Notes de cours *

Algorithmique parallèle et distribuée

Ecole Centrale

Algorithmique distribuée pour les réseaux

Master Parisien de Recherche en Informatique
(Univ. Paris Diderot, ENS, ENS Cachan, Ecole Polytechnique, Univ. Pierre et Marie Curie)

Pierre Fraigniaud

CNRS et Université Paris Diderot
pierre.fraigniaud@liafa.univ-paris-diderot.fr

Version du 5 octobre 2017

Résumé

Le contenu de ces notes de cours inclut le matériel présenté dans des cours d'algorithmique distribuée de l'école Centrale de Paris, et du MPRI. Ces notes présentent une introduction à différents aspects du calcul distribué et du calcul parallèle. Sont en particulier traités les problèmes algorithmiques posés par la distance entre les processeurs d'un réseau, et ceux posés par l'asynchronisme entre ces processeurs. L'algorithmique distribuée a comme principale objectif de résoudre ces problèmes lié à l'espace et au temps. L'algorithmique parallèle se focalise plutôt quant à elle sur les problèmes de performances, dont en particulier le facteur d'accélération que l'on peut espérer tirer de l'exécution d'un calcul sur plusieurs processeurs, comparé à l'exécution de ce même calcul sur un unique processeur.

*Ce document est encore un brouillon. Il est à diffusion restreinte, et n'est mis en ligne que pour fournir une aide aux étudiants des cours concernés. Il n'a pas pour objet remplacer les explications du cours, et certains arguments développés en cours peuvent ne pas apparaitre dans ce document.

Table des matières

1 Introduction

L'algorithmique distribuée est la branche de l'algorithmique s'intéressant à la conception d'algorithmes dédiés à tout ensemble d'unités de calcul (par exemple des processeurs) indépendants, exécutant chacun leur propre code, et dont l'objectif est la résolution commune d'un problème, ou la réalisation commune d'une tâche. La frontière entre algorithmique parallèle et algorithmique distribuée est parfois floue. Il convient néanmoins d'insister sur le caractère *indépendant* des processeurs impliqués dans l'exécution d'un algorithme distribué. Un tel algorithme ne doit pas supposer d'unité de contrôle centrale : le contrôle est lui-même distribué [1]. Par ailleurs, ou en conséquence de la nature distribuée du contrôle, un algorithme distribué ne doit pas se baser sur des hypothèses trop fortes relatives au degré de connaissance dont les processeurs disposent de leur environnement. L'algorithmique distribuée cherche donc moins la *performance* comme dans le parallélisme que savoir gérer l'*incertitude* liée à l'environnement [2]. Cette incertitude est générée par de multiples causes, dont la distance pouvant séparer les processeurs dans un réseau, la présence potentielle de pannes, le degré d'asynchronisme lié à des vitesses d'exécution variables, etc.

Internet est l'exemple typique de systèmes distribués. Le processeurs sont interconnectés par un réseau, ils sont asynchrones et susceptibles de tomber en panne. Internet est composé de sous-réseaux autonomes (les SA, pour « systèmes autonomes »), gérés individuellement par des administrateurs distincts, parfois concurrents, voire rivaux. A une toute autre échelle, un processeur multi-cœurs forme également un système distribué, dans lesquels les cœurs doivent s'auto-organiser pour exécuter les tâches du système d'exploitation en parallèle. De fait, la liste des systèmes distribués est immense, des réseaux de communication sans fil aux réseaux de capteurs, en passant par le « cloud computing » et les systèmes embarqués au sein d'une voiture ou d'un avion [3].

Dans ce cours, nous distinguerons principalement deux problématiques au cœur de l'algorithmique distribuée : (1) gérer les contraintes *spatiales*, et (2) gérer les contraintes *temporelles*. Dans le premier cas, il s'agira d'apprendre à concevoir des algorithmes distribués pour les réseaux, n'impliquant que des communications entre processeurs à faible distance les uns des autres, et/ou n'impliquant qu'un faible volume de communication entre les processeurs. Dans le second cas, il s'agira de concevoir des algorithmes distribués pour des systèmes asynchrones, c'est-à-dire impliquant des processeurs dont les vitesses d'exécution peuvent varier fortement entre elles et dans le temps, voire des processeurs pouvant tomber en panne (vitesse nulle).

Le cours abordera dans un second temps la conception de structure de données distribuées, en insistant sur la conception de structure de petite taille, dite *compact*. De telles structures de données peuvent être souhaitables par exemple pour la conception de table de routage dont la taille ne croît pas trop vite avec la taille du réseau, ou pour la stockage de données XML. Enfin, une dernière partie du document sera consacrée à une introduction à l'algorithmique parallèle, et le document se conclut par un aperçu des techniques probabilistes pour résoudre des problèmes de communications globales comme la diffusion d'un message d'une source à l'ensemble des processeurs d'un réseau.

Le lecteur est invité à consulter les ouvrages de référence suivant : [1, 3, 4].

1. Pour prendre une analogie militaire, la gestion d'une armée structurée sous les ordres d'un général a trait au parallélisme, alors que la gestion d'une foule de révolutionnaires sans leader pré-défini a trait au distribué.

2. Cette distinction entre performance et incertitude a été mise en évidence par M. Raynal, de l'Université de Rennes.

3. L'algorithmique distribuée est également un outil pour comprendre le comportement de systèmes non technologiques, dont par exemple les colonies de fourmis, les volées d'oiseaux ou les bancs de poissons.

2 Calcul distribué synchrone local

Ce chapitre a trait à la gestion de la localité, c'est-à-dire la conception d'algorithmes n'impliquant que des communications à distance faible dans un réseau.

2.1 Un modèle synchrone et la coloration distribuée

2.1.1 Le modèle LOCAL

Le modèle LOCAL s'applique à un réseau modélisé par un graphe $G = (V, E)$ dont les sommets (ou nœuds) modélisent des entités de calcul, et les arêtes modélisent des liens de communications. Chaque nœud possède un identifiant distinct entre 1 et $n = |V|$. On note $\mathrm{Id}(u)$ l'identité du nœud u. (On a donc $\mathrm{Id}(u) \in \{1, \ldots, n\}$ et $\mathrm{Id}(u) \neq \mathrm{Id}(v)$ pour tout $u \neq v$). Dans modèle LOCAL, les calculs s'effectuent par étapes synchrones. A chaque étape, chaque nœud u effectue trois opérations :

1. Envoyer un message à tous les voisins de u dans G ;
2. Recevoir le message de chacun des voisins de u dans G ;
3. Effectuer un calcul individuel.

Les communications ne sont pas a priori limitées en volume. Chaque message peut être estampillé par l'identité de son expéditeur. L'expéditeur d'un message peut distinguer les voisins destinataires de telle ou telle partie du message en estampillant ces parties par l'identifiant du destinataire. Les calculs individuels ne sont pas a priori limité en temps. On peut donc imaginer résoudre en chaque nœud un problème NP-difficile durant une seule étape dans le modèle LOCAL. Ce modèle n'a en effet pour objet que de mesurer la « localité » d'un algorithme distribué, pas sa complexité algorithmique séquentielle. Cela dit, lorsque l'on conçoit des algorithmes distribués dans le modèle LOCAL, il convient de prendre garde à n'effectuer que des calculs individuels raisonnables, c'est-à-dire de temps polynomial en fonction de la taille des données.

Remarque. Durant t étapes du modèle LOCAL, tout nœud u peut faire parvenir de l'information à tous les nœuds à distance au plus t dans G. De même, le nœud u peut récupérer toute l'information des nœuds à distance au plus t dans G, y compris les arêtes entre ces nœuds, sauf celles entre deux nœuds à distance exactement t de u. Tout algorithme $\mathcal{A}$ s'exécutant en temps t fixé, dans le modèle LOCAL, peut donc être transformé en un algorithme $\mathcal{B}$ résolvant le même problème et procédant en deux phases :

Phase 1 : récupérer toutes les données des nœuds à distance au plus t dans G (y compris la structure des liens entre ces nœuds) ;

Phase 2 : simuler les calculs des nœuds à distance au plus t effectué lors de l'exécution de $\mathcal{A}$.

L'algorithme $\mathcal{B}$ s'exécute également en t étapes dans le modèle LOCAL.

Briser la symétrie. Les nœuds d'un système distribué doivent se coordonner pour résoudre un problème ou effectuer une tâche. Une telle « auto-coordination » nécessite que les nœuds se distinguent les une des autres afin que certains effectuent telle opération pendant que d'autres effectuent telle autre opération. Evidemment, les identifiants deux-à-deux distincts attribués aux nœuds sont une clé permettant de briser la symétrie entre processeurs. L'utilisation des identifiants à cette fin peut toutefois se révéler difficile. Cette difficulté est illustrée dans le cas de la *coloration distribuée*.

Application. Non documenté.

2.1.2 Coloration distribuée

Rappelons qu'une k-coloration d'un graphe $G = (V, E)$ est une fonction

$$c : V \to \{1, \ldots, k\}$$

telle que, pour tout arête $\{u, v\} \in E$, $c(u) \neq c(v)$. Le plus petit k pour lequel il existe une k-coloration de G est appelé le *nombre chromatique* de G, noté $\chi(G)$. Etant donnés un graphe G et un entier k, décider si $\chi(G) \leq k$ est NP-complet. En fait, approximer $\chi(G)$ est également *très* difficile : pour tout $\epsilon > 0$, sauf si P = NP, approximer le nombre chromatique à un facteur $n^{1-\epsilon}$ est impossible en temps polynomial. En revanche, tout graphe admet une $(\Delta + 1)$-coloration, où $\Delta = \Delta(G)$ dénote le degré maximum des sommets de G :

$$\Delta = \max_{u \in V} \deg_G(u).$$

En particulier, un simple algorithme glouton permet d'obtenir une $(\Delta + 1)$-coloration :

Algorithme glouton de $(\Delta + 1)$-coloration de $G = (V, E)$:
début
$C \leftarrow \emptyset$
tant que $C \neq V$ **faire**
choisir $u \in V \setminus C$
$C \leftarrow C \cup \{u\}$
affecter à u la plus petite couleur non déjà affectée à un voisin de u dans C
fin.

L'algorithme ci-dessus retourne une $(\Delta+1)$-coloration de G car, puisque le degré de tout sommet u satisfait $\deg(u) \leq \Delta$, chaque sommet u possède au plus Δ voisins dans G. En conséquence, au moins une couleur parmi les $\Delta + 1$ couleurs autorisées est systématiquement disponible lorsque le sommet u est colorié par l'algorithme.

Notons que certains graphes n'admettent pas de Δ-coloration. C'est par exemple le cas de K_n, le graphe complet à n sommets, qui vérifie $\chi(K_n) = n = \Delta + 1$. C'est également le cas de C_{2n+1}, le cycle à $2n + 1$ sommets, qui vérifie $\chi(C_{2n+1}) = 3 = \Delta + 1$ pour tout $n \geq 1$.

Le problème de la $(\Delta+1)$-coloration permet de bien saisir la difficulté à laquelle on peut avoir à faire face en distribué lorsque l'on souhaite briser une symétrie. Ce problème est le suivant :

Le problème de $(\Delta + 1)$-COLORATION distribuée :

Entrée : un réseau $G = (V, E)$ de n nœuds, dont chaque nœud dispose d'une identité distincte dans $\{1, \ldots, n\}$;

Objectif : chaque nœud $u \in V$ retourne une valeur $c(u) \in \{1, \ldots, \Delta + 1\}$ telle que $c(u) \neq c(v)$ pour tout $\{u, v\} \in E$, où Δ dénote le degré maximum des sommets de G.

Ce problème semble a priori aisé à résoudre puisque qu'il suffit que chaque nœud se mette d'accord avec ses voisins afin de ne pas choisir une même couleur que l'un d'entre eux. Ceci est d'autant plus vrai que le choix d'une couleur par un nœud ne semble que faiblement contraindre les voisins de ce nœud puisque le nombre de couleurs à disposition de chaque nœud excède son

degré. En fait, quelque soit le choix des couleurs effectué par ses voisins, un nœud reste libre de choisir au moins une couleur. Nous allons néanmoins voir que la coloration distribuée ne peut se résoudre en ne communiquant qu'avec ses voisins, ni même en communiquant avec des nœuds à distance k, pour toute constante k. Mais avant de voir cela, étudions d'abord un algorithme simple, pour la 3-coloration de l'anneau C_n.

L'algorithme distribué de 3-coloration de l'anneau C_n ci-dessous est une variante d'un algorithme dû à Cole et Vishkin. Il suppose que les nœuds de l'anneau ont une notion commune de droite et de gauche, et qu'ils connaissent n. La couleur courante d'un nœud u est notée $c(u)$, dont l'écriture binaire est $c_1(u)c_2(u)\dots c_r(u)$. On note $|c(u)|$ la longueur de cette chaine binaire.

Algorithme de Cole et Vishkin pour la 3-coloration distribuée de C_n
pour un sommet u :
début
    $c(u) \leftarrow \mathrm{Id}(u)$
    pour $i = 1$ **à** N **faire** /* N *est une valeur fixée ultérieurement* */
        envoyer $c(u)$ au voisin de droite w
        recevoir $c(v)$ du voisin de gauche v
        $k \leftarrow$ plus petit indice tel que $c_k(v) \neq c_k(u)$
        $c(u) \leftarrow (c_k(u), k)$
    réduction-couleur()
fin.

Avant de décrire la fonction réduction-couleur() dont le but n'est que de finaliser la coloration, et de discuter de la valeur de N, montrons d'abord que si la coloration c est propre avant une itération de la boucle « pour » alors elle est propre après cette itération. (Une coloration c est *propre* si $c(u) \neq c(v)$ pour toute paire $\{u, v\}$ de nœuds voisins dans G). Initialement, la coloration $c = \mathrm{Id}$ est propre. Supposons que c est propre avant la $i$ème itération. Soit v le voisin de gauche de u, est soit $(c_k(u), k)$ et $(c_{k'}(v), k')$ les nouvelles couleurs respectives de u et v. Si $k \neq k'$, alors ces deux couleurs sont différentes. Si $k = k'$ alors les deux couleurs sont $(c_k(u), k)$ et $(c_k(v), k)$, qui sont également différentes puisque, par définition, $c_k(v) \neq c_k(u)$.

Pour calculer le nombre d'itérations, étudions la longueur des chaines binaires encodant les couleurs en binaire. Supposons que les couleurs soient sur K bits avant une itération. Les couleurs sont alors sur $|(c_k(u), k)| = 1 + \lceil \log K \rceil$ bits après cette itération. Notons alors que pour $K \geq 4$, on a $1 + \lceil \log K \rceil < K$. Soit $f : \mathbb{N} \to \mathbb{N}$ définie par $f(x) = 1 + \lceil \log x \rceil$. N est choisi comme le plus petit entier i tel qu'en itérant i fois f en partant de $x = \lceil \log n \rceil$ on obtienne une valeur ≤ 3.

Après N itérations, les couleurs sont donc sur au plus 3 bits. L'objectif de la fonction réduction-couleur() est simplement de réduire le nombre de couleurs de 8 à 3 :

fonction réduction-couleur() **pour un sommet u :**
début
    pour $i = 8$ **à** 4 **faire**
        envoyer $c(u)$ aux deux voisins de u
        recevoir $c(v)$ du voisin de gauche v et $c(w)$ du voisin de droite w
        si $c(u) = i$ **alors** $c(u) \leftarrow \min\{c : c \in \{1, 2, 3\} \setminus \{c(v), c(w)\}\}$
fin.

Le nombre d'étapes effectuées par l'algorithme de 3-coloration ci-dessus dans le modèle LOCAL est donc dominée par le nombre N d'itérations de la boucle « pour » réduisant exponentiellement le nombre de couleurs, car chaque itération de cette boucle correspond à une étape de calcul-communication, et la fonction réduction-couleur() s'exécute en 5 étapes. L'analyse du nombre N d'itérations nécessite d'introduire quelques notations. Soit $\log^{(k)}$ la fonction

log itérée k fois [4], c'est-à-dire :

$$\log^{(1)} n = \log n \ \text{ et } \ \log^{(i+1)} n = \log \log^{(i)} n \ \text{ pour tout } i \geq 1.$$

On définit alors :

$$\log^* n = \min\{k : \log^{(k)} n < 1\}.$$

Notez que si cette fonction croît très doucement (on a par exemple $\log^*(10^{100}) = 5$), elle n'est bornée par aucune constante.

Plus généralement, pour toute fonction f, définissons $f^{(1)} = f$, et $f^{(i+1)} = f \circ f^{(i)}$ pour tout $i \geq 1$. Soit $s > 0$ et $f : [s, +\infty[\to \mathbb{R}$ une fonction telle que, pour tout $x \geq s$, il existe $i \geq 1$ tel que $f^{(i)}(x) < s$. On définit $f^*(x)$ comme le plus petit indice i tel que $f^{(i)}(x) < s$.

