Calcul parallèle
Topologie des architectures parallèles

Florian Barbarin
Abdelkader Beldjilali
Nicolas Holvoet

13 janvier 2017

Table des matières

Introduction

Les premières architectures informatiques utilisaient des processeurs qui effectuaient les opérations arithmétiques les unes après les autres. Ces processeurs étaient ainsi qualifiés de séquentiels. L'amélioration de la puissance de calcul des ordinateurs passait alors par l'augmentation de la fréquence d'horloge des processeurs, c'est à dire le nombre d'opérations par seconde réalisées par le composant.

Cependant, des limites physiques à l'augmentation de la fréquence d'horloge ont vite été atteintes : la température des composants ne rendait par exemple plus possible l'utilisation de telles machines dans le cadre de l'informatique personnelle. C'est à ce moment que les architectures parallèles ont été largement implémentées dans les ordinateurs grâce à l'émergence des processeurs multi-coeurs.

Ce type d'architecture n'était pas nouveau mais était, jusqu'alors, plutôt réservé à des calculateurs professionnels et bien spécifiques : Météo-France utilisait par exemple des supercalculateurs *Cray* dans les années 1980-1990. La fréquence d'horloge n'étant plus vraiment améliorable, les constructeurs ont alors commencé à augmenter le nombre de cœurs et de processeurs pour aboutir à des architectures qualifiées de *massivement parallèles*.

Le nombre de nœuds de calcul augmentant alors de façon conséquente, la communication entre chacun de ces nœuds ne pouvait plus se faire de façon simple. L'organisation et les liens entre tous les processeurs devaient être au mieux adaptés et des topologies toujours plus efficaces devaient être mises en œuvre. C'est pour résoudre ces difficultés qu'il a été fait appel aux résultats de la théorie des graphes. Nous allons voir en quoi ces derniers ont largement contribué à améliorer les performances des architectures parallèles. Nous nous intéresserons tout d'abord à des résultats généraux sur les graphes et notamment aux algorithmes, bien connus, de Dijkstra et A* avant d'étudier certaines formes particulières de graphes : les grilles, les arbres, les grilles d'arbre et, enfin, les hypercubes.

1 Algorithmes de Dijkstra et A*

Algorithmes de Dijkstra : Edgser Wybe Dijkstra (EWD), physicien néerlandais reconverti à l'informatique en 1955, a proposé en 1959 un algorithme de recherche de chemin minimum dans un graphe dont la complexité est en $\mathcal{O}(n)$.

FIGURE 1 – Edgser Wybe Dijkstra (1930-2002)

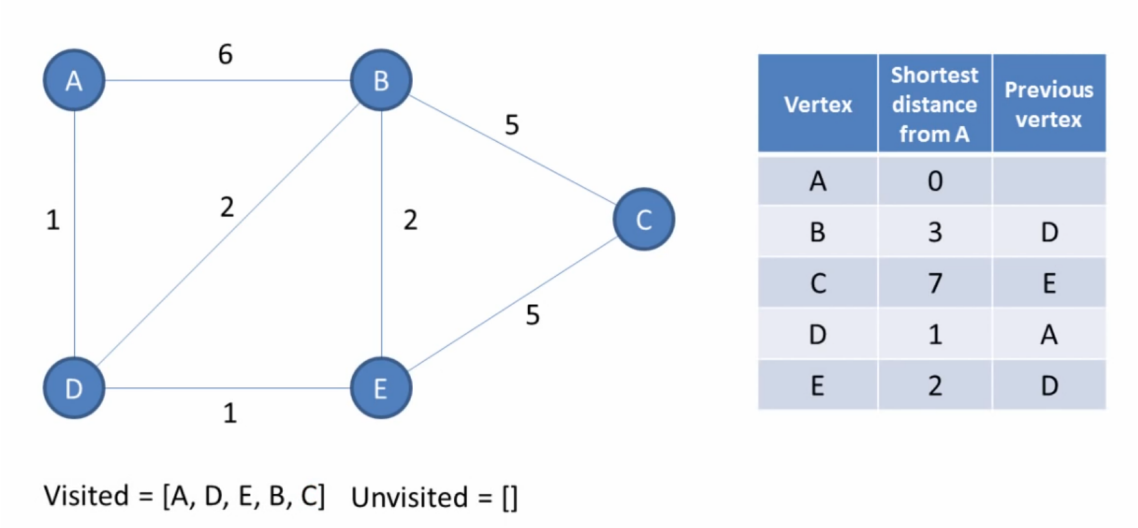
On doit à Dijkstra, qui avait la réputation d'avoir mauvais caractère et qui était notoirement allergique au "GOTO", quelques citations[1] telles que :

> *« Il est pratiquement impossible d'enseigner la bonne programmation aux étudiants qui ont eu une exposition antérieure au BASIC : comme programmeurs potentiels, ils sont mentalement mutilés, au-delà de tout espoir de régénération. »*
>
> *« Le plus court chemin d'un graphe n'est jamais celui que l'on croit, il peut surgir de nulle part, et la plupart du temps, il n'existe pas. »*

1. source : https://fr.wikipedia.org/wiki/Edsger_Dijkstra

« La programmation par objets est une idée exceptionnellement mauvaise qui ne pouvait naître qu'en Californie. »

L'algorithme donne le plus court chemin de la source à *tous les sommets* d'un graphe connexe pondéré (orienté ou non) dont le poids lié aux arêtes est positif ou nul.



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

FIGURE 2 – Exemple de calcul des plus courts chemins à partir du noeud A.

Sans entrer dans les détails d'implémentation que l'on trouve nombreux sur la toile et dans la littérature, l'algorithme de Dijkstra est un algorithme glouton qui utilise l'hypothèse qu'une décision prise sur la base d'un critère d'optimalité locale conduira à un optimum global. Ainsi, à chaque itération, l'algorithme choisit, parmi les nœuds non traités, le nœud du réseau dont la distance au nœud de départ est la plus faible. Ainsi, dans l'image 2, on voit que le plus court chemin pour aller de A à C a un coût de 7. On détermine ensuite de proche en proche le chemin le plus court grâce à la mémorisation des nœuds précédents. Ici, C a pour "previous vertex" E, E -> D et D -> A. Le plus court chemin pour aller de A à C est donc ADEC. On utilise la même technique pour déterminer tous les chemins optimaux d'origine A. Mais on peut n'avoir besoin que du calcul du chemin entre 2 sommets. De plus, la myopie de l'algorithme limite forcément ses performances. D'où l'idée d'utiliser une heuristique visant à guider les choix locaux. Ce qui nous amène à l'algorithme A* présenté brièvement dans la suite.

Algorithme A* : A* est l'algorithme qu'on utilise intuitivement pour se déplacer d'un endroit à un autre dans une ville connaissant la direction à prendre. Grâce à cette information supplémentaire, A* va privilégier le nœud qui minimise la somme de la distance déjà parcourue et de la distance estimée restant à parcourir, qui peut être ici la distance à vol d'oiseau.

