UNIVERSITÉ DE
SHERBROOKE

Département d'informatique
Faculté des sciences

IFT 630 - Processus concurrents et parallélisme

Chapitre 8

Algorithmiques parallèles

Gabriel Girard[1]

Sherbrooke

24 février 2023

[1] Gabriel.Girard@usherooke.ca

Table des matières

Chapitre 8

Algorithmes parallèles

Ce chapitre est inspiré de l'introduction du manuel de JàJà [9].
Les documents suivants ont aussi été consultés [15, 2, 19].

Pour déterminer la performance des algorithmes, il faut utiliser un modèle et un cadre de travail (framework) permettant de les présenter et de les analyser.

Un modèle communément accepté pour l'étude des algorithmes séquentiels est celui basé sur un processeur unique équipé d'une mémoire centrale unique et de périphériques d'entrées/sorties (modèle RAM - Randon-Access-Machine). Le succès de ce modèle est dû à sa simplicité et à son habileté à capturer les performances des algorithmes séquentiels.

Par contre, l'évaluation de la performance des algorithmes parallèles devenant plus complexe, plusieurs modèles ont été proposés. La difficulté provient du fait que la performance dépend d'un ensemble de facteurs interreliés tels que :

- la concurrence ;
- l'allocation des processeurs et la planification ;
- la communication ;
- la synchronisation.

8.0.1 Révision de l'algorithmique séquentiel

Lorsque l'on évalue la performance d'un algorithme, on analyse principalement son usage du temps UCT et celui de la mémoire en fonction de la taille des données en entrée. Pour les besoins de ce chapitre, nous ne considérons que le modèle RAM et uniquement la consommation de l'UCT. Pour ce faire, on dénombre ou estime le nombre d'instructions qu'un algorithme exécute en lien avec la quantité de données à traiter.

Rappelons que les langages de haut niveau contiennent plusieurs types d'énoncés dont :

- les énoncés d'assignation ;
- les blocs de code (begin/end ou {/}) ;
- les énoncés de sélection (if, switch, ...) ;
- les énoncés d'itération (while, for, do, ...) ;
- l'énoncé de fin.

Pour estimer le temps d'exécution seules les opérations de base sont considérées. Ainsi, on comptabilise principalement les instructions :

- de lecture (`load`) ;
- d'écriture (`store`) ;
- les opérations arithmétiques (addition, soustraction, division, multiplication) ;
- les opérations logiques (et, ou, décalage, ...).

S'il s'avère impossible d'identifier le nombre exact de ces opérations, on détermine alors celui dans le pire cas et celui d'un cas moyen.

Ainsi, si n est la taille des données en entrée, le résultat de l'analyse de la complexité d'une fonction $f(n)$ est exprimé en termes d'une ou de plusieurs limites asymptotiques parmi les suivantes :

1. La borne supérieure :

$$T(n) = O(f(n)) \iff \exists\ c, n_0 \text{ tel que } T(n) \leq c \times f(n),\ \forall n \geq n_0$$

2. La borne inférieure :

$$T(n) = \Omega(f(n)) \iff \exists\ c, n_0 \text{ tel que } T(n) \geq c \times f(n),\ \forall n \geq n_0$$

3. La limite exacte :

$$T(n) = \Theta(f(n)) \text{ si } T(n) = O(f(n)) \text{ et } T(n) = \Omega(f(n))$$

Il est important de noter que, ici, nous supposons que la taille des mots en mémoire n'affecte pas le coût ou l'évaluation (critère de coût uniforme).

8.1 Modèles

Plusieurs modèles ont été proposés pour développer, présenter et analyser les algorithmes parallèles. Toutefois, dans ce chapitre nous n'aborderons que trois de ces modèles, soit ceux qui sont les plus couramment utilisés :

- **les graphes orientés acycliques (DAG) ;**
  Un modèle simple et indépendant de l'architecture matérielle sous jacente.
- **le modèle à mémoire partagée (PRAM) ;**
  Un modèle attrayant par sa ressemblance au modèle séquentiel.
- **le modèle réseau.**
  Un modèle intéressant car il permet de capturer la complexité des communications en intégrant la topologie du réseau.

8.2 Critères pour l'évaluation et mesures

Le contexte dans lequel nous voulons faire nos analyses concerne les algorithmes parallèles qui s'exécutent sur un ordinateur multiprocesseurs ou sur un réseau d'ordinateurs afin de résoudre un problème donné.

Il est bon de rappeler que l'objectif ultime demeure de diminuer le temps de traitement.

L'analyse que nous désirons effectuer consiste à déterminer si un algorithme se «comporte bien» dans un environnement parallèle.

Lors d'une telle analyse dans un environnement centralisé, les critères employés sont :

- le temps d'exécution ;
- la quantité de mémoire occupée ;
- la facilité de programmation.

L'analyse des algorithmes parallèles utilise ceux-ci, mais considèrent en plus les critères suivants :

- le nombre de processeurs requis et leur capacité mémoire ;
- le modèle de communication ;
- les protocoles de synchronisation.

Il existe plusieurs mesures pour évaluer des algorithmes parallèles. Pour les présenter, nous admettons, dans les sous-sections suivantes,le contexte suivant :

- Soit Q le **problème à résoudre** et n la taille des données en entrée pour celui-ci.
- Soit A un **algorithme parallèle** qui résout Q.

Notre environnement parallèle

Pour toutes les mesures de nos exemples (à moins d'indications contraires), nous supposons l'existence d'un environnement comportant dix processeurs.

8.2.1 $T^*(n)$: la complexité séquentielle

La complexité séquentielle $T^*(n)$ représente le temps d'exécution optimal d'un algorithme séquentiel. Ceci signifie qu'il existe un algorithme séquentiel optimal qui résout Q dans cette limite de temps, i.e. qu'il est possible de prouver qu'il n'existe aucun algorithme qui le résout plus rapidement.

Exemple 1

Nous supposons $T^*(n) = 20$ secondes.

8.2.2 $T_p(n)$: le temps d'exécution parallèle

Le temps d'exécution parallèle $T_p(n)$ est défini par le temps d'exécution de l'algorithme A sur p processeurs.

À noter que $T_1(n)$ peut être différent de $T^*(n)$. En effet, il est fort possible que l'algorithme optimal produisant $T^*(n)$ soit difficilement parallélisable et qu'un autre algorithme sous-optimal soit choisi pour la parallélisation. Ainsi, on pourrait choisir le tri bulle pour produire une version parallèle par rapport à un autre qui s'avérerait plus rapide en séquentiel.

Exemple 1

Nous supposons que :

- $T_p(n) = T_{10}(n) = 4$ secondes.
- $T_p(n) = T_1(n) = 25$ secondes.

Il existe une borne inférieure au temps d'exécution parallèle, notée $T_\infty(n)$, au delà de laquelle aucune accélération n'est possible pour un algorithme particulier et cela peu importe le nombre de processeurs que l'on ajoute. Ainsi,

$$T_p(n) \geq T_\infty(n) \quad \forall p.$$

8.2.3 $S_p(n)$: l'accélération

L'accélération obtenue par l'algorithme A relativement au temps d'exécution séquentiel est décrite par la formule

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

L'accélération devrait généralement être supérieure à un, i.e. que le programme s'exécute plus rapidement que le programme séquentiel (en temps d'horloge). Idéalement, $S_p(n) \equiv p$, ce qui signifie que sur p processeurs, l'algorithme devrait s'exécuter p fois plus rapidement.

Exemple 1

Nous obtenons :

$$S_p(n) = S_{10}(n) = \frac{T^*(n)}{T_{10}(n)} = \frac{20}{4} = 5.$$

Toutefois, en réalité $S_p(n) \leq p$. Les facteurs qui introduisent des inefficacités et nuisent à l'accélération sont :

- les délais de communication ;
- la surcharge due à la synchronisation des activités des divers processeurs et par le contrôle du système ;
- insuffisamment de concurrence au niveau de l'algorithme lui-même.

En fait, il y a cinq variantes à l'accélération :

- $RS_p(n)$: l'**accélération relative** mesurée par rapport à la vitesse séquentielle et obtenue par :

$$RS_p(n) = \frac{T_1(n)}{T_p(n)}$$

- $S_p(n)$: l'**accélération réelle** mesurée par rapport à l'algorithme optimal et obtenue par (déjà présentée) :

$$S_p(n) = \frac{T^*(n)}{T_p(n)}$$

- $AS_p(n)$: l'**accélération absolue** mesurée par rapport à l'algorithme optimal sur le processeur séquentiel le plus puissant et décrit par :

$$AS_p(n) = \frac{meilleur\ algo + meilleur\ pcsr}{T_p(n)}$$

- $ARS_p(n)$: l'**accélération réelle asymptotique** décrite par :

$$ARS_p(n) = \frac{O(T^*(n))}{O(T_p(n))}$$

- $AVS_p(n)$: l'**accélération relative asymptotique** décrite par :

$$AVS_p(n) = \frac{O(T_1(n))}{O(T_p(n))}$$

Exemple 1

Nous obtenons :

$$RS_{10}(n) = \frac{25}{4} = 6,25$$

En pratique, il faudrait aussi pondérer l'accélération avec le coût de l'achat, l'installation et l'entretien des systèmes.

8.2.4 E_p : l'efficacité

Une autre mesure de performance importante est celle de l'efficacité. Celle-ci fournit une indication sur l'utilisation effective de tous les processeurs et est décrite par :

$$E_p(n) = \frac{T_1(n)}{pT_p(n)}$$

Exemple 1

Nous obtenons :

$$E_{10}(n) = \frac{25}{10 \times 4} = 0.625$$

Ainsi, une efficacité $E_p(n) = 1$ indique que $T_p(n)$ s'exécute p fois plus vite que sa version séquentielle et que les p processeurs font du travail utile pendant tout le calcul.

Généralement, l'efficacité est inférieure à un ($E_p(n) < 1$) indiquant que pendant le calcul, certains processeurs sont inactifs. Dans l'exemple 1, l'efficacité est de $0,625$ exprimant qu'en moyenne pendant tout le calcul, près du tiers des processeurs sont inactifs.

Tout comme le temps d'exécution, il existe une limite à partir de laquelle il n'y a plus aucun gain à espérer du côté de l'efficacité. Cette limite est habituellement fixée par :

$$E_p(n) \leq \frac{T_1(n)}{pT_\infty(n)}$$

où, si vous vous souvenez bien, $T_\infty(n)$ représente la limite inférieure sur le temps d'exécution parallèle. L'efficacité d'un algorithme diminue rapidement lorsque le nombre de processeurs est trop grand. Ce nombre de processeurs est fixé par :

$$\frac{T_1(n)}{T_\infty(n)}$$

Exemple 1

En supposant que $T_\infty(n) = 3$, alors la limite maximale du nombre de processeurs est fixée par :

$$\frac{T_1(n)}{T_\infty(n)} = \frac{25}{10 \times 3} = 0.833$$

Ainsi,

$$E_{10}(n) \leq \frac{25}{10 \times 3} = 0.833$$

car la nombre de processeurs est plus grand que la limite. Si nous augmentons le nombre de processeurs à 20, nous ne ferions que diminuer l'efficacité car il n'y aurait aucun gain en temps d'exécution. Ainsi :

$$E_{20}(n) \leq \frac{25}{20 \times 3} = 0.416$$

8.3 Modèles pour algorithmique parallèle

Le modèle RAM est couramment employé pour évaluer la performance des algorithmes séquentiels. Comme nous l'avons déjà mentionné, la modélisation des algorithmes parallèles est toutefois plus complexe.

Le modèle idéal décrirait et analyserait tous les aspects des algorithmes parallèles. Il devrait donc être :

- **simple ;**

  Sa simplicité permettrait de décrire les algorithmes parallèles et d'analyser mathématiquement leur performance en terme de vitesse, communication et occupation de la mémoire. Il se doit aussi d'être indépendant du matériel.

- **Implantable.**

  Les algorithmes développés sur ce modèle s'implanterait aisément dans un environnement parallèle, L'analyse devrait donc capturer de façon réaliste et significative la performance réelle de ces implantations.

Il existe peu de modèles qui satisfont à ces critères étant donné la très grande quantité d'algorithmes parallèles conçus pour des architectures spécifiques ou des ordinateurs spécifiques.

Les modèles les plus connus sont :

- **graphes orientés acycliques ;**
- **PRAM (Parallel RAM) ;**
- **réseau ;**
- arbres de comparaison ;
- réseaux de tri (sorting networks) ;
- circuits booléen.

Dans ce chapitre, nous abordons seulement les trois premiers modèles.

8.3.1 Graphes orientés acycliques (DAG)

Ce modèle ressemble au graphe de précédence abordé précédemment. Ainsi dans ce graphe :

- Chaque entrée est représentée par un nœud sans arc en entrée (figure 8.1) ;

Figure 8.1 – Nœuds d'entrée

- Chaque opération correspond à un nœud ayant des arcs en entrée provenant d'autres nœuds (figure 8.2) ;

Figure 8.2 – Nœuds d'opération

- Un arc reliant deux nœuds défini une contrainte de précédence (exécution séquentielle) ;
- Le nombre d'arcs entrant provenant de chaque nœud interne est d'au plus deux ;
- Un nœud n'ayant aucun arc de sortie détermine la fin du calcul (ou la sortie du résultat).

En supposant que la taille des données en entrée est de n, un graphe ayant n nœuds d'entrée correspond à un traitement sans instruction de branchement ou d'itération (boucles qui dépendent de n). Pour parvenir à ce type de graphe, il suffit de dérouler les boucles répétant les mêmes instructions.

Un algorithme est représenté par une famille de graphes $\{G_n\}$ où n correspond à la taille de l'entrée

Ce modèle est bien adapté au traitement d'algorithmes d'analyse numérique car ceux-ci contiennent des instructions de branchement matérialisant des boucles qui dépendent de la taille des données à traiter (n). On déroule la boucle en répétant l'instruction n fois. Le graphe spécifie donc les opérations faites par l'algorithme et les contraintes de précédence à respecter lors de leur exécution.