Lemma 1 *Soit $a \geq 1$, $b \geq 1$, et $s \geq 1$ tel que $a \log s + b < s$. Soit $f : [s, +\infty[\to \mathbb{R}$ définie par $f(x) = a \log x + b$. Alors $f^*(x) = \log^* x + O(1)$.*

Preuve. Le choix de $s \geq 1$ tel que $a \log s + b < s$ garantit que f^* est bien définie. Soit $x \in [s, +\infty[$ tel que $\log x \geq b/a$ et $\log \log x \geq 1 + \log a$. (Pour des valeurs bornées de x, i.e., telles que $\log x < b/a$ ou $\log \log x < 1 + \log a$, $f^*(x)$ est trivialement borné supérieurement par une constante). On a $f(x) = a \log x + b$, et donc

$$f(x) \leq 2a \log x.$$

Il suit que

$$f^{(2)}(x) \leq 2a \log(2a \log x) = 2a(1 + \log a + \log \log x) \leq 4a \log \log x,$$

et donc

$$f^{(2)}(x) \leq 4a \log^{(2)} x.$$

Il en découle que

$$f^{(3)}(x) \leq 2a \log(4a \log^{(2)} x) = 2a(2 + \log a + \log^{(3)} x).$$

Ainsi, si $\log^{(3)} x \geq 2 + \log a$, on obtient

$$f^{(3)}(x) \leq 4a \log^{(3)} x.$$

Plus généralement, si $f^{(k)}(x) \leq 4a \log^{(k)} x$ et $\log^{(k+1)} x \geq 2 + \log a$, alors

$$f^{(k+1)}(x) \leq 2a \log(4a \log^{(k)} x) = 2a(2 + \log a + \log^{(k+1)} x) \leq 4a \log^{(k+1)} x.$$

L'inégalité $\log^{(k+1)} x \geq 2 + \log a$ est nécessairement violée pour un $k \leq \log^* x$. Considérons le plus petit k pour lequel elle est violée. On a

$$f^{(k)}(x) \leq 4a \log^{(k)} x \ \text{ et } \ \log^{(k+1)} x < 2 + \log a,$$

d'où il suit que

$$f^{(k)}(x) \leq 4a \log^{(k)} x = 4a 2^{\log^{(k+1)} x} < 4a 2^{2 + \log a} = 16a^2.$$

En conséquence, après $k \leq \log^* x$ itérations de f, on obtient $f^{(k)}(x) \leq 16a^2 = O(1)$, et donc $f^*(x) = \log^* x + O(1)$. □

4. Rappelons que log désigne le logarithme en base 2, et ln le logarithme népérien.

Par le lemme 1, on obtient que le nombre d'itérations N de l'algorithme de 3-coloration ne dépasse pas $\log^* n + O(1)$. Nous concluons cette section par le théorème suivant, qui raffine légèrement ce que nous venons de voir.

Théorème 1 *Il existe un algorithme de 3-coloration de l'anneau C_n s'exécutant en $\frac{1}{2}\log^* n + O(1)$ étapes dans le modèle* LOCAL. *Cet algorithme suppose que les noeuds ont une connaissance a priori de n, et ont une notion consistante de droite et de gauche.*

Preuve. Nous avons vu que la variante de l'algorithme de Cole et Vishkin présentée ci-dessus s'exécute en $\log^* n + O(1)$ étapes. Il est en fait aisé de diminuer le nombre d'étapes d'un facteur 2, en utilisant le fait que deux étapes de re-coloration peuvent être effectuées en une seule étape de communication.

Algorithme distribué rapide de 3-coloration de C_n pour un sommet u :
début
    $c(u) \leftarrow \mathrm{Id}(u)$
    pour $i = 1$ **à** N **faire**
        envoyer $c(u)$ aux voisins de gauche v et de droite w
        recevoir $c(v)$ du voisin de gauche v et $c(w)$ du voisin de droite w
        $k \leftarrow$ plus petit indice tel que $c_k(v) \neq c_k(u)$
(1) $k' \leftarrow$ plus petit indice tel que $c_{k'}(w) \neq c_{k'}(u)$
        $c'(u) \leftarrow (c_k(u), k)$
(2) $c'(w) \leftarrow (c_{k'}(w), k')$
        $k \leftarrow$ plus petit indice tel que $c'_k(w) \neq c'_k(u)$
        $c(u) \leftarrow (c'_k(u), k)$
    réduction-couleur()
fin.

L'algorithme ci-dessus alterne entre une étape de re-coloration par comparaison avec le voisin de gauche, et une étape de re-coloration avec le voisin de droite. Afin de pouvoir effectuer une re-coloration avec le voisin de droite sans communiquer avec lui, il suffit de simuler son comportement. C'est ce qui est fait aux instructions (1) et (2). Ensuite, u se re-colore en utilisant la nouvelle couleur de w, qu'il connaît sans avoir eu besoin de communiquer avec w. On a vu que $c'(u) \neq c'(w)$. Pour les même raisons, $c(u) \neq c(w)$.

La valeur N est maintenant choisie comme le plus petit i tel que $f^{(2i)}(\lceil \log n \rceil) \leq 3$ où $f : \mathbb{N} \to \mathbb{N}$ est la fonction de réduction $f(x) = 1 + \lceil \log x \rceil$ de Cole et Vishkin. En conséquence, $N = \frac{1}{2}\log^* n + O(1)$. □

Remarque sur le danger de l'arrêt prématuré. L'algorithme de Cole et Vishkin, tout comme celui présenté dans la preuve du théorème 1, nécessite que chaque nœud connaisse n afin de pouvoir calculer le nombre d'itérations N. On pourrait imaginer une version plus simple basée sur le test d'arrêt vérifiant en chaque nœud si la couleur courante est codée sur au moins 3 bits. Ce test ne nécessite pas la connaissance de n. En revanche, il induit un arrêt prématuré de certains sommets. En particulier, le nœud d'identifiant 1 n'effectuerait aucune communication dans le cœur de l'algorithme. Or ce nœud peut être voisin d'un nœud de grand identifiant attendant la valeur de l'identifiant de u pour réduire sa couleur. Se passer de la valeur de n nécessite donc des modifications plus profondes de l'algorithme de Cole et Vishkin.

En résumé, nous venons de voir un algorithme très efficace de $(\Delta + 1)$-coloration, pour l'anneau. Néanmoins, même si le nombre d'étapes, $\frac{1}{2}\log^* n + O(1)$, est extrêmement faible pour

toute valeur raisonnable de n, il n'en reste pas moins que ce nombre d'étapes ne peut être borné par une constante pour tout n. Existe-t-il un algorithme de $(\Delta+1)$-coloration s'exécutant en temps constant pour toute taille d'anneau ? La section suivante répond par la négative.

2.2 Borne inférieure pour la $(\Delta+1)$-COLORATION

L'objectif de cette section est d'établir le résultat suivant, dû à N. Linial, présenté dans un article qui a valu à son auteur le prix Dijkstra en algorithmique distribuée 2013.

Théorème 2 *Tout algorithme de 3-coloration de l'anneau C_n s'exécute en au moins $\frac{1}{2}\log^* n+1$ étapes dans le modèle* LOCAL*, ce même si les noeuds ont une connaissance a priori de n, et ont une notion consistante de droite et de gauche.*

La preuve de cette borne inférieure repose sur la notion de *graphe des configurations* $B_{n,t}$, défini pour $t \geq 0$ comme suit. Les sommets de $B_{n,t}$ sont les $(2t+1)$-uplets $(x_1, \ldots, x_{2t+1})$ où $x_i \in \{1, \ldots, n\}$ pour tout $i = 1, \ldots, t$, et $x_i \neq x_j$ pour tout $i \neq j$. Il y a une arête entre deux sommets $(x_1, \ldots, x_{2t+1})$ et $(y_1, \ldots, y_{2t+1})$ de $B_{n,t}$ si et seulement si $(x_1, \ldots, x_{2t+1}) = (y_2, \ldots, y_{2t+1}, z)$ pour $z \notin \{y_1, \ldots, y_{2t+1}\}$, ou $(y_1, \ldots, y_{2t+1}) = (x_2, \ldots, x_{2t+1}, z)$ pour $z \notin \{x_1, \ldots, x_{2t+1}\}$. Notez que les sommets de $B_{n,t}$ ne sont rien d'autres que l'ensemble de toutes les boules de rayon t que l'on peut construire dans un anneau de n sommets dont les nœuds sont numérotés arbitrairement par un entier entre 1 et n. L'intérêt principal du graphe des configurations est que son nombre chromatique est fortement lié aux nombres d'étapes d'un algorithme distribué de coloration :

Lemma 2 *S'il existe un algorithme distribué $\mathcal{A}$ de k-coloration de C_n s'exécutant en t étapes, alors $\chi(B_{n,t}) \leq k$.*

Preuve. En effet, soit c la k-coloration de $B_{n,t}$ obtenue comme suit. Le sommet $(x_1, \ldots, x_{2t+1})$ prend comme couleur celle retournée par $\mathcal{A}$ au sommet x_{t+1} dans l'anneau C_n lorsque la boule de rayon t centrée en x_{t+1} est égale à $(x_1, \ldots, x_{2t+1})$. Soit $(x_1, \ldots, x_{2t+1})$ et $(x_2, \ldots, x_{2t+1}, x_{2t+2})$ deux sommets voisins de $B_{n,t}$. L'algorithme $\mathcal{A}$ retourne deux couleurs différentes en x_{t+1} et x_{t+2} car une portion de C_n peut bien avoir la séquence de nœuds consécutifs d'identifiants $x_1, \ldots, x_{2t+1}, x_{2t+2}$. La k-coloration de $B_{n,t}$ est donc propre. □

L'inégalité $\chi(B_{n,t}) \leq k$ du lemme 2 va nous servir à borner t inférieurement en montrant que, pour t trop petit, on a $\chi(B_{n,t}) > 3$ et qu'il n'existe donc pas d'algorithme de 3-coloration en temps t.

Avant de montrer la borne inférieure pour la 3-coloration, nous allons utiliser le lemme 2 à titre d'exercice pour montrer que la 2-coloration de C_n pour n pair requiert au moins $\frac{n}{2}-1$ étapes. La suite de sommets $u_1, u_2, \ldots, u_{n-1}$ ci-dessous forme un cycle dans le graphe des configurations

$B_{n,t}$ pour $t = \frac{n}{2} - 2$ (et donc $2t + 1 = n - 3$) :

$$\begin{array}{lcccccccc}
u_1 & = & x_1 & x_2 & x_3 & \dots & x_{n-5} & x_{n-4} & x_{n-3} \\
u_2 & = & x_2 & x_3 & x_4 & \dots & x_{n-4} & x_{n-3} & y_1 \\
u_3 & = & x_3 & x_4 & x_5 & \dots & x_{n-3} & y_1 & y_2 \\
u_4 & = & x_4 & x_5 & x_6 & \dots & y_1 & y_2 & x_1 \\
\vdots & & & & & & & & \\
\vdots & & & & & & & & \\
u_{n-3} & = & x_{n-3} & y_1 & y_2 & \dots & x_{n-8} & x_{n-7} & x_{n-6} \\
u_{n-2} & = & y_1 & y_2 & x_1 & \dots & x_{n-7} & x_{n-6} & x_{n-5} \\
u_{n-1} & = & y_2 & x_1 & x_2 & \dots & x_{n-6} & x_{n-5} & x_{n-4}
\end{array}$$

Comme n est pair, $n - 1$ est impair, et donc le cycle $u_1, u_2, \dots, u_{n-1}$ est de longueur impair, indiquant que le graphe $B_{n,\frac{n}{2}-2}$ n'est pas biparti, et donc $\chi(B_{n,\frac{n}{2}-2}) > 2$. D'après le lemme 2, on en déduit qu'il n'existe pas d'algorithme de 2-coloration des cycles pairs s'exécutant en moins de $\frac{n}{2} - 1$ étapes.

Etablir que, pour t trop petit, on a $\chi(B_{n,t}) > 3$, est un peu plus plus difficile car il n'existe pas de critère simple de non 3-colorabilité. C'est ce que fait la preuve ci-dessous.

Preuve du théorème 2. Soit $D_{n,t}$ le graphe orienté dont les sommets sont les t-uplets $(x_1, \dots, x_t)$ où $1 \leq x_1 < x_2 < \cdots < x_t \leq n$. Il y a une arc d'un sommet $(x_1, \dots, x_t)$ vers un sommet $(y_1, \dots, y_t)$ de $D_{n,t}$ si et seulement si $(y_1, \dots, y_t) = (x_2, \dots, x_t, z)$ pour $z > x_t$. Notez que le graphe orienté $D_{n,1}$ est le graphe complet à n sommets numérotés de 1 à n, et donc les arcs (i, j) satisfont $i < j$.

Pour $t > 1$, le graphe $D_{n,t}$ peut être obtenu à partir de $D_{n,t-1}$ par une opération élémentaire, comme suit. Pour un graphe orienté $G = (V, E)$, on note $L(G)$ le graphe orienté dont les sommets sont les arcs de G, et tel qu'il y a un arc de e vers e' dans $L(G)$ si et seulement si l'arc e' est incident à e dans G (voir la figure 1 pour une exemple). Le graphe $L(G)$ est appelé graphe représentatif des arcs (*line graph* en anglais). Par définition, on a, pour tout $t > 1$,

$$D_{n,t} = L(D_{n,t-1}).$$

L'égalité ci-dessus va maintenant nous servir à borner le nombre chromatique de $D_{n,t}$. Cette borne est obtenu à partir de l'inégalité

$$\chi(G) \leq 2^{\chi(L(G))}$$

valide pour tout graphe orienté G. Soit en effet une coloration c des sommets de $L(G)$. On définit la coloration $c'(u)$ pour tout sommet u de G par

$$c'(u) = \{c(e) : e \text{ arc sortant de } u\}.$$

Puisque $1 \leq c(e) \leq \chi(L(G))$ pour tout arc e de G, on obtient que la coloration c' utilise au plus $2^{\chi(L(G))}$ couleurs. Par ailleurs, si u et v sont voisins dans G, par exemple v voisin sortant de u, alors la couleur $c(e) \in c'(u)$ où $e = (u, v)$. En revanche $c(e) \notin c'(v)$ car tous les arcs sortants de v sont voisins de e dans $L(G)$. La coloration c' est donc une $2^{\chi(L(G))}$-coloration propre de G.

Il suit de $D_{n,t} = L(D_{n,t-1})$ et $\chi(G) \leq 2^{\chi(L(G))}$ que $\chi(D_{n,t}) \geq \log \chi(D_{n,t-1})$. Notons maintenant que D_{2t+1} est un sous-graphe de $B_{n,t}$: tous les sommets et toutes les arêtes

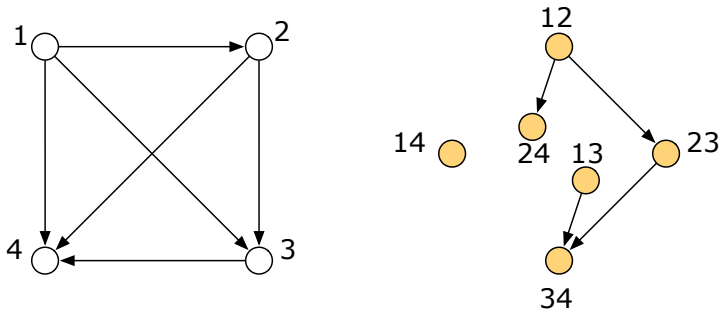

FIGURE 1 – *Une graphe orienté (à gauche) et son graphe représentatif des arcs (à droite)*

(arcs en fait) de D_{2t+1} apparaissent dans $B_{n,t}$. En conséquence, $\chi(B_{n,t}) \geq \chi(D_{n,2t+1})$. Enfin, $\chi(D_{n,1}) = \chi(K_n) = n$. Il découle que

$$\chi(B_{n,t}) \geq \log^{(2t)} n.$$

Ainsi, le plus petit t tel que $\chi(B_{n,t}) \leq 3$ satisfait $\log^{(2t)} n \leq 3$, et donc $\log^{(2t+2)} n < 1$. D'où $t \geq \frac{1}{2}\log^* n + 1$. Ceci conclut la preuve du théorème 2. □

2.3 Algorithmes déterministes de $(\Delta+1)$-COLORATION

Cette section décrit un certain nombre d'algorithmes déterministes de $(\Delta+1)$-coloration. Certains ont leur complexité exprimée en fonction du nombre de nœuds n du graphe et du degré maximum Δ de ces nœuds, alors que d'autres ont leur complexité exprimée uniquement en fonction du nombre de nœuds. Nous commençons par la première catégorie.

Dans cette sous-section, on supposera $\Delta \geq 2$ constant.

Un premier algorithme consiste à utiliser la stratégie développée dans l'algorithme de Cole et Vishkin, appliquée simultanément à tous les voisins à chaque étape. Après chaque étape de l'algorithme, la couleur d'un nœud u est un d-uplet $(c^{(1)}(u), \ldots, c^{(d)}(u))$ où $d = \deg(u)$. Pour $i = 1, \ldots, d$, la valeur de $c^{(i)}(u)$ évolue comme dans Cole et Vishkin par échange avec le $i$ème voisin de u.

pour un sommet u **:**
début
  $c(u) \leftarrow \mathrm{Id}(u)$
  pour $i = 1$ **à** N **faire**
    envoyer $c(u)$ aux voisins v_i de u, $i = 1, \ldots, d$
    recevoir $c(v_i)$ du voisin v_i, $i = 1, \ldots, d$
    pour $i = 1, \ldots, d$ **faire** $k_i \leftarrow$ plus petit indice j tel que $c_j(v_i) \neq c_j(u)$
    $c(u) \leftarrow ((c_{k_1}(u), k_1), \ldots, (c_{k_d}(u), k_d))$
fin.