Cet algorithme a été proposé pour la première fois par Peter E. Hart, Nils John Nilsson et Bertram Raphael en 1968. Il s'agit d'une extension de l'algorithme de Dijkstra et s'applique comme ce dernier à des graphes munis d'une distance positive. En outre, A* ne teste pas tous les chemins ; il ne fournit donc qu'un des meilleurs chemins. A* est cependant très performant dans le cas où le graphe comporte une faible densité de nœuds mais n'est pas ou peu efficace dans le cas de parcours de labyrinthes par exemple.

Il faut donc choisir, en fonction du problème, une heuristique h, c'est-à-dire ici une fonction qui estime la distance restante entre chaque nœud et l'arrivée, estimation par défaut qui ne doit jamais surestimer cette distance. La précision du résultat final dépendra de la précision de l'estimation. On peut citer les heuristiques suivantes : la distance euclidienne (norme 2) ou à "vol d'oiseau" déjà citée et la distance de Manhattan (norme 1) qui, sur une grille, correspond à la somme des cases verticales et horizontales qui sépare la position courante de l'arrivée. Les deux distances sont utilisables dans A* car elles sous-estiment la distance à l'arrivée.

À chaque étape, l'algorithme calcule g, la distance déjà parcourue pour chaque nœud n non visité puis la valeur $f(n) = g(n) + h(n)$. Le nœud sélectionné est celui pour lequel f est minimale. Lorsque

l'heuristique est la fonction nulle, A* est équivalent à l'algorithme de Dijkstra.

Utilisation L'application évidente de ces algorithmes dans le cadre du calcul parallèle est le transfert optimal de données d'un nœud *A* vers un nœud *B*. Cette application est d'ailleurs largement inspirée du domaine des réseaux de communication qui utilisent aussi la modélisation par graphes. On retrouve ainsi l'algorithme de Dijkstra dans le protocole de routage dynamique *OSPF*.

On peut donc envisager optimiser un réseau de nœuds distribués en permettant l'ajout d'un nœud et sa prise en compte automatique par le réseau.

Quant à l'algorithme A*, il permettrait de déterminer un chemin point à point optimal utilisé ensuite par exemple pour créer une liaison par commutation de circuit intéressante si la taille des données qui doivent être échangées est importante.

There are many different heuristic functions used for the grid maps. Some famous heuristics are Manhattan distance, diagonal distance, Euclidean distance. We are using the Manhattan distance to estimate h(x) because it works better on squared grids. It is the direct distance from current node to the goal node without considering obstacles in the path. In this way h(x) is giving us the lowest possible cos t to reach the goal node

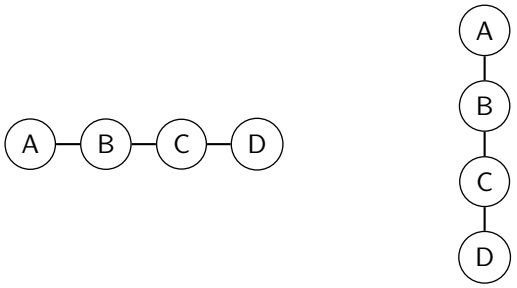
2 Les grilles

2.1 Défintion

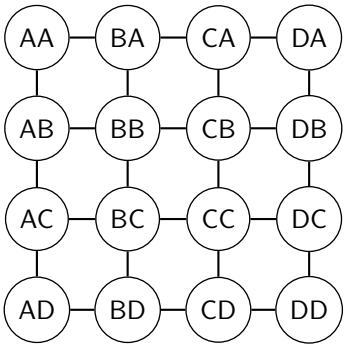
Définition. Une grille de dimension d possédant N nœuds suivant chaque coordonnée est le produit cartésien de d chaines ($d > 1$) de N sommets. On note cette grille $M(N)^d$ que l'on dira *de coté N*.

Remarque. Si l'on considère le produit cartésien de deux graphes, le graphe résultant est tel que :
- l'ensemble de ses nœuds est le produit cartésien des nœuds des deux premiers graphes ;
- deux de ses nœuds sont voisins s'ils sont composés de nœuds qui étaient voisins dans l'un des deux premiers graphes.

Exemple. Soient deux chaines composées des nœuds appartenant à l'ensemble $E = \{A, B, C, D\}$. Le produit cartésien de ces deux chaines ($d = 2$) de taille $N = Card(E) = 4$:



A pour résultat la grille $M(4)^2$:



2.2 Propriétés

2.2.1 Nombre total de nœuds

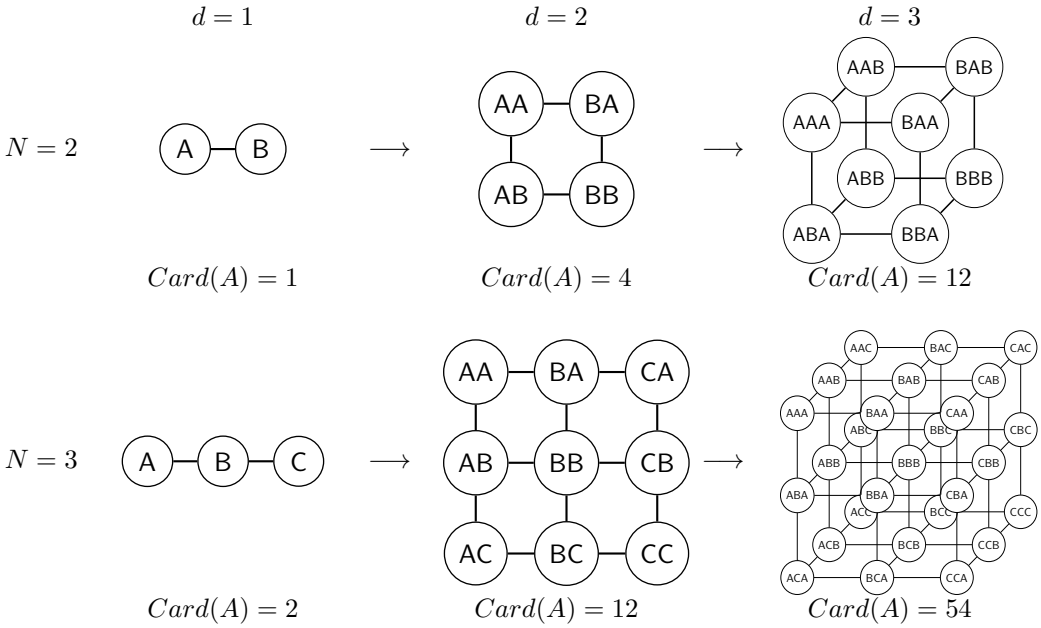
Il résulte de la définition ci-dessus que le nombre total de nœuds est égal au cardinal du produit cartésien de l'ensemble des nœuds de départ :

$$Card(S) = Card(\underbrace{E \times E \times ... \times E}_{\text{p fois}}) = \prod_{1}^{p} Card(E) = N^p$$

où S est l'ensemble des noeuds de la grille

2.2.2 Nombre total d'arêtes

Soit A l'ensemble des arêtes de la grille. Voici pour $N = 2$ et $N = 4$ le passage de la dimension 1 aux dimensions supérieures (2 et 3).



