λ n'est pas un entier). ■

Le lien avec les graphes à potentiel est désormais clair. D'après le Théorème 15, s'il existe un cycle $\mathcal{C} \in G$ tel que $\lambda < \rho(\mathcal{C})$, alors il n'existe pas d'ordonnancement de temps de cycle moyen λ. Réciproquement, si $\lambda \geq \rho(\mathcal{C})$ pour tous les cycles $\mathcal{C}$, on peut se restreindre aux ordonnancements de la forme $\sigma(v,k) = c_v + \lambda k$ (ou avec la partie entière). Les contraintes de dépendance se ramènent aux contraintes (9.7), qui définissent un problème d'ordonnancement à potentiel sur G'_λ. Le fait que les poids $w'_\lambda(e)$ puissent ne pas être entiers (si λ n'est pas un entier) n'est pas un problème, on peut quand même utiliser l'algorithme de Bellman-Ford.

Le Lemme 22 a deux conséquences importantes :

- Etant donnée une valeur de λ, il est facile de voir si elle détermine un ordonnancement cyclique valide, et si oui, de construire cet ordonnancement. On applique l'algorithme de Bellman-Ford. (Paragraphe 9.3.2) au graphe G'_λ.
- La valeur optimale de λ_∞ est la plus petite valeur λ telle que G'_λ n'a pas de cycle de poids strictement positif. Par suite $\lambda_\infty = 0$ si G est acyclique, et

$$\lambda_\infty = \max\{\rho(\mathcal{C}) \mid \mathcal{C} \text{ cycle élémentaire de } G\}$$

  sinon. Le lecteur peut vérifier directement que $\rho(\mathcal{C})$ est maximal pour des cycles élémentaires, ou se reporter à la preuve du Théorème 17.

Revenons à l'exemple introductif de la Figure 9.1. Il y a trois cycles élémentaires : $\rho(C_1) = \frac{7}{3}$ pour $C_1 : A \to B \to C \to A$, $\rho(C_2) = \frac{24}{2}$ pour $C_2 : B \to F \to D \to B$, et $\rho(C_3) = \frac{24}{2}$ for C_3 : $E \to F \to E$. Donc $\lambda_\infty = 12$. La preuve du Théorème 16 donne complètement l'ordonnancement ; on doit calculer des plus longs chemins dans le graphe G'_{12}, qui est représenté à la Figure 9.7. Les plus longs chemins $t(s,v)$ du sommet source s à tous les autres sommets v sont les valeurs à

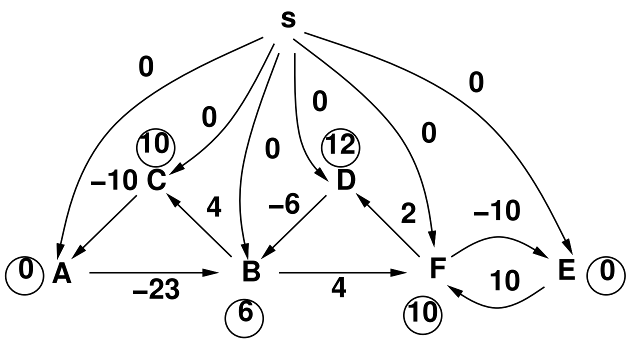


FIG. 9.7 – Le graphe G'_{12}.

l'intérieur des cercles sur la Figure 9.7. Par exemple, $t(s,D) = 12$, la longueur du chemin de s à D en passant par E et F. D'après le Théorème 16 et le Lemme 22, on définit un ordonnancement $\sigma(v,k) = c_v + \lambda k$ valide en posant $c_v = t(s,v)$. Deux tranches d'exécution consécutives (en régime permanent) sont représentées à la Figure 9.8. Comme pour la Figure 9.5, seules les dépendances au sein d'une même itération sont marquées (par des flèches). Pour faciliter la lecture, les temps d'exécution sont résumés dans la Table 9.2.

Instant de calcul	λk	$\lambda k{+}6$	$\lambda k{+}10$	$\lambda k{+}12$	$\lambda k{+}18$	$\lambda k{+}22$
Processeur P_1	(A,k)			$(A,k+1)$		
Processeur P_2	$(D,k{-}1)$	(B,k)	(C,k)	(D,k)	$(B,k{+}1)$	$(C,k{+}1)$
Processeur P_3	(E,k)		(F,k)			
Processeur P_4				$(E,k{+}1)$		$(F,k{+}1)$

TAB. 9.2 – Dates de début d'exécution pour l'ordonnancement à ressources illimitées.

Quatre processeurs sont nécessaires pour réaliser cet ordonnancement. Comme on initie une nouvelle instance d'opération tous les λ_∞ steps, on peut avoir besoin de plus de processeurs qu'il

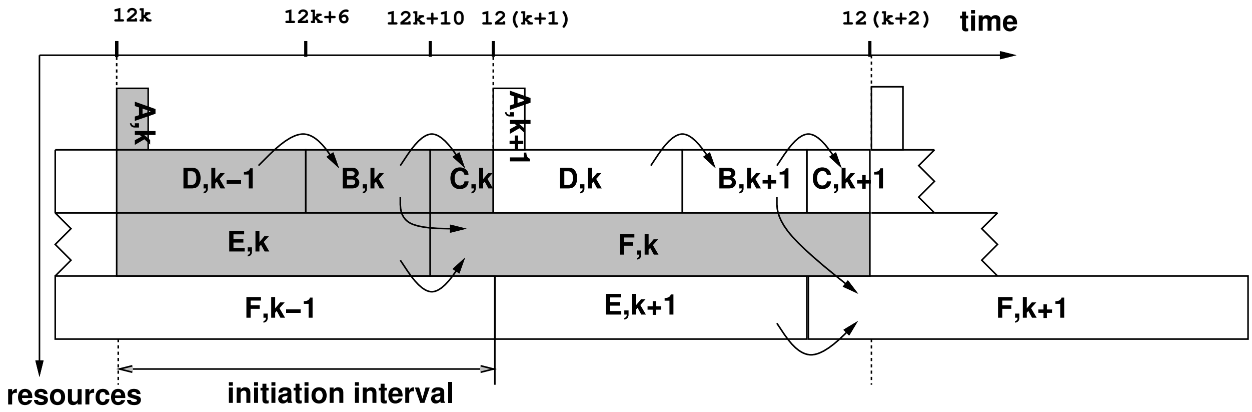


FIG. 9.8 – L'ordonnancement à ressources illimitées ($\lambda_\infty = 12$).

n'y a de tâches génériques. Ainsi $w(F) = 14 > \lambda_\infty = 12$, c'est pourquoi deux processeurs (P_3 et P_4 sur la Figure 9.8) calculent à tour de rôle les instances successive de F.

Nous concluons ce paragraphe en donnant le résultat principal sous forme de théorème :

Théorème 17 *Soit $G = (V, E, d, w)$ un graphe. Le temps de cycle moyen minimal λ_∞ d'un ordonnancement à ressources illimitées est $\lambda_\infty = 0$ si G est acyclique, et égal au maximum ρ des rapports des cycles de G sinon.*

Corollaire 2 *Etant donné un graphe $G = (V, E, d, w)$, BCS(∞) peut être résolu en temps*

$$O(|V||E|\log(|V|\Delta)),$$

où $\Delta = d_{\max}(w_{\max})^2$, $d_{\max} = \max_{v\in V} d(v)$ et $w_{\max} = \max_{e\in E} w(e)$.

Preuve On suppose que G a au moins un cycle, sinon $\lambda_\infty = 0$. Alors λ_∞ est égale au maximum des rapports des cycles de G. Vérifions en premier lieu que ce maximum est atteint pour un cycle élémentaire. Considérons un cycle non élémentaire $\mathcal{C}$, qui peut être décomposé en deux cycles C_1 et C_2, où C_1 est élémentaire. On utilise l'inégalité suivante : soient a, b, c, d quatre nombres strictement positifs et $m = \max(\frac{a}{b}, \frac{c}{d})$. Alors, $a \leq m.b$ et $c \leq m.d$, donc $(a + c) \leq m(b + d)$. D'où :

$$\frac{a+c}{b+d} \leq \frac{a}{b} \quad \text{ou} \quad \frac{a+c}{b+d} \leq \frac{c}{d}$$

Appliquons cette inégalité à $\mathcal{C}$: soit $\rho(\mathcal{C}) \leq \rho(C_1)$, soit $\rho(\mathcal{C}) \leq \rho(C_2)$. Dans le premier cas, on a terminé ; dans le second, on applique la décomposition à C_2, et à la fin, on obtiendra un cycle élémentaire de rapport égal ou meilleur.