Si la couleur de u est codée sur K bits avant une itération de la boucle « pour », alors cette couleur passe à au plus $f(K) = \Delta(\lceil \log K \rceil + 1)$ bits après cette itération. Pour une constante α suffisamment grande, on a $f(K) < K$ pour tout $K > \alpha\Delta\log\Delta$ avec $\Delta \geq 2$. En conséquence, la valeur N est maintenant choisie comme le plus petit i tel que $f^{(i)}(\lceil \log n \rceil) \leq \alpha\Delta\log\Delta$ où

$f : \mathbb{N} \to \mathbb{N}$ est la fonction $f(x) = \Delta(\lceil \log x \rceil + 1)$. Par le lemme 1, on obtient que le nombre d'itérations de l'algorithme ne dépasse pas $\log^* n + O(1)$.

Montrons que l'algorithme produit une $\Delta^{\alpha\Delta}$-coloration propre. Il suffit de montrer qu'il produit une coloration propre, car les couleurs en fin d'exécution sont évidemment sur au plus $\alpha\Delta \log \Delta$ bits. Comme précédemment pour l'anneau, on suppose que la coloration est propre avant une itération, et on montre qu'elle reste propre après l'itération. Soit v un voisin de u. Supposons que v soit le $i$ème voisin de u, et comparons les $i$ème blocs des couleurs de ces deux nœuds, $(c_{k_i}(u), k_i)$ et $(c_{k'_i}(v), k'_i)$. L'argumentation est la même que pour l'anneau : si $k_i \neq k'_i$ alors les couleurs de u et de v restent distinctes, et si $k_i = k'_i$, alors $c(u)$ et $c(v)$ diffèrent au k_i-ème bit, et donc $c_{k_i}(u) \neq c_{k'_i}(v)$, impliquant que les couleurs restent distinctes.

Pour passer de $\Delta^{\alpha\Delta}$ couleurs à $\Delta + 1$ couleurs, on utilise la réduction-couleur() comme pour l'anneau :

fonction réduction-couleur() **pour un sommet** u **:**
début
  pour $i = \Delta^{\alpha\Delta}$ **à** $\Delta + 1$ **faire**
    envoyer $c(u)$ aux voisins de u
    recevoir $c(v_j)$ des voisins v_j de u, $j = 1, \ldots, d$
    si $c(u) = i$ **alors** $c(u) \leftarrow \min\{c : c \in \{1, \ldots, \Delta + 1\} \setminus \{c(v_1), \ldots, c(v_d)\}\}$
fin.

On en conclut donc qu'il existe un algorithme $\mathcal{A}$ de $(\Delta + 1)$-coloration s'exécutant en $\Delta^{O(\Delta)} + \log^* n$ étapes dans le modèle $\mathcal{LOCAL}$. Cet algorithme suppose que les noeuds ont une connaissance a priori de n et de Δ. Il est possible de diminuer significativement la dépendance en Δ du nombre d'étapes. Nous montrons en effet le résultat suivant :

Théorème 3 *Il existe un algorithme $\mathcal{A}$ de $(\Delta+1)$-coloration s'exécutant en $O(\Delta^2 \log \Delta + \log^* n)$ étapes dans le modèle* LOCAL. *Cet algorithme suppose que les noeuds ont une connaissance a priori de n et de Δ.*

L'algorithme $\mathcal{A}$ est basé sur le lemme suivant :

Lemma 3 *Pour tout k et Δ avec $k > \Delta \geq 2$, il existe une famille $J = \{S_1, \ldots, S_k\}$ de k sous-ensembles de $\{1, \ldots, 5\lceil \Delta^2 \log k \rceil\}$ telle que pour tout $\Delta + 1$ sous-ensembles $F_0, F_1, \ldots, F_\Delta$ de J, on a $F_0 \not\subseteq \cup_{i=1}^{\Delta} F_i$.*

Preuve. Soit $m = 5\lceil \Delta^2 \log k \rceil$. Nous allons construire de façon probabiliste les sous-ensembles S_i de $\{1, \ldots, m\}$, pour $i = 1, \ldots, k$. Pour $i \in \{1, \ldots, k\}$ et $x \in \{1, \ldots, m\}$, on place x dans S_i avec probabilité $1/\Delta$. Ces placements sont effectués de manières mutuellement indépendantes. De cette construction résulte une collection (aléatoire) J de k sous-ensembles de $\{1, \ldots, m\}$. Soit $x \in \{1, \ldots, m\}$ et soit $F_0, F_1, \ldots, F_\Delta$ une collection quelconque de $\Delta + 1$ ensembles de J. On a

$$\Pr\{x \in F_0 \setminus \cup_{i=1}^{\Delta} F_i\} = \frac{1}{\Delta}(1 - \frac{1}{\Delta})^{\Delta} .$$

Or $\frac{1}{\Delta}(1 - \frac{1}{\Delta})^{\Delta} \geq \frac{1}{4\Delta}$ pour $\Delta \geq 2$. Donc

$$\Pr\{x \in F_0 \setminus \cup_{i=1}^{\Delta} F_i\} \geq \frac{1}{4\Delta} .$$

On a $F_0 \not\subseteq \cup_{i=1}^{\Delta} F_i$ si et seulement si il existe $x \in F_0 \setminus \cup_{i=1}^{\Delta} F_i$. Dit autrement, on a $F_0 \subseteq \cup_{i=1}^{\Delta} F_i$ si et seulement si, pour tout $x \in \{1, \ldots, m\}$, on a $x \notin F_0 \setminus \cup_{i=1}^{\Delta} F_i$. Il découle de la dernière inégalité ci-dessus que

$$\Pr\{x \notin F_0 \setminus \cup_{i=1}^{\Delta} F_i\} \leq 1 - \frac{1}{4\Delta} .$$

On a donc, par indépendance des choix $x \in S_i$,

$$\Pr\{F_0 \subseteq \cup_{i=1}^{\Delta} F_i\} \leq (1 - \frac{1}{4\Delta})^m .$$

Par sous-linéarité de la probabilité ($\Pr\{A \cup B\} \leq \Pr\{A\} + \Pr\{B\}$), on obtient que la probabilité qu'il existe une collection $F_0, F_1, \ldots, F_\Delta$ de $\Delta + 1$ ensembles de J telle que $F_0 \subseteq \cup_{i=1}^{\Delta} F_i$ est au plus

$$N\ (1 - \frac{1}{4\Delta})^m$$

où N dénote le nombre de choix possibles pour la collection $F_0, F_1, \ldots, F_\Delta$, et pour le choix de F_0 dans cette collection. On obtient donc que la probabilité qu'il existe une collection $F_0, F_1, \ldots, F_\Delta$ de $\Delta + 1$ ensembles de J telle que $F_0 \subseteq \cup_{i=1}^{\Delta} F_i$ est au plus

$$(\Delta + 1)\binom{k}{\Delta + 1}(1 - \frac{1}{4\Delta})^m .$$

Une analyse basée sur l'estimation de $\binom{k}{\Delta+1}$ utilisant la formule de Stirling permet de vérifier que, pour $m = 5\lceil \Delta^2 \log k \rceil$, on a $(\Delta+1)\binom{k}{\Delta+1}(1-\frac{1}{4\Delta})^m < 1$. Ainsi, la probabilité que l'ensemble J ne satisfasse pas l'énoncé du lemme est strictement inférieure à 1. Il existe donc au moins un ensemble J satisfaisant l'énoncé du lemme. □

Preuve du théorème 3. L'algorithme $\mathcal{A}$ procède en $O(\log^* n)$ étapes. Initialement, l'intervalle des couleurs utilisées est $[1, n]$ et le nœud d'identifiant i prend comme couleur initiale la couleur i. A chaque étape, cet intervalle est réduit de $[1, k]$ à $[1, 5\lceil \Delta^2 \log k \rceil]$, tout en maintenant une coloration propre. A cette fin, soit $J = \{S_1, \ldots, S_k\}$ une collection de sous-ensembles de $\{1, \ldots, 5\lceil \Delta^2 \log k \rceil\}$ satisfaisant le lemme 3. Les noeuds de couleur $i \in \{1, \ldots, k\}$ prennent $S_i \subseteq \{1, \ldots, 5\lceil \Delta^2 \log k \rceil\}$ comme étiquette. Soit u un noeud de couleur $i \in \{1, \ldots, k\}$. Les $d \leq \Delta$ voisins de u ont des couleurs différentes de i, et ont donc comme étiquettes des ensembles $S_j \subseteq \{1, \ldots, 5\lceil \Delta^2 \log k \rceil\}$, $j \neq i$. Soit $S_{j_1}, \ldots, S_{j_d}$ ces d étiquettes (avec potentiellement des répétitions). On a $S_i \not\subseteq \cup_{r=1}^{d} S_{j_r}$. Le noeuds i prend comme nouvelle couleur le plus petit x tel que $x \in S_i \setminus \cup_{r=1}^{d} S_{j_r}$. Comme $x \notin S_{j_r}$ pour tout $r = 1, \ldots, d$, les voisins de u prennent des couleurs différentes de x. La nouvelle coloration est donc propre, et le nombre de couleurs utilisées a été réduit de k à $5\lceil \Delta^2 \log k \rceil$. Chaque étape induit une réduction logarithmique du nombre de couleurs. D'après le lemme 1, le nombre de couleurs après $\log^* n + O(1)$ étapes est au plus $10\lceil \Delta^2 \log \Delta \rceil$. Lorsque $k \leq 10\lceil \Delta^2 \log \Delta \rceil$, on n'a plus nécessairement $5\lceil \Delta^2 \log k \rceil < k$, et donc le nombre de couleur ne décroît plus nécessairement. On réduit donc le nombre de couleurs à $\Delta + 1$ par application de la fonction réduction-couleur() en $O(\Delta^2 \log \Delta)$ étapes. Ceci termine la preuve du théorème. □

A ce jour, le meilleur algorithme de $(\Delta + 1)$-coloration connu dont la complexité s'exprime en n et Δ s'exécute en $\tilde{O}(\sqrt{\Delta}) + \log^* n$ étapes dans le modèle LOCAL. En l'absence de borne inférieure, on ne sait pas s'il est possible de développer un algorithme de $(\Delta + 1)$-coloration s'exécutant en $O(\Delta^{\frac{1}{2}-\epsilon}) + \log^* n$ étapes avec $\epsilon > 0$.

Le meilleur algorithme de $(\Delta+1)$-coloration connu dont la complexité s'exprime uniquement en n s'exécute en $2^{\sqrt{\log n}}$ étapes dans le modèle LOCAL. Cette fonction de n croît moins vite que n'importe quel polynôme n^ϵ, quelque soit $\epsilon > 0$ arbitrairement petit, mais plus vite que n'importe quel polylogarithme $\log^c n$, quelque soit $c > 0$ arbitrairement grand. Dans la section suivante, nous décrivons un algorithme *probabiliste* de $(\Delta + 1)$-coloration s'exécutant en un nombre polylogarithmique d'étapes, avec forte probabilité.

Exercice 1 *Montrer qu'il existe un algorithme de 3-colorations des arbres enracinés dans lesquels les noeud ont une notion consistante de haut et de bas, s'exécutant en $O(\log^* n + O(1))$ étapes.*

2.4 Un algorithme probabiliste de $(\Delta + 1)$-COLORATION

L'algorithme probabiliste présenté ci-après est algorithme de type Las Vegas, au sens où il produit nécessairement une $(\Delta + 1)$-coloration, mais son nombre d'étapes est une variable aléatoire. La garantie d'obtenir une coloration en un faible nombre d'étapes est toutefois forte : la probabilité que le nombre d'étapes soit au plus $O(\log^2 n)$ est au moins $1-O(\frac{1}{n})$. On dit d'une telle garantie qu'elle est *avec forte probabilité*, abrégé en « a.f.p. ». Notons qu'il existe un algorithme probabiliste s'exécutant, a.f.p., en $O(\log n)$ étapes. Son analyse est toutefois significativement plus compliquée que celle de l'algorithme présenté ci-dessous, quoiqu'il soit basé sur les mêmes idées conceptuelles.

2.4.1 Equivalence $(\Delta + 1)$-coloration et ensemble indépendant maximal (MIS)

Le problème de coloration est très lié à celui de la construction d'un ensemble indépendant maximal (« maximal independent set » (MIS) en anglais). Etant donné $G = (V, E)$, un ensemble indépendant dans G est un ensemble $I \subseteq V$ tels que toute paire quelconque de nœuds de I ne sont pas adjacent. I est maximal s'il est maximal pour l'inclusion. Le problème MIS est donc définit comme suit.

Le problème ENSEMBLE INDÉPENDANT MAXIMAL (MAXIMAL INDEPENDENT SET – MIS) :

Entrée : un réseau $G = (V, E)$;

Objectif : chaque nœud $u \in V$ retourne une valeur $mis(u) \in \{0, 1\}$ telle que pour tout $u \in V$:
- $mis(u) = 1 \Rightarrow mis(v) = 0$ pour tout $v \in V$ tel que $\{u, v\} \in E$,
- $mis(u) = 0 \Rightarrow$ il existe $v \in V$ tel que $\{u, v\} \in E$ et $mis(v) = 1$.

La connexion entre coloration et MIS est établi par le lemme suivant.

Lemma 4 *Pour tout algorithme $\mathcal{M}$ de MIS, il existe un algorithme $\mathcal{C}$ de $(\Delta + 1)$-coloration tel que, si $\mathcal{M}$ s'exécute en $t(n, \Delta)$ étapes dans tout graphe de n nœuds et degré maximum Δ, alors $\mathcal{C}$ s'exécute en $t((\Delta + 1)n, 2\Delta)$ étapes dans tout graphe de n nœuds et degré maximum Δ.*

Preuve. La transformation de $\mathcal{M}$ en $\mathcal{C}$ s'effectue comme suit. Chaque nœud u simule le comportement des nœuds d'une clique K_u de taille $\Delta + 1$. (Voir la figure 2). Plus spécifiquement, u donne aux nœuds virtuels de K_u les identifiants $\mathrm{Id}(u).i$ pour $i = 1, \ldots, \Delta + 1$. Chaque nœud virtuel $\mathrm{Id}(u).i$ est (virtuellement) connecté aux sommets $\mathrm{Id}(v).i$ pour tous les voisin v de u dans

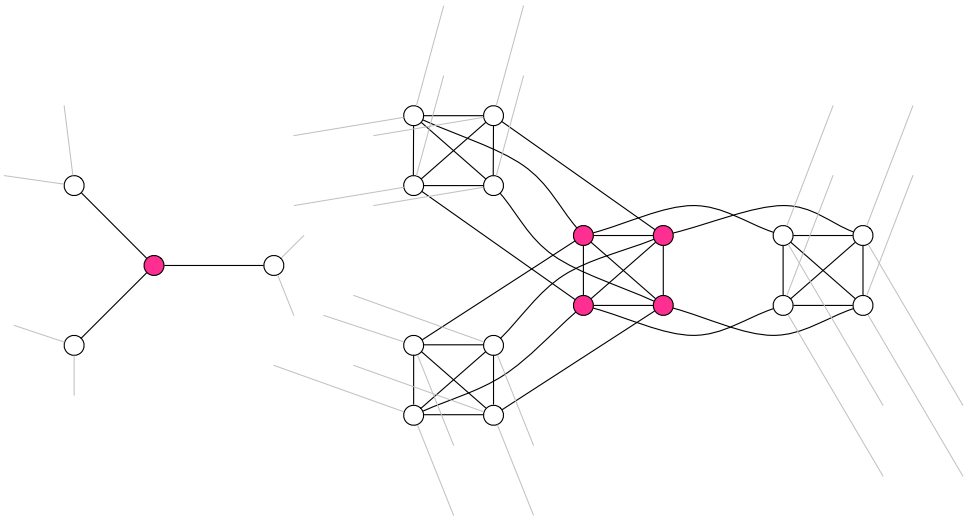

FIGURE 2 – *Réduction de* MIS *à* $(\Delta+1)$-COLORATION

G. $\mathcal{C}$ simule le comportement de $\mathcal{M}$ sur le graphe G' des nœuds et liens virtuels. Remarquons qu'un MIS I dans G' inclut un et un seul nœud virtuel par clique. En effet, il ne peut pas y avoir plus d'un membre de I par clique, et comme chaque clique K_u contient $\Delta+1$ nœuds, au moins un d'entre eux est non adjacent aux sommets de I dans les cliques K_v adjacentes à K_u. Ainsi, si $\mathcal{M}$ place le nœud virtuel $\mathrm{Id}(u).i$ dans le MIS, le nœud u prend la couleur i. □

Exercice 2 *Montrer que pour tout algorithme $\mathcal{C}$ de $(\Delta+1)$-coloration, il existe un algorithme $\mathcal{M}$ de MIS tel que, si $\mathcal{C}$ s'exécute en t étapes dans un graphe G, alors $\mathcal{M}$ s'exécute en $t+\Delta-1$ étapes dans G.*

2.4.2 Algorithme probabiliste pour la $(\Delta+1)$-coloration

Algorithme distribué de $(\Delta+1)$-coloration pour un sommet u :
début
$c(u) \leftarrow \perp$
$C(u) \leftarrow \emptyset$
tant que $c(u) = \perp$ **faire**
choisir une couleur $\ell(u) \in \{0,1,\ldots,\Delta+1\} \setminus C(u)$ avec
$\Pr[\ell(u)=0] = \frac{1}{2}$, et $\Pr[\ell(u)=\ell] = \frac{1}{2(\Delta+1-|C(u)|)}$ pour $\ell \in \{1,\ldots,\Delta+1\}\setminus C(u)$
envoyer $\ell(u)$ aux voisins et recevoir la couleur $\ell(v)$ de chaque voisin v
si $\ell(u) \neq 0$ et $\ell(v) \neq \ell(u)$ pour tout voisin v **alors** $c(u) \leftarrow \ell(u)$ **sinon** $c(u) \leftarrow \perp$
envoyer $c(u)$ aux voisins et recevoir la couleur $c(v)$ de chaque voisin v
ajouter à $C(u)$ les couleurs des voisins v tels que $c(v) \neq \perp$
fin.