On remarque à présent que, pour N fixé, le passage d'une dimension d à une dimension $d + 1$ se fait en deux étapes :

— on "copie" N fois la grille de dimension d ;

— on relie, par une arête, les nœuds de la grille 1 avec les nœuds correspondants de la grille 2 puis les nœuds de la grille 2 avec les nœuds correspondants de la grille 3, et ainsi de suite jusqu'à la grille N.

Cette méthode de construction nous donne une relation de récurrence pour le nombre d'arêtes :

$$Card(A)_{d+1} = \underbrace{N \times Card(A)_d}_{\text{On copie } N \text{ fois la grille de dimension } d} + \underbrace{(N-1) \times Card(S)_d}_{\text{On relie les arêtes de chaque grille}}$$

On ne peut aisément déterminer une expression générale de la suite ci-dessus. Or les exemples précédents nous permettent de déduire une expression du nombre d'arêtes en fonction de d et de N :

$$Card(A)_d = d \times (N-1) \times N^{d-1}$$

Nous pouvons démontrer par récurrence cette expression.

Preuve de la relation.

Initialisation : Pour $d = 1$, on a $Card(A)_1 = 1 \times (N-1) \times N^{1-1} = N - 1$ ce qui correspond bien au nombre d'arêtes dans une chaine.

Hypothèse de récurrence : On fait l'hypothèse qu'il existe un rang d tel que $Card(A)_d = d \times (N-1) \times N^{d-1}$. Montrons que cette relation est vraie au rang $d+1$.

Hérédité : Nous avons :

$$\begin{aligned} Card(A)_{d+1} &= N \times Card(A)_d + (N-1) \times N^d \\ &= N \times d \times (N-1) \times N^{d-1} + N^d \times (N-1) \\ &= N^d \times d \times (N-1) + N^d \times (N-1) \\ &= (d+1) \times (N-1) \times N^d \end{aligned}$$

Nous retrouvons bien l'hypothèse de récurrence au rang $d+1$. On en déduit que $\forall d \in \mathbb{N}^*$, on a $Card(A)_d = d \times (N-1) \times N^{d-1}$. □

D'après l'ensemble des éléments qui précèdent, le nombre d'arêtes d'une grille est telle que :

$$\forall d \in \mathbb{N}^*,\ Card(A)_d = d \times (N-1) \times N^{d-1}$$

2.2.3 Diamètre de la grille

La construction de la grille telle qu'envisagée jusque là nous permet de faire l'observation suivante : un nœud de la grille diffère exactement d'une seule coordonnée de ses voisins. Pour reprendre l'exemple déjà vu ci-dessus avec $N = 2$ et $d = 2$, le nœud AA est voisin des nœuds AB et BA. Dans un cas plus général où les nœuds de la grille seraient formés de coordonnées entières, le nœud de coordonnées $(x_0, x_1, ..., x_{d-1})$ aurait pour voisin dans la dimension i le noeud de coordonnées $(x_0, x_1, ..., y_i, ..., x_{d-1})$ avec $y_i = x_i \pm 1$.

Or par définition, le diamètre d'un graphe est la plus grande distance entre deux sommets. Dans le cas général d'une grille composée de coordonnées entières, les deux nœuds les plus éloignés sont le nœud $\underbrace{(0, 0, ..., 0)}_{q \text{ fois}}$ et le nœud $\underbrace{(N, N, ..., N)}_{q \text{ fois}}$. Le plus long chemin entre ces deux nœuds est donc de passer par l'ensemble des coordonnées possibles, ce qui correspond à q fois $N - 1$ possibilités.

Au final, le diamètre D d'une grille est :

$$D = q \times (N-1)$$

2.2.4 Bissection de la grille

La bissection, ou plus précisément la largeur de la bissection, est le nombre minimum d'arêtes qu'il faut enlever à la grille pour la diviser en deux moitiés avec un nombre de nœuds identique (à un près).

On remarque déjà que ce problème dépend de la parité du nombre de nœuds de la grille : dans le cas où celui-ci est pair, il est possible de diviser la grille en deux avec un nombre de nœuds identique ; dans le cas où celui-ci est impair, les deux grilles résultantes auront un nombre de nœuds égal, à plus ou moins un nœud près.

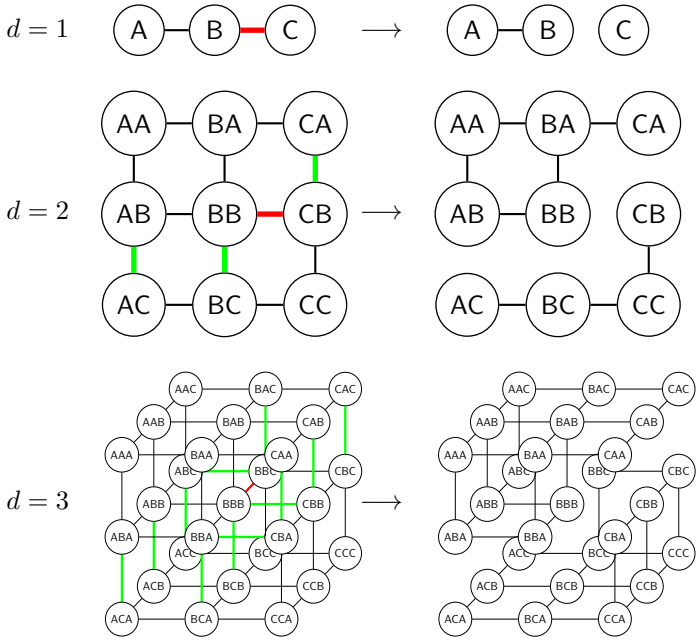
Cas où N est pair : La séparation en deux grilles avec un nombre de nœuds identique est triviale. La question est cependant de connaître le nombre exact d'arêtes à enlever pour obtenir cette séparation.

Or on a vu précédemment que passer de la dimension $d-1$ à la dimension d se faisait en deux étapes : recopie N fois de la grille de dimension $d-1$ et création des nouvelles arêtes entre chaque paire de nœuds correspondants dans les deux grilles, c'est à dire N^{d-1} arêtes.

Ainsi, lorsque l'on sépare une grille en deux, on la sépare à une jonction entre deux grilles de dimension $d-1$ ce qui nous conduit à enlever exactement N^{d-1} arêtes.

On peut donc dire que la largeur de la bissection d'une grille de côté N (N étant pair) est N^{d-1}.