Dans la suite on note ρ le maximum des rapports de cycles dans G. Puisque $\rho(\mathcal{C})$ est maximal pour un cycle élémentaire $\mathcal{C}$, $\rho(\mathcal{C}) \leq d(\mathcal{C}) \leq |V|d_{\max}$ (et $\rho(\mathcal{C}) > 0$ car on a supposé que G a au moins un cycle). Montrons d'abord comment calculer $\overline{\rho} = \lceil\rho\rceil$, on étendra ensuite la technique au cas où ρ n'est pas entier. Dans tous les cas on effectue une recherche binaire dans l'intervalle $]b_{\text{inf}}, b_{\text{sup}}]$, en commençant avec l'intervalle $]0, |V|d_{\max}]$, et en garantissant qu'il existe au moins un cycle $\mathcal{C}$ tel que $\rho(\mathcal{C}) \in]b_{\text{inf}}, b_{\text{sup}}]$, mais qu'il n'existe pas de cycle $\mathcal{C}$ tel que $\rho(\mathcal{C}) > b_{\text{sup}}$.

Dans le premier cas, on cherche une valeur entière pour $\overline{\rho}$. Tant que $b_{\text{sup}} - b_{\text{inf}} > 1$, on pose $\lambda = \lceil(b_{\text{inf}} + b_{\text{sup}})/2\rceil$, et on appelle l'algorithme de Bellman-Ford dans G'_λ. Si l'algorithme renvoie l'existence d'un cycle de poids strictement positif, alors $\lambda < \rho$, et on continue la recherche dans l'intervalle $]\lambda, b_{\text{sup}}]$. Sinon, $\lambda \geq \rho$, et on continue la recherche dans l'intervalle $]b_{\text{inf}}, \lambda]$. L'algorithme s'arrête quand $b_{\text{sup}} = b_{\text{inf}}+1$, et dans ce cas, $\overline{\rho} = b_{\text{sup}}$. Chaque pas de recherche a un coût $O(|V||E|))$, et le nombre de pas est $O(\log(|V|d_{\max}))$, d'où la complexité totale.

Dans le second cas, quand ρ n'est pas un entier, on effectue une recherche binaire sur un ensemble de nombres rationnels. On donne d'abord une borne inférieure pour la différence minimale η entre les rapports de deux cycles : si C_1 et C_2 sont deux cycles de rapports différents :

$$|\rho(C_1) - \rho(C_2)| = \frac{|d(C_1)w(C_2) - d(C_2)w(C_1)|}{w(C_1)w(C_2)} \geq \frac{1}{(w_{\max})^2} = \eta$$

On utilise alors la même technique que dans le premier cas. Tant que $b_{\text{sup}} - b_{\text{inf}} > \eta$, on pose $\lambda = (b_{\text{inf}} + b_{\text{sup}})/2$, et on on appelle l'algorithme de Bellman-Ford afin de décider si $\lambda > \rho$. Quand $b_{\text{sup}} - b_{\text{inf}} \leq \eta$, nous sommes certains qu'il existe une et une seule valeur $\rho(\mathcal{C})$ pour un cycle $\mathcal{C}$ telle que $b_{\text{inf}} < \rho(\mathcal{C}) \leq b_{\text{sup}}$. On conclut avec un dernier appel à l'algorithme de Bellman-Ford, avec $\lambda = b_{\text{inf}}$. Comme $\rho > b_{\text{inf}}$, l'algorithme renvoie l'existence d'un cycle de poids strictement positif. Soit $\mathcal{C}$ un tel cycle (construit, par exemple, comme expliqué au Paragraphe 9.3.2) : $w'_\lambda(\mathcal{C}) = d(\mathcal{C}) - \lambda w(\mathcal{C}) > 0$, donc $\rho(\mathcal{C}) > \lambda$, et $\rho(\mathcal{C})$ est la valeur cherchée pour ρ. Le nombre de pas de recherche est maintenant $\log(|V|d_{\max}(w_{\max})^2)$.

L'algorithme pour calculer ρ est emprunté au livre de Gondran et Minoux [38]. ■

9.4 Résolution de BCS(p)

Nous commençons par une brève digression pour démontrer la NP-complétude de BCS(p). Désolé d'interrompre le fil conducteur de l'exposé, mais il faut bien déterminer la complexité du problème à résoudre avant d'introduire des heuristiques d'approximation.

9.4.1 NP-complétude de BCS(p)

Le problème de décision associé à BCS(p) est de déterminer, étant donnés un nombre de ressources p et une borne $K \in \mathbb{N}^*$, s'il existe un ordonnancement de temps de cycle moyen au plus K. Nous ne savons pas si ce problème est dans la classe NP ou non : nous ne savons pas si une solution (un certificat) peut être codé avec une taille polynômiale en l'instance du problème. Néanmoins, on peut se restreindre au cas des ordonnancements cycliques entiers, de la forme $\sigma(v,k) = c_v + \lambda k$ où $\lambda \in \mathbb{N}$:

Définition 24 *Le problème de décision Dec-BCS(p) associé au problème BCS(p) est le suivant : étant donnés un graphe $G = (V, E, d, w)$, un nombre $p \geq 1$ de ressources identiques, et une borne $K \in \mathbb{N}^*$, existe-t-il un ordonnancement cyclique entier σ dont le temps cyclique moyen λ_σ est au plus K ?*

Théorème 18 *Dec-BCS(p) est NP-complet.*

Preuve On montre d'abord que Dec-BCS(p) est bien dans NP. On se restreint aux graphes $G = (V, E, d, w)$ tels que $K \leq \sum_{v \in V} d(v)$; sinon la réponse pour G est clairement "OUI". Etant donné un ordonnancement cyclique σ dont le temps de cycle moyen λ est au plus K (et $\lambda > 0$ parce que $p\lambda \geq \sum_{v \in V} d(v)$), on peut vérifier ne temps linéaire que les contraintes de dépendances sont satisfaites : il faut vérifier que chaque inégalité (9.7) est satisfaite ; il y en a $|E|$, et chacune ne met en jeu que des nombres qui sont bornés polynômialement ($\log(\lambda)$ est linéaire en $\log(d)$ parce que $\lambda \leq \sum_{v \in V} d(v)$).

Vérifier qu'au plus p ressources sont actives à chaque instant est plus délicat. Une vérification polynômiale en p ou λ n'est pas admissible, il faut une vérification polynômiale en $|V|$, $|E|$, éventuellement en $\log(p)$, $\log(d)$, en $\log(w)$. Comme l'ordonnancement est périodique de période λ, il suffit de vérifier les contraintes de ressource sur un intervalle de temps $I = [T,\ T+\lambda[$ où $T \geq \max_{v \in V} c_v$. A un instant donné t, le nombre $\mathcal{A}(v)$ d'opérations (v,k) actives est le nombre d'entiers positifs ou nuls dans l'intervalle $](t-c_v)/\lambda - d(v)/\lambda,\ (t-c_v)/\lambda]$. On écrit $d(v)-1 = r(v)\lambda + s(v)$ avec $0 \leq s(v) < \lambda$. On peut alors vérifier que $\mathcal{A}(v) = r(v)$ si $(t-c_v) - \lambda\lfloor(t-c_v)/\lambda\rfloor \geq s(v)+1$, et $\mathcal{A}(v) = r(v)+1$ sinon. Donc, pour chaque $v \in V$, il y a ua plus deux instants importants à vérifier : l'instant $t_1 \in I$ tel que $t_1 - c_v$ est un multiple de λ (et pour lequel $\mathcal{A}(v) = r(v)+1$), et l'instant $t_2 \in I$ (s'il existe) tel que $(t_2 - c_v) - \lambda\lfloor(t_2-c_v)/\lambda\rfloor = s(v)+1$. Il ya donc seulement $2 * V$ instants à vérifier, et les contraintes de ressource peuvent bien être vérifiées en temps polynômial : $O(V)$ additions, chacune avec $O(V)$ termes mettant en jeu des nombres polynômialement bornés.