Théorème 4 *Algorithme distribué de $(\Delta+1)$-coloration est un algorithme probabiliste de Las Vegas s'exécutant, a.f.p., en $O(\log n)$ étapes.*

Preuve. Soit u un sommet quelconque. Montrons que, à chaque étape, u à une probabilité au moins $\frac{1}{4}$ de terminer. Soit $N(u)$ l'ensemble des voisins de u. Rappelons que, par définition,

$\Pr[A \mid B] = \Pr[A \wedge B]/\Pr[B]$, et donc $\Pr[A \wedge B] = \Pr[A \mid B] \cdot \Pr[B]$. On en déduit :

$$\begin{aligned}\Pr[u \text{ termine}] &= \Pr[\ell(u) \neq 0 \text{ et aucun } v \in N(u) \text{ satisfait } \ell(v) = \ell(u)] \\ &= \Pr[\forall v \in N(u), \ell(v) \neq \ell(u) \mid \ell(u) \neq 0] \cdot \Pr[\ell(u) \neq 0] \\ &= \frac{1}{2} \cdot \Pr[\forall v \in N(u), \ell(v) \neq \ell(u) \mid \ell(u) \neq 0]\end{aligned}$$

Rappelons également que $\Pr[A] = \Pr[A \mid B] \cdot \Pr[B] + \Pr[A \mid \bar{B}] \cdot \Pr[\bar{B}]$. Soit $v \in N(u)$ n'ayant pas encore terminé. On a

$$\begin{aligned}\Pr[\ell(v) = \ell(u) \mid \ell(u) \neq 0] &= \Pr[\ell(v) = \ell(u) \mid \ell(u) \neq 0 \wedge \ell(v) = 0]\ \Pr[\ell(v) = 0] \\ &\quad + \Pr[\ell(v) = \ell(u) \mid \ell(u) \neq 0 \wedge \ell(v) \neq 0]\ \Pr[\ell(v) \neq 0] \\ &= \Pr[\ell(v) = \ell(u) \mid \ell(u) \neq 0 \wedge \ell(v) \neq 0]\ \Pr[\ell(v) \neq 0] \\ &\leq \frac{1}{2}\ \Pr[\ell(v) = \ell(u) \mid \ell(u) \neq 0 \wedge \ell(v) \neq 0] \\ &= \frac{1}{2}\ \frac{1}{\Delta + 1 - |C(u)|}.\end{aligned}$$

En conséquence,

$$\Pr[\exists v \in N(u) : \ell(v) = \ell(u) \mid \ell(u) \neq 0] \leq (\Delta - |C(u)|)\frac{1}{2(\Delta + 1 - |C(u)|)} < \frac{1}{2}$$

Donc $\Pr[u \text{ termine}] > \frac{1}{4}$. Ainsi, la probabilité que u n'ait pas terminé après $k \ln n$ étapes est au plus $\left(\frac{3}{4}\right)^{k \ln n} = e^{k \ln n \ln 3/4} = n^{k \ln 3/4} = n^{-k \ln 4/3}$. Par sous-linéarité des probabilités, on obtient donc que la probabilité qu'il existe un sommet u qui n'ait pas terminé après $k \ln n$ étapes est au plus $n^{1-k \ln 4/3}$. Soit $c \geq 1$. En prenant $k = \frac{1+c}{\ln 4/3}$, on obtient qu'avec probabilité $1 - 1/n^c$, tous les sommets ont terminé après $k \ln n$ étapes. □

2.4.3 Algorithmes probabilistes pour la construction d'un MIS

Nous allons donc maintenant présenter un algorithme distribué de MIS. Dans l'algorithme ci-dessous, chaque nœud u possède une variable $mis(u) \in \{-1, 0, 1\}$ de valeur initiale -1. A la fin de l'algorithme, on a $mis(u) \in \{0, 1\}$ où $mis(u) = 1$ (resp., $mis(u) = 0$) si u a joint le MIS (resp., n'a pas joint le MIS). L'idée générale de l'algorithme ci-dessous est due à Luby.

a) Algorithme de Luby. L'algorithme de Luby procède par phase. A chaque phase, chaque nœud u se propose pour joindre le MIS avec probabilité de l'ordre de $1/\deg(u)$, ce qui, intuitivement, permet d'équilibrer entre voisins la probabilité de se proposer. Cet algorithme repose sur un ordre entre les sommets. Pour toute paire de sommet $u \neq v$, on pose

$$v \succ u \iff \deg(v) > \deg(u) \vee (\deg(v) = \deg(u) \wedge \mathrm{Id}(v) > \mathrm{Id}(u))$$

Algorithme distribué de Luby pour le calcul d'un MIS
– code d'un sommet u tel que $mis(u) = -1$:
début
(1) **si** $\deg_H(u) = 0$ **alors** $mis(u) \leftarrow 1$
sinon
$join(u) \leftarrow true$ avec probabilité $\frac{1}{2 \deg_H(u)}$ (sinon $join(u) = false$)

envoyer $join(u)$ aux voisins de u dans H
recevoir $join(v)$ de tous les voisins v de u dans H
si $join(u) = true$ et $\nexists v \in N(u) : v \succ u \wedge join(v) = true$ **alors** $mis(u) \leftarrow 1$
envoyer $mis(u)$ aux voisins de u dans H
recevoir $mis(v)$ de tous les voisins v de u dans H
si $mis(u) \neq 1$ et $\exists v \in N(u) : mis(v) = 1$ **alors** $mis(u) \leftarrow 0$
envoyer $mis(u)$ aux voisins de u dans H
recevoir $mis(v)$ de tous les voisins v de u dans H

fin.

Théorème 5 *L'algorithme probabiliste Las Vegas de Luby pour le calcul d'un MIS s'exécute, avec forte probabilité, en $O(\log n)$ étapes dans le modèle* LOCAL.

Preuve. Soit u un sommet quelconque. On a

$$\begin{aligned}
\Pr[mis(u) \neq 1 \mid join(u)] &= \Pr[\exists v \in N(u) : v \succ u \wedge join(v) \mid join(u)] \\
&= \Pr[\exists v \in N(u) : v \succ u \wedge join(v)] \\
&\leq \sum_{v \in N(u) : v \succ u} \Pr[join(v)] \\
&= \sum_{v \in N(u) : v \succ u} \frac{1}{2\deg(v)} \\
&\leq \sum_{v \in N(u) : v \succ u} \frac{1}{2\deg(u)} \\
&\leq \frac{\deg(u)}{2\deg(u)} \\
&\leq \frac{1}{2}
\end{aligned}$$

En conséquence, comme $\Pr[mis(u) = 1] = \Pr[mis(u) = 1 \mid join(u)] \cdot \Pr[join(u)]$, on obtient

$$\Pr[mis(u) = 1] \geq \frac{1}{2} \cdot \frac{1}{2\deg(u)} = \frac{1}{4\deg(u)}. \tag{1}$$

Un sommet u est dit *grand* si

$$\sum_{v \in N(u)} \frac{1}{2\deg(v)} \geq \frac{1}{6}$$

Un sommet qui n'est pas grand est dit *petit*. Intuitivement, un grand sommet a beaucoup de voisins de petit degré. C'est donc un sommet qui a beaucoup de chance qu'un de ses voisins entre dans le MIS. Nous allons effectivement montrer qu'un grand sommet a une probabilité au moins 1/36 d'avoir un de ses voisins qui entre dans le MIS ou de rentrer lui-même dans le MIS. Notons que, d'après Eq. (1), cela est vrai si u a un voisin v tel que $\deg(v) \leq 2$. On considère donc un grand sommet u dont tous ses voisins $v \in N(u)$ ont degré au moins 3, et satisfont donc

$$\frac{1}{2\deg(v)} \leq \frac{1}{6}$$

Comme u est grand, il existe donc un sous-ensemble de sommets $S \subseteq N(u)$ tel que

$$\frac{1}{6} \leq \sum_{v \in S} \frac{1}{2\deg(v)} \leq \frac{1}{3}$$

On en déduit que

$$\begin{aligned}\Pr[mis(u) \neq -1] &\geq \Pr[\exists v \in S : mis(v) = 1] \\ &\geq \sum_{v \in S} \Pr[mis(v) = 1] - \sum_{v,w \in S, v \neq w} \Pr[mis(v) = 1 \wedge mis(w) = 1].\end{aligned}$$

où la seconde inégalité provient du principe d'inclusion-exclusion :

$$\Pr[E_1 \vee E_2 \vee \cdots \vee E_r] = \sum_i \Pr[E_i] - \sum_{i \neq j} \Pr[E_i \wedge E_j] + \sum_{i \neq j \neq k} \Pr[E_i \wedge E_j \wedge E_k] - \ldots \\ \cdots + (-1)^{r+1} \Pr[E_1 \wedge \cdots \wedge E_r].$$

On obtient donc

$$\begin{aligned}\Pr[mis(u) \neq -1] &\geq \sum_{v \in S} \Pr[mis(v) = 1] - \sum_{v,w \in S, v \neq w} \Pr[join(v) \wedge join(w)] \\ &\geq \sum_{v \in S} \Pr[mis(v) = 1] - \sum_{v \in S} \sum_{w \in S} \Pr[join(v)] \cdot \Pr[join(w)] \\ &\geq \sum_{v \in S} \frac{1}{4 \deg(v)} - \sum_{v \in S} \sum_{w \in S} \frac{1}{2 \deg(v)} \cdot \frac{1}{2 \deg(w)} \\ &\geq \left(\sum_{v \in S} \frac{1}{2 \deg(v)} \right) \left(\frac{1}{2} - \sum_{w \in S} \frac{1}{2 \deg(w)} \right) \\ &\geq \frac{1}{6} \left(\frac{1}{2} - \frac{1}{3} \right) = \frac{1}{36}.\end{aligned}$$

Ainsi, les grands sommets ont une probabilité constante de terminer à chaque étapes. Malheureusement, le nombre de grands sommets n'est pas forcément élevé à chaque étape, et on ne peut pas directement appliquer le même raisonnement que dans le cas de l'algorithme de coloration. Pour contourner cet obstacle, on considère les arêtes plutôt que les sommets, de la façon suivante.

On dit qu'une arête est *grande* si au moins une de ses extrémités est grande. Par ailleurs, chaque arête $\{u, v\}$ avec $u \prec v$ est orientée de u vers v. Rappelons que $\deg^+(u)$ et $\deg^-(u)$ dénotent respectivement le nombre de voisins v de u tels que l'arête $\{u, v\}$ est orientée de u vers v, et de v vers u. Avec l'orientation définie ci-dessus, tout petit sommet u vérifie

$$\deg^+(u) \geq 2 \ \deg^-(u).$$

En effet, sinon, il existerait un petit sommet u tel que $\deg^+(u) < 2 \ \deg^-(u)$, c'est-à-dire tel que $\deg(u) < 3 \ \deg^-(u)$. En ce cas, soit $S = \{v \in N(u) : \deg(v) \leq \deg(u)\}$, impliquant $|S| \ \geq \deg^-(u)$, et donc $|S| \geq \frac{1}{3}|N(u)|$. De l'existence d'un petit sommet u tel que $\deg^+(u) < 2 \ \deg^-(u)$, il découlerait que

$$\sum_{v \in N(u)} \frac{1}{2 \deg(v)} \geq \sum_{v \in S} \frac{1}{2 \deg(v)} \geq \sum_{v \in S} \frac{1}{2 \deg(u)} \geq \frac{\deg(u)}{3} \cdot \frac{1}{2 \deg(u)} = \frac{1}{6},$$

ce qui impliquerait que u est grand. Donc tout petit sommet u vérifie $\deg^-(u) \leq \frac{1}{2} \deg^+(u)$. Il en découle que

$$\sum_{u \text{ petit}} \deg^-(u) \leq \frac{1}{2} \sum_{u \text{ petit}} \deg^+(u) \leq \frac{m}{2}$$

où m désigne le nombre d'arête du graphe résiduel au début de l'itération. En conséquence,

$$\sum_{u \text{ grand}} \deg^-(u) \geq \frac{m}{2},$$

et donc au moins la moitié des arêtes sont grandes. Par ailleurs, nous avons vu que, si $u \in V$ est grand, alors $\Pr[mis(u) \neq -1] \geq \frac{1}{36}$. Donc, pour chaque arête grande, soit X_e la variable de Bernouilli égale à 1 si e disparait de H durant une phase de l'algorithme. On a $\Pr[X_e = 1] \geq \frac{1}{36}$, et donc $\mathbb{E}X_e \geq \frac{1}{36}$. Soit X la variable aléatoire égale au nombre d'arêtes éliminées à une phase. On a $X \geq \sum_{e \text{ grande}} X_e$, et donc, comme $\mathbb{E}X = \sum_{e \text{ grande}} \mathbb{E}X_e$, on obtient $\mathbb{E}X \geq \frac{m}{72}$. Ainsi, en moyenne, une fraction constante des arêtes restantes sont éliminées à chaque étapes.

Soit $p = \Pr[X \leq \frac{1}{2}\,\mathbb{E}X]$. On a

$$\mathbb{E}X = \sum_{x=0}^{m} x\ \Pr[X = x] = \sum_{x=0}^{\frac{1}{2}\mathbb{E}X} x\ \Pr[X = x] + \sum_{x=\frac{1}{2}\,\mathbb{E}X+1}^{m} x\ \Pr[X = x] \leq \tfrac{1}{2}\, p\, \mathbb{E}X + (1-p)m$$

d'où

$$p \leq \frac{m - \mathbb{E}X}{m - \frac{1}{2}\mathbb{E}X} \leq \frac{m - \frac{1}{2}\mathbb{E}X}{m} \leq 1 - \frac{1}{144}.$$

Donc, si $\mathcal{E}$ dénote l'évènement « au moins $\frac{m}{144}$ arêtes sont éliminées en une itération », on a

$$\Pr[\mathcal{E}] \geq \frac{1}{144}.$$

Soit $\alpha = \frac{144}{143}$. Soit X_i, $i = 1, \ldots, k$, une suite de variables aléatoires de Bernouilli mutuellement indépendantes, de probabilité $q = \frac{1}{144}$, avec $k = c \log_\alpha n$, pour c constante dont nous allons fixer la valeur ultérieurement. Soit $Y = \sum_{i=1}^{k} X_i$. Notez que si $Y \geq \log_\alpha m$, où m denote le nombre d'arête du graphe original G sur lequel l'algorithme s'exécute, alors cet algorithme termine car toutes les arêtes auront disparu. Nous allons montrer que pour $c > 0$ suffisamment grand, l'évènement $Y \geq \log_\alpha m$ est vrai avec forte probabilité. On a $\mathbb{E}Y = \sum_{i=1}^{k} \mathbb{E}X_i = kq$. L'inégalité de Chernoff stipule que, pour tout $\delta \in]0, 1[$,

$$\Pr[Y \leq (1-\delta)\mathbb{E}Y] \leq e^{-\frac{1}{2}\delta^2 \mathbb{E}Y}.$$

Dans notre cas de figure, avec $\delta = \frac{1}{2}$, on obtient :

$$\Pr[Y \leq \tfrac{kq}{2}] \leq e^{-\frac{kq}{8}},$$

c'est-à-dire

$$\Pr[Y \leq \tfrac{cq \log_\alpha n}{2}] \leq e^{-\frac{cq \log_\alpha n}{8}}.$$

Soit $c \geq 4/q$. De ce choix, il découle que $\frac{cq \log_\alpha n}{2} \geq \log_\alpha m$ car $m \leq n^2$. Il découle également que $cq \geq 8 \ln \alpha$. En conséquence,

$$e^{-\frac{cq \log_\alpha n}{8}} = \frac{1}{n^{\frac{cq}{8 \ln \alpha}}} \leq \frac{1}{n}.$$

Donc,

$$\Pr[Y \leq \log_\alpha m] \leq \frac{1}{n}.$$

Donc, avec probabilité $1 - \frac{1}{n}$, on a $Y \geq \log_\alpha m$. L'algorithme termine donc en $O(\log n)$ étapes avec forte probabilité, i.e., probablilité $1 - \frac{1}{n}$. □

b) Algorithme de Peleg. L'algorithme de Luby ci-dessus repose sur un ordre sur les sommets utilisant les identifiants des sommets. L'algorithme présenté dans [4] ne nécessite pas un tel ordre, et peut s'exécuter dans un réseau *anonyme*, c'est-à-dire en l'absence d'identifiants. Cet algorithme garde la même idée générale que celui de Luby : chaque nœud u propose de joindre le MIS avec probabilité $1/\mathrm{d}(u)$ où $\mathrm{d}(u)$ dénote le degré maximum des nœuds à distance au plus 2 de u.