Cas où N est impair : On rappelle que la bissection est le nombre minimum d'arêtes qu'il faut enlever à la grille pour la diviser en deux moitiés avec un nombre de nœuds identique (a un près). Dans le cas où $N = 3$, on peut, en prenant en compte cette définition, effectuer la division de la façon suivante :



On remarque à travers cet exemple que la division en deux grilles se fait en deux étapes :

— pour une grille de dimensions d, on effectue une bissection dans chaque dimension $d-1$ (arêtes vertes) ;

— on obtient deux blocs reliés par une unique arête (arête rouge) que l'on retire afin d'obtenir deux blocs distincts avec le même nombre de nœuds (à un près).

En effet, si l'on prend le cas ci-dessus où $d = 2$, on a vu précédemment que la grille était composée de 3 chaines de dimension $d-1 = 1$. On effectue dans ces chaines les bissections que l'on avait trouvées en dimension 1 ce qui nous amène à supprimer les arêtes en vert. Ne reste alors qu'une arrête joignant les deux blocs (arête rouge $(BB) - (CB)$) qu'il faut supprimer pour obtenir une bissection conforme à la définition.

De même, en prenant le cas où $d = 3$, la grille est composée de trois grilles de dimension $d-1 = 2$. On effectue dans chacune de ces grilles les bissections que nous venons de voir en dimension 2 ce qui nous amène à supprimer les arêtes en vert. Ne reste alors qu'une arrête joignant les deux blocs (arête rouge $(BBB) - (BBC)$) qu'il faut supprimer pour obtenir une bissection conforme à la définition.

Si l'on écrit ce procédé mathématiquement, on obtient, en notant B_d la bissection de la grille en dimensions d, la relation de récurrence suivante :

$$B_d = N \times B_{d-1} + 1$$

En effet, de façon générale, le nombre d'arêtes à enlever pour obtenir une bissection en dimension d est égale à N fois le nombre d'arêtes à enlever en dimension $d-1$ plus une arête pour séparer les deux blocs.

On obtient alors une suite arithmético-géométrique que l'on peut facilement résoudre. On obtient alors dans le cas général :

$$B_d = N^d(B_0 - \frac{1}{1-N}) + \frac{1}{1-N} = \frac{1-N^d}{1-N} \text{ avec la convention } B_0 = 0$$

Remarque importante : Il est à noter que la valeur $\frac{1-N^d}{1-N}$ est obtenue suite à l'application d'une méthode empirique qui ne prouve en rien qu'il s'agisse de la bissection optimale. Cela avait été clairement noté par [Lei92] dans sa liste de problèmes restés ouverts. Il ne s'agit en effet que d'une borne inférieure

de la valeur de la bissection. Le problème est resté ouvert pendant plusieurs années avant que [EF07] ne prouve qu'il s'agisse également d'une borne supérieure, montrant par la même occasion que $B_d = \frac{1-N^d}{1-N}$ est la valeur optimale d'une bissection pour une grille de coté N (N étant impair).

2.2.5 Exemple d'algorithme sur une grille

Dans [Lei92], des exemples d'algorithmes s'implémentant sur des grilles ou des grilles toriques sont donnés au chapitre 3. L'un des exemples les plus répandus est l'implémentation parallèle d'un produit matriciel pour des matrices d'ordre N. L'ouvrage propose un principe d'implémentation pour effectuer ce produit en N étapes de calcul : on crée une grille de dimension 2 et de coté N sur laquelle on répartit les calculs des coefficients $c_{i,j} = \sum_{k=1}^{n} a_{i,k} \times b_{k,j}$. Ainsi, le nœud (i, j) de la grille en dimension 2 est en charge du calcul du coefficient $c_{i,j}$.

Différents algorithmes proposent de mettre en œuvre cette méthode. C'est notamment le cas de l'algorithme de Fox que nous allons détailler ci-dessous.

Initialisation de l'algorithme de Fox (étape 0) : On considère pour simplifier la présentation de cet algorithme que nous sommes en présence de deux matrices A et B d'ordre N ainsi que de N^2 nœuds de calculs. Chaque nœud se voit attribuer la mémoire nécessaire pour stocker un élément de A, un élément de B et un élément de C ($C = AB$).

Étape 1 de l'algorithme de Fox : La première étape du calcul consiste à calculer dans chaque nœud (i, j) le coefficient $c_{i,j} = a_{i,i} \times b_{i,j}$. Il est donc nécessaire de diffuser les coefficients diagonaux $a_{i,i}$ de la matrice A à travers les lignes de la grille ainsi que les coefficient $b_{i,j}$ de la matrice B à chaque nœud (i, j) de la grille.

Étape k de l'algorithme de Fox : Quelle que soit l'étape k qui suit, le calcul dans chaque nœud (i, j) de la grille consiste à calculer le coefficient $c_{i,j} = c_{i,j} + a_{i,i+k} \times b_{i+k,j}$. Cela se fait en diffusant sur chaque ligne de la grille l'élément de la colonne suivante (modulo N) de A et en translatant sur la grille (toujours modulo N), du bas vers le haut, les éléments de la matrice B.

Après N étapes, chaque nœud (i, j) de la grille détient le coefficient $c_{i,j}$ de la matrice $C = AB$.

Cet algorithme parallèle a donc une complexité en $\mathcal{O}(N)$ ce qui est bien meilleur que les algorithmes séquentiels de produit matriciel et ce qui justifie son usage lorsque la taille des matrices devient importante. En effet, une implémentation naïve d'un produit matriciel séquentiel (trois boucles "for" imbriquées) est en $\mathcal{O}(N^3)$ et des versions plus optimisées n'améliorent que très peu cette complexité. Par exemple, on apprend dans [Wik16b] que l'algorithme de Strassen a une complexité en $\mathcal{O}(N^{2,807})$ et dans [Wik16a] que celui de Coppersmith-Winograd a une complexité en $\mathcal{O}(N^{2,376})$, ce qui en fait l'algorithme séquentiel de produit matriciel le plus efficace asymptotiquement.

3 Les arbres et grilles d'arbres

3.1 Les arbres binaires

Diamètre d'un arbre binaire : Un arbre est constitué d'un ensemble de niveaux, le niveau 0 étant occupé par la racine de l'arbre. Les niveaux se répartissent donc de 0 à p où p est par définition la hauteur ou la profondeur de l'arbre. Chaque niveau i est occupé par k^i nœuds dans un arbre k-naire, par 2^i pour un arbre binaire. Par conséquent, à la profondeur p, un arbre binaire complet aura donc $f = 2^p$ feuilles. Un arbre binaire complet sera constitué de $2^{p+1} - 1 = 2f - 1$ nœuds y compris les feuilles. La plus grande distance entre deux nœuds, i.e. le diamètre de l'arbre, est donc le nombre d'arêtes du chemin passant par la racine et joignant deux feuilles. Ce qui donne :

$$D = 2p = 2\log_2(f)$$

puisque la profondeur est liée au nombre de feuilles par

$$p = \log_2(f)$$

Bissection d'un arbre binaire : La bissection est le nombre minimal d'arêtes à supprimer pour qu'un graphe initialement connexe se sépare en deux composantes connexes possédant le même nombre de nœuds à une unité près. Pour un arbre binaire, ce nombre minimal est obtenu en supprimant une des deux arêtes liées à la racine. On trouve donc dans ce cas une bissection égale à 1.