Exemple 1 : somme de n valeurs

On veut calculer la somme des n éléments d'un tableau. Les figures 8.3 et 8.4 introduisent deux graphes (DAGs) solutionnant ce problème pour $n = 8$. Ceux-ci, qui forment une famille de graphes pour l'algorithme «somme», servent à analyser leur performance en supposant que chaque processeur peut accéder aux données des autres sans coût additionnel.

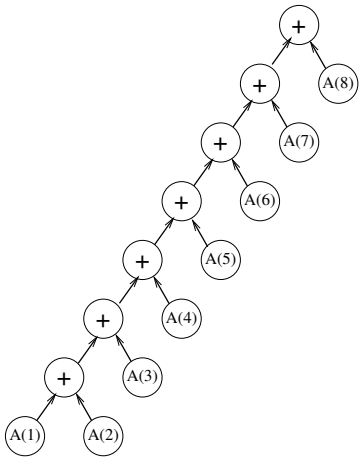


Figure 8.3 – Solution 1 pour la somme de n valeurs (graphe 1)

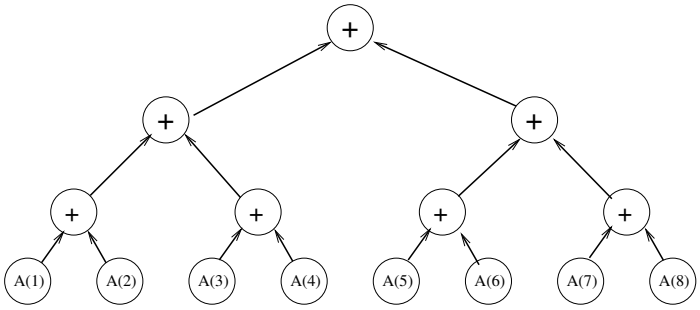


Figure 8.4 – Solution 1 pour la somme de n valeurs (graphe 2)

Une implantation possible pour ces deux graphes consiste à associer chaque nœud à un processeur particulier. Cette étape s'appelle la planification et implique aussi d'associer un temps d'exécution à chaque opération (nœud interne).

Soit p le nombre de processeurs, la planification consiste à associer à chaque nœud interne i une paire (j_i, t_i) où $j_i < p$, J_i étant un numéro de processeur et t_i une unité de temps (le processeur j traite l'opération i au temps t). Le temps t_i associé à un nœud de départ (nœud d'entrée) est 0 et aucun processeur ne lui est associé.

Lors de cette planification, les conditions suivantes doivent être respectées :

1. si $t_i = t_k$ pour $i \neq k$ alors $j_i \neq j_k$;
   Chaque processeur traite une seule opération par unité de temps.
2. si (i, k) est un arc alors $t_k \geq t_i + 1$.
   L'opération du nœud k doit s'exécuter après l'opération du nœud j.

On appelle la séquence $\{(j_i, t_i) | i \in n\}$ un horaire d'exécution pour le programme parallèle où n est l'ensemble de nœuds. Les figures 8.5 et 8.6 introduisent respectivement les horaires de nos deux solutions. Le tableau 8.1 résume et compare ces deux horaires.

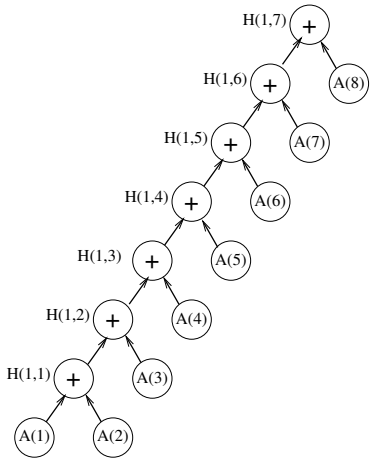


Figure 8.5 – Horaire pour la représentation 1

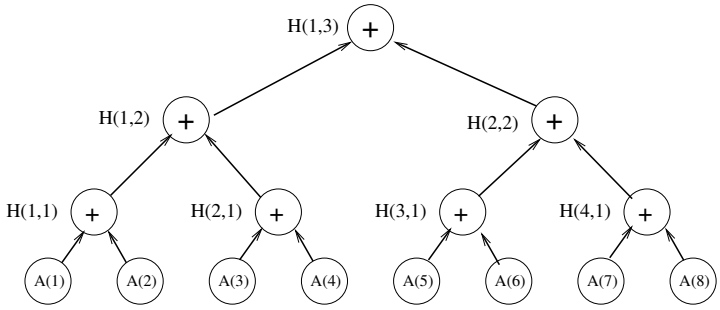


Figure 8.6 – Horaire pour la représentation 2

Pour chaque horaire proposé, le temps pour exécuter le programme est $Max_{i \in n} t_i$. Clairement, ce temps correspond donc à la profondeur du graphe. Ainsi pour le graphe 1, $Max_{i \in n} t_i = 7$ et pour

t_i	Horaire graphe 1	Horaire graphe 2
i=1	$p_1 \rightarrow B_1 := A_1 + A_2$	$p_1 \rightarrow B_1 := A_1 + A_2$ $p_2 \rightarrow B_2 := A_3 + A_4$ $p_3 \rightarrow B_3 := A_5 + A_6$ $p_4 \rightarrow B_4 := A_7 + A_8$
i=2	$p_1 \rightarrow B_1 := B_1 + A_3$	$p_1 \rightarrow C_1 := B_1 + B_2$ $p_2 \rightarrow C_2 := B_3 + B_4$
i=3	$p_1 \rightarrow B_1 := B_1 + A_4$	$p_1 \rightarrow D_1 := C_1 + C_2$
i=4	$p_1 \rightarrow B_1 := B_1 + A_5$	
i=5	$p_1 \rightarrow B_1 := B_1 + A_6$	
i=6	$p_1 \rightarrow B_1 := B_1 + A_7$	
i=7	$p_1 \rightarrow B_1 := B_1 + A_8$	

Table 8.1 – Horaire pour la seconde représentation

le graphe 2, $Max_{i \in n} t_i = 3$.

La complexité parallèle de l'algorithme que l'on extrait de cette famille de graphes est le moindre de ces maximums soit :

$$T_p(n) = Min(Max_{i \in n} t_i)$$

où Min est choisi parmi tous les horaires possibles pour exécuter le programme sur p processeurs. Dans notre cas, $T_p(n) = 3$ en faisant appel à quatre processeurs.

Nous pouvons tirer les conclusions suivantes à partir de l'analyse de ces deux représentations :

- La performance de la solution 1 (graphe 1) est $O(n)$ et cela peu importe le nombre de processeurs utilisés.
- La performance de la représentation 2 (graphe 2) est $O(log_2 n)$ si on emploie $n/2$ processeurs, Il est important de remarquer qu'utiliser plus de $n/2$ processeurs n'augmentera jamais la performance de ce algorithme.

Exemple 2 : multiplication de matrices

Soit deux matrices A et B de taille $n \times n$. Nous désirons effectuer l'opération de multiplication sur ces deux matrices pour obtenir

$$C = A \times B \ .$$

Pour y parvenir on doit calculer n^2 produits scalaires $C(i, j)$, chacun défini par la formule

$$C(i, j) = \sum_{l=1}^{n} A(i, l) \times B(l, j).$$

La figure 8.7 fournit une représentation graphique (DAG) pour un produit scalaire d'une matrice de taille 4×4 dont la profondeur est $log_2 n$ (soit 3 pour ce cas particulier).

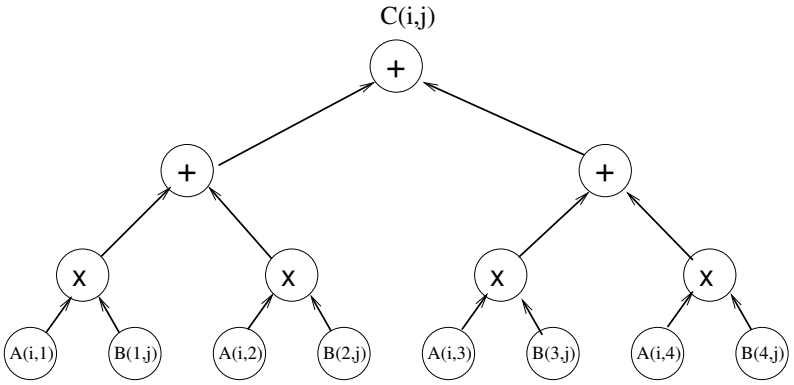


Figure 8.7 – Horaire pour la représentation 2

Il y aura donc n^2 graphes, un pour chaque produit scalaire. À partir de ce constat et de la représentation donnée, nous pouvons conclure que :

- la performance de l'algorithme sera de $O(log_2 n)$ sur n^3 processeurs.
  Cette performance est ahurissante et peu réaliste. Qu'il s'agisse seulement de penser que la multiplication de deux matrices 100×100 (de petites matrices à l'échelle scientifique) exigera un million de processeurs.
- Quelle sera la performance si on utilise n^2 processeurs : ?
  Il faut envisager que nous avons n^2 produits scalaires à effectuer (DAG). La planification de chaque graphe se fera donc sur un seul processeur.
- Quelle sera la performance si on a recours à n processeurs ?
  Il est possible d'assigner un processeur à chaque produit scalaire. Chaque processeur devra donc faire n produits scalaires.
- Quelle sera la performance avec un seul processeur ?

8.3.2 PRAM (Parallel Random Access Machine)

Le modèle PRAM [9, 6, 14] est une extension naturelle du modèle séquentiel RAM. En fait le modèle PRAM est un cas particulier du modèle à mémoire partagée, illustré à la figure 8.8, pour lequel nous supposons la présence de plusieurs processeurs ayant une mémoire locale privée et un mémoire globale partagée. De façon générale avec le modèle à mémoire partagée, on fait l'hypothèse que chaque processeur exécute son propre programme et communique via la mémoire partagée. Chaque processeur possède un identificateur unique accessible localement (même situation que lorsque l'on programme avec les langages/bibliothèques SR, JR, MPI et OpenCL).

Dans les modèles à mémoire partagée, il existe deux modes de fonctionnement :

- le mode asynchrone
  Selon ce mode, chaque processeur est indépendant et n'opère qu'avec sa propre horloge. Selon

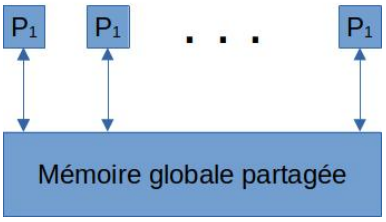


Figure 8.8 – Structure d'une architecture PRAM

cette approche, chaque processeur peut exécuter des programmes distincts et la synchronisation lors de l'accès à la mémoire est sous la responsabilité de la personne en charge du développement de l'algorithme. Ce modèle est donc équivalent au modèle MIMD (Multiple Program Multiple Data).

En fait ce modèle est celui des multiprocesseurs modernes (Intel, Arm, ...).

- Le mode synchrone ;

  Selon ce mode, tous les processeurs opèrent de façon synchrone contrôlés par une même horloge. Ce mode est fréquemment appelé PRAM [9, 10, 16, 13] et dans certains cas PRAM synchrone ([17]). Pour les besoins du cours nous y ferons référence tout simplement par PRAM. C'est ce modèle que nous détaillons dès à présent dans cette section.

Le modèle PRAM est particulièrement populaire pour l'analyse d'algorithmes parallèles. Il permet la conception d'algorithmes en faisant abstraction des contraintes architecturales et en ignorant la complexité des communications inter-processus. Aussi, ce modèle est plutôt théorique, mais il sert de base de développement pour des algorithmes parallèles efficaces.

PRAM

Le modèle PRAM est privilégié par la communauté d'informatique théorique car il soutient la création d'algorithmes parallèles sans se préoccuper des contraintes architecturales (puissance des processeurs, complexité des communications, ...).

Selon le modèle PRAM, tous les processeurs exécutent le même programme et ce, au même rythme. En cela, il agit exactement comme le mode SIMD (Single Instruction Multiple Data). De plus, chaque processeur est actif ou non. Un processeur inactif ne participe plus à au traitement du programme.

Généralement, un programme «PRAM» s'exécute selon trois phases :

1. la lecture : chaque processeur lit des données en mémoire partagée ;
   Nous définissons l'instruction de lecture en mémoire partagée par `Global read(X,Y)`
2. le calcul : chaque processeur effectue des calculs sur ses données locales ;
3. l'écriture : chaque processeur écrit les résultats dans la mémoire partagée.
   L'instruction d'écriture en mémoire partagée est définie ici par `Global write(U,V)`. La taille des données transférées représente la quantité de communication

Même si les processeurs exécute le même programme (et aussi les mêmes instructions) de façon synchrone, chacun possède un identificateur unique qui sert particulièrement à accéder à des

adresses distinctes en mémoire globale. Il est tout de même possible que plusieurs processeurs tentent d'accéder à la même zone de mémoire partagée simultanément. Comme la synchronisation n'est pas à la charge de la personne qui développe, le modèle propose différentes solutions pour gérer les accès concurrents :

- **EREW (Exclusive Read Exclusive Write) ;**
  EREW ne permet aucun accès simultané. Tous les programmes «corrects» doivent s'assurer de ne jamais accéder à la même adresse en mémoire simultanément car alors le résultat n'est pas défini.
- **CREW (Concurrent Read Exclusive Write) ;**
  Ici, les accès concurrents sont permis mais seulement en lecture. Tous les processeurs lisant simultanément la même adresse obtiendront le même résultat.
- **ERCW (Exclusive Read Concurrent Write) ;**
  Dans ce cas, les accès concurrents sont permis mais seulement en écriture. Ce mode n'est pas vraiment employé.
- **CRCW (common, arbitrary, priority).**
  Ce mode permet les accès concurrents autant en lecture qu'en écriture. Lors d'écritures concurrentes, plusieurs solutions sont proposées afin de résoudre les conflits potentiels :
  - **common CRCW ;**
    CRCW permet les écritures concurrentes si et seulement si tous les processeurs tentent d'écrire la même valeur.
  - **arbitrary CRCW ;**
    Ici, une valeur arbitraire est choisie parmi toutes celles que les processeurs tentent d'écrire.
  - **priority CRCW ;**
    La valeur choisie pour l'écriture est celle du processeur le plus prioritaire (fréquemment celui avec le plus petit identificateur).
  - **combining CRCW.**
    Dans ce dernier cas, la valeur emmagasinée est une combinaison des valeurs écrites (on applique souvent une opération associative et commutative telle l'addition ou le maximum).