Comme nous l'avons dit, l'algorithme ci-dessous procède par phase. Soit U_t l'ensemble des nœuds n'ayant pas encore décidé de joindre ou pas le MIS au début de la phase t. C'est-à-dire

$$U_t = \{u \in V : mis(u) = -1 \text{ au début de la phase } t\}.$$

En particulier $U_1 = V$. (U est pour « undecided »). On note alors $H_t = G[U_t]$ le sous-graphe de G induit par les nœuds de U_t. Le code ci-dessous décrit la phase t de l'algorithme de MIS. Seuls le nœuds de H_t sont concernés, les autres ayant fixé leur variable mis à 0 ou 1 lors d'une étape précédente. L'indice t de U et H est omis pour faciliter la lecture. On note

$$N_d(u) = \{v \in V(H) : \mathrm{dist}_H(u, v) \leq d\}.$$

Algorithme distribué de Peleg pour le calcul d'un MIS – code d'un sommet u tel que $mis(u) = -1$:
début
(1) **si** $\deg_H(u) = 0$ **alors** $mis(u) \leftarrow 1$
sinon
envoyer $\deg_H(u)$ aux sommets de $N_2(u)$ */* deux étapes de communication */*
recevoir $\deg_H(v)$ de tous les sommets $v \in N_2(u)$
$\mathrm{d}(u) \leftarrow \max_{v \in N_2(u)} \deg_H(v)$
$p(u) \leftarrow 1/\mathrm{d}(u)$
$join(u) \leftarrow true$ avec probabilité $p(u)$ (sinon $join(u) = false$)
envoyer $join(u)$ aux voisins de u dans H
recevoir $join(v)$ de tous les voisins v de u dans H
si $join(u) = 1$ **alors**
(2) **si** $join(v) = 0$ pour tous les sommets $v \in N_1(u)$ **alors** $mis(u) \leftarrow 1$
envoyer $mis(u)$ aux voisins de u dans H
recevoir $mis(v)$ de tous les voisins v de u dans H
(3) **si** au moins un voisin v de u dans H satisfait $mis(v) = 1$ **alors** $mis(u) \leftarrow 0$
envoyer $mis(u)$ aux voisins de u dans H
recevoir $mis(v)$ de tous les voisins v de u dans H
fin.

Correction. Pour montrer que l'algorithme ci-dessus est correct, soit

$$I_t = \{u \in V : mis(u) = 1 \text{ au début de la phase } t\}.$$

En particulier $I_1 = \emptyset$. Nous allons montrer que, pour tout $t \geq 1$, l'ensemble I_t forme un MIS dans $K_t = G[V \setminus U_t]$. En fait, nous allons montrer un peu plus, pour faciliter la récurrence. Nous allons montrer la propriété $P(t)$ qui stipule que l'ensemble I_t forme un MIS dans $K_t = G[V \setminus U_t]$, et que les nœuds de $V \setminus U_t$ ayant un voisin dans U_t satisfont $mis = 0$ (c'est-à-dire ne sont pas dans I_t). $P(t)$ est vraie pour $t = 1$. Supposons que $P(t)$ est vraie pour t, et considérons la phase t, dont il résultera I_{t+1}.

Observons que $mis(u)$ passe à 1 lors de la phase t uniquement aux instructions (1) ou (2). Ces nœuds rentrent dans I_{t+1}. Dans les deux cas, tous les voisins v de u dans H_t satisfont $join(v) = 0$, et donc il n'y a pas de conflit avec les voisins dans H_t. L'absence de conflit avec des voisins dans K_t est assurée par le fait que les nœuds de $V \setminus U_t$ ayant un voisin dans U_t satisfont $mis = 0$. Par ailleurs, l'instruction (3) stipule que lorsqu'un nœud se place dans I_{t+1}, il force tous ses voisins à décider de ne pas se placer dans I_{t+1}. En particulier, cela assure que tous les nœuds de K_{t+1} ayant un voisin dans H_{t+1} ne sont pas dans le MIS. Pour s'assurer de $P(t+1)$, il ne s'agit plus que de vérifier la condition de maximalité, c'est-à-dire qu'un nœud u satisfaisant $mis(u) = 0$ a au moins un voisin dans satisfaisant $mis(v) = 1$. Cela découle directement de l'instruction (3) qui stipule justement que mis ne passe à 0 que si un voisin entre dans le MIS.

Temps d'exécution. Afin de mesurer le temps d'exécution de l'algorithme, nous allons d'abord étudier la probabilité d'un nœud encore indécis au début d'une phase à décider durant cette phase. A cette fin, on se place dans le cas où le degré maximum Δ de H est non nul, car si $\Delta = 0$ alors tous les nœuds de H joignent le MIS par l'instruction (1). Définissons l'évènement $\mathcal{E}_{u,v}$ pour une arête $\{u, v\}$ de H par

$$\mathcal{E}_{u,v} = \text{« } join(v) = 1 \text{ et } join(w) = 0 \text{ pour tout } w \in N_1(u) \cup N_1(v) \setminus \{v\} \text{ ».}$$

Notons que si $\mathcal{E}_{u,v}$ est vrai, alors v se place dans le MIS et u se place en dehors du MIS, et que donc les deux nœuds décident. Notons également que, pour deux voisins distincts v et v' de u, les évènements $\mathcal{E}_{u,v}$ et $\mathcal{E}_{u,v'}$ sont disjoints. Soit alors

$$\mathcal{E}_u = \bigcup_{v \in N_1(u)\setminus\{u\}} \mathcal{E}_{u,v}.$$

Par définition, si $\mathcal{E}_u$ est vrai, alors u se place en dehors du MIS et exactement un voisin de u se place dans le MIS. L'observation suivante indique que si le degré d'un nœud dans H est élevé, alors il a une probabilité constante de décider de se placer hors du MIS.

Observation 1 *Soit Δ le degré maximum de H. Pour tout u tel que $\deg_H(u) \geq \Delta/2$, on a $\Pr[\mathcal{E}_u] \geq \frac{1}{2e^4}$ où e est la constante de Néper ($\ln(e) = 1$).*

Preuve. On a

$$\begin{aligned}
\Pr[\mathcal{E}_{u,v}] &= p(v) \prod_{w \in N_1(u) \cup N_2(v) \setminus \{v\}} (1 - p(w)) \\
&= \frac{1}{\mathrm{d}(v)} \prod_{w \in N_1(u) \cup N_2(v) \setminus \{v\}} (1 - \frac{1}{\mathrm{d}(w)}) \\
&\geq \frac{1}{\Delta} \prod_{w \in N_1(u) \cup N_2(v) \setminus \{v\}} (1 - \frac{2}{\Delta}) \qquad \text{car } \mathrm{d}(v) \leq \Delta \text{ et } \mathrm{d}(w) \geq \Delta/2 \\
&= \frac{1}{\Delta}(1 - \frac{2}{\Delta})^{|N_1(u) \cup N_2(v) \setminus \{v\}|} \\
&\geq \frac{1}{\Delta}(1 - \frac{2}{\Delta})^{2\Delta - 1} \\
&\geq \frac{1}{\Delta}\frac{1}{e^4}
\end{aligned}$$

Or, puisque $\mathcal{E}_{u,v}$ et $\mathcal{E}_{u,v'}$ sont disjoints pour $v \neq v'$, il découle que

$$\Pr[\mathcal{E}_u] = \sum_{v \in N_1(u)\setminus\{u\}} \Pr[\mathcal{E}_{u,v}].$$

En conséquence

$$\Pr[\mathcal{E}_u] \geq \deg_H(u)\frac{1}{\Delta}\frac{1}{e^4} \geq \frac{1}{2e^4}.$$

□

Au vu de l'observation ci-dessus, étudions le comportement de degré des nœuds de H_t en fonction de t. A cette fin, pour $i \geq 0$, définissons

$$M_i(t) = \{u \in U_t : \deg_{H_t} \geq n/2^i\}.$$

Soit $k = 6e^4 \ln n$. Nous allons étudier l'évolution des degrés non pas à chaque phase, mais toutes les k phases. Nous nous focalisons donc sur les phases $t_i = ki$, pour $i = 0, 1, 2, \ldots$ Considérons l'évènement

$$\mathcal{F}_i = \text{« } M_i(t_i) = \emptyset \text{ »}.$$

Notons que $\mathcal{F}_0$ est toujours vrai.

Observation 2 *Pour tout $i \geq 0$, on a $\Pr[\mathcal{F}_{i+1} \mid \mathcal{F}_i] \geq 1 - \frac{1}{n^2}$.*

Preuve. Notons que si $\mathcal{F}_i$ est vrai, alors $\Delta(H_{t_i}) < n/2^i$, et donc $\Delta(H_t) < n/2^i$ pour tout $t \geq t_i$. Par ailleurs, si $\mathcal{F}_{i+1}$ est faux, alors $\Delta(H_{t_{i+1}}) \geq n/2^{i+1}$, et donc $\Delta(H_t) \geq n/2^{i+1}$ pour tout $t \leq t_{i+1}$. En conséquence, si $u \in M_{i+1}(t_{i+1})$, alors $u \in M_{i+1}(t)$ pour tout $t = t_i, t_i + 1, \ldots, t_{i+1}$, et donc, pour tous ces t, on a $\deg_{H_t}(u) \geq \Delta(H_t)/2$. Ainsi, d'après la précédente observation, à chaque phase t, $t_i \leq t \leq t_{i+1}$, on a $\Pr[\mathcal{E}_u] \geq \frac{1}{2e^4}$. Les tirages de variables aléatoires sont mutuellement indépendants, donc pour tout $u \in M_{i+1}(t_i)$ on a

$$\Pr[u \in M_{i+1}(t_{i+1})] \leq (1 - \frac{1}{2e^4})^k = (1 - \frac{1}{2e^4})^{6e^4 \ln n}.$$

Comme $\ln(1 - x) \leq -x$ pour tout x, il en découle que $6e^4 \ln n \ln(1 - \frac{1}{2e^4}) \leq -3 \ln n$, d'où

$$\Pr[u \in M_{i+1}(t_{i+1})] \leq \frac{1}{n^3}.$$

Du fait de la sous-linéarité des probabilités, on en déduit donc en sommant sur les au plus n nœuds de $M_{i+1}(t_i)$ que

$$\Pr[M_{i+1}(t_{i+1}) \neq \emptyset \mid \mathcal{F}_i] \leq \frac{1}{n^2}$$

comme désiré. □

Nous sommes maintenant prêts à montrer le résultat suivant :

Théorème 6 *L'algorithme probabiliste Las Vegas de Peleg pour le calcul d'un MIS s'exécute, avec forte probabilité, en $O(\log^2 n)$ étapes dans le modèle* LOCAL.

Preuve. Nous allons montrer que dans l'algorithme probabiliste décrit dans ce chapitre, la probabilité que tous les nœuds décident d'entrer ou non dans le MIS en au plus $O(\log^2 n)$ étapes est au moins $1 - 1/n$. Pour cela, posons $\ell = \lceil \log n \rceil$, et définissons

$$\mathcal{F} = \bigcap_{i=1}^{\ell} \mathcal{F}_i.$$

Soit $\mathbf{p}$ la probabilité que tous les nœuds aient décidé au temps t_ℓ. Par définition, $\mathbf{p} \geq \Pr[\mathcal{F}]$. Or, nous avons

$$\overline{\mathcal{F}} = \overline{\mathcal{F}_1} \cup (\mathcal{F}_1 \cap \overline{\mathcal{F}_2}) \cup (\mathcal{F}_1 \cap \mathcal{F}_2 \cap \overline{\mathcal{F}_3}) \cup \ldots \cup (\mathcal{F}_1 \cap \mathcal{F}_2 \cap \cdots \cap \mathcal{F}_{\ell-1} \cap \overline{\mathcal{F}_\ell}).$$

Donc

$$\begin{aligned}
1 - \mathbf{p} &\leq \Pr[\overline{\mathcal{F}}] \\
&= \Pr[\overline{\mathcal{F}_1}] + \Pr[\mathcal{F}_1 \cap \overline{\mathcal{F}_2}] + \Pr[\mathcal{F}_1 \cap \mathcal{F}_2 \cap \overline{\mathcal{F}_3}] + \cdots + \Pr[\mathcal{F}_1 \cap \mathcal{F}_2 \cap \cdots \cap \mathcal{F}_{\ell-1} \cap \overline{\mathcal{F}_\ell}] \\
&\leq \Pr[\overline{\mathcal{F}_1}] + \Pr[\mathcal{F}_1 \cap \overline{\mathcal{F}_2}] + \Pr[\mathcal{F}_2 \cap \overline{\mathcal{F}_3}] + \cdots + \Pr[\mathcal{F}_{\ell-1} \cap \overline{\mathcal{F}_\ell}] \\
&\leq \Pr[\overline{\mathcal{F}_1} \mid \mathcal{F}_0] + \Pr[\overline{\mathcal{F}_2} \mid \mathcal{F}_1] + \Pr[\overline{\mathcal{F}_3} \mid \mathcal{F}_2] + \cdots + \Pr[\overline{\mathcal{F}_\ell} \mid \mathcal{F}_{\ell-1}] \\
&\qquad \text{car } \Pr[A \mid B] = \Pr[A \cap B] / \Pr[B] \geq \Pr[A \cap B] \\
&\leq \ell \cdot \frac{1}{n^2} \\
&= \lceil \log n \rceil / n^2
\end{aligned}$$

Donc, la probabilité $\mathbf{p}$ que tous les nœuds aient décidé au temps $t_\ell = k\ell = O(\log^2)$ est au moins $1 - O(1/n)$, ce qui conclut la preuve. □

Corollary 1 *Il existe un algorithme probabiliste Las Vegas de $(\Delta + 1)$-coloration s'exécutant, avec forte probabilité, en $O(\log^2 n)$ étapes dans le modèle* LOCAL.

Pb ouvert : couplage max dans graphes bi-partis, bi-colorés, de degré max Δ. (borne inf relaxed matching ne s'applique pas).

3 Gérer la congestion dans le modèle synchrone

Dans le chapitre précédent, nous avons vu que certains problèmes pouvaient se résoudre localement, c'est-à-dire en n'impliquant que des communications à petite distance (typiquement poly-logarithmique). C'est le cas du problème de la $(\Delta + 1)$-coloration et de celui du MIS. Malheureusement, de nombreux problèmes ne peuvent pas être résolus localement, au sens où leur résolution implique des communications entre nœuds à distance D, où D dénote le diamètre $diam(G)$ du graphe $G = (V, E)$:

$$D = \max_{(u,v) \in V \times V} \text{dist}_G(u, v).$$

Le problème typique ne pouvant pas être résolu localement est la construction d'un arbre couvrant de poids minimal (MST, pour « minimum spanning tree » en anglais).

Applications. Non documenté.

Le problème ARBRE COUVRANT DE POIDS MINIMUM (MST) :

Entrée : un réseau $G = (V, E)$ de n nœuds, dont chaque nœud dispose d'une identité distincte dans $\{1, \ldots, n\}$, et dont chaque arête e dispose d'un poids $w(e) \geq 0$; chaque nœud connaît initialement le poids de chacune de ses arêtes incidentes.

Objectif : chaque nœud $u \in V$ retourne un ensemble $mst(u) \subseteq E$ d'arêtes incidentes à u tel que $\cup_{u \in V} mst(u)$ induise un arbre couvrant de poids minimum dans G.

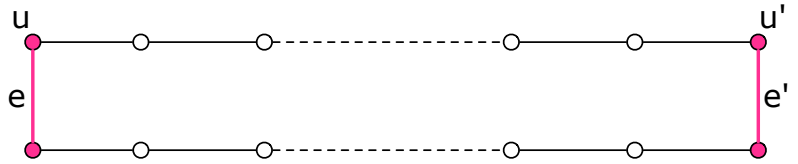

FIGURE 3 – *Le problème du MST ne peut pas se résoudre localement. Toutes les arêtes ont poids 0 sauf e et e' de poids respectifs $w(e) > 0$ et $w(e') > 0$.*

Dans la suite, nous supposerons sans perte de généralité que toutes les arêtes ont des poids deux-à-deux différents. Si ce n'est pas le cas, on peut toujours modifier le poids $w(e)$ de chaque arête $e = \{x, y\}$ comme suit : $w'(e) = (w(e), \min\{\mathrm{Id}(x), \mathrm{Id}(y)\}, \max\{\mathrm{Id}(x), \mathrm{Id}(y)\})$. Sous l'hypothèse des poids deux-à-deux distincts, il est facile de montrer qu'il ne peut y avoir qu'un unique MST. Notez que nous donnerons quelques exemples dans lesquels on supposera par exemple que, pour ensemble d'arêtes F, on a $w(e) = 0$ pour tout $e \in F$. Pour satisfaire la condition de poids distincts, il suffit de donner à chacune des arêtes de F un poids différent, arbitrairement petit.

Le problème du MST ne peut pas se résoudre localement, même dans l'anneau. Considérons en effet l'anneau C_n dessiné sur la figure 3. Le MST dépend de la valeur relative de $w(e)$ et $w(e')$. Ainsi, pour que u décide s'il doit inclure e dans $mst(u)$, il doit connaître $w(e')$. Plus formellement, considérons un algorithme distribué $\mathcal{A}$ de MST communiquant à distance au plus $\frac{n}{2} - 2$. Considérons trois instances du problème, définies par les paires $(w(e), w(e'))$:

$$I_1 = (1, 3) \quad I_2 = (3, 2) \quad I_3 = (1, 2)$$

Notons que n'importe quel algorithme de MST sur l'exemple de la figure 3 doit inclure toutes les arêtes de poids nuls. Supposons que $\mathcal{A}$ est correct pour I_1 et I_2. Dans I_1, on a $e \in mst(u)$. Dans I_2, on a $e' \in mst(u')$. Du fait que u et u' ne communiquent pas lors de l'exécution de $\mathcal{A}$ (ils sont à distance $\frac{n}{2} - 1$), le nœud u décide la même chose pour I_1 et I_3. De même, le nœud u' décide la même chose pour I_2 et I_3. En conséquence, lorsque $\mathcal{A}$ s'exécute sur I_3, on a $e \in mst(u)$ et $e' \in mst(u')$. La structure retournée pas $\mathcal{A}$ n'est donc pas un arbre.