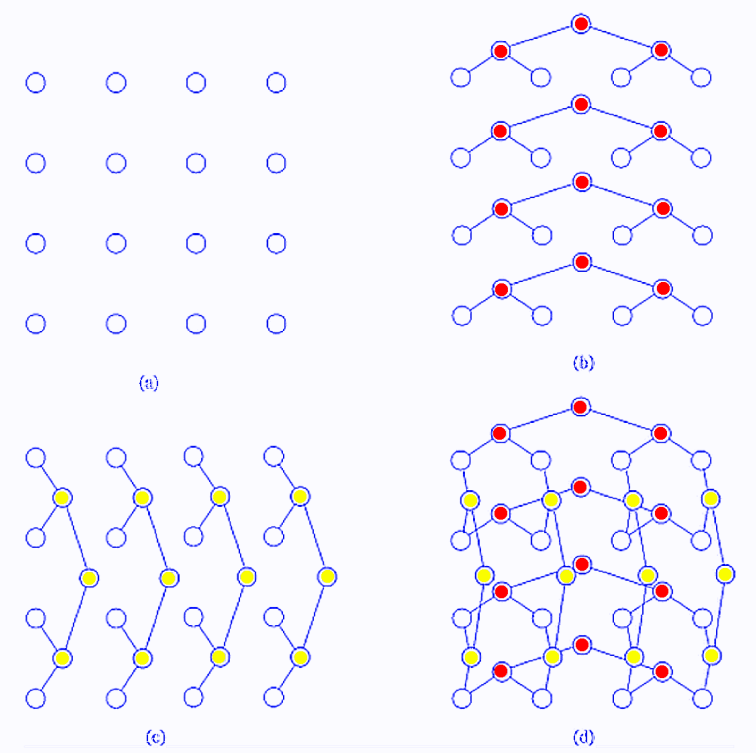


FIGURE 3 – Grille d'arbre 2-D construite sur une grille $N \times N$ avec $N = 4$ (a) à laquelle on connecte 4 arbres binaires horizontaux (b) et 4 arbres binaires verticaux (c) pour obtenir la grille d'arbres (d).

3.2 Les grilles d'arbres

Définition d'une grille d'arbres bidimensionnelle : Une grille d'arbre bidimensionnelle, *mesh of trees* en anglais ou MOT en abrégé, est construite à partir d'une grille bidimensionnelle reliée par des arbres binaires, verticalement et horizontalement comme illustré figure 3. Les nœuds de la grille constituent les feuilles des arbres. On a donc $2N$ arbres de profondeur $p = \log_2(N)$. Chacun des $2N$ arbres est composé de $(N-1)$ nœuds internes et de N feuilles. On a donc un nombre total de nœuds dans la grille d'arbres égal à la somme des nœuds de la grille $N \times N$, soit N^2 et du nombre de nœuds ajoutés par les arbres, soit $2N(N-1)$. Une grille d'arbres $N \times N$ est donc composée de $3N^2 - 2N$ nœuds. On montre enfin qu'une grille d'arbres comporte $2N(2^{p+1} - 2) = 4N^2 - 4N$ arêtes.

Diamètre d'une grille d'arbres : Le diamètre est la distance "diagonale" entre le nœud supérieur gauche et inférieur droit. La géométrie carrée implique que cette distance est le double de la distance qui sépare le nœud supérieur gauche et le nœud inférieur gauche. Or ces 2 nœuds sont les feuilles d'un arbre binaire qui compte N feuilles dont le diamètre est $D = 2p = 2\log_2(N)$. Le diamètre d'une grille d'arbres est donc :

$$D' = 4\log_2(N)$$

Bissection d'une grille d'arbres : La figure 3 (c) montre qu'en déconnectant les $N = 4$ racines des arbres verticaux, on aboutit à deux sous-réseaux de même taille si on affecte une fois sur deux les nœuds racines ainsi déconnectés au sous-réseau supérieur puis au sous réseau inférieur. La bissection est donc N.

Bilan : Les grilles d'arbres sont des architectures performantes car elles jouissent à la fois d'un diamètre faible, donc de connexions plus rapides et directes, d'un degré constant (1, 2 ou 3) et d'un nombre d'arêtes modéré, ce qui les rendent particulièrement évolutives. Le bissection, que l'on peut définir par rapport à la connectivité entre deux parties de la structure et la maîtrise du risque lié aux "goulots d'étranglement", n'est pas la plus élevée mais reste acceptable (voir 4.1 Comparaison des différentes architectures).

3.3 Implémentation du produit matrice-vecteur sur une grille d'arbres :

On suppose qu'on dispose d'une grille d'arbres $N \times N$ sur laquelle on cherche à effectuer le produit matrice-vecteur $Y = AX$ où $A = (a_{ij})$, $X = (x_i)$ et $Y = y_j$ avec $0 \leq i, j \leq N - 1$.

1. On introduit x_i par la racine de l'arbre horizontal i non représenté sur la figure 4(b), $0 \leq i \leq N - 1$.
2. Les x_i sont transmis aux feuilles au travers des arbres horizontaux de telle sorte que chaque feuille de l'arbre de la ligne i reçoit x_i à l'étape $\log N$.
3. On introduit (a_{ij}) dans la feuille (i, j) via les entrées I_i à l'étape $\log N$.
4. La feuille (i, j) peut alors calculer le produit $a_{ij}x_j$.
5. Les résultats sont sommés vers la racine des arbres verticaux.

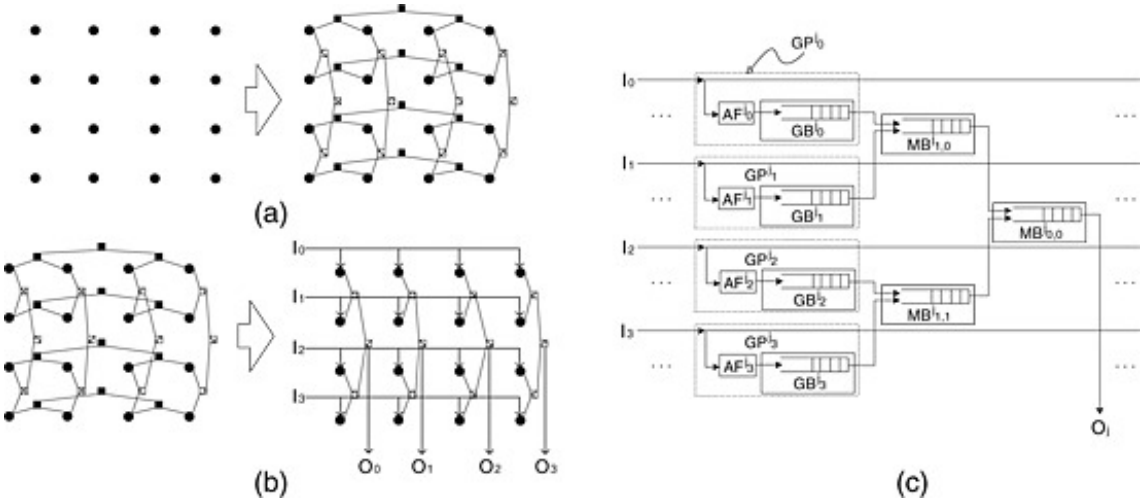


FIGURE 4 – Grille d'arbres pour illustrer le principe du calcul d'un produit matrice-vecteur.