Considérons maintenant une instance arbitraire Inst_1 de $\text{Dec}(p)$, le problème de décision associé à $\text{Pb}(p)$, l'ordonnancement d'un DAG avec p processeurs (Paragraphe 4.4) : soit un DAG $G = (V,E,w)$, un nombre de ressources $p \geq 1$, et une borne $K \in \mathbb{N}^*$ sur le temps d'exécution ; , existe-t-il un ordonnancement σ dont la latence est au plus K : $MS(\sigma,p) \leq K$?

Le théorème 6 montre que $\text{Dec}(p)$ est NP-complet. Nous allons réduire polynômialement $\text{Dec}(p)$ à $\text{Dec-BCS}(p)$. Nous construisons une instance Inst_2 de $\text{Dec-BCS}(p)$ comme suit : on pose $G = (V',E',d',w')$ avec $V' = V$, $d' = w$ (les sommets ont la même durée), et $E' = E \cup E''$, où $E'' = \{(u,v) \mid u \text{ sommet d'entrée de } V,\ v \text{ sommet de sortie de } V\}$. On ajoute donc une arête de tout sommet de sortie de V à chaque sommet d'entrée de V. Par définition des sommets d'entrée et de sortie, ces arêtes n'étaient pas présentes dans E. On pose $w'(e) = 0$ pour chaque arête $e \in E$ et $w(e) = 1$ pour chaque arête $e \in E''$. Enfin, on pose $p' = p$ et $K' = K$ et on demande s'il existe un ordonnancement cyclique pour G' avec p' processeurs et dont le temps de cycle moyen est au plus K'. La construction de Inst_2 est polynômiale (and même quadratique, car on ajoute au plus $O(|V|^2)$ arêtes) en la taille de Inst_1. De plus, si Inst_1 admet comme solution un ordonnancement σ, alors l'ordonnancement cyclcique $\sigma_c(v,k) = \sigma(v) + MS(\sigma,p)k$ est une solution de Inst_2.

Réciproquement, s'il existe un ordonnancement cyclique $\sigma_c(v,k) = c_v + \lambda k$, tel que $\lambda \leq K$, on définit l'ordonnancement σ pour le DAG G par $\sigma(v) = c_v - \min_{u \in V} c_u$, de manière à commencer l'exécution au top 0. σ est un ordonnancement valide parce que les arêtes de E de poids 0 imposent le respect de toutes les dépendances de G. De plus, grâce aux arêtes de E'', il n'y a aucun recouvrement entre deux instances différentes : la dernière opération (u,k) doit être terminée avant que la première opération $(v,k+1)$ ne puisse commencer. Par suite $\max_{u \in V}\{\sigma(u)+w(u)\} - \min_{v \in V}\sigma(v) \leq \lambda$, donc $MS(\sigma,p) \leq \lambda \leq K$, et Inst_1 admet bien une solution. ■

La seconde partie de la preuve peut être adaptée aux ordonnancements non cycliques : donc le problème de décision pour des ordonnancements généraux est au moins aussi difficile que $\text{Dec-BCS}(p)$. Mais la première partie de la preuve, qui montre que $\text{Dec-BCS}(p)$ est dans NP, ne se généralise pas à un ordonnancement quelconque, parce que nous ne savons pas si un tel ordonnancement peut être codé polynômialement en taille.

Nous allons maintenant présenter une heuristique dont la performance peut être garantie. On note λ_p, le temps de cycle moyen minimal réalisable par tout ordonnancement :

$$\lambda_p = \inf\{\lambda_\sigma \mid \sigma \text{ est a ordonnancement, à } p \text{ ressources, de temps de cycle moyen } \lambda_\sigma\}$$

Le Théorème 15 montre que $p\lambda_p \geq \sum_{v \in V} d(v)$ et que pour tout cycle $\mathcal{C}$ in G, $\lambda_p \geq \rho(\mathcal{C})$, i.e., $\lambda_p \geq \lambda_\infty$.

Soit $G = (V, E, d, w)$ un graphe de dépendances.

1. Définir $A(G) = (V, E', d)$ où $E' = \{e \in E \mid w(e) = 0\}$.
2. Effectuer un ordonnancement de liste σ_a sur $A(G)$.
   Calculer la latence de σ_a : $\lambda = \max_{v \in V}(\sigma_a(v) + d(v))$.
3. Définir l'ordonnancement cyclique σ par :

$$\forall v \in V,\ \forall k \in \mathbb{N},\ \sigma(v, k) = \sigma_a(v) + \lambda k$$

FIG. 9.9 – Algorithme de compactage de boucle.

9.4.2 Compactage de boucle

La technique de compactage de boucle, ou *loop compaction*, consiste à ordonnancer le corps de la boucle sans essayer de mélanger plusieurs itérations. Le principe général est le suivant : on considère le graphe $A(G)$ qui résume les dépendances figurant dans le corps de la boucle, boucle-indépendantes avec le vocabulaire du Chapitre 8 ; ce sont les dépendances dont la distance est égale à 0. Le graphe $A(G)$ est acyclique ; il peut être ordonnancé à l'aide de techniques de listes vues au Chapitre 4. Le schéma d'ordonnancement ainsi construit est répété plusieurs fois pour définir l'ordonnancement cyclique. Nous avons utilisé cette technique au Paragraphe 9.2.3. Nous la formalisons à la Figure 9.9.

Lemme 23 *L'ordonnancement σ donné par l'algorithme de compactage de boucle est un ordonnancement cyclique valide pour G. De plus, $p\lambda \leq p\lambda_p + (p-1)\Phi(G)$ où $\Phi(G)$ est le poids maximal d'un chemin dans $A(G)$.*

Preuve Tout d'abord, $A(G)$ est acyclique : s'il y avait un cycle $\mathcal{C}$, alors son poids $w(\mathcal{C})$ serait égal à 0. Mais $\mathcal{C}$ est aussi un cycle de G, et on a supposé que tous les cycles de G ont un poids strictement positif. L'ordonnancement σ_a est bien défini, soit λ sa latence. Le Théorème 7 assure de l'existence d'un chemin $\mathcal{P}$ dans $A(G)$ tel que $p\lambda \leq \sum_{v \in V} d(v) + (p-1)d(\mathcal{P})$. Comme $\sum_{v \in V} d(v) \leq p\lambda_p$ et que $d(\mathcal{P}) \leq \Phi(G)$ par définition de $\Phi(G)$, on a le résultat.

Il reste à prouver que σ est un ordonnancement cyclique valide. Les contraintes de ressources sont satisfaites par choix de l'ordonnancement de liste et de la définition de λ, qui garantit que deux itérations différentes ne se recouvrent pas. Pour les contraintes de dépendance, il faut pour chaque $e = (u, v)$ de E vérifier l'inégalité (9.7) :

$$\forall e = (u, v) \in E,\ \sigma_a(u) + d(u) \leq \sigma_a(v) + \lambda w(e)$$

Supposons d'abord que $e \in E'$, i.e., $w(e) = 0$. Comme σ_a est un ordonnancement pour $A(G)$, $\sigma_a(u) + d(u) \leq \sigma_a(v)$, et l'inégalité est satisfaite. Maintenant, if e n'apparaît pas dans $A(G)$, alors $w(e) > 0$, et $\lambda w(e) \geq \lambda$. Par définition de λ, on a $\sigma_a(u) + d(u) - \sigma_a(v) \leq \sigma_a(u) + d(u) \leq \lambda$. L'inégalité est encore satisfaite. ■

On ne peut pas garantir directement l'heuristique comme au Paragraphe 4.4. En effet dans ce paragraphe nous avions réussi à borner $\Phi(G)$ grâce à la Proposition 7. Ici, $\Phi(G)$ n'a aucune relation avec λ_p a priori . Il va falloir passer par une étape de mélange d'itérations, à travers des décalages, pour obtenir un graphe acyclique $A(G)$ dont le plus long chemin $\Phi(G)$ soit le plus petit possible.