Ces trois modèles diffèrent peu en terme de puissance de calcul ou d'expression. Le plus puissant des trois est CRCW. Le modèle CREW est également plus puissant que EREW mais de peu ($\leq log_2 n$).

Exemple 1 : multiplication matrice-vecteur

Soit A une matrice $n \times n$ et X un vecteur d'ordre n. Nous voulons calculer $Y = A \times X$.

Nous savons que la complexité de l'algorithme séquentiel est de $O(n^2)$. En ce qui concerne l'algorithme parallèle, sa complexité dépend de l'environnement. Pour la déterminer, nous supposons la présence de p processeurs, tel que $p < n$, que $r = n/p$ est entier et que l'on opère de façon asynchrone.

Exemple de lien entre n et p

Pour respecter la contrainte r est un entier, nous choisirons, par exemple pour 2 processeurs, des matrices de taille 4×4 ou de taille 8×8.
Dans ces deux cas, $r = 2$ et $r = 4$ respectivement.

Pour effectuer le calcul, on partitionne A en p matrices $A_1, A_2, ..., A_p$ de taille $r \times n$

Exemple de partitionnement d'une matrice 4×4

Illustrons la procédure pour multiplier une matrice A, de taille 4×4, par un vecteur d'ordre 4 sur deux processeurs. La valeur r étant 2, le partitionnement crée deux matrices A_1 et A_2, chacune de taille 2×4 (de taille $r \times n$ où $r = n/p = 4/2 = 2$).

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 4 & 5 & 6 & 7 \\ 1 & 2 & 3 & 4 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

$$A_1 \times X = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

$$A_2 \times X = \begin{pmatrix} 4 & 5 & 6 & 7 \\ 1 & 2 & 3 & 4 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

Suite au partitionnement, chaque processeur p_i lit la matrice A_i et le vecteur X, puis exécute le calcul $Z_i = A_i \times X$. À la fin de chaque calcul, chaque processeur emmagasine Z_i aux endroits appropriés dans Y.

Calcul du processeur p_1

$$Z_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 1 + 4 + 9 + 16 \\ 5 + 12 + 21 + 32 \end{pmatrix} = \begin{pmatrix} 30 \\ 70 \end{pmatrix}$$

Calcul du processeur p_2

$$Z_2 = \begin{pmatrix} 4 & 5 & 6 & 7 \\ 1 & 2 & 3 & 4 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 4 + 10 + 18 + 28 \\ 1 + 4 + 9 + 16 \end{pmatrix} = \begin{pmatrix} 60 \\ 30 \end{pmatrix}$$

Le programme 8.1 décrit le code pour effectuer ce calcul. La notation $A(l : u, s : t)$ employée dans le programme signifie que la matrice A contient :

- les lignes $l, l+1, l+2, ..., u$;
- les colonnes $s, s+1, s+2, ...t$.

De plus, l'expression $(i-1)r+1$ permet de calculer la position du premier élément de la matrice. Pour notre exemple, avec $r = 2$, ce calcul produira :

- pour le processus p_1, l'indice de départ $1 = (1-1) \times 2 + 1$;
  La processus p_1 utilisera donc la matrice $A(1 : 2, 1 : 4)$ et produira son résultat dans $Y(1, 2)$.
- pour le processus p_1, l'indice de départ $3 = (2-1) \times 2 + 1$.
  Le processus p_2 utilisera donc la matrice $A(3 : 4, 1 : 4)$ et produira son résultat dans $Y(3, 4)$.

```
1  /*********************************************************
2  ---------      Code pour le processus Pi    ---------------
3
4  Entrée : A : matrice
5           X : vecteur
6           n : taille de la matrice et du vecteur
7           i : le numero du processeur
8           p : le nombre de processeurs
9           r : n/p
10 Sortie : z : composants (i-1)r+1, ..., ir de Y
11 *********************************************************/
12 global read(X, w)                        // Étape 1
13 global read(A((i-1)r+1:ir, 1:n), B)      // Étape 2
14 z = B x w                                // Étape 3
15 global write(w, Y((i-1)r+1:ir)           // Étape 4
```

Programme 8.1 – Multiplication de matrices en parallèle sur PRAM.

Exemple 2

Soit une matrice de taille 10×10 ($n = 10$) et un environnement comprenant cinq processeurs ($p = 5$).
Dans ce cas, $r = 2$. Selon l'algorithme proposé, chacun des processeurs est responsable des parties suivantes de la matrice :

$\longrightarrow P_1$: read $(A(1 : 2, 1 : n), B)$
$\longrightarrow P_2$: read $(A(3 : 4, 1 : n), B)$
$\longrightarrow$ ...
$\longrightarrow P_5$: read $(A(9 : 10, 1 : n), B)$

Quelle est la complexité de cet algorithme ? Pour l'évaluer, on suppose que l'algorithme est exécuté en parallèle sur chaque processeur et qu'un accès concurrent en lecture à la matrice A est possible. De plus, l'algorithme, tel que défini, ne présente aucun conflit lors des écritures et donc celles-ci se réalisent également en parallèle sans recours à une synchronisation. L'évaluation de la complexité pour chacune des étapes est :

- Étape 3 (ligne 14)
  Lors de cette étape, il s'effectue $O(n^2/p)$ opérations arithmétiques. En effet, on divise le nombre d'opérations arithmétiques par le nombre de processeurs travaillant en parallèle. Voyons cela en détails :
  - Chaque produit scalaire requiert n multiplications et $n-1$ additions ;
  - Chaque processeur effectue r produits scalaires ;
  - Chaque processeur effectue donc
    * $r \times n = n^2/p$ multiplications (car $r = n/p$) ;
    * $r \times (n-1) = n(n-1)/p$ additions ;
    * $(n^2/p) + n(n-1)/p$ opérations $= O(n^2/p)$.
- Étape 1 et 2 (Lignes 12 et 13)
  Lors des deux étapes de lecture, $O(n^2/p)$ valeurs sont transférées. En effet, le vecteur X contient n valeurs et la partie de la matrice A transférée (A_i) contient $r \times n = n/p \times n = n^2/p$ valeurs.
- Étape 4 (ligne 15)
  À cette étape, chaque processeur écrit n/p valeurs

N.B : Le calcul précédant est valide en autant qu'aucune synchronisation ne soit requise. Il est possible qu'un autre partitionnement requiert de la synchronisation.

Exemple avec la matrice 4×4

Selon notre exemple de multiplication d'une matrice 4×4 par un vecteur, chaque processus effectue :

- $r \times n = n/p \times n = 4/2 \times 4 = 8$ multiplications $(= n^2/p = 4^2/2 = 8)$;
- $r \times (n-1) = n(n-1)/p = 4 \times 3/2 = 6$ additions ;
- $(n^2/p) + n(n-1)/p = 4^2/2 + 6 = 14$, ce qui est l'ordre de $O(n^2/p)$.

Exemple : somme de n valeurs

On souhaite additionner tous les éléments d'un tableau A de taille $n = 2^k$. On suppose la présence d'un environnement doté de n processeurs $p_1, p_2, ...p_n$, chacun exécutant le même algorithme. À la fin, le résultat doit être $S = A(1) + A(2) + ... + A(n)$.

Le programme 8.2 implante une solution à ce problème. Aux étapes 1 et 2, chaque processus lit un ensemble de valeurs et en crée une copie en mémoire globale afin d'éviter de modifier le tableau original. De cette façon, tous les processus ont accès à la copie (B).

L'étape 3 contient la boucle qui fait le calcul et, comme indiqué par la figure 8.9, ce calcul requiert $log(n)$ itérations. La sélection (ligne 11) permet de déterminer les processeurs qui effectueront le travail. Comme constaté à la figure 8.9, à chaque étape de la boucle le nombre de processeurs utiles diminue. Le travail accompli aux lignes 13 à 16, sert à lire le résultat du calcul précédant dans le

tableau B, faire la somme et emmagasiner le nouveau résultat dans ce même tableau. Toutefois les indices varient selon le processeur qui fait le calcul. Ainsi, si nous évaluons la somme des 8 valeurs, tel qu'illustré à la figure 8.9, lorsque h est égal à :

1. les processeurs 1 à 4 calculent et emmagasinent les résultats aux positions 1 à 4. Les lectures se font aux positions 1 et 2 pour p_1, 3 et 4 pour p_2, etc.
2. les processeurs 1 à 2 calculent et emmagasinent les résultats aux positions 1 à 2. Les lectures s'effectuent aux positions 1 et 2 pour p_1, 3 et 4 pour p_2.
3. le processeur p_1 procède au dernier calcul, emmagasine le résultat à la position 1, puis comme étape finale place le résultat dans S (étape 4, ligne 18).

```
1  /*********************************************************
2  ---------      Code pour le processus Pi    ---------------
3  Entrée : A : vecteur de taille n
4           n : taille de la matrice et du vecteur
5           i : le numéro du processeur
6  Sortie : S : la somme
7  *********************************************************/
8  global read(A(i), a)          // Étape 1
9  global write(a, B(i))         // Étape 2
10 For h=1 to log n  do          // Étape 3
11    if (i <= n/2^h)
12      begin
13        global read(B(2i-1), x)
14        global read(B(2i), y)
15        z = x+y
16        global write(z, B(i))
17      end --- barrière ----
18 If i=1 global write(z,S)      // Étape 4
```

Programme 8.2 – Exemple de la somme de n valeurs.

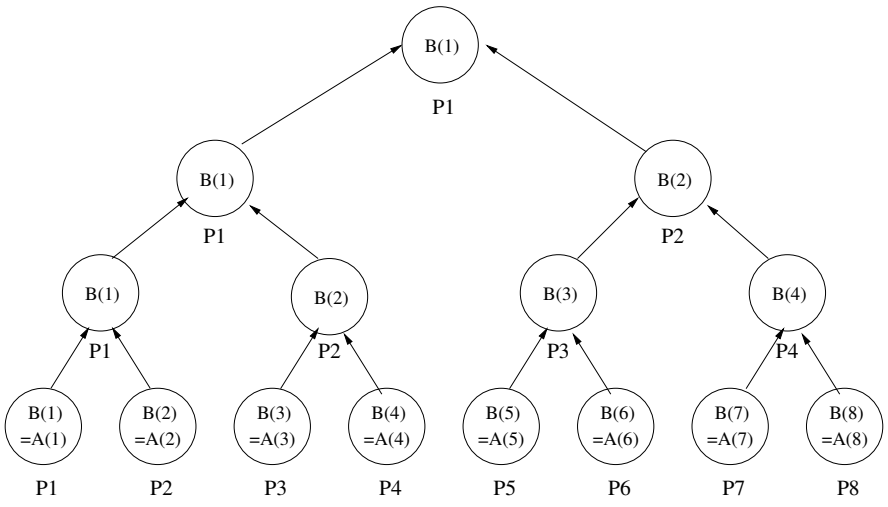


Figure 8.9 – DAG pour la somme de n valeurs

À noter que les tableaux A et B sont tous les deux en mémoire partagée. De plus, l'algorithme est synchrone, i.e. il y a une barrière entre chaque itération.

Exemple : Multiplication de matrices carrées

On désire implanter une multiplication de deux matrices sur le modèle PRAM, i.e. soit calculer $C = A \times B$.

Pour simplifier le problème, les matrices choisies sont carrées de taille «n X n» où $n = 2^k$. Nous supposons aussi que notre environnement fournit n^3 processeurs numérotés $P_{i,j,l}$.

Selon notre algorithme (programme 8.3), chaque ensemble de n processeurs $p_{i,j,*}$ calcule un produit scalaire ($O(log_2 n)$). Ainsi, chaque $P_{i,j,l}$ effectue l'opération $A(i,l) \times B(l,j)$. La ligne 11 de ce programme multiplie toutes les valeurs requises pour les produits scalaires. Ensuite, on remarque plusieurs similitudes entre cet algorithme et celui effectuant la somme. Ainsi, l'itération comprend $log(n)$ étapes pour faire la somme des n éléments du produit scalaire. À la fin, le résultat est copié dans la matrice destination (ligne 15).

```
1  /*********************************************************
2  ---------      Code pour le processus Pi    ---------------
3  Entrée :     A : matrice de taille  n x n
4               B : matrice de taille  n x n
5               n : taille de la matrice et du vecteur
6           i,j,l : le no du processeur
7  Sortie :     C : A x B
8  *********************************************************/
9  // on omet les global read et global write
10
11 C'(i,j,l) := A(i,l) x B(l,j)
12 For h=1 to log n  do
13       if (l <= n/2^h)
14            C'(i,j,l) = C'(i,j,2l-1) + C'(i,j,2l)
15 If l=1 C(i,j) = C'(i,j,l)
```

Programme 8.3 – Exemple de la somme de n valeurs.

Les $log_2(n)$ itérations impliquent que la complexité de l'algorithme est $O(log_2 n)$.

Cet algorithme requiert un modèle PRAM de type CREW. En effet, les mêmes valeurs pourraient être lues en parallèle par plusieurs processeurs, mais l'écriture finale (ligne 15) ne se fait qu'une seule fois pour chaque position (à cause de l'énoncé de sélection `if`). Si on retire cet énoncé, il faudra plutôt utiliser un modèle PRAM de type «common CRCW» puisqu'alors il y aura plusieurs écritures simultanées à la même position, mais tous écriront la même valeur.