Après $2 \log N$ étapes, la valeur

$$y_i = \sum_{j=1}^{N} a_{ij}x_j$$

est disponible dans la racine de la i^e colonne désignée par O_i sur la figure 4 (b).

4 Les hypercubes

Un hypercube est une grille de dimension d ne possédant que deux sommets selon chaque coordonnée. Ainsi, il possède 2^d nœuds de degré d et $d2^{d-1}$ arêtes. On construit un hypercube de dimension d récursivement à partir de deux hypercubes de dimension $d - 1$ en connectant les sommets similaires,

un hypercube de dimension 0 correspondant à un nœud de calcul unique. On obtient ainsi ces quatre premières grilles :

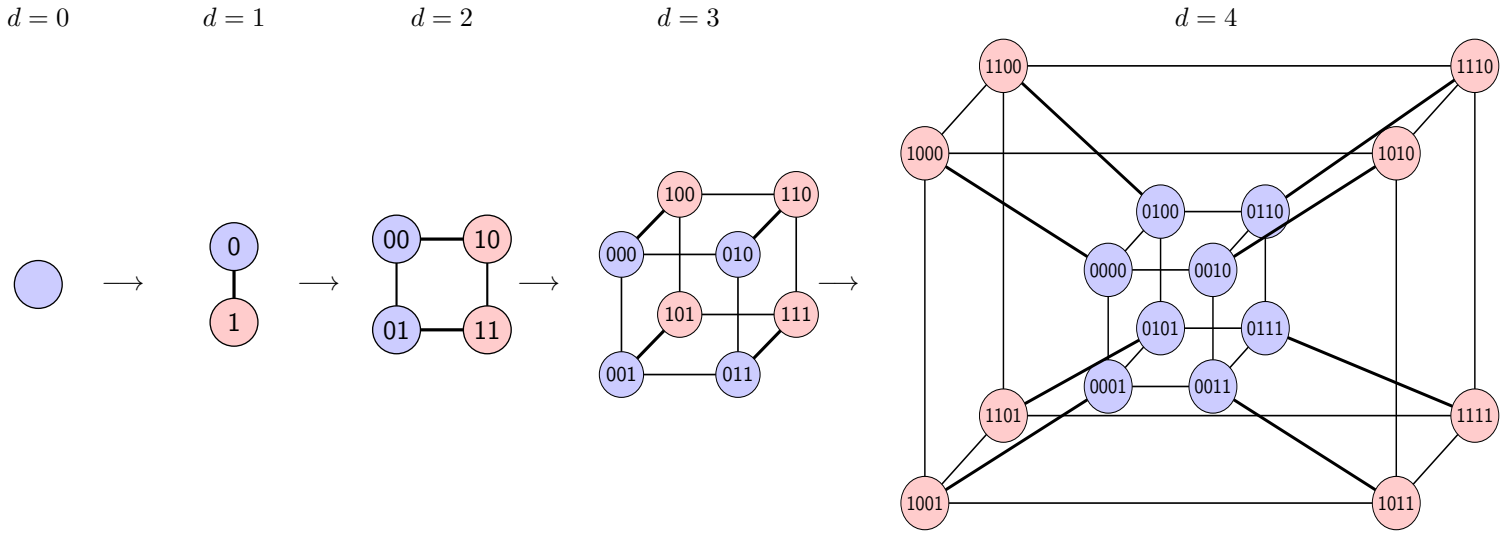


FIGURE 5 – Construction des hypercubes de dimension 0 à 4

On étiquette généralement les nœuds par une séquence de d bits. Pour étiqueter un hypercube de dimension d, on part d'un hypercube de dimension $d-1$ dont les sommets auront été préfixés d'un 0 (en bleu sur la figure) puis on ajoute un hypercube de dimension $d-1$ dont les sommets auront été préfixés d'un 1 (en rouge). On a alors deux propriétés intéressantes :

- Chaque bit correspond à une dimension de l'hypercube. Ainsi, dans notre exemple $d = 3$ et par construction, le premier bit (en partant de la droite) correspond à la coordonnée classique z (apparue dans l'hypercube $d = 1$), le deuxième correspond à la coordonnée y (issue de l'hypercube $d = 2$) et enfin le troisième à x. On peut donc voir les étiquettes sous une forme « xyz ».
- Un nœud est voisin d'un autre nœud si et seulement si leurs étiquettes diffèrent d'un seul bit. Ainsi, la distance entre deux nœuds correspond à la distance de Hamming de leurs étiquettes.

Ces propriétés permettent d'implémenter un algorithme de routage très simple dans lequel un message transite via les nœuds dont les étiquettes se rapprochent à chaque étape d'un bit de l'étiquette destination. Enfin, cela permet, à l'aide de codes de Gray bien définis, de plonger une grille classique ou un arbre dans un hypercube...

Diamètre Comme nous l'avons vu pour les grilles, le diamètre d'un hypercube de dimension d correspond à la distance entre le nœud $\underbrace{(0, 0, \ldots, 0)}_{d \text{ fois}}$ et le nœud $\underbrace{(1, 1, \ldots, 1)}_{d \text{ fois}}$ qui, compte tenu de la deuxième propriété énoncée ci-dessus, est égale à d.

Bissection D'après les résultats obtenus sur les grilles, la bissection d'un hypercube de dimension d est de 2^{d-1} (on partitionne l'hypercube en deux hypercubes de dimension $d-1$ en retirant les 2^{d-1} arêtes qui ont permis de les lier).

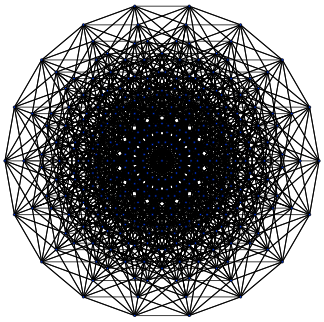


FIGURE 6 – Une projection de l'*enneract* (9-cube)

4.1 Comparaison des différentes architectures

À partir des résultats précédents, on peut manipuler les équations pour obtenir une estimation du diamètre, du degré des nœuds, du nombre d'arêtes et de la bissection des différentes architectures étudiées jusqu'ici, directement en fonction du nombre de nœuds de calcul n souhaité.