Le but du décalage d'instructions exposé ci-dessous est double :

- réduire la valeur du plus long chemin $\Phi(G)$ dans $A(G)$, parce que cette quantité est étroitement liée à la borne de garantie de l'ordonnancement de liste ;
- réduire au maximum le nombre d'arêtes dans le graphe acyclique $A(G)$,de manière à réduire le nombre de contraintes de dépendance pour le problème d'ordonnancement acyclique.

9.4.3 Décalage de boucle

Reprenons notre exemple introductif, celui du Paragraphe 9.2.3 :

```
PRELUDE

DO k=1, N-1
    (A) : a(k) = c(k-1)
    (D) : d(k-1) = f(k-2)/3
    (B) : b(k) = a(k-2) * d(k-1)
    (C) : c(k) = b(k) + 1
    (E) : e(k) = sin(f(k-2))
    (F) : f(k-1) = log(b(k-1) + e(k-1))
ENDDO

POSTLUDE
```

Avec PRELUDE :

```
a(0) = c(-1)
b(0) = a(-2) * d(-1)
c(0) = b(0) + 1
e(0) = sin(f(-2))
```

et POSTLUDE :

```
d(N-1) = f(N-2)/3
f(N-1) = log(b(N-1) + e(N-1))
```

Ce code exécute exactement les mêmes calculs que le code original. Le calcul de (F, k) et (D, k) est retardé d'une itération. Donc, (B, k) et (E, k) sont toujours calculés avant (F, k), et $(F, k-1)$ est toujours calculé avant (D, k). Cependant, il faut vérifier que (F, k) et (D, k) ne sont pas calculés après des opérations qui doivent utiliser leur résultat. (F, k) doit être calculé avant $(E, k+2)$, ce qui est toujours le cas (il est maintenant calculé à l'itération d'avant). Mais (D, k) est maintenant calculé à la même itération que $(B, k+1)$; c'est pourquoi le corps de la boucle a été ré-ordonné de sorte que (D, k) apparaisse textuellement avant $(B, k+1)$ (la nouvelle distance de dépendance est 0, i.e., la dépendance devient boucle-indépendante).

Le nouveau graphe de dépendance est donné à la Figure 9.10. On peut appliquer l'algorithme de compactage de boucle, et on retrouve la solution avec $\lambda = 20$ donnée au Paragraphe 9.2.3. Le poids maximal d'un chemin dans le graphe acyclique de départ $A(G)$ était 24 (le chemin $E \to F$) ; il vaut maintenant 14 (le "chemin" réduit au sommet F).

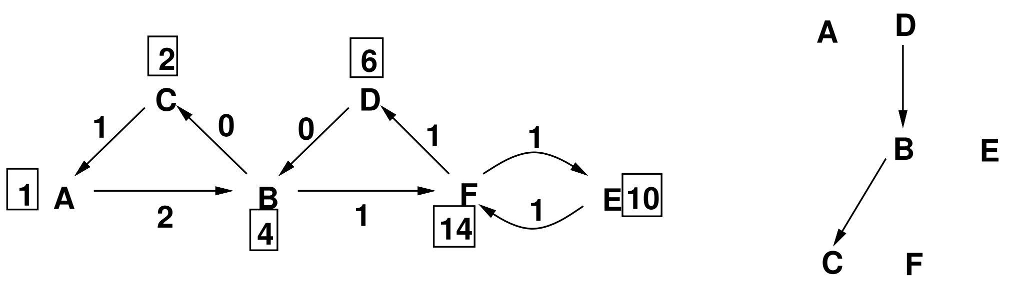


FIG. 9.10 – Le graphe de dépendance après décalage de boucle, et le graphe acyclique associé.

Le décalage de boucle se formalise ainsi : au lieu de considérer que le sommet $v \in V$ du graphe G représente toutes les opérations de la forme (v, k), on considère qu'il représente maintenant toutes les opérations de la forme $(v, k - r(v))$, ce qui revient à retarder l'opération (v, k) de $r(v)$ itérations.

Dans l'exemple $r(A) = r(B) = r(C) = r(E) = 0$ et $r(D) = r(F) = 1$. La nouvelle distance de dépendance $w_r(e)$ pour une arête $e = (u, v) \in E$ est alors $w_r(e) = w(e) + r(v) - r(u)$ car elle relie les sommets $(u, k - r(u))$ et $(v, k - r(u) + w(e)) = (v, k - r(v) + w_r(e))$. Ceci définit un graphe transformé $G_r = (V, E, d, w_r)$. Pour tout cycle $\mathcal{C}$, $w(\mathcal{C}) = w_r(\mathcal{C})$.

Si $w_r(e) \neq 0$, les deux opérations en dépendance sont exécutées à différentes itérations dans le code. Si $w_r(e) > 0$, les deux opérations sont calculées dans l'ordre original, et la dépendance est portée par la boucle ; si $w_r(e) = 0$, les deux opérations sont calculées à la même itération, et on place l'instruction correspondant à u textuellement avant celle correspondant à v pour respecter la dépendance, qui est boucle-indépendante. Ce ré-arrangement est toujours possible parce que le graphe transformé $G_r = (V, E, d, w_r)$ n'a pas de cycle de poids nul (les poids des cycles dans G et G_r sont les mêmes) ; enfin si $w_r(e) < 0$, le décalage de boucle n'est pas valide.

Lemme 24 *Il y a une correspondance bijective entre les ordonnancements pour G et pour G_r.*

Preuve La fonction σ_r est un ordonnancement pour G_r si et seulement si la fonction σ, définie par $\sigma(v, k) = \sigma_r(v, k + r(v))$, est un ordonnancement pour G. Clairement, les contraintes de ressource sont satisfaites pour σ ssi elles le sont pour σ_r. Il reste à vérifier toutes les contraintes de dépendance, i.e., les inégalités (9.2) :

$$\begin{array}{ll} \sigma(u,k) + d(u) \leq \sigma(v, k + w(e)) & \Leftrightarrow \\ \sigma_r(u, k + r(u)) + d(u) \leq \sigma_r(v, k + w(e) + r(v)) & \Leftrightarrow \\ \sigma_r(u, k') + d(u) \leq \sigma_r(v, k' + w_r(e)) \text{ où } k' = k + r(u) & \end{array}$$

Ainsi, G et G_r sont deux représentations du même problème. ∎

En d'autres mots, raisonner sur G_r n'est qu'un changement de représentation du problème, qui ne peut pas nous empêcher de trouver le bon ordonnancement. Nous utilisons le décalage de boucle comme une étape de pré-traitement avant la mise en oeuvre de l'algorithme de compactage de boucle.

Le décalage de boucle est appelé re-synchronisation, ou *retiming*, dans le contexte des circuits VLSI synchrones [49]. Nous allons garder la terminologie anglaise. Chaque sommet v représente un opérateur de logique combinatoire, de délai $d(v)$. Le poids $w(e)$ d'une arête e est interprété comme un nombre de registres. Le graphe $A(G)$ utilisé lors du compactage de boucle est le graphe des arêtes "sans registres." Quand $A(G)$ est acyclique (comme nous le supposons dans notre contexte), G n'a pas de cycle de poids nul, on dit qu'il est synchrone. Le retiming revient à enlever $r(u)$ registres au poids de chaque arête quittant u and à ajouter $r(v)$ registres à chaque arête entrante du sommet v. La contrainte $w(e) + r(v) - r(u) \geq 0$ signifie qu'un nombre négatif de registres n'est pas autorisé.

Avec cette nouvelle formulation, nos objectifs deviennent :

Objectif 1 : Trouver un retiming r qui minimise le plus long chemin dans in $A(G_r)$, i.e., en termes de circuits, qui minimise la période d'horloge $\Phi(G_r)$ du graphe re-synchronisé (voir le Paragraphe 9.4.4).