8.3.3 Modèle réseau

Un modèle réseau se perçoit tel un graphe $G = (N, E)$ dans lequel chaque nœud $i \in N$ représente un processeur et chaque arc (i, j) représente un lien de communication bidirectionnel.

Chaque processeur du réseau :

- opère de façon synchrone ou asynchrone ;
- possède sa propre mémoire (aucune mémoire partagée n'est disponible) ;

Dans ce modèle, la communication et la synchronisation se font entièrement par l'envoi et la réception de messages. Les primitives de communication utilisées sont `Send(x,i)` et `Receive(y,j)`.

De plus, les communications ne se font pas nécessairement entre voisins directs. Le processus de livraison des messages d'une source vers une destination passe éventuellement par plusieurs intermédiaires. Ce processus de livraison qui comprend plusieurs intermédiaires s'appelle le routage.

Le modèle réseau incorpore la topologie. Ainsi il est possible d'évaluer les caractéristiques de la topologie du réseau via quelques métriques :

- le diamètre ;
  Le diamètre est la distance maximale entre deux nœuds du réseau.
- le degré maximum ;
  Le degré maximum représente le plus grand nombre de liens de communication en entrée et en sortie (arc) qu'un nœud possède.
- la connectivité.
  La connectivité d'un nœud A est le nombre de chemins différents allant de ce nœud vers un autre nœud. C'est en fait le nombre de liens à couper pour déconnecter entièrement deux nœuds.

Les topologies les plus populaires que nous allons présenter dans cette section sont :

- le réseau linéaire ;
- le maillage 2D ;
- l'hypercube.

Réseau linéaire

Le réseau linéaire est composé de n processeurs, $p_1, p_2, ..., p_n$, chacun d'eux étant connecté à un maximum de deux voisins. Ainsi, chaque p_i est connecté à p_{i-1} et p_{i+1}. La figure 8.10 illustre un réseau linéaire. Dans ce réseau, si on connecte p_n à p_1 (la ligne pointillée), on obtient un anneau. Les caractéristiques du réseau linéaire sont :

- Diamètre $= n - 1$;
  Cela signifie que la distance maximale entre deux nœuds, en l'occurrence p_1 et p_n, est de $n-1$ (arcs). Si l'on transforme le réseau en un anneau, cette distance devient $n/2$.
- Degré maximum $= 2$;
  Ainsi le nombre de liens en entrée ou en sortie de chaque nœud est au plus 2. Seuls p_1 et p_n n'ont qu'un lien en entrée ou en sortie. Cependant, si le réseau devient un anneau, chaque nœud possédera exactement deux liens en entrée et en sortie.
- Connectivité $= 1$.
  Selon la topologie du réseau linéaire, couper un seul lien parvient à isoler deux nœuds. La connectivité devient deux si la topologie est un anneau.

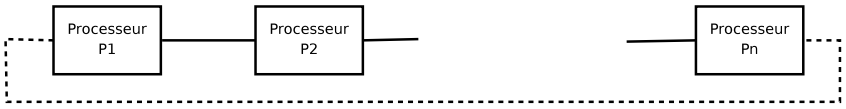


Figure 8.10 – Réseau linéaire

Illustrons le fonctionnement d'un algorithme sur cette topologie en implantant l'opération $Y = A \times X$, dans laquelle A est une matrice et X un vecteur (multiplication matrice-vecteur). Pour simplifier le problème, nous supposons que A est une matrice carré de taille $n \times n$ et que X est un

vecteur d'ordre n. Nous supposons également que l'environnement comprend p processeurs organisés en anneau tel que $r = n/p$ est un entier.

Pour effectuer le calcul parallèle, on partitionne A et X en p blocs $A = (A_1, A_2, ..., A_p)$ et $X = (X_1, X_2, ..., X_p)$ de façon à ce que :

- chaque A_i soit de taille $n \times r$;
- chaque X_i soit d'ordre r.

Partitionnement d'une matrice 4×4 et d'un vecteur d'ordre 4 pour 2 processeurs

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 4 & 5 & 6 & 7 \\ 1 & 2 & 3 & 4 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} \quad \nearrow \quad \begin{pmatrix} 1 & 2 \\ 5 & 6 \\ 4 & 5 \\ 1 & 2 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

$$\searrow \quad \begin{pmatrix} 3 & 4 \\ 7 & 8 \\ 6 & 7 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \end{pmatrix}$$

Selon cette approche, implantée par le programme 8.4, chaque processeur p_i calcule $z_i = A_i \times X_i$. À la fin, on cumule la somme $z_1 + z_2 + ... + z_p$. Ainsi, à l'étape 1, chaque processeur multiplie la matrice partitionnée avec le vecteur (aussi partitionné). À l'étape 2, on reçoit le résultat de notre voisin afin de poursuivre l'opération. Seul le premier processus (sans voisin direct) ne reçoit rien. À l'étape 3, on effectue la somme des résultats partiels et on la transmet au prochain voisin (étape 4). À la fin (étape 5) le nœud p_1 reçoit le résultat final.

```
1  /*********************************************************
2  ---------      Code pour le processus Pi    ---------------
3  Entrée : B : matrice de taille n x r
4                   -> A(1:n, (i-1)r+1:ir)
5           w : vecteur d'ordre r -> X((i-1)r+1:ir)
6           p : le nombre de processeurs
7           i : le no du processeur
8  Sortie : Y : Y = A1*X1 + A2*X2 + ... + Ai*Xi (Pi)
9           Y : Y = A*X (P1)
10 *********************************************************/
11 z := B * w                          // Étape 1
12 if i=1 then Y := 0                  // Étape 2
13        else receive(y, left)        // Étape 2
14 Y := Y + z                          // Étape 3
15 send(Y, right)                      // Étape 4
16 if i=1 then receive(y, left)        // Étape 5
```

Programme 8.4 – Exemple de la somme de n valeurs.

Partitionnement d'une matrice 4×4 et d'un vecteur d'ordre 4 pour 2 processeurs

Illustrons la procédure complète de l'algorithme par l'exemple suivant. Soit une matrice «4 × 4» à multiplier par un vecteur donné dans un environnement comprenant 2 processeurs (r = n/p = 2).

$$\text{À évaluer : } \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 1 & 2 & 3 & 4 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix}$$

On partitionne la matrice et le vecteur, et on envoie le résultat sur les noeuds P_1 et P_2. Ces deux noeuds font leur calcul (multiplication matrice-vecteur) en parallèle (Étape 1).

$$P_1 \text{ exécute : } \begin{pmatrix} 1 & 2 \\ 5 & 6 \\ 9 & 10 \\ 1 & 2 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 1 + 4 \\ 5 + 12 \\ 9 + 20 \\ 1 + 4 \end{pmatrix} = \begin{pmatrix} 5 \\ 17 \\ 29 \\ 5 \end{pmatrix} = Z_1$$

$$P_2 \text{ exécute : } \begin{pmatrix} 3 & 4 \\ 7 & 8 \\ 11 & 12 \\ 3 & 4 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 9 + 16 \\ 21 + 32 \\ 33 + 48 \\ 9 + 16 \end{pmatrix} = \begin{pmatrix} 25 \\ 53 \\ 81 \\ 25 \end{pmatrix} = Z_2$$

Le noeud (ou processus) P_2 reçoit le résultat de P_1 (Étape 2) pour ensuite effectuer l'addition (Étape 3).

$$P_2 \text{ exécute : } Z_1 + Z_2 = \begin{pmatrix} 5 \\ 17 \\ 29 \\ 5 \end{pmatrix} + \begin{pmatrix} 25 \\ 53 \\ 81 \\ 25 \end{pmatrix} = \begin{pmatrix} 30 \\ 70 \\ 110 \\ 30 \end{pmatrix}$$

Le résultat final est transmis à P_1 (Étape 4) qui le recevra à l'étape 5.

Quelle est la complexité du calcul de ce produit matriciel ? Évaluons la complexité du calcul effectué à chacune des étapes :

- **Calcul à l'étape 1** : $(n^2/p) \rightarrow O(n^2/p)$;
  La multiplication matrice-vecteur est d'ordre n^2. En divisant la tâche entre p processeurs, on obtient la complexité $O(n^2/p)$.
- **Calcul à l'étape 3** : $(n(p-1)) \rightarrow O(\alpha\, n)$ où α est une constante ;
  Le calcul effectué à l'étape 3 comprend n additions, donc $O(n)$. Toutefois, toutes les étapes 3 elles, sont exécutées de façon séquentielle. En effet, chaque processeur à cette étape doit patienter jusqu'à ce que son prédécesseur ait terminé son étape 5 :
  - A_p attend que le calcul de A_{p-1} soit terminé (étape 5) pour débuter le sien (étape 3) ;
  - A_{p-1} attend que le calcul de A_{p-2} soit terminé (étape 5) pour débuter le sien (étape 3) ;
  - ...

  On doit donc additionner les temps, ce qui produira une complexité de $O(n(p-1))$.
- **Temps total de calcul** $= T_{calcul} = (n^2/p) + n(p-1) = \alpha(n^2/p) \rightarrow O(n^2/p)$;
- **Temps de communication** $= T_{comm} = p \times comm(n)$.
  Le modèle réseau a cette capacité de permettre de calculer la charge en communication d'un algorithme. Pour ce faire, on considère que le temps pour transmettre un message est $comm(n) = \sigma + n\tau$ où
  - σ est le temps pour initialiser la transmission
  - τ est le taux de transmission

  Considérant les communications entre les p processeurs, le temps total en communication est $p \times comm(n)$.
- **Temps d'exécution total** $= T = T_{calcul} + T_{comm} = \alpha(n^2/p) + p(\sigma + n\tau)$

Maillage 2D (Mesh)

Un maillage 2D est une version en deux dimensions d'un réseau linéaire. Un tel maillage contient $p = n^2$ processeurs organisés dans une grille de taille «$n \times n$» telle que le processeur $P_{i,j}$ est connecté aux processeurs $P_{i\pm1,j}$ et $P_{i,j\pm1}$. La figure 8.11 illustre ce type de réseau.

Les caractéristiques d'une telle topologie comprenant p nœuds sont :

- Diamètre $= 2\sqrt{p} - 2$;
  La valeur $\sqrt{p}$ nous donne le nombre de nœuds pour chaque réseau linéaire (une dimension du maillage). Pour se rendre aux deux nœuds d'un maillage 2D, il y a à traverser au plus deux de ces réseaux linéaires (horizontal et vertical). La distance pour parcourir un réseau linéaire étant de $\sqrt{p} - 1$ (liens ou arcs), la distance maximale entre deux nœuds du maillage est de $2\sqrt{p} - 2$. Par exemple, si $p = 16$, le diamètre est de 6.
- Degré maximum $= 4$;
  Dans un maillage, chaque nœud possède au maximum 4 liens en entrée ou en sortie.

Un maillage 2D possède quelques autres caractéristiques intéressantes pour l'analyse d'algorithme, en particulier :

- il supporte plusieurs types d'algorithmes synchrones et asynchrones ;
- la simplicité ;
- la régularité ;

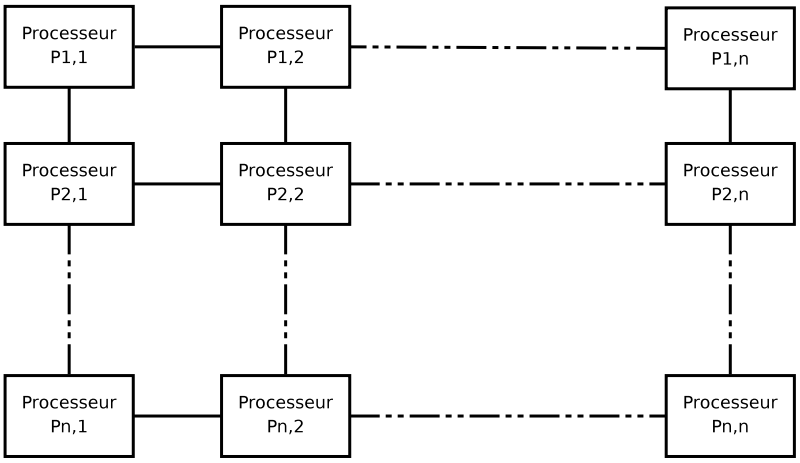


Figure 8.11 – Un maillage 2D

- la capacité d'expansion ;
- il correspond bien à la structure de calcul de plusieurs applications.

Notons qu'étant donné son diamètre, tous les calculs non triviaux requièrent $\Omega(\sqrt{p})$ étapes.

L'exemple choisi pour fin d'analyse est la multiplication de deux matrices carrées d'ordre n, soit A et B. La matrice résultante C est donc telle que $C = A \times B$. La solution est basée sur l'utilisation d'un réseau systolique.

Réseau systolique [18, 20]

Un réseau systolique est un réseau homogène de noeuds de calcul étroitement couplés (maillage). Chaque nœud calcule indépendamment un résultat partiel en fonction des données reçues de ses voisins en amont, emmagasine le résultat et le transmet à ses voisins en aval. Le nom «systolique» est choisi par analogie avec le pompage régulier du sang par le cœur car la propagation des données à travers un réseau systolique ressemble au pouls du système circulatoire humain.
Les premiers réseaux systoliques ont été implantés pour la première fois dans **Colossus**, un des premiers ordinateurs conçus pour casser les chiffrements allemands de Lorenz pendant la Seconde Guerre mondiale. Ils ont été réhabilités dans les années 1970 pour résoudre de nombreux calculs d'algèbre linéaire (produit matriciel, résolution de systèmes d'équations linéaires, décomposition LU, etc.).
Ils sont parfois classés comme architectures MISD selon la taxonomie de Flynn, mais cette classification est discutable car de nombreux arguments vont dans le sens que les réseaux systoliques ont des caractéristiques telles qu'il serait envisageable de les classer dans toutes les autres classes de Flynn (SISD, SIMD, MISD ou MIMD).
Plus récemment, dans le but d'accélérer les calculs en intelligence artificielle (principalement pour les réseaux de neurones), cette architecture a été privilégiée lors de la conception du processeur TPU (Tensor Processing Unit) de Google [4, 21, 12, 8].