Architecture	Diamètre	Degré	Nombre d'arêtes	Bissection
Graphe complet	1	$n-1$	$\frac{n(n-1)}{2}$	$\approx \frac{n^2}{4}$
Grille de dimension d	$d(n^{1/d}-1)$	$2d$	$dn(1-n^{-1/d})$	$\approx n^{1-1/d}$
Grille de dimension 2	$2(\sqrt{n}-1)$	4	$2(n-\sqrt{n})$	$\approx \sqrt{n}$
Grille d'arbres bidimensionnelle	$\approx 2\log_2(\frac{n}{3})$	1, 2 ou 3	$\approx \frac{4}{3}n - \frac{4}{\sqrt{3}}\sqrt{n}$	$\approx \frac{\sqrt{n}}{\sqrt{3}}$
Hypercube	$\log_2 n$	$\log_2 n$	$\frac{1}{2}n\log_2 n$	$\frac{n}{2}$

FIGURE 7 – Comparaison des architectures pour n nœuds

On constate que l'hypercube offre de bons diamètre et bissection, au détriment d'un degré non constant et d'arêtes plus nombreuses et plus longues par rapport à une grille classique, ce qui implique un câblage plus complexe et coûteux. Pour ces raisons et selon l'application, on pourra préférer utiliser une grille d'arbres.

4.2 Template de communication

On trouve dans [Fos95] un *template* de communication classiquement utilisé dans de nombreux problèmes SIMD implémentés sur un hypercube :

Procédure 1 hypercube($task\_id$, n, $input$, $output$)

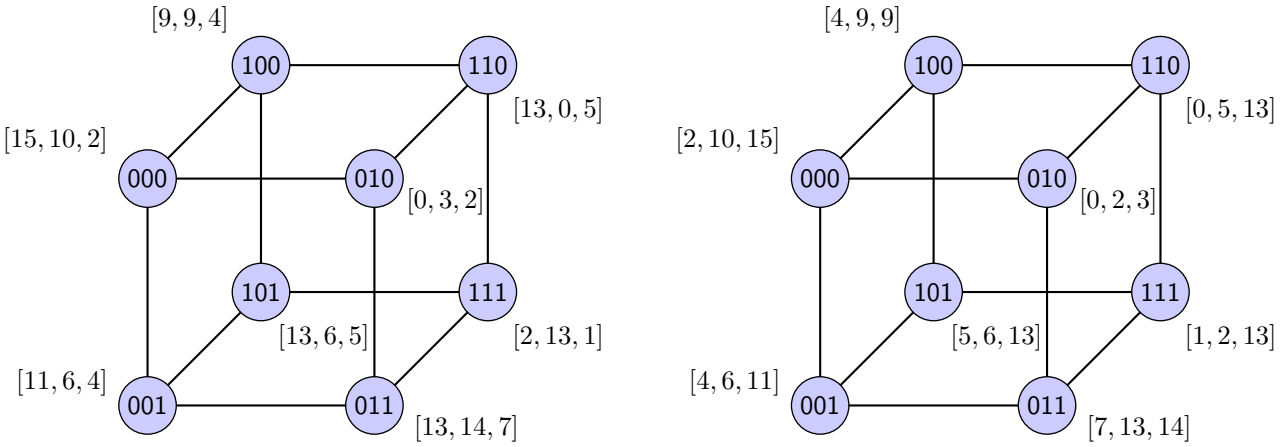
Input: $task\_id$, n, $input$, $output$
  $state \leftarrow input$
  for $i = 0$ **to** $\log_2 n - 1$ **do**
    $partner \leftarrow task\_id \oplus 2^i$
    send$(partner, state)$
    $message \leftarrow$ receive$(partner)$
    $state \leftarrow$ OP$(state, message)$
  end for
  $output \leftarrow state$

Cette procédure est exécutée par chacune des n tâches. La tâche $task\_id$, dont l'état initial vaut $input$, communique avec ses $\log_2 n$ voisins qu'elle identifie simplement avec une opération XOR sur chaque dimension. Elle envoie son état courant à son partenaire et reçoit de ce dernier le sien. La tâche met à jour son état en effectuant une opération « OP » combinant son état courant et l'information reçue de son voisin.

4.3 Tri bitonique

Dans ce paragraphe, nous décrivons l'algorithme de tri par fusion bitonique, particulièrement adapté aux réseaux de tri et au calcul parallèle. Pour simplifier les explications, on considère un tableau de taille multiple de n, notée $l = Mn = M2^d$ avec M entier. On commence par distribuer le tableau sur les n nœuds d'un hypercube de dimension d, chaque nœud ayant pour état initial un sous-tableau de taille M.

Prenons par exemple le tableau de 24 éléments suivant :
$[15, 10, 2, 11, 6, 4, 0, 3, 2, 13, 14, 7, 9, 9, 4, 13, 6, 5, 13, 0, 5, 2, 13, 1]$.
On distribue ce tableau en 8 sous-tableaux de trois éléments, de manière arbitraire sur un hypercube de dimension 3 (figure 8a) puis chaque tâche trie ensuite, de manière séquentielle, son sous-tableau (figure 8b).



(a) Distribution du tableau

(b) Tri séquentiel des sous-tableaux

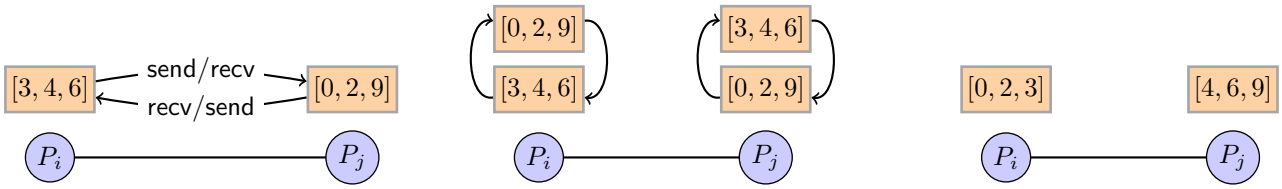
FIGURE 8 – Initialisation du tri bitonique sur un hypercube ($M = 3$)

4.3.1 *Compare-split*

Nous aurons besoin de faire communiquer les nœuds pour fusionner les sous-tableaux de manière à pouvoir obtenir la séquence triée attendue. L'idée est de maintenir des relations d'ordre sur les états des nœuds : pour des sous-tableaux T_i et T_j vivant respectivement sur les tâches P_i et P_j, on écrira $T_i \leq T_j$ si chaque élément de T_i est inférieur ou égal à chacun des éléments de T_j. On cherchera alors, pour deux nœuds voisins P_i et P_j, à obtenir des sous-tableaux $T_i' \leq T_j'$ en combinant leurs sous-tableaux initiaux T_i et T_j. L'algorithme permettant de réaliser cette opération porte le nom de *compare-exchange* ou *compare-split* et est notamment décrit dans [GGKK02] :

1. Chaque nœud envoie à l'autre nœud son sous-tableau de M éléments, si bien que les deux tâches disposent d'une copie locale de T_i et T_j.
2. Par un critère bien défini, chacun des nœuds sait s'il joue le rôle de P_i (resp. P_j) et fusionnera les deux tableaux T_i et T_j de manière à sélectionner les M plus petits (resp. plus grands) éléments de $T_i \cup T_j$ (on parlera de *compare-split-low* ou *compare-split-high*).
3. À l'issue, les deux nœuds contiendront l'un des sous-tableaux T_i' et T_j' avec $T_i' \leq T_j'$ (et bien entendu $T_i' \cup T_j' = T_i \cup T_j$).