Objective 2 : Trouver un retiming r tel que le nombre d'arêtes dans $A(G_r)$ est minimal, i.e., distribuer les registres de manière à laisser aussi peu d'arêtes sans registre que possible (voir le Paragraphe 9.4.5).

9.4.4 L'algorithme de retiming de Leiserson-Saxe

Nous présentons ici l'algorithme de retiming proposé par Leiserson et Saxe pour minimiser la période d'horloge d'un circuit VLSI i.e., le poids maximal d'un chemin sans registre. Toutes les preuves sont données dans [49].

On note toujours $u \overset{\mathcal{P}}{\rightsquigarrow} v$ un chemin $\mathcal{P}$ de G de u à v, $w(\mathcal{P}) = \sum_{e\in\mathcal{P}} w(e)$ la somme des poids des arêtes de arêtes de $\mathcal{P}$, et $d(\mathcal{P}) = \sum_{v\in\mathcal{P}} d(v)$ la somme des délais des sommets de $\mathcal{P}$. On définit W et D comme suit :

$$\begin{aligned} W(u,v) &= \min\{w(\mathcal{P}) \mid u \overset{\mathcal{P}}{\rightsquigarrow} v\} \\ D(u,v) &= \max\{d(\mathcal{P}) \mid u \overset{\mathcal{P}}{\rightsquigarrow} v \text{ et } w(\mathcal{P}) = W(u,v)\} \end{aligned}$$

W et D se calculent à l'aide d'un algorithme de plus courts chemins entre tous les sommets de G, où le poids d'une arête $u \xrightarrow{e} v$ est la paire ordonnée lexicographiquement $(w(e), -d(u))$. Enfin, soit

$$\Phi(G) = \max\{d(\mathcal{P}) \mid \mathcal{P} \text{ chemin de } G,\ w(\mathcal{P}) = 0\}$$

$\Phi(G)$ est la longueur du plus long chemin de poids nul dans G, la période d'horloge de G en suivant la terminologie des circuits VLSI.

Théorème 19 (Théorème 7 de [49]) *Soit $G = (V, E, d, w)$ un circuit synchrone, c un réel strictement positif quelconque et r une fonction de V dans les entiers. Alors r est un retiming valide de G tel que $\Phi(G_r) \leq c$ si et seulement si*

1. *$r(u) - r(v) \leq w(e)$ pour toute arête $u \xrightarrow{e} v$ de G, et*
2. *$r(u) - r(v) \leq W(u,v) - 1$ pour tous sommets $u, v \in V$ tels que $D(u,v) > c$.*

Intuitivement, la première inégalité signifie que le retiming laisse un nombre de registres positif ou nul sur chaque arête (retiming valide), tandis que la seconde inégalité signifie que le retiming laisse au moins un registre sur chaque chemin $\mathcal{P}$ dont le poids $d(\mathcal{P})$ est supérieur à c (sinon la période d'horloge serait plus grande que $d(\mathcal{P})$).

Le Théorème 19 est à la base de l'algorithme OPT1 de la Figure 9.11, qui détermine un retiming tel que la période d'horloge du graphe re-synchronisé est minimale.

1. Calculer W et D (voir l'Algorithme WD de [49]).
2. Trier les différentes valeurs $D(u,v)$.
3. Parmi les valeurs $D(u,v)$, effectuer une recherche binaire pour trouver la période d'horloge minimale qui peut être réalisée. Pour vérifier qu'une période d'horloge c est réalisable, appliquer l'algorithme de Bellman-Ford pour déterminer si les conditions du Théorème 19peuvent être satisfaites.
4. Pour la période d'horloge minimale trouvée en 3, utiliser pour $r(v)$ les valeurs calculées par l'algorithme de Bellman-Ford, elles fournissent le retiming optimal.

FIG. 9.11 – Algorithm OPT1 de [49]

Le coût de l'algorithme OPT1 est $O(|V|^3 \log |V|)$. Il en existe un autre dont le coût est seulement $O(|V||E| \log |V|)$, ce qui constitue une amélioration significative pour les graphes creux : l'idée est de remplacer, dans le pas 3, l'appel à l'algorithme de Bellman-Ford par l'utilisation de l'algorithme de la Figure 9.12.

Dans la suite, on note Φ_{opt} la période d'horloge minimale réalisable pour G. Pour le graphe de Figure 9.3, $\Phi_{\text{opt}} = 14$ et le retiming r qui réalise cette période d'horloge est obtenu en deux itérations par l'algorithme FEAS. Les Figures 9.13 (a), (b), and (c) montrent les graphes re-synchronisés obtenus successivement. Voila comment nous avons trouvé la solution donnée à la Figure 9.10.

Etant donné un circuit synchrone $G = (V, E, d, w)$ et une période d'horloge souhaitée c, l'algorithme produit un retiming r de G tel que G_r est a circuit synchrone de période d'horloge $\Phi(G_r) \leq c$, si un tel retiming existe.

1. Pour chaque sommet $v \in V$, initialiser $r(v)$ à 0.
2. Répéter $|V| - 1$ fois :
   (a) Calculer le graphe G_r avec les valeurs courantes pour r.
   (b) pour tout sommet $v \in V$ calculer $\Delta(v)$, la somme maximale $d(\mathcal{P})$ des durées des sommets le long d'un chemin orienté $\mathcal{P}$, de poids nul dans G_r et se terminant en v. Ce calcul peut s'effectuer en $O(|E|)$.
   (c) Pour chaque sommet v tel que $\Delta(v) > c$, changer $r(v)$ en $r(v) + 1$.
3. Exécuter l'algorithme utilisé au pas 2b pour calculer $\Phi(G_r)$. Si $\Phi(G_r) > c$, aucun retiming valide n'existe. Sinon, r est le retiming cherché.

FIG. 9.12 – Algorithm FEAS de [49]

Revenons maintenant à la borne de garantie donnée au Lemme 23. Par décalage puis compactage de boucle, on peut définir un ordonnancement cyclique σ dont le temps de cycle moyen vérifie $p\lambda \leq p\lambda_p + (p-1)\Phi_{\text{opt}}$ où Φ_{opt} est la période d'horloge minimale réalisable pour G. Il reste à établir un lien entre Φ_{opt} et λ_p pour obtenir une heuristique pour laquelle le rapport λ/λ_p est borné.

Lemme 25 *Φ_{opt} et λ_∞ sont liés par les inégalités $\lambda_\infty \leq \Phi_{opt} \leq \lceil \lambda_\infty \rceil + d_{\max} - 1$ où $d_{\max} = \max_{v \in V} d(v)$.*

Preuve Appliquons le décalage/compactage de boucle avec des ressource illimitées. Pour cela, on définit un retiming r tel que $\Phi(G_r) = \Phi_{\text{opt}}$, et on construit le graphe $A(G_r)$ en supprimant dans G toutes les arêtes e telles que $w_r(e) > 0$. On construit alors un ordonnancement pour $A(G_r)$ à ressources illimitées en posant $\sigma_a(v) = \max\{d(\mathcal{P}) : \mathcal{P}$ chemin de $A(G)$ se terminant en $v\}$. La latence de σ_a est Φ_{opt} par construction. On obtient enfin un ordonnancement pour G en posant $\sigma(v, k) = \sigma_a(v) + (r(v) + k)\Phi_{\text{opt}}$. Par définition λ_∞ est le plus petit temps de cycle moyen pour $p = \infty$, on a donc $\lambda_\infty \leq \Phi_{\text{opt}}$.