Pour effectuer la multiplication de matrices :

- Les rangées de A entrent de façon synchrone à gauche du réseau (voir figure 8.12)
- les colonnes de B entrent de façon synchrone au sommet du réseau (voir figure 8.12)

Lorsqu'un nœud $P_{i,j}$ reçoit deux entrées $A(i,l)$ et $B(l,j)$ (en amont) :

- il calcule $C(i,j) = C(i,j) + A(i,l) \times B(l,j)$;
- il envoie $A(i,l)$ à sa droite (en aval) ;
- il envoie $B(l,j)$ vers le bas (en aval) ;

Après n étapes, $P_{i,j}$ contiendra les résultat $C_{i,j}$.

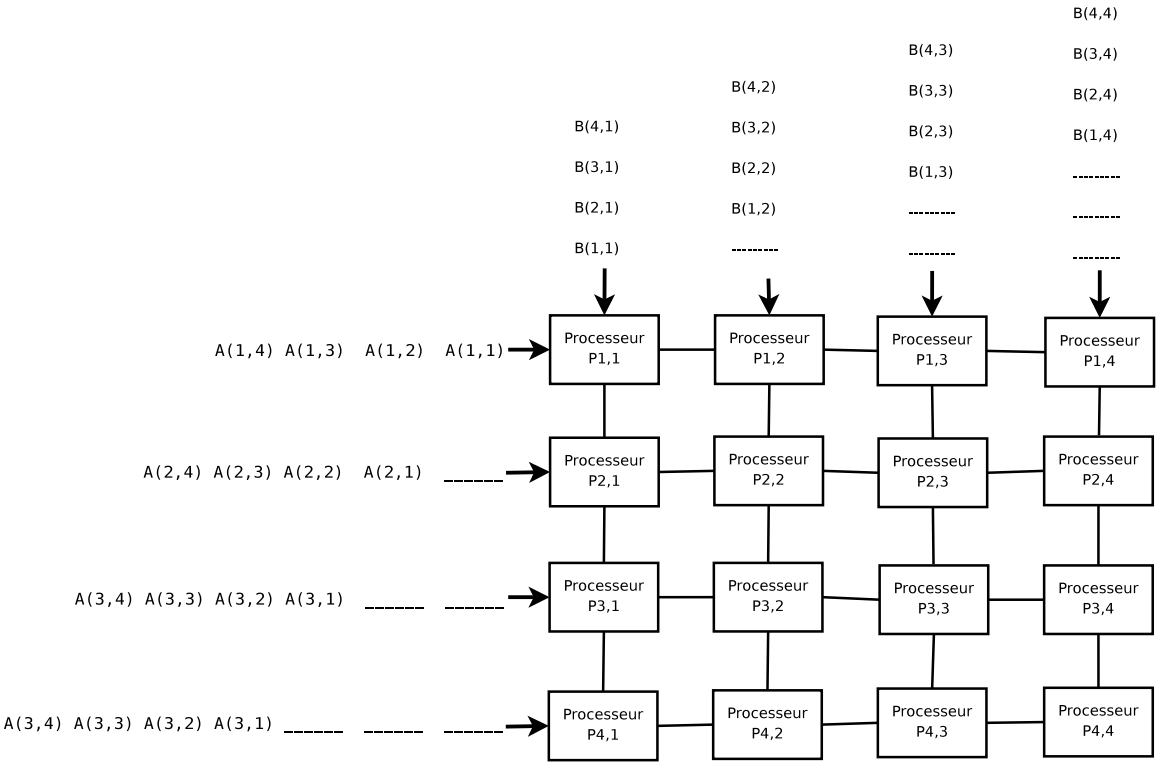


Figure 8.12 – Multiplication des matrices A et B sur un réseau systolique.

La figure 8.13 introduit un exemple dans lequel on procède à la multiplication de deux matrices carrées de taille 2×2, soit :

$$C = A \times B = \begin{pmatrix} 2 & 2 \\ 3 & 3 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 2 & 2 \end{pmatrix} = \begin{pmatrix} 6 & 8 \\ 9 & 12 \end{pmatrix}$$

Chaque nœud du réseau systolique effectue un produit scalaire. La figure 8.13 a présente l'initialisation du réseau. Par la suite, à chaque étape, les données avancent d'un nœud afin que chacun

puisse poursuivre son calcul. Après quatre étapes (figure 8.13 e), qui est fonction de la taille de la matrice, le réseau contient le résultat final du calcul.

Quelle est la complexité d'un calcul sur un réseau systolique ? Comme le réseau exige n étapes pour multiplier des matrices de taille $n \times n$, la complexité est de $O(n)$ avec n^2 processeurs (plutôt que $O(n^3)$). Avec plus de processeurs, la complexité restera la même car le calcul dépend toujours de la taille de la matrice. Si on a moins de n processeurs, la complexité augmentera.

8.3.4 Hypercube

Un hypercube consiste en $p = 2^d$ processeurs connectés en un cube booléen de dimension d. Chacun des processeurs p_i possède un identificateur i dont la représentation binaire est donnée par :

$$i_{d-1} i_{d-2} ... i_0$$

Par exemple, si nous avons un hypercube de dimension 3, les adresses des différents processeurs seront :

- $p_0 = 000$
- $p_1 = 001$
- $p_2 = 010$
- $p_3 = 011$
- $p_4 = 100$
- $p_5 = 101$
- $p_6 = 110$
- $p_7 = 111$

Ce type d'adressage est important car chaque processeur p_i est connecté aux processeurs $p_{i^{(j)}}$ où $i^{(j)} = i_{d-1}...\overline{i_j}...i_0$ où $\overline{i_j} = 1 - i_j$ pour $0 \leq j \leq d-1$. Cela implique que deux processeurs sont connectés si leurs indices n'ont qu'un seul bit de différence [1].

Un hypercube est une structure récursive, ce qui signifie que l'on crée un hypercube de dimension $d+1$ en connectant deux hypercubes de dimension d. Les adresses de chacun des deux cubes sont modifiées de façon à que le bit le plus significatif des adresses d'un des deux cubes, soit à 0 et celui de l'autre à 1.

La figure 8.14 comprend trois hypercubes de dimensions 1, 2 et 3. On remarque que pour chacun d'eux, les voisins directs d'un nœud sont exactement ceux dont l'adresse binaire varie d'un seul bit. La figure 8.15 présente un hypercube de dimension 4 formé à partir de deux hypercubes de dimension 3.

La caractéristiques d'un hypercube sont :

- Diamètre $= di = log_2(p)$;

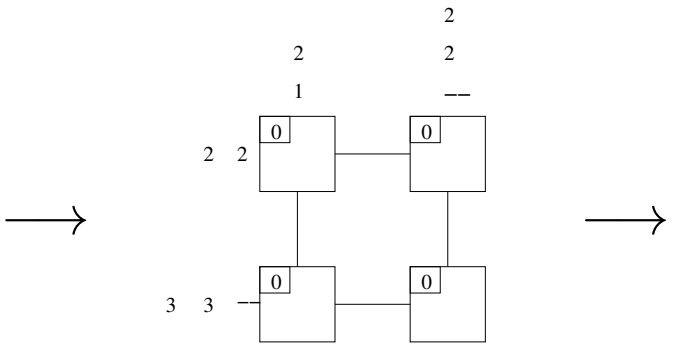
  La distance entre les deux nœuds (source et destination) les plus éloignés est le nombre de bits de l'adresse source à complémenter afin de la transformer en l'adresse de destination (donc tous les bits). Puisqu'un hypercube de taille d contient $p = 2^d$ nœuds, le nombre de bits de l'adresse est $log_2(p)$.

- Degré maximum $= dg = log_2(p)$;

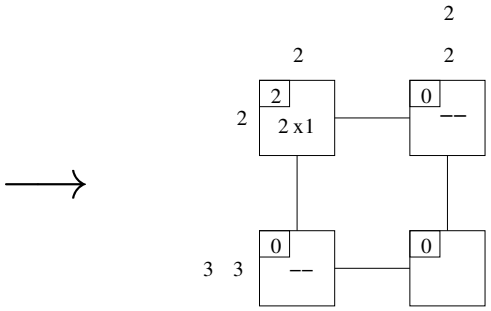
  Chaque nœud est connecté à tous les autres nœuds dont l'adresse varie d'un seul bit. Chaque nœud possède donc une connexion avec d voisins, donc $log_2(p)$;.

L'hypercube possède aussi d'autres caractéristiques intéressantes : sa régularité, son petit diamètre, sa rapidité en calcul et différentes autres propriétés reliées à la théorie des graphes.

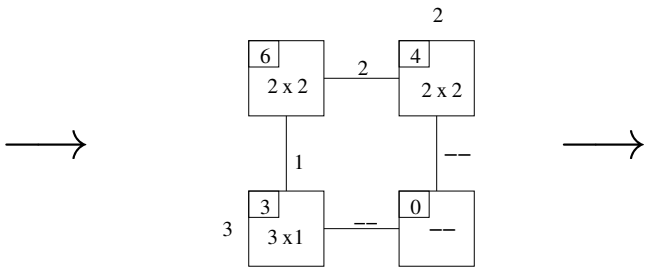
1. Dans les programmes à venir, la notation $i \sim j$ sera utilisée pour signifier le complément sur la position j.



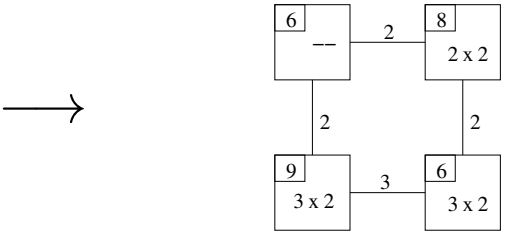
(a) $A \times B$: étape 0 : initialisation.



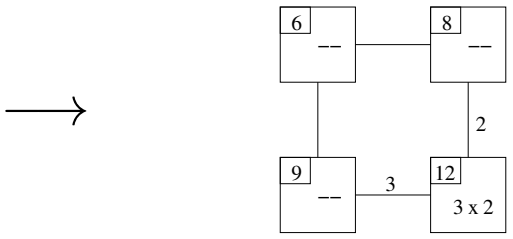
(b) $A \times B$: étape 1.



(c) $A \times B$: étape 2



(d) $A \times B$: étape 3.



(e) $A \times B$: étape 4.

Figure 8.13 – Multiplication des matrices A et B sur un réseau systolique.

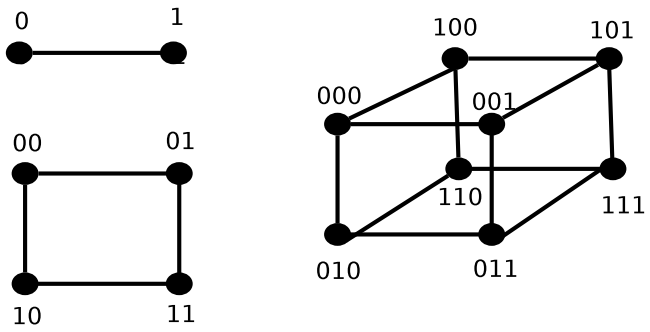


Figure 8.14 – Trois hypercubes de dimension respective 1, 2 et 3.

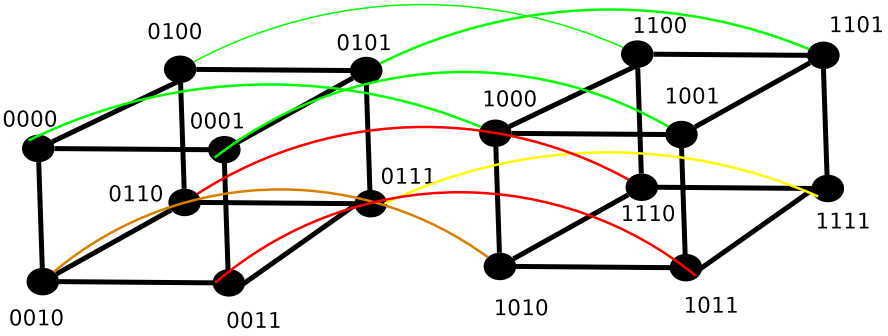


Figure 8.15 – Un hypercube de dimension 4.

Pour bien illustrer le fonctionnement d'un hypercube, nous effectuerons la somme de tous les éléments d'un tableau sur un hypercube. Soit A un tableau contenant n éléments et un environnement offrant n processeurs ($n = 2^d$). À calculer

$$S = \sum_{i=0}^{n-1}(A_i).$$

À la fin , le résultat S sera disponible sur le processeur p_0.
Voici les différentes étapes du calcul :

1. Initialisation ;
   Lors de l'initialisation, on emmagasine chaque A_i sur un processeur p_i ;
2. Lors du calcul, on effectue d itérations
   (a) **itération 1** : calcul de la somme des paires sur le cube dD ;
   Les sommes sont calculées et emmagasinées dans le cube $(d-1)D$ dont le bit le plus significatif est à 0.
   (b) **itération 2** : calcul de la somme de paires sur le cube $(d-1)D$;
   Les sommes sont calculées et emmagasinées dans le cube $(d-2)D$ dont les deux bits les

plus significatifs sont à 0.

(c) ...

(d) **itération** d : calcul de la somme finale.
La somme est calculée et emmagasinée dans le nœud p_0 dont tous les bits sont à 0.

Le programme 8.5 implante un algorithme effectuant la somme sur un hypercube. Chaque processeur exécute le même algorithme. La boucle s'exécute d fois (i.e. $log_2(p)$), permettant ainsi aux données de circuler à travers tout l'hypercube. Ainsi, pour un hypercube de dimension 3 :

- à l'étape $l = 2$: les processeurs de p_0 à p_3 calculent ;
- à l'étape $l = 1$: les processeurs de p_0 à p_1 calculent ;
- à l'étape $l = 0$: seulement le processeur de p_0 calcule.