Les sous-tableaux initiaux étant triés, l'opération de fusion ne présente pas de difficulté et sa complexité est en $\mathcal{O}(M)$. Pour cette même raison, il n'est pas toujours nécessaire de disposer des M éléments de l'autre sous-tableau pour obtenir le nouveau sous-tableau et on pourrait imaginer que les tâches s'échangent plus ou moins à la volée les éléments nécessaires à la décision, à condition que les coûts engendrés par le surplus de messages ne soient pas supérieurs aux gains liés à la réduction du volume des échanges.



(a) Les deux nœuds s'envoient leur tableau

(b) Chaque nœud dispose d'une copie locale des deux tableaux, qu'il fusionne

(c) Résultat

FIGURE 9 – *Compare-split* entre deux tâches P_i et P_j

4.3.2 *Parallel merge*

L'algorithme *compare-split* permet de trier un tableau distribué sur deux nœuds. La problématique qui se pose maintenant est commune à tous les réseaux de tri et plus généralement au calcul parallèle. Il s'agit d'assembler les résultats, ici en trouvant le réseau de tri c'est-à-dire la séquence (nombre, ordre...) d'opérations *compare-split* deux à deux permettant de trier le tableau correctement.

On trouve ainsi dans [GGKK02] cette séquence, exécutée par chacun des n processus :

Procédure 2 parallel-merge($task\_id$, n, $input$, $output$)

Input: $task\_id$, n, $input$, $output$
  $list \leftarrow input$
  for $i = 0$ **to** $\log_2 n - 1$ **do**
    for $j = i$ **downto** 0 **do**
      $partner \leftarrow task\_id \oplus 2^j$
      if $(task\_id \;\&\; 2^{i+1}) \oplus (task\_id \;\&\; 2^j)$ **then**
        $list \leftarrow$ compare-split-high$(partner, list)$
      else
        $list \leftarrow$ compare-split-low$(partner, list)$
      end if
    end for
  end for
  $output \leftarrow list$

Ci-dessous, on représente les séquences d'opérations nécessaires pour trier des éléments distribués sur un 2-cube puis sur un 3-cube, comme le ferait l'algorithme. Les flèches indiquent une opération *compare-split*, le nœud à la base de la flèche opère un *compare-split-high*, le nœud pointé un *compare-split-low*. En quelques mots, chaque itération sur i permet de fusionner deux tableaux triés sur deux hypercubes de dimension i pour obtenir un tableau trié sur un seul hypercube de dimension $i + 1$.

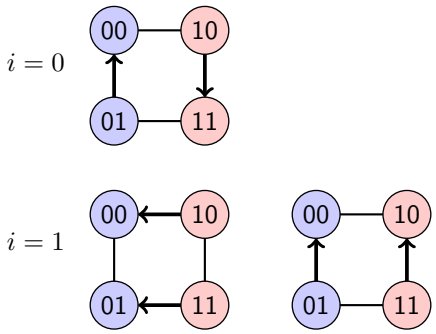


FIGURE 10 – Séquence du tri bitonique sur un 2-cube

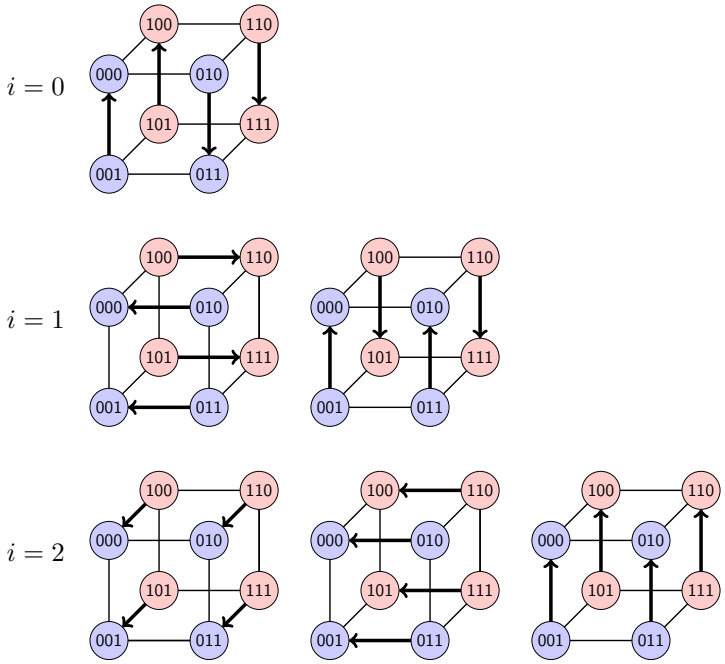


FIGURE 11 – Séquence du tri bitonique sur un 3-cube

Ainsi, en appliquant la séquence de *compare-split* représentée en figure 11 sur notre 3-cube initial, on obtiendrait le tableau trié, en parcourant nos nœuds par étiquette croissante.

La complexité de cet algorithme parallèle est en $\mathcal{O}(\log^2 n)$ et sa correction se prouve à l'aide de propriétés sur les suites dites bitoniques. Les communications étant nombreuses, cette implémentation n'est pas forcément évolutive et son usage doit être limité aux très grands tableaux.

4.4 Quelques autres exemples d'algorithmes

On peut utiliser une structure en hypercube pour résoudre bien d'autres problèmes qui se prêtent bien au parallélisme : multiplication de matrices, traitement d'images (rotation, produit de convolution, etc.). On peut alors définir un code de Gray permettant de définir la notion de « voisins » pour l'application que l'on souhaite en faire. On trouve aussi des hypercubes dans les problématiques de réseaux et de *broadcast*. Enfin, par plongements, l'hypercube peut être soit simulé sur une architecture le permettant, soit simulant en servant de support à un algorithme basé sur une grille, un arbre...

Références

[EF07] Kemal Efe and Gui-Liang Feng. A proof for bisection width of grids. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 1(3) :597–602, 2007.

[Fos95] Ian Foster. *Designing and Building Parallel Programs : Concepts and Tools for Parallel Software Engineering*. Pearson Addison-Wesley, 1995. https://www.mcs.anl.gov/~itf/dbpp/.

[GGKK02] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Pearson Addison-Wesley, 2nd edition, 2002. http://parallelcomp.uw.hu/index.html.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures : Array, Trees, Hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[Pac97] Peter S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers, San Francisco (Calif.), 1997.

[Wik16a] Wikipedia. Algorithme de coppersmith-winograd — Wikipedia, the free encyclopedia, 2016. [Online ; accessed 10-January-2017].

[Wik16b] Wikipedia. Algorithme de strassen — Wikipedia, the free encyclopedia, 2016. [Online ; accessed 10-January-2017].