Maintenant, considérons un ordonnancement σ à ressources illimitées, dont la période est $\overline{\rho} = \lceil \lambda_\infty \rceil$, comme défini au Paragraphe 9.3.3 : $\sigma(v, k) = t(s, v) + \overline{\rho}k$. Soit $s(v) = t(s, v) \bmod \overline{\rho}$ et $r(v) = \lfloor \frac{t(s,v)}{\overline{\rho}} \rfloor$. Puisque σ est un ordonnancement, pour toute arête $e = (u, v)$:

$$t(s, u) + d(u) \leq t(s, v) + \overline{\rho}\, w(e)$$

ce qu'on ré-écrit en :

$$s(u) + d(u) - s(v) \leq \overline{\rho}\,(w(e) + r(v) - r(u))$$

Comme $s(u) + d(u) - s(v) \geq -s(v) > -\overline{\rho}$, on a $w(e) + r(v) - r(u) > -1$, i.e., $w(e) + r(v) - r(u) \geq 0$ car ces nombres sont entiers. Donc r est un retiming valide. Calculons $\Phi(G_r)$, le poids maximal d'un chemin de $A(G_r)$. Quand $r(v) - r(u) + w(e) = 0$, $d(u) \leq s(v) - s(u)$. Donc pour tout chemin $\mathcal{P}$ dans $A(G_r)$, $\mathcal{P} = (v_1, \ldots, v_n)$, on a $d(\mathcal{P}) \leq s(v_n) - s(v_1) + d(v_n) < \overline{\rho} + d_{\max}$. Par définition, $\Phi(G_r)$ est le poids maximal d'un chemin dans $A(G_r)$, donc $\Phi(G_r) \leq \overline{\rho} + d_{max} - 1$. Enfin, comme $\Phi_{\text{opt}} \leq \Phi(G_r)$, on obtient l'inégalité cherchée. ∎

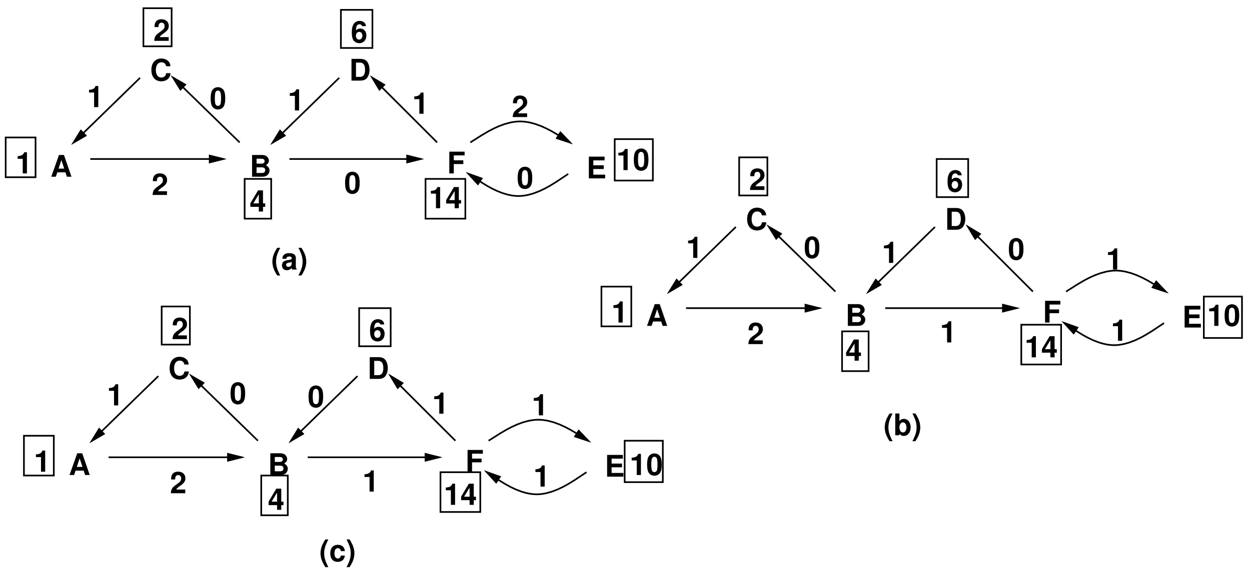


FIG. 9.13 – (a) Graphe de dépendance initial G. (b) et (c) Itérations successives de l'algorithme de Leiserson-Saxe.

Théorème 20 *Un ordonnancement cyclique de temps de cycle moyen* λ *vérifiant*

$$p\lambda \leq p\lambda_p + (p-1)(\lceil \lambda_\infty \rceil + d_{\max} - 1) \quad \textit{et } \lambda_\infty \leq \lambda_p$$

peut être construit en temps $O(|V||E| \log |V|)$.

Preuve La preuve est immédiate à partir des Lemmes 23 et 25. La complexité de l'heuristique est dominée par l'algorithme de Leiserson-Saxe pour le retiming. ■

9.4.5 Minimisation du nombre de contraintes dans $A(G_r)$

Parmi tous les graphes re-synchronisés dont la période d'horloge est la plus petite possible, à savoir Φ_{opt}, nous voulons en trouver un avec le nombre minimal d'arêtes de poids nul. Ce graphe réalisera les objectifs fixés à la fin du Paragraphe 9.4.3.

Revenons à notre exemple fétiche. L'algorithme de Leiserson-Saxe nous a conduit au graphe re-synchronisé G_r de la Figure 9.10. G_r minimise bien le poids maximal Φ d'un chemin de $A(G_r)$, i.e. il a une période d'horloge minimale ; mais G_r ne minimise pas pour autant le nombre d'arêtes de poids nul. Sur la Figure 9.10, on peut appliquer encore un nouveau retiming pour obtenir le graphe de la Figure 9.14. La longueur du plus long chemin de poids nul dans ce graphe est toujours $\Phi = 14$, mais le nombre d'arêtes de poids nul a diminué. En conséquence, le graphe acyclique $A(G_r)$ correspondant (voir Figure 9.14) contient moins d'arêtes que le graphe acyclique de la Figure 9.10, et est donc susceptible de conduire à une période plus petite lors du compactage de boucle. Bien sûr, un ordonnancement de liste conduit sur un sous-graphe d'un graphe ne conduit pas toujours à un temps d'exécution inférieur. Mais l'intuition nous dit que ce sera souvent le cas : moins il y a de contraintes de dépendance, plus il y a de liberté pour l'ordonnancement. C'est le cas dans l'exemple : on trouve une période égale à 19, comme le montre la Figure 9.15. IL s'avère que $\lambda = 19$ est le meilleur temps de cycle moyen réalisable avec $p = 2$ processeurs : la somme des délais de toutes les opérations est 37, et $\lceil \frac{37}{2} \rceil = 19$. Cependant, on peut trouver un ordonnancement temps de cycle moyen égal à 18.5, comme on le verra à la fin de ce paragraphe.

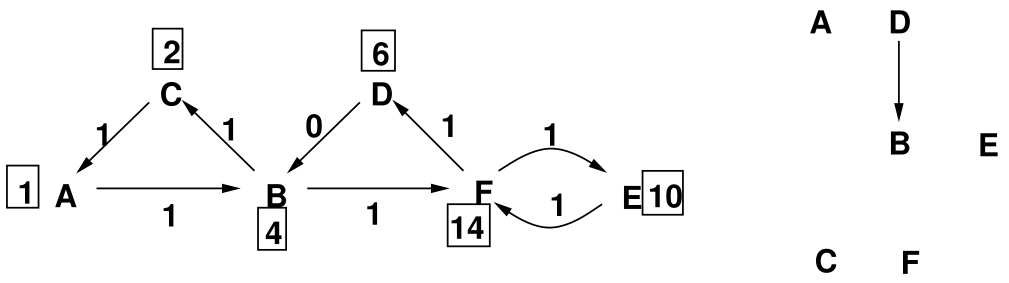


FIG. 9.14 – Le graphe re-synchronisé final, et le graphe acyclique associé.