L'énoncé de sélection permet de choisir les nœuds qui feront un calcul. Les notations suivantes sont utilisées :

- $A(i \sim l)$ indique $A(i^{(l)})$;
  Ainsi, `i~l` signifie le complément du bit en position l de l'identificateur i (ex. : `000~1 = 010`)
- `2^l` $\rightarrow 2^{\texttt{l}}$ (2 à la puissance `l`)

L'opération de la ligne 10 se fait en deux étapes. À la première, le processeur p_i copie $A(i^{(l)})$ du processeur $p(i^{(l)})$ vers p_i. À la seconde étape, p_i effectue l'addition et emmagasine le résultat dans la mémoire de p_i.

```
1  /*********************************************************
2  ---------      Code pour le processus Pi    ---------------
3  Entrée : A(i) sur chaque pi
4  Sortie : S : somme des A(i) sur p0
5
6  *********************************************************/
7  begin
8     For l = d-1 to 0 do
9         if (0<= i <= (2^l)-1) do
10              A(i) := A(i) + A(i~l))
11 end
```

Programme 8.5 – Exemple de la somme de n valeurs.

Les figures 8.16 à 8.19 montrent le fonctionnement d'une somme sur un hypercube de dimension 3. La figure 8.16 indique la configuration initiale, alors que les figures suivantes représentent chacune des itérations.

La complexité de l'algorithme effectuant la somme est de $O(\log_2 n)$, i.e. la dimension du cube.

Un second exemple permettant d'illustrer le fonctionnement d'un hypercube, est celui de la diffusion d'information. Le programme 8.6 implante la diffusion d'un valeur D sur l'hypercube. Les notations sont telles que définies à l'exemple de sommation.

Les figures 8.20 à 8.23 illustrent la diffusion d'une valeur A sur un hypercube de dimension 3 : la figure 8.20 est celle de l'initialisation de l'hypercube au début de la diffusion alors que les figures 8.21 à 8.23 affichent chacune des étapes de la diffusion de la valeur.

La complexité de cet algorithme est la même que celle de la sommation soit $O(\log_2 p)$ (la dimension de l'hypercube).

Les algorithmes de sommation et de diffusion sont catégorisés sous l'appellation des algorithmes

```
d = 3
i~l = complément du bit en position l

for l = d−1 to 0 do
  if ( 0 <= i <= (2^l)−1 ) do
    A[i] = a[i] + A[i~l]
```

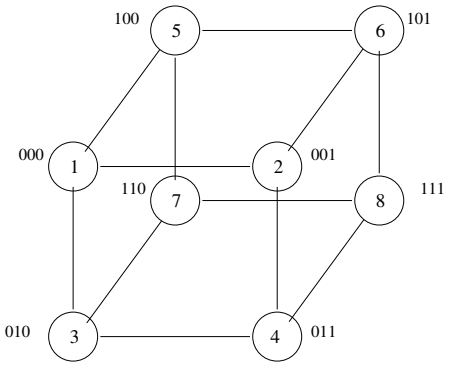


Figure 8.16 – Exemple d'une sommation sur un hypercube - Initialisation.

```
d = 3
l = 2
i~2 = complément bit position 2

for l = 2 to 0 do
  if (0 <= i <= 3) do
    A[i] = A[i] + A[i~2]

A[0] = A[0] + A[4]
A[1] = A[1] + A[5]
A[2] = A[2] + A[6]
A[3] = A[3] + A[7]
```

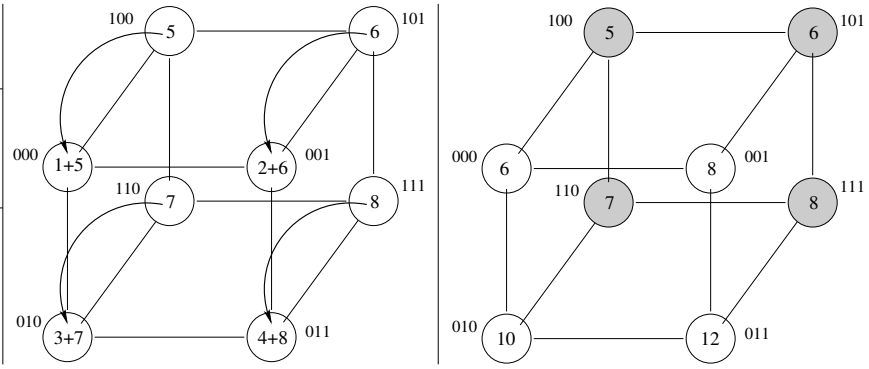


Figure 8.17 – Exemple d'une sommation sur un hypercube - première itération

```
d = 3
l = 1
i~1 = complément bit position 1

for l = 2 to 0 do
  if (0 <= i <= 1) do
    A[i] = A[i] + A[i~1]

A[0] = A[0] + A[2]
A[1] = A[1] + A[3]
```

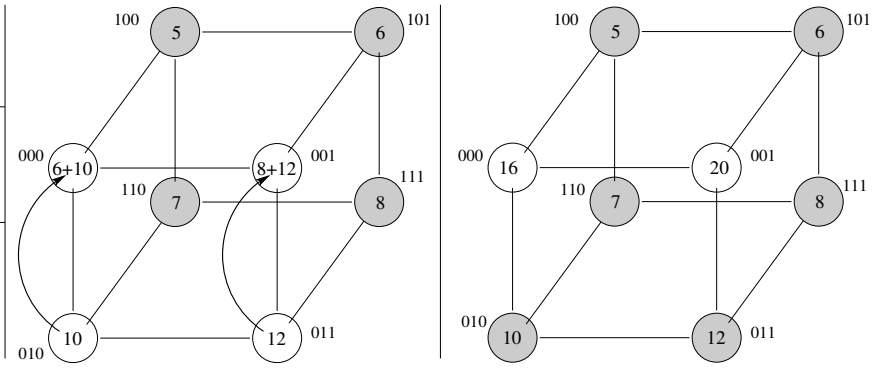


Figure 8.18 – Exemple d'une sommation sur un hypercube - seconde itération

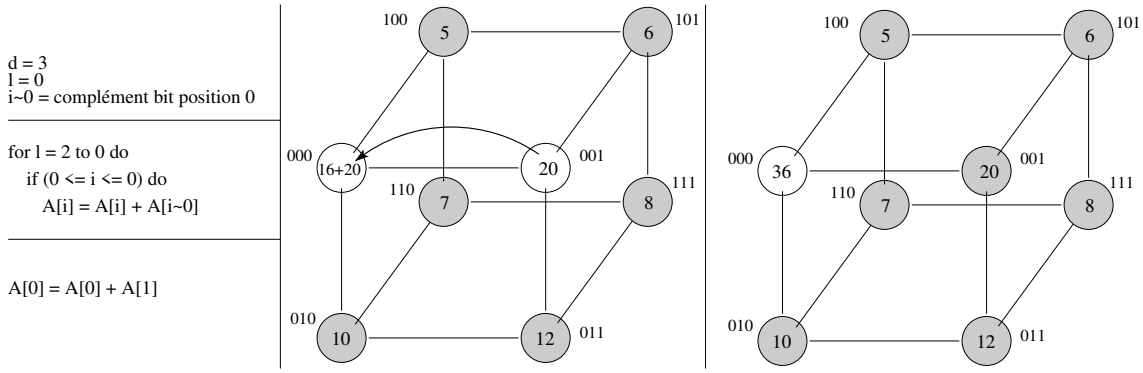


Figure 8.19 – Exemple d'une sommation sur un hypercube - dernière itération.

```
1  /*********************************************************
2  ---------     Code pour le processus Pi    ---------------
3  Entrée : p0 contient x dans D(0)
4  Sortie : pi contient x : pour tout i
5
6  *********************************************************/
7  begin
8     for l = 0 to d-1 do
9         if (0<= i <= (2^l)-1) do
10             D(i~l) := D(i)
11 end
```

Programme 8.6 – Exemple de la diffusion d'information.

«**normaux**». En ce qui concerne les hypercubes, cela signifie qu'ils emploient une dimension de l'hypercube à chaque unité de temps de telle façon à ce que les dimensions consécutives soient utilisées en séquence dans le temps. En fait, ces deux algorithmes appartiennent aussi à une autre catégories, celles des algorithmes «**complètement normaux**». Ces derniers sont des algorithmes normaux mais répondant à une contrainte additionnelle, soit celle que toutes les dimensions de l'hypercube agissent en séquence dans le temps.

Comme dernier exemple, entreprenons la multiplication de matrices sur un hypercube. Soit A, B et C des matrices carrées de taille $n \times n$. Notre but : calculer $C = A \times B$ sur un hypercube fournissant $p = n^3$ processeurs.

Comme déjà indiqué, pour faciliter la présentation, nous privilégions une taille de matrice étant une puissance de 2. Ainsi, si $n = 2^q$ alors $p = 2^{3q}$ processeurs. Conséquemment, nous pouvons identifier chaque processeur à partir d'un indice à trois dimensions (l, i, j) tel que $p_{l,i,j}$ représente p_r où $r = ln^2 + in + j$. Selon cette désignation, les q bits les plus significatifs sont ceux de l'indice l et les q bits les moins significatifs ceux de l'indice j. En fixant deux des trois valeurs, nous obtenons un hypercube de dimension q.

Initialement, afin de démarrer la multiplication :

- la matrice A est emmagasinée dans le cube formé par les processeurs $p_{l,i,0} : 0 \leq i, l \leq n-1$ tel que $A(i, l)$ est sur le processeur $p_{l,i,0}$;

```
d = 3
i~l = complément du bit en position l

for l = 0 to d−1 do
   if (0 <= i <= (2^l)−1) do
      D[i~l] = D[i]
```

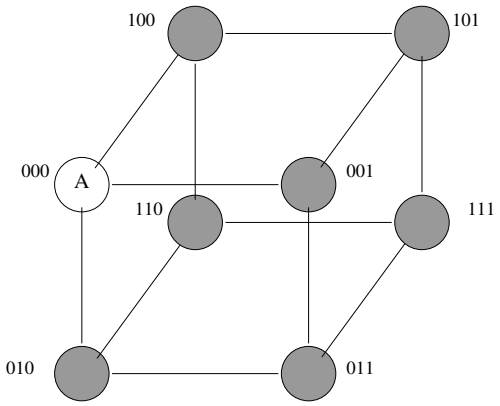


Figure 8.20 – Exemple de diffusion sur un hypercube - Initialisation.

```
d = 3
l = 0
i~0 = complément bit position 0

for l = 0 to d−1 do
   if (0 <= i <= 0) do
      D[i~0] = D[i]

D[1] = D[0]
```

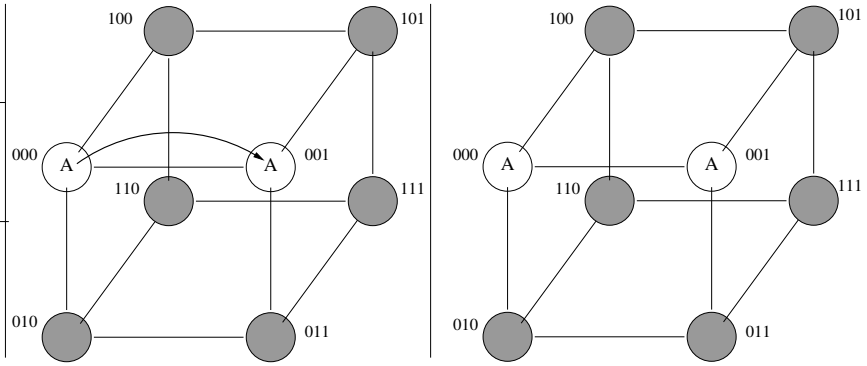


Figure 8.21 – Exemple de diffusion sur un hypercube - première itération

```
d = 3
l = 1
i~1 = complément bit position 1

for l = 0 to 2 do
   if (0 <= i <= 1) do
      D[i~1] = D[i]

D[2] = D[0]
D[3] = D[1]
```

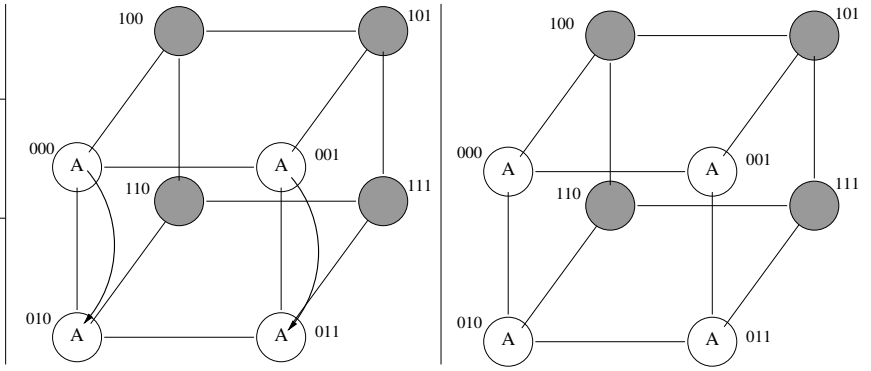


Figure 8.22 – Exemple de diffusion sur un hypercube - seconde itération

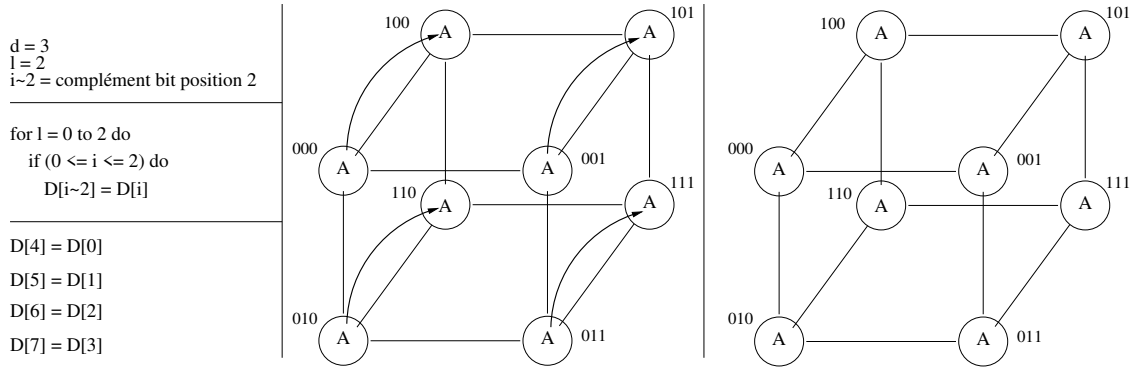


Figure 8.23 – Exemple de diffusion sur un hypercube - dernière itération

- la matrice B est emmagasinée dans le cube formé par les processeurs $p_{l,0,j} : 0 \leq j, l \leq n-1$ tel que $B(l,j)$ est sur le processeur $p_{l,0,j}$.