Ce chapitre s'intéresse à la complexité des problèmes nécessitant de communiquer à distance D pour les résoudre. Dans le modèle LOCAL utilisé dans le chapitre précédent, tous les problèmes (Turing-calculable) peuvent être résolus en $O(D)$ étapes en concentrant les données en un nœud, résolvant le problème en ce nœud, et distribuant le résultat à tous les nœuds. Un tel algorithme suppose toutefois de potentiellement transmettre un grand nombre d'informations sur chaque lien à chaque étape, ce qui peut être irréaliste. Nous utiliserons donc un modèle plus raffiné dans ce chapitre, décrit dans la section suivante.

Notons que le fait que les nœuds aient ou non une connaissance de la taille n du réseau, voire même de son diamètre D n'a pas d'impact sur la borne ci-dessus pour le MST. Dans ce chapitre, nous supposerons donc que les nœuds connaissent n et D.

3.1 Un modèle à congestion, et MST distribué

3.1.1 Le modèle CONGEST

Le modèle CONGEST est une restriction du modèle synchrone qui impose une contrainte supplémentaire sur le volume des communications à chaque étape : à chaque étape, la quantité de données pouvant transiter sur une arête ne peut excéder $O(\log n)$ bits.

Le choix de la borne $O(\log n)$ sur le nombre de bits pouvant transiter sur chaque lien à chaque étape est relativement arbitraire. On peut en fait considérer le modèle CONGEST(B) qui stipule qu'à chaque étape, la quantité de données pouvant transiter sur une arête ne peut excéder B bits. Le choix de $B = O(\log n)$ est néanmoins raisonnable car il permet de faire transiter l'identité de l'expéditeur sur un lien en une étape, et, plus généralement, de faire transiter toute donnée de la taille d'un identifiant sur chaque lien à chaque étape.

Notez que si tout problème (calculable) peut être résolu en $O(D)$ étapes dans le modèle synchrone général, il n'en est pas de même dans le modèle CONGEST car concentrer toutes les données en un nœud peut nécessiter de faire transiter un gros volume de données vers ce nœud. De même, diffuser la solution calculée en un nœud à tous les autres nœuds peut être coûteux en termes de nombre de bits transitant sur chaque lien.

Nous pouvons supposer dans la suite que nous disposons d'une arbre couvrant T enraciné par exemple au sommet d'identifiant 1. Par exemple, T peut être un arbre BFS à partir de 1, donc de diamètre $\leq 2D$. En fait, étant donné un sous ensemble $U \subseteq V$ de nœuds tel que le sous-graphe $G[U]$ de G graphe induit par les nœuds de U soit connexe, on peut aisément construire un arbre couvrant T de U. Pour cela, tous les nœuds de U diffusent leurs identifiants dans $G[U]$. Lorsqu'un nœud reçoit des identifiants de ses voisins, il ne propage que le plus petit d'entre eux, et ce seulement s'il n'a pas déjà propagé préalablement un identifiant plus petit. En au plus $diam(G[u])$ étapes, tous les sommets connaissent le plus petit identifiant dans $G[U]$. Le sommet de plus petit identifiant initie alors la construction d'un arbre BFS, qui s'effectue également en au plus $diam(G[u])$ étapes. Remarquons toutefois que les nœuds de $G[U]$ ne connaissent pas nécessairement $diam(G[u])$. Ils peuvent toutefois borner $diam(G[u])$ par $|U| - 1$, et si même $|U|$ n'est pas connu, alors il peuvent borner $diam(G[u])$ par $n - 1$. Attention, on n'a pas nécessairement $diam(G[u]) \leq D$ car les plus courts chemins dans $G[u]$ peuvent être plus longs que ceux dans G.

Afin d'illustrer le modèle CONGEST, nous présentons ci-après un algorithme distribué pour le MST. On suppose qu'il existe $c > 0$ tel que pour toute arête e, $w(e) \leq n^c$. Cette hypothèse est posée uniquement pour assurer que le poids de chacune des arêtes soit sur $O(\log n)$ bits, et puisse donc transiter en une étape le long d'un lien.

3.1.2 Construction distribuée d'un MST (algorithme de Borůvka)

Cet algorithme est une version distribué de l'algorithme de Borůvka, datant de 1926, consistant à réduire le nombre d'arbres d'une forêt couvrante jusqu'à obtenir un unique arbre couvrant. Initialement, la forêt consiste en n arbres, chacun réduit à un unique nœud. L'algorithme procède en $\lceil \log n \rceil$ phases. A chaque phase, la taille minimale des arbres de la forêt est au moins doublée : à la fin de la phase i, $i \geq 1$, tous les arbres de la forêt ont taille au moins 2^i. On appelle *fragment* le graphe induit par les nœuds d'un arbre de la forêt. L'algorithme part donc de n fragments, et les fusionne jusqu'à obtenir un unique fragment.

A chaque phase (voir la figure 4), chaque fragment F détermine l'arête de poids minimum incidente à F. A cette fin, chaque fragment F maintient un nœud leader $lead(F)$ auquel est enraciné un arbre T_F couvrant F, et les noeuds de F font remonter le poids de leur arête incidente de poids minimum le long de T_F jusqu'à la racine $lead(F)$. Ces messages sont de la forme $(\mathrm{Id}(u), \mathrm{Id}(v), w(e))$ pour une arête $e = \{u, v\}$. Pour limiter le coût de cette remontée, chaque nœud filtre les poids, et ne fait remonter que le poids minimum parmi tous ceux qu'il a appris. Plus spécifiquement, chaque nœud du fragment F attend de recevoir un poids de chacun de ses enfants dans T_F. Une fois reçus tous ces poids, il en calcule le minimum (en incluant

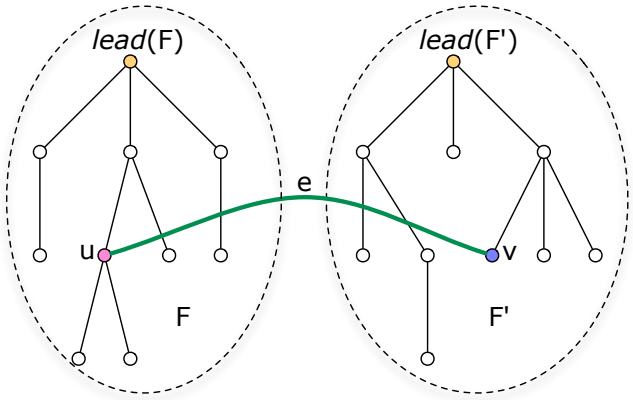

FIGURE 4 – *Fusion de deux fragments*

le poids minimum de ses propres arêtes incidentes ayant leur autre extrémité à l'extérieur du fragment), puis transmet ce minimum à son parent dans T_F. Une fois que $lead(F)$ a reçu une valeurs de chacun de ses enfants, il en calcule le minimum (toujours en incluant le poids minimum de ses propres arêtes incidentes ayant leur autre extrémité à l'extérieur du fragment), et diffuse ce minimum dans T_F. Ainsi, en $O(diam(T_F))$, tous les sommets du fragment F connaisse l'arête incidente $e = \{u, v\}$ à ce fragment, de poids minimum.

Supposons $u \in F$, et $v \in F'$, avec $F' \neq F$. Les fragments F et F' fusionnent, pour constituer un nouveau fragment $\widehat{F}$. En conséquence, l'arête e est rajoutée à T_F et $T_{F'}$, résultant en un arbre $T_{\widehat{F}}$ couvrant $\widehat{F}$. Les fragments F et F' sont identifiés par les identifiants respectifs de leurs leaders. Le fragment $\widehat{F}$ est identifié par l'identité du nœud ayant la plus petite identité entre $lead(F)$ et $lead(F')$. Ce nœud devient le leader de $\widehat{F}$. On diffuse alors le longs de T_F et $T_{F'}$ l'identité de ce nouveau leader afin que les nœuds de $\widehat{F}$ mettent à jour leur identifiant de fragment. Durant cette mise à jour, les nœuds modifient potentiellement l'identité de leur parent et celles de leurs enfants dans l'arbre couvrant le fragment. Une fois cela fait, la phase se termine.

Le déroulement d'une phase peut toutefois être plus complexe qu'exposé ci-dessus. En effet, lors de la fusion de F et F', si $e = \{u, v\}$ est l'arête de poids minimum incidente au fragment F, il n'y a pas a priori de raison que e soir l'arête de poids minimum incidente au fragment F'. La figure 5 montre par exemple le cas de six fragments. La direction de chaque arête de F_i vers F_j indique que l'arête est celle incidente à F_i de poids minimum. Parfois, c'est aussi celle de poids minimum incidente à F_j, mais pas toujours. Ainsi, lors du déroulement d'une phase, la fusion ne se fait donc pas nécessairement par paire de fragments, mais une même fusion peut impliquer plus de deux fragments. Ceci a deux conséquences majeures. D'une part, l'algorithme lui même est rendu plus complexe. (Nous ne rentrerons toutefois pas ici dans les détails de son implémentation qui, sans être trivial, ne fait apparaître qu'une série de détails fastidieux à prendre en compte). D'autre part, la fusion de plus de deux fragments à chaque phase ne permet plus de borner ni la taille ni le diamètre de chaque fragment. Or, si nous savions que la taille de tous les fragments n'excède pas 2^i à la fin de la phase i, nous pourrions borner le nombre total d'étapes de l'algorithme par $O(\sum_{i=1}^{\lceil \log n \rceil} 2^i) = O(n)$. Malheureusement, la fusion de multiple fragments lors d'une même phase ne nous permet pas de donner une meilleure borne que celle exprimée dans le résultat ci-dessous, où le temps de chaque phase est simplement borné par $O(n)$ étapes afin d'être certain que toutes les fusions de la phase i ont eut lieu avant de commencer la phase $i + 1$.

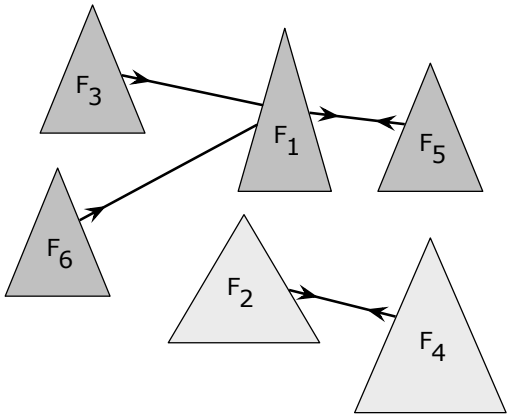

FIGURE 5 – *Fusion de plusieurs fragments*

Théorème 7 *L'algorithme de Borůvka distribué construit un MST en $O(n \log n)$ étapes dans le modèle* CONGEST.

Preuve. L'arbre retourné par l'algorithme est l'arbre couvrant construit lorsqu'il ne reste plus qu'un seul fragment. Le fait que cet arbre soit effectivement un MST provient de la *règle de la coupe* qui stipule que, pour toute partition des nœuds d'un graphe $G = (V, E)$ en deux ensembles S et $V \setminus S$, l'arête de poids minimum parmi toutes les arêtes connectant S a $V \setminus S$ est nécessairement dans un MST. La correction de l'algorithme découle de cette règle puisque pour tout fragment F, la paire $(F, V \setminus F)$ forme une partition de V, et l'arête choisie par l'algorithme est précisément l'arête de poids minimum incidente à F, donc l'arête de poids minimum parmi celles de la coupe $(F, V \setminus F)$. □

3.2 Un algorithme de MST sous-linéaire

Nous avons vu dans la section précédente un algorithme distribué pour la construction d'un MST s'exécutant en $O(n \log n)$ étapes. Nous allons maintenant voir qu'il est possible de faire mieux, et de construire un MST en temps $O(n^c + D)$ avec $0 < c < 1$. Notons que cette complexité ne contredit pas la borne inférieure $\Omega(n)$ vue en introduction, car cette borne inférieure est basée sur une famille de graphes pour lesquels $D = \Theta(n)$. L'obtention de la borne $O(n^c + D)$ nécessite d'introduire une autre construction, de performances similaires à celles de la version distribuée de Borůvka, mais qui, combinée habilement avec cette dernière, permet d'obtenir le résultat souhaité.

3.2.1 Algorithme Matroïde pour la construction d'un MST

Le nom de cet algorithme provient du fait qu'il peut également s'appliquer à la construction d'un ensemble indépendant maximal de poids minimum dans un matroïde (un arbre n'est qu'un ensemble maximal dans le matroïde des co-cycles d'un graphe).

L'algorithme Matroïde suppose l'existence d'un arbre BFS enraciné en un nœud r. Notons T cet arbre. Nous avons vu qu'un tel arbre pouvait être construit en $O(D)$ étapes. L'arbre T va servir à concentrer en r un ensemble d'arêtes pertinent, duquel il sera possible d'extraire un MST. Notons que concentrer en r toutes les arêtes pourrait prendre jusqu'à $\Omega(n^2)$ étapes dans

un graphe dense. Il convient donc de filtrer les arêtes en chaque nœud afin de ne faire remonter vers r que les arêtes pertinentes. A cette fin, nous utilisons le résultat facile à obtenir spécifiant que, étant donné un cycle C dans G, l'arête de C de poids maximum parmi toutes les arêtes de C ne peut être dans aucun MST.

L'algorithme Matroïde procède en deux phases : une phase de montée, et une phase de descente. Dans l'algorithme de remontée ci-dessous, parent(u) et enfant(u) dénotent respectivement le parent de u dans l'arbre T, et l'ensemble de ses enfants dans ce même arbre. Une arête $e = \{x, y\}$ est identifiée par le triplé $(w(e), \mathrm{Id}(x), \mathrm{Id}(y))$. L'algorithme décrit comment les arêtes sont remontées vers la racine r en étant filtrées.

Algorithme Matroïde pour un nœud u (montée) :
début
$K \leftarrow$ {arêtes incidentes à u dans G} /* *K is for « know »* */
attendre d'avoir reçu au moins une arête de chaque enfant
répéter
$K \leftarrow K \cup$ {arêtes reçues des enfants}
$U \leftarrow$ {arêtes déjà envoyées à parent(u)} /* *U is for « up »* */
$R \leftarrow \{e \in K \setminus U \ : \ U \cup \{e\}$ contient un cycle$\}$ /* *R is for « remove »* */
$C \leftarrow K \setminus (U \cup R)$ /* *C is for « candidate »* */
si $C \neq \emptyset$ **alors**
envoyer à parent(u) l'arête de poids minimum dans C
recevoir une arête de chacun de enfants de u
/* *si aucune arête est envoyée par un enfant alors ignorer cette réception* */
sinon stop.
fin.

Notons qu'un nœud u ne commence à envoyer à son parent qu'à l'étape $t = h(u) + 1$ où $h(u)$ dénote la hauteur du sous-arbre T_u de T pendant du nœud u, c'est-à-dire la plus longue distance entre u et une feuille de T_u. Afin d'établir la correction de l'algorithme Matroïde, nous montrons le lemme suivant. Fixons une étape $t \geq 1$. Un nœud qui n'a pas terminé durant l'étape $t - 1$ est dit *actif* à l'étape t.

Lemma 5 *Pour tout enfant actif v de u, l'ensemble candidat C de u lors de l'étape t contient au moins une arête envoyée précédemment par v.*

Preuve. On procède par induction sur la hauteur $h(u)$ de u. Le lemme est vrai trivialement pour $h(u) = 0$ car une feuille n'a pas d'enfants. Supposons le lemme vrai pour tous les nœuds de hauteur au plus k, et considérons un nœud u de hauteur $k + 1$. Dit autrement, nous supposons que les enfants de u satisfont le lemme. Soit v un enfant actif de u. Soit E_v (respectivement, E_u) l'ensemble des arêtes envoyée par v à u (respectivement, par u à son parent) jusqu'à l'étape $t-1$ inclue. Notons que $h(v) \leq h(u) - 1$, donc v a commencé à envoyer des arêtes vers u au moins une étape avant que u ne commence à envoyer des arêtes vers son parent. Donc $|E_v| > |E_u|$. Il en découle qu'il existe une arête $e \in E_v$ telle que $E_u \cup \{e\}$ est acyclique[5]. En effet, E_u est une forêt $F = \{T_1, \ldots, T_\ell\}$ non couvrante de G, avec $\ell \geq 1$. Si toutes les arêtes de E_v étaient incluses dans les composantes connexes de F, c'est-à-dire dans $\cup_{i=1}^{\ell} \big(V(T_i) \times V(T_i)\big)$, alors on aurait $\big|\big(V(T_i) \times V(T_i)\big) \cap E_v\big| \geq |V(T_i)|$ pour au moins un indice i, ce qui impliquerait l'existence d'un cycle dans E_v, ce qui est impossible. Il existe donc une arête $e \in E_v$ dont les deux extrémités ne sont pas dans un même $V(T_i)$, et $E_u \cup \{e\}$ est donc acyclique. On a donc $e \in C$ à l'étape t, ce qui prouve le lemme. □

5. C'est pour cette propriété, qui est une propriété de matroïde, que l'algorithme est appelé Matroïde.

Lemma 6 (a) *Si u fait remonter une arête de poids p à son parent à l'étape t, alors, d'une part, toutes les arêtes qu'il a reçues à l'étape $t-1$ de ses enfants actifs étaient de poids au moins p, et d'autre part, toutes les arêtes qu'il apprendra à l'avenir (à toute étape $\geq t$) seront de poids au moins p.* (b) *Les poids des arêtes envoyées par u à son parent sont en ordre croissant.*

Preuve. On procède de nouveau par induction sur la hauteur $h(u)$ de u. La propositions (a) est vraie trivialement pour $h(u) = 0$ car une feuille n'a pas d'enfants. La proposition (b) est vraie par le choix de l'arête à faire remonter parmi celles de C. Supposons le lemme vrai pour tous les nœuds de hauteur au plus k, et considérons un nœud u de hauteur $k+1$. Dit autrement, nous supposons que les enfants de u satisfont le lemme.