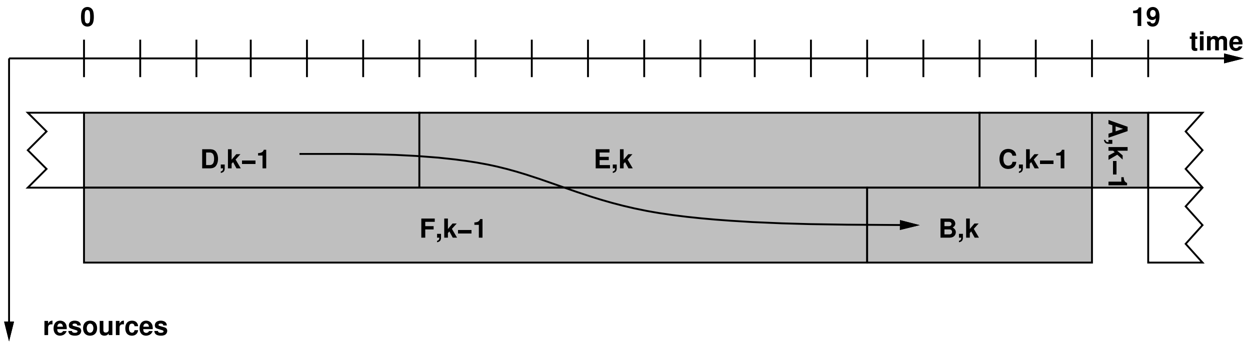


FIG. 9.15 – Une tranche de l'ordonnancement avec $\lambda = 19$.

Tout retiming r tel que $\Phi(G_r) = \Phi_{\text{opt}}$ est une solution entière du système suivant (voir la formulation du Théorème 19) :

$$\begin{cases} r(v) - r(u) + w(e) \geq 0 \text{ pour toute arête } u \xrightarrow{e} v \in E \\ r(v) - r(u) + W(u,v) \geq 1 \text{ pour tous sommets } u, v \in V \text{ tels que } D(u,v) > \Phi_{\text{opt}} \end{cases} \tag{9.8}$$

Parmi ces retimings, on veut trouver un retiming particulier r pour lequel le nombre d'arêtes de poids nul dans le graphe G_r est minimal. Pour ce faire, on utilise la technique suivante :

Lemme 26 *Soit $G = (V, E, d, w)$ un circuit synchrone. Un retiming r tel que $\Phi(G_r) = \Phi_{opt}$ et tel que le nombre d'arêtes de poids nul dans le graphe G_r est minimal peut être trouvé en résolvant le problème de programmation linéaire en nombres entiers :*

$$\begin{cases} \min \sum_{e \in E} v(e) \\ 0 \leq v(e) \leq 1 \\ r(v) - r(u) + w(e) + v(e) \geq 1 \textit{ pour toute arête } u \xrightarrow{e} v \in E \\ r(v) - r(u) + W(u,v) \geq 1 \textit{ pour tous sommets } u, v \in V \textit{ tels que } D(u,v) > \Phi_{opt} \end{cases} \tag{9.9}$$

Preuve Soit (r, v) une solution entière optimale du système linéaire (9.9). r définit un retiming pour G avec $\Phi(G_r) = \Phi_{\text{opt}}$ parce que le système (9.8) est vérifié ; en effet $r(v) - r(u) + w(e) + v(e) \geq 1$ et $v(e) \leq 1$ impliquent $r(v) - r(u) + w(e) \geq 0$.

Chaque valeur $v(e)$ est seulement contrainte par l'équation : $r(v) - r(u) + w(e) + v(e) \geq 1$. Distinguons deux cas :

– L'arête e de G_r a un poids nul, i.e. $r(v) - r(u) + w(e) = 0$. Alors $v(e) = 1$ est la seule possibilité.
– L'arête e de G_r a un poids strictement positif, i.e., $r(v) - r(u) + w(e) \geq 1$ (en effet r et d sont entiers). Dans ce cas, la valeur minimale pour v est 0.

Ainsi, étant donné un retiming r, la somme $\sum_{e \in E} v(e)$ est minimale quand elle est égale au nombre d'arêtes de poids nul dans G_r.

Il reste à montrer qu'une solution entière optimale peut être trouvée en temps polynômial. Le système (9.9), écrit sous forme matricielle $\min\{cx \mid Ax \leq b\}$, est tel que la matrice A est totalement unimodulaire. La résolution du problème de programmation linéaire en nombres entiers 9.9 n'est pas NP-complète ; le système (9.9) considéré comme problème de programmation linéaire sur les rationnels a une solution optimale entière (Corollaraire 19.1a de [62]), et cette solution entière peut être déterminée en temps polynômial (Théorème 16.2 de [62]). ■

Une optimisation directe pour réduire le coût est de pré-calculer les composantes fortement connexes G_i de G et de résoudre le problème séparément pour chaque composante G_i. Un retiming qui minimise le nombre d'arêtes de poids nul dans G_r est alors construit en ajoutant des constantes adéquates à chaque retiming r_i de manière à ce que les arêtes reliant les différentes composantes aient un poids positif.

Conclusion

Résumons : comment incorporer la minimisation du nombre de contraintes dans notre heuristique pour l'ordonnancement cyclique ? D'abord on calcule Φ_{opt}, la période d'horloge minimale réalisable pour G ; puis on résout le système (9.9) et on obtient un retiming r (décalage de boucle). On définit $A(G_r)$ comme le graphe acyclique des arêtes de poids nul dans G_r ; le plus long chemin dans $A(G_r)$ est minimisé, ainsi que le nombre d'arêtes dans $A(G_r)$. Enfin, on ordonnance $A(G_r)$ à l'aide d'un ordonnancement de liste (compactage de boucle).

Nous nous sommes restreints à des périodes λ entières, ce qui est suffisant pour obtenir une garantie de performance. La recherche pour des périodes *rationelles* pourrait donner de meilleurs résultats, au prix d'un coût accru. Chercher des valeurs $\lambda = \frac{p}{q}$ est possible en déroulant la boucle d'un facteur q, ce qui impose de traiter un graphe de dépendance étendu avec beaucoup plus de sommets et d'arêtes. Pour notre exemple, un ordonnancement de temps de cycle moyen égal à 18.5 existe, il est représenté à la Figure 9.16. Cette valeur ne peut pas être améliorée : les deux processeurs sont toujours actifs, car $\sum_{v \in V} d(v) = 37 = 2\lambda$. Nous avons obtenu cette solution en concaténant deux ordonnancements de liste différents de $A(G_r)$, l'un au plus tôt et l'autre au plus tard.

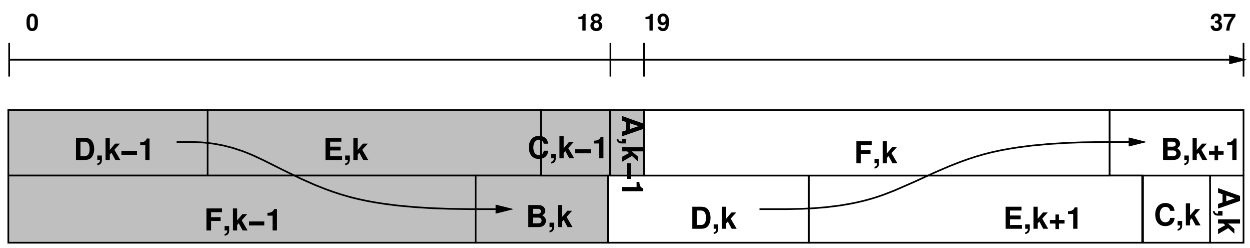


FIG. 9.16 – Un ordonnancement avec ($\lambda = 18.5$).

Notes bibliographiques

Ce chapitre s'inspire honteusement de l'excellent livre de Darte, Robert et Vivien [28] : en fait, nous avons traduit en français le troisième chapitre de cet ouvrage immortel, auquel nous renvoyons pour les références bibliographiques précises des notions présentées ici.

L'ordonnancement cyclique est une technique particulière pour aborder le pipeline logiciel, ou *software pipelining*. Le lecteur intéressé pourra se reporter à l'article de synthèse de V. H. Allan, R. B. Jones, R. M. Lee, et S. J. Allan [4], et profiter de l'imposante bibliographie proposée par B. R. Rau dans [59].

Bibliographie

[1] R. Agarwal, F. Gustavson, and M. Zubair. A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM J. Research and Development*, 38(6) :673–681, 1994.