Le but de l'algorithme est de calculer chaque $C(i,j) := \sum_{l=0}^{n-1} A(i,l) \times B(l,j)$, $0 \leq i, j \leq n-1$. Cela se fait en trois étapes :

1. **La diffusion des données ;**
   Lors de cette étape, les données sont diffusées sur l'hypercube de façon à ce que chaque $p_{l,i,j}$ contienne $A(i,l)$ et $B(l,j)$.
2. **Le calcul d'un produit ;**
   Chaque processeur $p_{l,i,j}$ calcule $C'(l,i,j) = A(i,l) \times B(l,j),\ \forall i,j,l \mid 0 \leq i,j,l \leq n-1$
3. **La sommation.**
   Chaque processeur $p_{l,i,j}, 0 \leq l \leq n-1$ calcule $C(i,j) = \sum_{l=0}^{n-1} C'(l,i,j)$.

Examinons la complexité de l'algorithme de multiplication :

1. **Étape 1** : la diffusion ;
   Cette étape consiste en :
   (a) la diffusion de $A(i,l),\ \forall\ i$ et l, par $p(l,i,0)$ à tous les processeurs $p(l,i,j)$ $(0 \leq j \leq n-1)$ de son «sous-cube» contenant n processeurs. La complexité d'une opération de diffusion est de $O(log_2 n)$.
   (b) la diffusion de $B(l,j),\ \forall\ l$ et j, par $p(l,0,j)$ à tous les processeurs $p(l,i,j)$ $(0 \leq i \leq n-1)$ de son «sous-cube» contenant n processeurs. La complexité d'une opération de diffusion est de $O(log_2 n)$.
   À la fin de cette étape, chaque processeur $p(l,i,j)$ détient les deux entrées $A(i,l)$ et $B(l,j)$.
2. **Étape 2** : la multiplication ;
   À cette étape, chaque processeur $p_{l,i,j}$ effectue une seule multiplication, donc la complexité de cette opération est de $O(1)$. À la fin de cette étape, chaque processeur $p(l,i,j)$ détient un résultat $C'(l,i,j)$.
3. **Étape 3** : la sommation ;

À cette étape, on calcule n^2 sommes $C(i,j)$. Les termes $C'(l,i,j)$ résident dans un «sous-cube» $p(l,i,j)$ $(0 \leq l \leq n-1)$. À la fin, $p(0,i,j)$ contiendra les n^2 résultats. Chaque somme (comme nous l'avons déjà déterminer dans notre exemple précédant) sur un hypercube est d'une complexité de l'ordre de $O(log_2 n)$.

4. **Complexité totale.**
   La complexité totale du calcul est alors de $O(log_2 n)$.

Réalisme associé à l'hypercube

Procéder à un calcul quelconque sur un hypercube (du moins basé sur cette analyse) est peu réaliste. En effet, pour implanter une telle solution, le nombre de processeurs requis est beaucoup trop élevé. Cela est envisageable pour des matrices de petites dimensions, par exemple, pour une matrice de taille 2×2 car alors le calcul nécessite neuf processeurs ce qui est concevable.
Toutefois pour une matrice de taille 100×100, il faudrait un million de processeurs. Ce qui est invraisemblable considérant la technologie actuelle (2022).

8.3.5 Évaluation

Nous venons de présenter trois modèles tous aptes à évaluer la performance d'algorithmes parallèles. Mais une question s'impose : y en a-t-il un meilleur que les autres ?

Passons donc en revue les avantages et inconvénients de chacun d'eux.

Graphes orientés acycliques (DAG)

Le modèle basé sur les graphes orientés acycliques (DAG) est simple, mais :

- il s'applique seulement à une classe spécialisée d'algorithmes (algorithmes réguliers) ;
- il ne fait ressortir qu'une information partielle en ce sens qu'il :
  - ne modélise pas la planification et l'allocation des ressources (servant à confirmer les mesures de performance) ;
  - ne contient aucun mécanisme pour évaluer les coûts en communications.

Réseau

De façon générale, les différents modèles réseau présentés (linéaire, maillage 2D et hypercube) sont tous plus appropriés que le modèle DAG pour représenter le calcul et les communications.

Toutefois la description et l'analyse des algorithmes s'avèrent plus complexes qu'avec les modèles PRAM ou DAG. De plus, les algorithmes dépendent de la topologie. En effet, différentes topologies (maillage ou hypercube) impliquent différents algorithmes pour résoudre le même problème.

PRAM

Selon plusieurs, le modèle PRAM est le plus puissant des trois car :

- Il existe plusieurs techniques bien développées pour modéliser les types variés de problèmes.
- Il élimine les détails algorithmiques reliés à la synchronisation et à la communication. On se concentre alors uniquement sur la résolution du problème lui-même.
- Il capture plusieurs paramètres importants du calcul parallèle, soit :
  - la planification ;

    Il est apte à déterminer le moment (unité de temps) où chacune des opérations doit être exécutée.
  - l'allocation.

    Il permet déterminer quand et comment les processeurs sont alloués aux différentes tâches.
- Il est robuste.

  La robustesse d'un modèle se définit de maintes façons mais celle qui semble la plus pertinente dans ce cas [5, 13] est la capacité d'un algorithme développé sur un modèle particulier de se projeter ou de s'adapter à un autre modèle tout en conservant son efficacité en terme de complexité algorithmique.

  Ainsi, plusieurs algorithmes du modèle réseau dérivent des algorithmes développés sur PRAM. De plus, les algorithmes PRAM se projettent sur certains types de réseau.
- Il est possible d'incorporer la synchronisation et la communication au modèle.

8.4 Performance des algorithmes parallèles

Maintenant que nous connaissons des modèles pour représenter des algorithmes parallèles, nous sommes en mesure d'introduire d'autres mesures de performance.

8.4.1 Le coût

Le coût est une mesure dérivée du temps d'exécution et du nombre de processeurs total utilisés pour le calcul.

Définition : Le coût

Soit Q un problème à résoudre sur PRAM qui s'exécute en temps $T(n)$ sur $P(n)$ processeurs pour des données de taille n.
Le **coût** de l'algorithme parallèle correspond alors à $C(n) = T(n) \times P(n)$.

La calcul du coût consiste en fait à convertir l'exécution parallèle en son équivalent séquentiel. Dans notre cas, plutôt que de faire appel à $P(n)$ processeurs, nous simulerons ces P_n exécutions parallèles de Q en les traitant chacune séquentiellement sur un seul processeur. Ainsi, ce processeur exécutera les $P(n)$ étapes en une temps proportionnel à $O(P(n))$, chacune des étapes prenant un temps d'ordre $T(n)$. Ainsi, on conclu que l'algorithme parallèle se convertit en algorithme séquentiel et que la performance en temps de ce dernier est de $O(C(n))$.

Il est possible de généraliser cet argument pour chaque nombre de processeur $p \leq P(n)$. Ainsi, pour chaque étape d'une durée de $T(n)$, il est envisageable d'utiliser p processeurs pour simuler les

$P(n)$ processeurs originaux en $O(P(n)/p)$ étapes. Ainsi. à l'étape :

1. on simule p des $P(n)$ processeurs en faisant les étapes $1, 2, 3, ..., p$;
2. on simule les étapes $p+1, p+2, ..., 2p$

Cette simulation prend donc $O(T(n)P(n)/p)$ unités de temps. Quand $p > P(n)$, on atteint $T(n)$ avec seulement $P(n)$ processeurs.

Suite à cela on arrive à la conclusion que les quatre méthodes suivantes pour mesurer la performance sont asymptotiquement équivalentes :

- $P(n)$ processeurs et temps d'exécution $= T(n)$;
- Si $p = P(n)$, coût $= C(n) = P(n) \times T_p(n)$ et temps d'exécution $= T(n)$;
- $\forall\ p \leq P(n)$ processeurs, temps d'exécution $= O(T(n)P(n)/p)$
- $\forall\ p$ processeurs, temps d'exécution $= O(C(n)/p + T(n))$

La dernière expression fixe une borne minimale au temps d'exécution, soit $T(n)$. En effet, sans cette limite, si p devient très grand, le temps d'exécution pourrait tendre vers 0, ce qui ne fait aucun sens. En fait cela signifie que l'ajout de processeurs supplémentaires au dessus de $P(n)$ n'apporte rien de significatif au niveau performance.

Pour notre exemple d'algorithme qui calcule la somme de n éléments sur PRAM, cela signifie que les énoncés suivants sont équivalents :

- $n = P(n)$ processeurs et un temps d'exécution de $O(log_2 n)$;
- Si $p = P(n) = n$, coût $= C(n) = O(n\ log_2 n)$ et temps de $O(log_2 n)$
- $\forall\ p \leq n$, temps d'exécution $= O(\frac{n\ log_2 n}{p})$
- $\forall\ p$, temps d'exécution $= O(\frac{n\ log_2 n}{p} + log_2 n)$

De même, pour l'algorithme qui calcule la multiplication de matrices sur PRAM, cela signifie que les énoncés suivants sont équivalents :

- $n^3 = P(n))$ processeurs et $O(log_2 n)$ unités de temps
- Si $p = P(n) = n^3$, coût $= C(n) = O(n^3\ log_2 n)$ et temps d'exécution $= O(log_2 n)$
- $\forall\ p \leq n^3$, temps d'exécution $= O(\frac{n^3\ log_2 n}{p})$
- $\forall\ p$, temps d'exécution $= O(\frac{n^3\ log_2 n}{p} + log_2 n)$

8.4.2 Le travail (worktime paradigm)

Les différentes mesures de performance présentées servent à évaluer principalement des mesures dérivées du temps d'exécution par rapport au nombre de processeurs (temps d'exécution, accélération, efficacité, coût). En mode séquentiel, ces mesures en temps sont directement dépendantes du nombre d'opérations à traiter. En ce qui concerne les algorithmes parallèles, la dépendance entre le nombre d'opérations et le temps requis pour les traiter est beaucoup moins évident.

Le modèle de «travail», noté WT (pour «WorkTime paradigm») cherche à estimer cette quantité d'opérations exécutées par un algorithme parallèle. Il permet de décrire un algorithme en deux niveaux : le niveau présentation et le niveau planification.

Le premier niveau : niveau présentation (niveau le plus abstrait)

Ce premier niveau supprime les détails spécifiques de l'algorithme. Il permet de spécifier un algorithme en termes d'unités de temps, chacune des unités incluant un nombre quelconque d'opérations concurrentes. Ce premier niveau établit la quantité de travail, $W(n)$, fait par un algorithme parallèle en terme du nombre total d'opérations nécessaires pour traiter n données en entrée.

Pour faciliter la représentation des algorithmes à ce niveau, la notation suivante :

for $l \leq i \leq u$ **pardo** opérations

définie que les «*opérations*» de l'énoncé **pardo** correspondant aux valeurs de i incluses entre l et u seront toutes exécutées en parallèle. Par exemple, le programme 8.7 implante la somme de n valeurs en utilisant cette notation. À noter que l'opérateur «^» signifie «exposant». Dans ce programme, la ligne 7 (étape 1) sert à initialiser le tableau de travail et se fait toujours en parallèle. La ligne 8 (étape 2) démarre la sommation. Celle-ci se fait en $log_2 n$ itérations séquentielles. À chaque itération, les processeurs p_i choisis (ceux dont l'identificateur se situe entre $1 \leq i \leq n/2^h$) réalisent la somme.

```
1  /*********************************************************
2  ---------     Code pour le processus Pi    ---------------
3  Entree: A : un tableau contenant n nombres
4  Sortie: S : la somme des nombres
5  *********************************************************/
6  begin
7  for 1 <= i <= n pardo B(i) := A(i)    // Étape 1
8  for h:=1 to log n  do                 // Étape 2
9      for 1 <= i <= n/2^h  pardo
10         B(i) := B(2i-1) + B(2i)
11 S := B(i)
```

Programme 8.7 – Exemple de la somme de n valeurs.

Il est important de constater qu'à ce niveau, l'algorithme ne donne aucune indication sur le nombre de processeurs ni comment les opérations sont allouées aux processeurs. Il n'est décrit qu'en terme d'unités de temps, chacune d'elles contenant un nombre variable d'opérations. Ainsi, l'algorithme se divise en trois étapes :

- Étape 1 : assignation ;

  Lors de cette étape, l'algorithme traite en parallèle toutes les assignations en une seule unité de temps. Il exécute donc n opérations dans la première unité de temps (ligne 7).

- Étape 2 : sommation ;

  Lors de cette seconde étape (débutant à l'unité de temps 2), l'algorithme effectue $log_2 n$ itérations, chacune occupant une unité de temps. À l'unité de temps $j-1$ (l'itération h correspond à l'unité de temps $j-1$), il exécute en parallèle $n/2^{j-1}$ opérations (pour $2 \leq j \leq log_2 n + 1$).