Pour établir (a), commençons par établir la première partie de la proposition (a). Soit v un enfant actif de u, et soit e' l'arête envoyée par v à u à l'étape $t-1$. Par ailleurs, soit $e \in C$ dont l'existence viens d'être établi dans le lemme 5. Par induction, la proposition (b) implique que $w(e) \leq w(e')$. Par ailleurs, par le choix de l'arête envoyée par u à son parent, on a $p \leq w(e)$. Donc $w(e') \geq p$ comme souhaité.

La seconde partie de la proposition (a) découle alors de la première partie et de l'induction (b).

Finalement, (b) découle directement de la seconde partie de (a), par le choix de l'arête de C à remonter au parent. □

Les lemmes ci-dessus permet en particulier d'établir qu'un nœud ne s'arrête pas prématurément. Plus précisément, montrons que si $C = \emptyset$ à l'étape t, alors u ne recevra plus d'arêtes aux étapes $\geq t$. On montre cela de nouveau par induction sur la hauteur $h(u)$. La proposition est vraie trivialement pour $h(u) = 0$. Supposons la proposition vraie pour tous les nœuds de hauteur au plus k, et considérons un nœud u de hauteur $k+1$. Par le lemme 5, si $C = \emptyset$ alors tous les enfants de u sont inactifs (c'est-à-dire ont terminé à l'étape $t-1$). Le nœud u ne recevra donc plus d'arêtes aux étapes $\geq t$, et il est donc légitime qu'il s'arrête.

Nous sommes prêts maintenant à prouver le résultat suivant.

Théorème 8 *L'algorithme Matroïde construit un MST en $O(n + D)$ étapes dans le modèle* CONGEST.

Preuve. Puisque tout nœud fait remonter les arêtes dans un ordre croissant de poids, ne pas considérer comme candidates à la remontée les arêtes de R est légitime puisque toute arête e de R forme un cycle avec des arêtes déjà remontées. Comme e est l'arête de poids maximum de ce cycle (puisque les arêtes remonte en poids croissants), elle ne peut apparaître dans un MST. Il est donc légitime de ne pas la faire remonter vers la racine. En conséquence, la racine r finit par recevoir un ensemble d'arêtes duquel elle peut extraire un MST qui est de fait un MST du graphe G.

Le nombre d'étapes de la montée dans l'algorithme Matroïde est borné par $O(n + D)$. En effet, d'une part, la racine r à reçu un message de chacun de ses enfants après au plus D étapes (sa hauteur est au plus D). D'autre part, un nœuds ne peut faire remonter plus de $n-1$ arêtes vers son parent car n arêtes induisent nécessairement un cycle dans G, en contradiction avec le filtrage effectué dans l'ensemble R. La racine termine donc après au plus $O(n + D)$ étapes. Pour la descente, la racine diffuse à tous les sommets l'ensemble des arêtes du MST calculé. Un pipeline dans l'arbre BFS permet d'effectuer cette diffusion en au plus $O(D + n)$ étapes car la hauteur de r est au plus D, et le nombre d'arêtes à envoyer au plus $n-1$. □

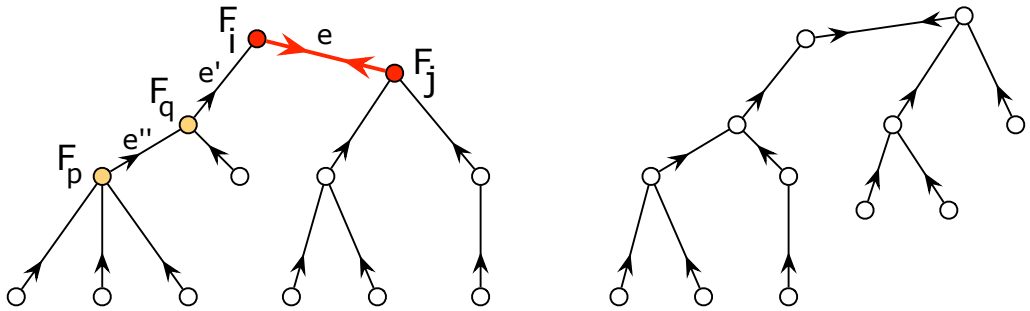

FIGURE 6 – *Fusion de plusieurs fragments. Chaque sommet de ces deux arbres représente un fragment. L'arête incidente de poids minimum incidente au fragment F_p est l'arête e'', alors que celle incidente au fragment F_q est e'. Les fragments F_i et F_j ont en commun d'avoir la même arête e incidente de poids minimum incidente. A l'issue de la fusion de tous ces fragments, il ne restera plus que deux fragments, correspondant à chacun de ces deux arbres.*

3.2.2 Combinaison de Borůvka et de Matroïde

La version distribuée de l'algorithme de Borůvka est peu efficace car les fragments peuvent potentiellement devenir très grands, entrainant de nombreuses étapes (potentiellement un nombre linéaire d'étapes) pour identifier l'arête incidente au fragment de poids minimum. L'algorithme Matroïde est quant à lui peu efficace car il concentre toutes les données en un seul nœud. Même filtrée, ces données sont de taille linéaire (au moins n arêtes remontent jusqu'à la racine, pouvant nécessiter un temps $\Omega(n)$ si la racine est de petit degré). Afin de développer un algorithme sous-linéaire, l'idée est d'effectuer quelques phases de Borůvka en contrôlant la taille maximum des fragments, puis de finir la construction avec Matroïde lorsque la taille des fragment dépasse un certain seuil.

Pour contrôler la taille des fragments, il faut revenir à la structure des arêtes inter-fragments identifiées lors d'une phase de fusion de fragments. La figure 5 donne un aperçu de ces liens. Plus généralement, la figure 6 décrit la forme de ces liens. Chaque fragment F à une arête incidente e de poids minimum, représentée par un arc sortant du sommet représentant F. L'ensemble de ces arêtes forme ainsi un 1-facteur, c'est-à-dire un graphe orienté tel que chaque sommet possède un et un seul arc sortant. La structure d'un 1-facteur est telle que représentée sur la figure 6 : une collection d'arbres attachés à des cycles orientés. Dans le cas particulier de l'algorithme de Borůvka, chaque cycle ne contient que deux sommets. En effet, un cycle $e_1, \ldots, e_k$ de $k > 2$ sommets, composé d'arcs incidents à $F_1, \ldots, F_k$, respectivement, impliquerait que $w(e_1) > w(e_2) > \cdots > w(e_k)$ car F_{i+1} sélectionne e_{i+1} et pas e_i, pour tout $i = 1, \ldots, k-1$. Or F_1 sélectionne e_1 et pas e_k, donc $w(e_k) > w(e_1)$, ce qui conduit à la contradiction $w(e_k) > w(e_1) > w(e_2) > \cdots > w(e_k)$.

Il est aisé d'orienter l'arc e choisi par deux fragments du fragment d'identité plus petite vers le fragment d'identité plus grande. On obtient ainsi un collection d'arbres enracinés, telle que représentée sur la figure 7.

Nous avons vu dans l'exercice 1 qu'il existe un algorithme de 3-colorations des arbres enracinés dans lesquels les noeud ont une notion consistante de haut et de bas, s'exécutant en $O(\log^* n + O(1))$ étapes dans le modèle synchrone. On peut de plus vérifier qu'il est possible de concevoir un tel algorithme s'exécutant en $O(\log^* n + O(1))$ étapes dans le modèle CONGEST. L'équivalence entre $(\Delta+1)$-coloration MIS établie dans la section 2.4.1 permet d'obtenir un MIS

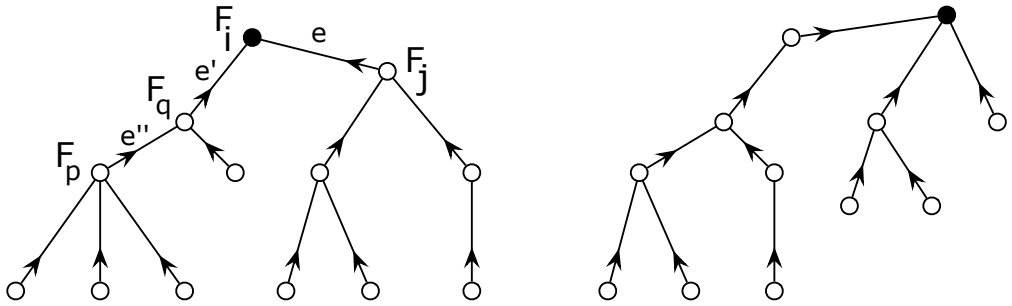

FIGURE 7 – *Le 1-facteur composé de l'arête choisie par chaque fragment est transformé en une forêt dont chaque arbre est enraciné.*

en $O(\log^* n + O(1))$ étapes dans le modèle CONGEST.

Construction d'un petit ensemble dominant. Observons qu'un MIS forme un ensemble dominant. (Dans un graphe $G = (V, E)$, un ensemble $D \subseteq V$ est *dominant* si, pour tout $u \in V$, on a $u \in D$ ou u est adjacent à un sommet $v \in D$). Nous disposons ainsi d'un algorithme construisant un ensemble dominant D dans les arbres enracinés en $O(\log^* n + O(1))$ étapes. Voyons maintenant comment raffiner cette construction afin de s'assurer que $|D| \leq \nu/2$ dans un arbre enraciné T de ν sommets. Pour $i \in \{0, 1, 2\}$, soit X_i l'ensemble des sommets de T à distance i d'une feuille de T (en particulier X_0 est l'ensemble des feuilles de T). Soit Y l'ensemble formé par le reste des sommets de T. Soit I le MIS construit par notre algorithme en $O(\log^* \nu + O(1))$ étapes, lorsque restreint aux sommets de Y. Soit alors

$$D = I \cup X_1.$$

D est bien un ensemble dominant car I domine Y, et X_1 domine $X_0 \cup X_1 \cup X_2$. Par ailleurs $|X_1| \leq |X_0|$, et donc

$$|X_1| \leq \frac{1}{2}(|X_0| + |X_1|).$$

Enfin, on peut associer à chaque sommet de I son enfant dans $(Y \cup X_2) \setminus I$ afin d'obtenir un injection de I dans $(Y \cup X_2) \setminus I$. Donc $|I| \leq |Y| + |X_2| - |I|$, et ainsi

$$|I| \leq \frac{1}{2}(|Y| + |X_2|).$$

Il en découle que

$$|D| \leq \frac{1}{2}(|X_0| + |X_1| + |X_2| + |Y|) = \frac{\nu}{2}.$$

Construction de fragments de petit diamètre. L'algorithme rapide procède comme suit. On débute par un certain nombre de phases de Borůvka, en contrôlant la nombre et le diamètre des fragments. Soit $N(t)$ le nombre maximum de fragments construit à la phase t de Borůvka, et soit $diam(t)$ le diamètre maximum d'un fragment construit à la phase t. (Initialement, $N(0) = n$ et $diam(0) = 0$). On construit un petit ensemble dominant dans chaque « méta-arbre » de fragments obtenu à la phase $t + 1$, permettant de découper chaque méta-arbre de ν nœuds en au plus $\nu/2$ sous-arbres de diamètre au plus $3 \cdot diam(t) + 2$. (Pour cela, on découpe chaque méta-arbre en « étoiles » de centre les fragments dominants). Ainsi, après t phases, on obtient :

$$N(t) \leq \frac{n}{2^t}$$

et

$$diam(t) \leq 3^t - 1.$$

Le temps de ces t phases est

$$\sum_{i=1}^{t} \Big[diam(i) \cdot \Big(\log^* N(i) + O(1) \Big) \Big] = O(3^t \log^* n).$$

Après un nombre t de phases bien choisi, on termine par l'algorithme Matroïde pour déterminer les arêtes entre les $N(t)$ fragments construits par Borůvka modifié. Ceci se fait en au plus

$$O(N(t) + diam(G)) = O\Big(\frac{n}{2^t} + diam(G) \Big)$$

étapes. Soit $t = \log_6 n$. On a alors $3^t = \frac{n}{2^t} = n^\alpha$ avec $\alpha = \frac{\log 3}{\log 6} = 0,6131\ldots$. On en déduit donc ce qui suit.

Théorème 9 *Il existe un algorithme de MST s'exécutant en $O(n^{0,6131} \log^* n + D)$ étapes dans le modèle* CONGEST *pour tout graphe de n nœuds et diamètre D.*

Il est donc possible de construire un MST de façon distribuée en un nombre sous-linéaire d'étapes. La question naturelle qui se pose alors est : quel est le plus petit α, $0 \leq \alpha \leq 1$, tel qu'il soit possible de construire un MST en $O(n^\alpha + D)$ étapes dans le modèle CONGEST ? Nous avons montré que $\alpha < \frac{\log 3}{\log 6} + \epsilon$, pour tout $\epsilon > 0$. En fait, il existe un algorithme de MST s'exécutant en $O(\sqrt{n} \log^* n + D)$ étapes dans le modèle CONGEST. Dans la section suivante, nous allons voir que cette dernière borne asymptotique est quasiment optimale.

3.3 Bornes inférieures dans le modèle CONGEST

3.3.1 Une technique générale

La technique générale pour obtenir des bornes inférieures dans la modèle CONGEST consiste à généraliser celle que nous avons vue en introduction de la section 3, mais en argumentant non pas sur la distance séparant deux parties du graphe, mais sur le volume de données devant transiter d'une partie du graphe à une autre. Cette technique est schématisée sur la figure 8. Sur cette figure sont indiquées deux ensembles de nœuds A et B, supposés distants dans le graphe. Supposons qu'une « décision » devant être prise par les nœuds de A dépende de données disponibles sur les nœuds de B. Par exemple, en introduction de la section 3, nous étions dans le cas où un nœud u devait décider de mettre ou pas son arête incidente e dans le MST, et cette décision dépendait du poids d'une arête e' à distance $D = diam(G)$. Cela nous avait permis de conclure à une borne inférieure $\Omega(D)$, ce qui n'est pas forcément d'une grande pertinence pour des graphes de petit diamètre. En revanche, si nous pouvons établir que les nœuds de la partie A ont besoin de « beaucoup » de données provenant de B, alors nous pouvons potentiellement exprimer une borne inférieure dépendante de n, même pour de petits diamètres. En effet, si A a besoin de x bits provenant de B, et que la coupe minimale séparant A et B est de k arêtes, alors le transfert de ces x bits de A vers B nécessitera au moins $x/(k \log n)$ étapes, car, à chaque étape, au plus $k \log n$ bits peuvent transiter sur k liens dans le modèle CONGEST.

L'argument ci-dessus n'est toutefois pas forcément trivial à utiliser. En effet, outre qu'il convient d'identifier correctement les ensembles A et B, la notion du « nombre de bits de B

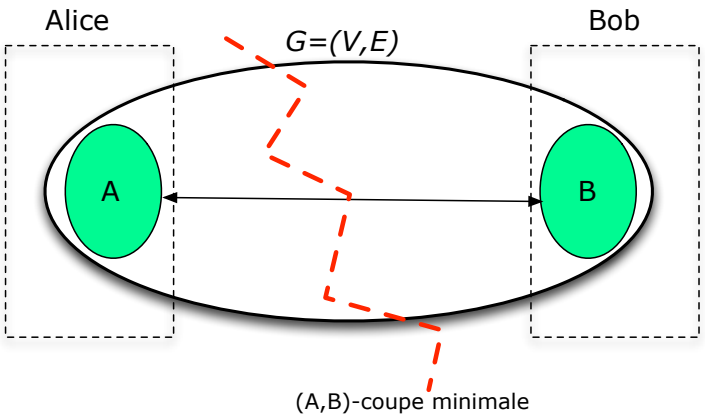

FIGURE 8 – *Réduction du calcul d'une borne inférieure dans le modèle* CONGEST *au calcul d'une borne inférieure pour un problème de complexité des communications.*

nécessaire à A » pour prendre un décision n'est pas formellement définie ici. En fait, ce questionnement fait précisément l'objet d'une théorie appelée la *complexité de la communication* (« communication complexity » en anglais). Dans sa version classique, la complexité des communications fait intervenir deux joueurs, Alice et Bob, devant calculer une fonction f de leurs entrées. Plus précisément, Alice et Bob disposent chacun d'une donnée respective a et b, et la question est : combien de bits doivent échanger Alice et Bob pour calculer $f(a, b)$? Par exemple f peut être l'indicatrice de $a = b$. Le point crucial est que, pour certaines fonctions f, Alice n'a pas nécessairement besoin de tous les bits nécessaires à encoder b pour calculer $f(a, b)$. Cela est particulièrement vrai dans un contexte probabiliste. Ainsi, la technique générale évoquée ci-dessus et conduisant à la borne inférieure $x/(k \log n)$ ne doit pas définir x comme le nombre de bits pour encoder les données de B, mais x comme la complexité de la communication permettant à A de prendre la décision convenable. Ainsi, la partie A du graphe devra correspondre à un joueur, par exemple Alice, et la partie B devra correspondre à un second joueur, par exemple Bob.

Nous allons appliquer cette technique à la mise en évidence d'une borne inférieure pour la construction d'un MST.