[2] M. Ajtai, J. Komlos, and E. Szemeredi. An $o(n \log n)$ sorting network. In *Proceedings of the 15th ACM Symposium on the Theory of Computing STOC*, pages 1–19. ACM Press, 1983.

[3] S. G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, 1989.

[4] Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3) :367–432, September 1995.

[5] John Allen, David Callahan, and Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual Symposium on Principles of Programming Languages*, pages 63–76, Munich, Germany, January 1987.

[6] John R. Allen and Ken Kennedy. PFC : A program to convert Fortran to parallel form. In *Proceedings of IBM Conference on Parallel Computing and Scientific Computations*, 1982.

[7] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Second Edition.* SIAM, Philadelphia, PA, 1995. See also the URL : `http://www.netlib.org/lapack`.

[8] C. Arcelli and G. Sanniti Di Baja. Finding local maxima in a pseudo-euclidian distance transform. *Computer Vision, Graphics and Image Processing*, 43 :361–367, 1988.

[9] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4) :345–420, 1994.

[10] Uptal Banerjee, Rudolph Eigenmann, Alexandru Nicolau, and D.A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2) :211–243, 1993.

[11] B.A.Shirazi, A.R. Hurson, and K.M. Kavi. *Scheduling and load balancing in parallel and distributed systems.* IEEE Computer Science Press, 1995.

[12] K. Batcher. Sorting networks and their applications. In *Spring Joint Computing Conference*, pages 307–314. AFIPS Procedings vol. 32, 1968.

[13] F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid : Blueprint for a New Computing Infrastructure*, pages 279–309. Morgan-Kaufmann, 1999.

[14] Arthur J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15 :757–762, October 1966.

[15] F. Bitz and H.T. Kung. Path planning on the warp computer : using a linear systolic array in dynamic progamming. *Inter. J. Computer Math*, 25 :173–188, 1988.

[16] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* SIAM, 1997.

[17] R. Buyya. *High Performance Cluster Computing. Volume* 1 *: Architecture and Systems.* Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[18] R. Buyya. *High Performance Cluster Computing. Volume* 2 *: Programming and Applications.* Prentice Hall PTR, Upper Saddle River, NJ, 1999.

[19] L.E. Cannon. *A cellular computer to implement the Kalman filter algorithm.* PhD thesis, Montana State University, 1969.

[20] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, and S.M. Watt. *Maple Reference Manual*, 1988.

[21] P. Chrétienne, E.G. Coffman Jr., J.K. Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications.* John Wiley and Sons, 1995.

[22] Philippe Chrétienne. Task scheduling with interprocessor communication delays. *European Journal of Operational Research*, 57 :348–354, 1992.

[23] E. G. Coffman. *Computer and job-shop scheduling theory.* John Wiley & Sons, 1976.

[24] R. Cole. Parallel merge sort. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science FOCS*, pages 511–516. IEEE Computer Society, 1986.

[25] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[26] M. Cosnard and D. Trystram. *Algorithmes et Architectures Parallèles.* Interéditions, 1993.

[27] D. E. Culler and J. P. Singh. *Parallel Computer Architecture : A Hardware/Software Approach.* Morgan Kaufmann, San Francisco, CA, 1999.

[28] A. Darte, Y. Robert, and F.Vivien. *Scheduling and Automatic Parallelization.* Birkhaüser, 2000.

[29] F. Desprez. *Procédures de base pour le calcul scientifique sur machines parallèles à mémoire distribuée.* PhD thesis, Ecole Normale Supérieure de Lyon, January 1994.

[30] J. Reif (editor). *Synthesis of Parallel Algorithms.* Morgan Kaufmann, 1993.

[31] H. El-Rewini, T.G. Lewis, and H.H. Ali. *Task scheduling in parallel and distributed systems.* Prentice Hall, 1994.

[32] I. Foster and C. Kesselman, editors. *The Grid : Blueprint for a New Computing Infrastructure.* Morgan-Kaufmann, 1999.

[33] G. Fox, S. Otto, and A. Hey. Matrix algorithms on a hypercube i : matrix multiplication. *Parallel Computing*, 3 :17–31, 1987.

[34] Michael R. Garey and Davis S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1991.

[35] M. Gengler, S. Ubéda, and F. Desprez. *Initiation au Parallélisme.* Masson, 1996.

[36] Apostolos Gerasoulis and Tao Yang. A comparison of clustering heuristics for scheduling DAGs on multiprocessors. *J. Parallel and Distributed computing*, 16(4) :276–291, December 1992.

[37] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[38] M. Gondran and M. Minoux. *Graphs and Algorithms.* John Wiley & Sons, 1984.

[39] E. Goubault, F. Nataf, and M. Schoenauer. *Polycopié - Calcul parallèle.* Ecole Polytechnique, 2000.

[40] C. Hanen and A. Munier. Cyclic scheduling on parallel processors : an overview. In P. Chrétienne, E. G. Coffman, J. K. Lenstra, and Z. Liu, editors, *Scheduling Theory and its Applications*, pages 193–226. John Wiley & Sons, 1995.

[41] Claire Hanen and Alix Munier. An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. In *EFTA 95 : INRIA / IEEE Symposium on Emerging Technology and Factory Animation*, pages 167–189. IEEE Computer Science Press, 1995.

[42] K. Hwang and Z. Xu. *Scalable Parallel Computing.* McGraw-Hill, 1998.

[43] R. W. Kenyon. Tiling a rectangle with the fewest squares. *J. Combin. Theory A*, 76 :272–291, 1996.

[44] D. E. Knuth. *The Art of Computer Progrmming, volume 3 : Sorting and Searching.* Addison-Wesley, 1973.

[45] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook.* The MIT Press, 1994.

[46] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[47] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2) :83–93, February 1974.

[48] C.E. Leiserson. Fat-trees : universal networks for hardware-efficient supercomputing. *IEEE Trans. Computers*, 34(10) :892–901, 1985.

[49] C.E. Leiserson and J.B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6 :5–35, 1991.

[50] C. Lin and L. Snyder. A matrix product algorithm and its comparative performance on hypercubes. In *Scalable High Performance Computing Conference*, pages 190–193. IEEE Computer Society, 1992.

[51] Fredrik Manne and Tor Sørevik. Partitioning an array onto a mesh of processors. In *In proceedings of Para'96, Workshop on Applied Parallel Computing in Industrial Problems and Optimization*, volume 1184, pages 467–477. Lecture Notes in Computer Science, Springer, 1996.

[52] Serge Miguet and Yves Robert. Path planning on a ring of processors. *Intern. Journal Computer Math*, 32 :61–74, 1990.

[53] Serge Miguet and Yves Robert. Parallélisation d'algorithmes de balayage d'image sur un anneau de processeurs. *Technique et Science Informatiques*, 10(4) :287–296, 1991.

[54] Morris Newman. *Integral Matrices.* Academic Press, 1972.

[55] J.K. Peir and R. Cytron. Minimum distance : a method for partitioning recurrences for multiprocessors. *IEEE Transactions on Computers*, 38(8) :1203–1211, August 1989.

[56] C. Picouleau. Task scheduling with interprocessor communication delays. *Discrete Applied Mathematics*, 60(1-3) :331–342, 1995.

[57] William Pugh. The Omega test : a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8 :102–114, August 1992.

[58] F. Rastello. *Partitionnement : optimisations de compilation et algorithmique hétérogène.* PhD thesis, Ecole Normale Supérieure de Lyon, September 2000.

[59] B. R. Rau. Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1) :3–64, 1996.

[60] Y. Robert. *The impact of vector and parallel architectures on the Gaussian elimination algorithm.* Manchester University Press and John Wiley, 1991.

[61] A. Rosenfeld and A.C. Kak. *Digital Picture Processing.* Academic Press, New York, 1982.

[62] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, New York, 1986.

[63] Michael Wolfe. *High Performance Compilers For Parallel Computing.* Addison-Wesley, 1996.

[64] A. Y. Wu, S. K. Bhaskar, and A. Rosenfeld. Parallel computation of geometric properties from the medial axis transform. *Computer Vision, Graphics and Image Processing*, 41 :323–332, 1988.

[65] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.