- Étape 3 : assignation finale

  Lors de cette étape (ou unité de temps), l'algorithme traite une seule opération.

Le nombre d'unités de temps total utilisé par l'algorithme est de $log_2 n+2$. Son temps d'exécution est donc

$$T_p(n) = O(\log_2 n).$$

Le travail (ou nombre d'opérations) effectué par l'algorithme est de :

$$W(n) = n + \sum_{j:=1}^{\log_2 n} (n/2^j) + 1 = O(n).$$

Précisons quelques détails par rapport à cette méthode de calcul. Le travail est défini par la somme des opérations exécutées à chacune des étapes, soit n à la première étape, $\sum_{j:=1}^{\log_2 n}(n/2^j)$ à la seconde et une seule à la dernière étape. En combien d'opérations la sommation de l'étape 2 est-elle effectuée ?

Pour l'évaluer, procédons ainsi :

$$\begin{aligned}
\sum_{j=1}^{\log_2 n} (n/2^j) &= \frac{n}{2^1} + \frac{n}{2^2} + \frac{n}{2^3} + ... + \frac{n}{2^{log_2 n}} \\
&= \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + ... + \frac{n}{2^{log_2 n}} \\
&= n\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^{log_2 n}}\right) \quad \text{(série géométrique convergente)} \\
&\longrightarrow n \times 1 = n \text{ (lorsque n est suffisamment grand, voir note sur la série géométrique)}
\end{aligned}$$

Série géométrique convergente de raison 1/2

$$\lim_{k\to\infty} \sum_{j=1}^{k} (1/2^j) = \lim_{k\to\infty} \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + ... + \frac{1}{2^k}\right) = 1$$

Le second niveau : niveau planification

Ce second niveau fourni les détails de planification du calcul. Nous ne nous attarderons pas sur ce niveau et vous référons à JàJà [9] pour plus d'informations.

8.4.3 Travail vs Coût

Le travail et le coût sont deux mesures proches mais tout de même assez différentes. Ainsi :

- le coût $C(n)$ mesure le temps d'exécution de l'algorithme relativement au nombre de processeurs [2] ;
- le travail $W(n)$ mesure le nombre total d'opérations et n'a aucun lien avec le nombre de processeurs.

2. Rappelons que le coût d'un algorithme est $C(n) = T_p(n) \times p$ où $T_p(n)$ est son temps d'exécution sur p processeurs.

Exemple : somme de n valeurs

Soit un algorithme qui évalue la somme de n nombres pour lequel les mesures de performance sont :

- $W(n) = O(n)$
- $T_p(n) = O(log_2 n)$
- $C(n) = O(p \cdot log_2 n) = O(n \cdot\ log_2 n)$ si $p = n$

Le travail ($O(n)$ opérations) est ici différent du coût car l'algorithme n'utilise pas efficacement les processeurs. Ainsi, dans un environnement comprenant p processeurs, au plus $p/2$ sont actifs à la première itération, au plus $p/4$ à l'itération suivante, etc.

8.4.4 Notion d'optimalité

Soit un problème Q dont la complexité séquentielle optimale théorique est de $T^*(n)$. En parallélisme, on définit deux notions d'optimalité. Un algorithme parallèle est optimal si :

1. le **travail** $W(n)$ satisfait $W(n) = \Theta(T^*(n))$;
   Un algorithme parallèle est optimal si le nombre total d'opérations qu'il utilise est asymptotiquement le même que celui de son algorithme séquentiel optimal, et ceci peu importe le temps d'exécution.
2. **l'accélération** $S_p(n) = \Theta(p)$
   Un algorithme parallèle obtient une accélération optimale s'il est possible de prouver que son temps d'exécution $T(n)$ ne peut être amélioré, et ce, peu importe l'algorithme.

Exemple : somme de n valeurs

Reprenons notre exemple d'algorithme effectuant la somme de n valeurs afin de déterminer s'il est optimal. Pour cette algorithme :

- Le temps d'exécution $T_p(n) = O(log_2 n)$;
  L'algorithme obtient une accélération optimale lorsque le nombre de processeurs
  $$p = O(n/log_2 n).$$
- Le travail $W(n) = O(n)$
  Comme $T^*(n) = n$, l'algorithme est optimal (pour le travail).

8.5 Complexité des communications

Lorsque l'on évalue un algorithme parallèle, il ne suffit pas de déterminer son temps d'exécution, il faut également déterminer sa complexité en terme de coûts de communication.

Sur le modèle PRAM, le coût en communication est le pire cas de trafic entre la mémoire commune et la mémoire privée d'un processeur quelconque.

Soit A un algorithme optimal adapté avec succès pour s'exécuter sur p processeurs. Même s'il est optimal en terme de temps d'exécution, il ne l'est pas nécessairement en terme de complexité des communications.

Par exemple, le programme 8.8 implante une multiplication de matrices. Une analyse rapide de cet algorithme permet de déterminer que :

- son temps d'exécution est de $O(log_2 n)$ pour $O(n^3)$ opérations ;
- son temps d'exécution est de $O(n^2)$ avec n processeurs ;
- sa complexité en communication est de $O(n^2)$ avec n processeurs car :
  1. (ligne 7-8) Chaque processeur lit une ligne de la matrice A et la matrice B au complet : complexité de communication = $O(n^2)$ valeurs ;
  2. (ligne 10-11-12) Chaque processeur effectue un produit scalaire et emmagasine le résultat en mémoire : complexité de communication = $O(n^2)$;
  3. (ligne 14-15) Chaque processeur emmagasine une valeur : complexité de communication = $O(n)$.

```
1  /*****************************************
2  Entree: A et B : matrices n x n
3          n : entier = 2^k
4  Sortie: C = A x B
5  *****************************************/
6  begin
7  for  1 <= i,j,l <n pardo
8       C'(i,j,1) := A(i,l) x B(l,j)
9
10 for h=1 to log n  do
11      for (1 <= i,j <= n , 1<= l <= n/2^h) pardo
12          C'(i,j,k) :=  C'(i,j,2l-1) + C'(i,j,2l)
13
14 for 1 <= i,j <n  pardo
15       C(i,j) :=  C'(i,j,1)
16 end
```

Programme 8.8 – Multiplication de matrice.

Il est possible d'effectuer le calcul différemment pour diminuer le coût en communication sans nuire au temps de calcul. Dans notre précédant algorithme, chacun des processeurs effectuait son calcul à partir d'une ligne de la matrice A et de la matrice B en entier. Cette fois, on choisit un autre partitionnement. Ainsi, divisons plutôt les matrices en sous-blocs de taille égale comme l'illustre la figure 8.24. Sous l'hypothèse que les deux matrices à multiplier de taille $n \times n$ sont telles que $\alpha = \sqrt[3]{n}$ est un entier (comme celles de la figure 8.24 b), on partitionne les deux matrices à multiplier en $\sqrt[3]{n} \times \sqrt[3]{n}$ blocs (sous-matrices) de taille $n^{2/3} \times n^{2/3}$. Pour effectuer la multiplication, on distribue ces blocs par paires (au total n paires de blocs) aux différents processeurs (voir figure 8.25).

Pour cet algorithme spécifique, on obtient les mesures de performance suivantes :

- temps d'exécution, $T_p(n) = O(n^2)$ sur n processeurs ;
- coût en communication, $C = O(n^{4/3} \cdot log_2 n)$.
  À la première étape, chaque processeur doit lire une paire de blocs de taille $n^{2/3} \times n^{2/3}$ (au total $n^{4/3}$ valeurs) pour effectuer la multiplication initiale. La seconde étape consiste en une sommation parallèle qui prend $log_2(n)$ étapes. À chacune de ces étapes, les processeurs doivent lire deux matrices de taille $n^{2/3} \times n^{2/3}$ (au total $n^{4/3}$ valeurs). Le coût total de communication

$$\left(\begin{array}{cc|cc} x & x & x & x \\ x & x & x & x \\ \hline x & x & x & x \\ x & x & x & x \end{array}\right)$$

(a) 4 sous-matrices 2×2

$$\left(\begin{array}{cccc|cccc} x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ \hline x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \end{array}\right)$$

(b) 4 sous-matrices 4×4

Figure 8.24 – Partitionnement de matrices 4×4 et 8×8

$$\left(\begin{array}{cccc|cccc} x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ \hline x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \end{array}\right) \times \left(\begin{array}{cccc|cccc} x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ \hline x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \\ x & x & x & x & x & x & x & x \end{array}\right)$$

Figure 8.25 – Multiplication de matrices par blocs de taille $n^{2/3} \times n^{2/3}$

est donc $O(n^{4/3} \cdot log_2 n)$.

Cet algorithme se révèle aussi efficace que le premier en terme de temps de calcul mais fournit une meilleure performance en terme de coûts de communication. Il serait même possible selon JàJà [9] de réduire davantage les coûts de communication à $O(n^{4/3})$.

8.6 Exemples d'algorithmes

Voici quelques exemples d'algorithmes parallèles associés à leur performance respective :

- Recherche dans une liste triée : $O(\frac{log(n+1)}{log(p+1)})$ (CREW PRAM) ;
- Tris parallèles :
  - tri bulle : $O(n)$ sur $O(n)$ processeurs (PRAM EREW) ;
  - tri insertion : $O(n^2)$;
  - heap sort : $O(n \cdot log_2 n)$ (pire cas) ;
  - tri fusion : $O(log_2^2 n)$ (PRAM CREW) – d'autres variations font mieux ;
  - tri bitonique : $O(log_2^2 n)$ (PRAM EREW) ;

8.7 Conclusion

Dans ce chapitre, différents modèles servant à évaluer la performance d'algorithmes parallèles (PRAM, DAG et réseau) ont été présentés. Les mesures suivantes permettant de les comparer ont également été abordées :

- Temps d'exécution séquentiel optimal : $T^*(n)$;
- Temps d'exécution sur p processeurs : $T_p(n)$;
- Temps d'exécution sur un processeur : $T_1(n)$;
- Accélération absolue : $S_p(n)$;
- Accélération relative : $RS_p(n)$;
- Efficacité : $E_p(n)$;
- Coût : $C(n)$;
- Travail : $W(n)$.

Sur les systèmes parallèles, une autre mesure de performance s'ajoute (entièrement absente sur les systèmes centralisés), soit le coût en terme de complexité de communication.

En fait, nous avons seulement effleuré le domaine de l'algorithmie parallèle et cela de manière très informelle. Ce sujet est abordé beaucoup plus en profondeur et plus formellement dans JàJà [9], Alsuwaiyel [1], Casanova et al.[3], Kumar et al. [11], Grama et al. [7] et plusieurs autres.

Bibliographie

[1] Muhammad Hamad Alsuwaiyel : *Parallel Algorithms*. Lecture Notes Series on Computing : Volume 16. World Scientific, 2022.

[2] Blaise Barney et Donald Frederick : Introduction to parallel computing tutorial. https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial##MemoryArch, 2022.

[3] Henri Casanova, Arnaud Legrand et Yves Robert : *Parallel Algorithms*. 07 2008.

[4] Google Cloud : Cloud tensor processing units (tpus). https://cloud.google.com/tpu/docs/tpus, 2022.

[5] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice E. Santos, Ramesh Subramonian et Thorsten von Eicken : Logp : Towards a realistic model of parallel computation. *In Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), San Diego, California, USA, May 19-22, 1993*, pages 1–12, 1993.

[6] GeeksForGeeks : Pram or parallel random access machines. https://www.geeksforgeeks.org/pram-or-parallel-random-access-machines/, 2022.

[7] A. Grama, V. Kumar, A. Gupta et G. Karypis : *Introduction to Parallel Computing*. Pearson Education. Addison-Wesley, 2003.

[8] Mostafa Ibrahim : What is a tensor processing unit (tpu) and how does it work ? https://towardsdatascience.com/what-is-a-tensor-processing-unit-tpu-and-how-does-it-work-dbbe6ecbd8ad, 2021.

[9] Joseph JáJá : *An Introduction to Parallel Algorithms*. Addison Wesley Longman Publishing Co., Inc., USA, 1992.

[10] Arvind Krishnamurthy : Cs-424 - parallel computing : Readings. https://homes.cs.washington.edu/~arvind/cs424/readings/pram.pdf, 2002.

[11] Vipin Kumar, Ananth Grama, Anshul Gupta et George Karypis : *Introduction to Parallel Computing : Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., USA, 1994.

[12] Jayric Maning : What is a tpu (tensor processing unit) and what is it used for ? https://www.makeuseof.com/what-is-tpu-how-is-it-used/, 2022.

[13] Marc Moreno Maza : Cs3101 - parallel random-access machines. http://www.csd.uwo.ca/~moreno/cs3101_Winter_2015/PRAMs.pdf, 2015.

[14] Encyclopedia of Mathematics : Parallel random access machine. https://encyclopediaofmath.org/wiki/Parallel_random_access_machine, 2018.

[15] TutorialsPoint : Parallel algorithm tutorial. https://www.tutorialspoint.com/parallel_algorithm/index.htm, 2021.

[16] Sathish Vadhiyar : Pram algorithms. http://cds.iisc.ac.in/wp-content/uploads/PRAM.pdf, 2009.

[17] Robert van Engelen : Hpc-ii : Advanced topics in hpc - parallel algorithms & the pram model. https://www.cs.fsu.edu/~engelen/courses/HPC-adv/PRAM.pdf, 2009.

[18] Wikipedia : Réseau systolique. https://fr.wikipedia.org/wiki/R%C3%A9seau_systolique, 2021.

[19] Wikipedia : Analysis of parallel algorithms. https://en.wikipedia.org/wiki/Analysis_of_parallel_algorithms, 2022.

[20] Wikipedia : Systolic array. https://en.wikipedia.org/wiki/Systolic_array, 2022.

[21] Wikipedia : Tensor processing unit. https://fr.wikipedia.org/wiki/Tensor_Processing_Unit, 2022.