3.3.2 Borne inférieure pour la construction d'un MST

Pour établir une borne inférieurs sur le temps minimum nécessaire à la construction d'un MST dans le modèle CONGEST, nous allons considérer une famille d'instances, indexées par un entier positif k et par un vecteur x de k^2 bits. Le vecteur x ne sert que pour l'affectation des poids, alors que k contrôle uniquement la structure d'un graphe G_k.

La construction de G_k est illustrée sur la figure 9. Il consiste en k^2 chemins disjoints, chacun formé de $k^2 + 1$ nœuds, notés $v_{i,j}$, pour $i = 0, \ldots, k^2$ et $j = 1, \ldots, k^2$, plus $k + 1$ nœuds notés c_i pour $i = 0, \ldots, k$ formant un chemin $(c_0, c_1, \ldots, c_k)$. Chaque c_i est de plus le centre d'une étoile à k^2 branches obtenue en connectant c_i au nœuds $v_{ik,j}$ pour $j = 1, \ldots, k^2$. On a donc $n = k^2(k^2 + 1) + k + 1 = \Theta(k^4)$. Remarquons que le diamètre de G_k est $2k + O(1) = \Theta(n^{1/4})$.

Chaque arête de G_k a un poids nul, sauf les arêtes incidentes aux c_i, $i = 0, \ldots, k$, n'appartenant pas au chemin connectant les c_i. Les arêtes incidentes à c_0 ont leurs poids dictés par $x = (x_1, \ldots, x_{k^2})$, avec $x_i \in \{0, 1\}$ pour tout $i = 1, \ldots, k^2$, comme suit : pour tout $j = 1, \ldots, k^2$,

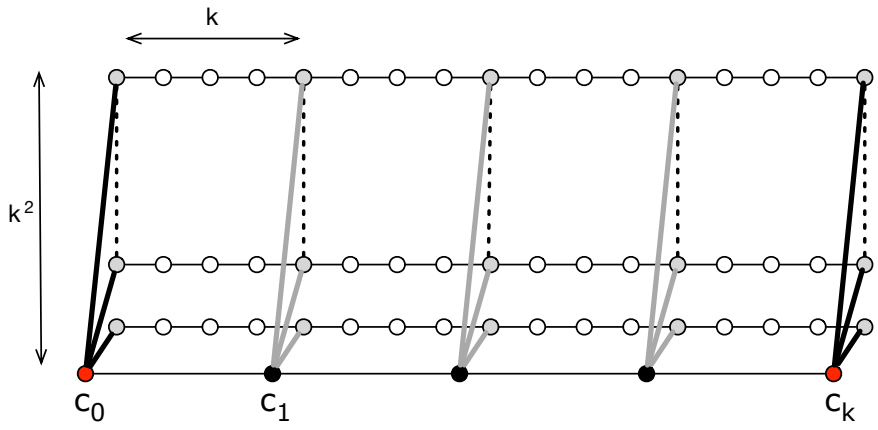

FIGURE 9 – *Les instances (G_m, x). Toutes les arêtes sont de poids nul, sauf celles indiquées en gras. Parmi celles-ci, celles en noir ont un poids 1, 2, ou 3, et celles en gris ont un poids infini.*

on fixe $w(\{c_0, v_{0,j}\}) = 2x_j + 1$. Ces poids valent donc 1 ou 3 selon la valeur de x_j. Pour $i = 1, \ldots, k-1$, les arêtes $\{c_i, v_{ik,j}\}$ ont un poids ∞. Enfin, les arêtes incidentes à c_k ont toutes un poids 2. Les nœuds de l'instance (G_k, x) ne connaissent pas x. Ainsi, pour tout j, la présence de $\{c_k, v_{k,j}\}$ dans l'unique MST dépend seulement du poids de $\{c_0, v_{0,j}\}$, et on vérifie facilement que, pour tout j,

$$\{c_k, v_{k,j}\} \in \text{MST} \iff w(\{c_0, v_{0,j}\}) = 3.$$

La borne du diamètre ne donne qu'une borne inférieure $\Omega(n^{1/4})$. Nous allons montrer presque $\Omega(n^{1/2})$ en utilisant le volume de communication devant transiter entre c_0 et c_k.

Théorème 10 *Tout algorithme de construction de MST nécessite au moins $\Omega(\sqrt{n}/\log n)$ étapes dans le modèle* CONGEST*, même restreint aux instances (G_k, x).*

Preuve. Notons qu'en une unique étape, on peut transférer un bit de chacun des nœuds $v_{k^2,j}$ à c_k, et de chacun des nœuds $v_{0,j}$ à c_0, $j = 1, \ldots, k^2$. On peut donc réduire la construction d'un MST dans (G_k, x) au transfert de x de c_0 à c_k. Plus précisément, étant donné x en c_0, le nœud c_k doit déterminer l'ensemble des indices i pour lesquels $x_i = 1$. En supposant qu'Alice a x en entrée, la théorie de la complexité des communications dit que le calcul d'une telle fonction par Bob nécessite le transfert de k^2 bits d'Alice à Bob. La construction d'un MST dans (G_k, x) nécessite donc bien le transfert de k^2 bits de c_0 à c_k.

Nous allons montrer que le transfert de k^2 bits de c_0 à c_k dans G_k nécessite $\Omega(\sqrt{n}/\log n)$ étapes dans le modèle CONGEST. En fait, nous n'allons montrer ce résultat que si l'on se restreint à un algorithme de communication *explicite*, c'est-à-dire pour lequel chacun des k^2 bits suit un chemin spécifique de c_0 à c_k, sans être altéré. La borne $\Omega(\sqrt{n}/\log n)$ étapes reste vraie en générale, même si l'on autorise les bits à être combinés et recomposés au cours du transfert de c_0 à c_k. La preuve serait néanmoins plus complexe.

Soit donc P_i le chemin de c_0 à c_k suivi par le $i$ème bit devant être transféré entre ces deux nœuds. Appelons *autoroute* le chemin $(c_0, c_1, \ldots, c_k)$. (Notez que si un chemin P_i n'emprunte pas d'arêtes de l'autoroute, alors ce chemin est de longueur $\Omega(\sqrt{n})$, impliquant $\Omega(\sqrt{n})$ étapes). Soit n_i le nombre d'arêtes de l'autoroute empruntées par le chemin P_i. On a $0 \leq n_i \leq k$. La

longueur de P_i est au moins $n_i + (k - n_i)k$. Donc, si $n_i \leq k/2$ pour un i quelconque, alors la longueur de P_i est $\geq k^2/2$, impliquant impliquant $\Omega(\sqrt{n})$ étapes. Si $n_i > k/2$ pour chacun des k^2 bits, alors le nombre total d'arêtes de l'autoroute emprunté par les k^2 bits est

$$\sum_{i=1}^{k^2} n_i > k^3/2.$$

Or, l'autoroute n'a que k arêtes, donc l'une d'entre elles est empruntée par plus de $k^2/2$ chemins. La bande passante de cette arête étant restreinte à $O(\log n)$ dans le modèle CONGEST, le passage des $k^2/2$ bits correspondant à ces $k^2/2$ chemins nécessite $\Omega(\sqrt{n}/\log n)$ étapes. □

4 Structures de données distribuées

Tout comme en calcul séquentiel, le calcul distribué nécessite des structures de données appropriées. Dans ce chapitre, nous étudions deux types de structures de données. La première structure de données étudiée est celle permettant d'acheminer (on utilise souvent la terminologie « router ») des messages au sein d'un réseau. Au fin de routage, chaque nœud u est muni d'une *table de routage* contenant suffisamment d'information à propos du réseau pour router les messages à partir de u. Nous nous intéresserons à la conception de tables *compactes* car l'accès séquentiel à une table prend un temps qui croît avec la taille de la table. Or router de multiples messages rapidement demande un temps de réponse faible pour calculer le port de sortie de chaque message. Concevoir des tables compacts à accès séquentiel rapide est une voie pour répondre à ce problème.

Nous nous intéresserons également à la conception de schémas d'étiquetage informatif. Il s'agit, étant donné un type d'information f relatif à toute paire de nœuds (par exemple $f(u, v) = \text{dist}(u, v)$), de donner à chaque nœud u une *étiquette* (« label » en anglais) $L(u)$ telle que, pour toute paire (u, v) de nœuds, la valeur de $f(u, v)$ puisse être calculée uniquement à partir des étiquettes $L(u)$ et $L(v)$. Comme pour le routage, nous chercherons à concevoir des schémas d'étiquetage informatif compacts, c'est-à-dire utilisant des étiquettes de petite taille. Une des motivations pour la conception des schémas d'étiquetage compacts est l'utilisation de tels schémas au sein de fichiers XML, afin de ne pas faire exploser la taille de ces fichiers. Une autre motivation est l'utilisation de schémas d'étiquetage pour la conception d'algorithmes distribués *auto-stabilisants*. Dans ce dernier cadre, l'utilisation de schémas compacts peut permettre de limiter les risques de corruption mémoire.

4.1 Routage compact

4.1.1 Schéma de routage

Etant donné un réseau modélisé par un graphe $G = (V, E)$, router dans G demande de concevoir trois objets :

— Un étiquetage des $\deg(u)$ arêtes incidentes à chaque sommet $u \in V$, de 1 à $\deg(u)$. Le numéro associé à une arête en u est appelé numéro de *port*, par analogie avec les numéros de port Internet. Notons que le numéro de port d'une arête $e = \{u, v\}$ en u peut être différent de celui en v. On note $\texttt{port}(u, v)$ le numéro de port de l'arête $\{u, v\}$ en son extrémité u.

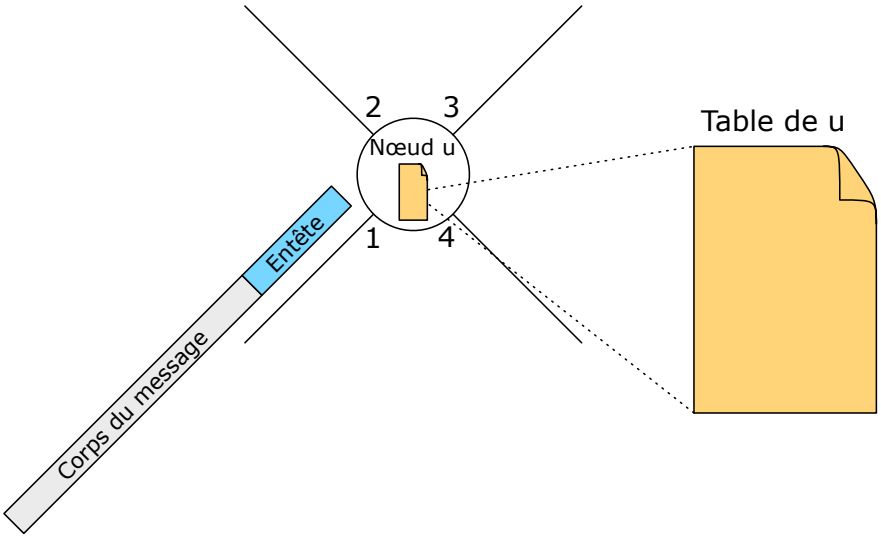

FIGURE 10 – *Un message arrivant au nœud u est routé selon les informations locales dont dispose u à propos de ce message, à savoir les informations stockées dans l'entête du message, celles stockées dans la table de routage de u, et, potentiellement, le port d'entrée du message en u (ici le port 1). Ces informations permettent de calculer le port de sortie du message, si celui-ci n'est pas à destination de u. Une fois cette information calculée, le message est renvoyé par le port de sortie en question.*

— Une affectation d'un *nom* à chaque nœud de G. On note $\texttt{nom}(u)$ le nom du nœud u.
— Une affectation d'une *table de routage* à chaque nœud. Chaque table peut être spécifique du nœud auquel elle est affectée. On note $\texttt{table}(u)$ la table de routage du nœud u.

Un message consiste en deux parties : l'*entête* et le *corps* du message. L'entête du message doit contenir des informations susceptibles de permettre de router le message de sa source à sa destination. Nous nous limiterons ici au cas où l'entête d'un message est égal au nom de la destination.

Une *fonction de routage* $\mathcal{R}$ est une fonction qui prend en entrée un nom, un numéro de port et une table, et qui retourne en sortie un numéro de port (voir la figure 10). Un message M à destination de v (donc d'entête $\texttt{nom}(v)$) entrant en u par le port $in(M) \in \{1, \ldots, \deg(u)\}$ sera routé sur le port $out(M) \in \{0, 1, \ldots, \deg(u)\}$ tel que :

$$out(M) = \mathcal{R}(in(M), \texttt{nom}(v), \texttt{table}(u))$$

Ici, $out(M) = 0$ indique que le message M est arrivé à destination, ce qui ne doit arriver que si $u = v$. En général, on ne considérera pas de fonction dépendant du port d'entrée. En ce cas,

$$out(M) = \mathcal{R}(\texttt{nom}(v), \texttt{table}(u)).$$

Definition 1 *Un* schéma *de routage pour une famille $\mathcal{F}$ de graphes est la donnée du quadruplé de fonctions $(\texttt{port}, \texttt{nom}, \texttt{table}, \mathcal{R})$ dont les trois premières peuvent dépendre du graphe $G \in \mathcal{F}$ auxquelles elles sont appliquées, et dont la dernière (la fonction de routage $\mathcal{R}$) ne peut dépendre que de $\mathcal{F}$.*

Notons que l'on peut s'intéresser au cas où les ports ne sont pas libres d'être fixés par le concepteur du schéma de routage (« designer-port model » en anglais), mais sont fixés a priori

(« fixed-port model » en anglais). Dans les deux cas, l'objectif est de concevoir des schémas de routage compacts, c'est-à-dire minimisant

$$\max_{u \in V} (|\texttt{nom}(u)| + |\texttt{table}(u)|).$$

La valeur ci-dessus est appelée *taille* du schéma. Notons qu'il existe un schéma trivial de taille $O(n \log n)$ bits pour tous les graphes d'au plus n nœuds, et ce même dans le modèle à ports fixés à priori. Les nœuds sont numérotés arbitrairement de 1 à n, et $\texttt{table}(u)$ consiste en un tableau de n entrées tel que $\texttt{table}(u)[i]$ contient le port de sortie de u sur lequel router tous les messages à destination du nœud de nom $i \in [n]$. Construire un tel schéma s'effectue en temps polynomial car il suffit de calculer les plus courts chemins entre toutes paires de nœuds pour remplir les tables de routage. Ce schéma peut donc être conçu pour router le long de plus courts chemins.

On peut montrer que, pour tout schéma de routage de plus courts chemins, il existe un graphe de n nœuds nécessitant $\Omega(n \log n)$ bits. Le schéma trivial ci-dessus est donc optimal en général. Il est toutefois possible de faire beaucoup mieux que $\Theta(n \log n)$ bits, par exemple en étudiant des familles de graphes particulières, ou en relaxant la contrainte de plus courts chemins. C'est ce que nous allons voir dans la suite.

4.1.2 Routage compact dans les arbres

L'étude de la construction de schémas de routage compacts dédiés aux arbres est au centre de la construction de schémas de routage compacts généraux. En effet, des schémas de routage compacts pour des graphes arbitraires peuvent être obtenus à partir de couvertures de graphes par des familles d'arbres : la fonction de routage pour un graphe G est obtenue à partir de la fonction de routage de chacun des arbres $T_1, T_2, \ldots$ couvrant G.

Nous allons construire un schéma de routage dans les arbres utilisant des noms et des tables de taille $O(\log n)$ bits, où n désigne le nombre de nœuds de l'arbre. Soit donc T un arbre de n nœuds. Soit r un nœud arbitraire de T. On considère maintenant T comme enraciné en r. Pour affecter des noms aux nœuds, on commence par associer à chaque nœud u une étiquette $\textsc{dfs}(u) \in [n]$, définie par l'ordre de visite des nœuds dans un parcours DFS visitant les enfants de tout nœud dans l'ordre décroissant de la taille des sous-arbres de T enraciné en ces enfants. Voir la figure 11(a) pour un exemple d'un tel étiquetage. Les numéros de port associés à chaque arête sont alors fixés comme suit. Soit u un nœud, et soient $v_1, \ldots, v_k$ ses enfants, avec $\textsc{dfs}(v_1) < \textsc{dfs}(v_2) < \cdots < \textsc{dfs}(v_k)$. On défini $\texttt{port}(u, v_i) = i$. Par ailleurs, si v est le parent de u, alors $\texttt{port}(u, v) = \deg(u)$.

On définit ensuite deux entiers $N_0(u)$ et $N_1(u)$ associés à chaque sommet u, désignant respectivement le nombre de nœuds dans le sous-arbre de T enraciné en u, et le nombre de nœuds dans le sous-arbre de T enraciné en l'enfant v_1 de u. (Si u est une feuille, alors $N_1(u) = 0$; Par ailleurs, $N_0(r) = n$). Dans l'exemple de la figure 11(a), $N_0(u) = 9$ et $N_1(u) = 4$. L'intérêt de N_0 apparaît immédiatement lorsque l'on note que tout nœud v dans le sous-arbre de T enraciné en u satisfait $\textsc{dfs}(v) \in [\textsc{dfs}(u), \textsc{dfs}(u) + N_0(u) - 1]$. Ainsi, si u reçoit un message à destination d'un nœud dont l'étiquette DFS n'est pas dans cet intervalle, alors u peut simplement transmettre ce message sur le port $\deg(u)$ conduisant au parent de u.

L'intérêt de N_1 apparaîtra lorsque nous aurons vu les dernières structures de données dont nous avons besoin. Ces structures sont notées $\textsc{gl}(u)$ et $\textsc{pl}(u)$, et appelées respectivement *grande liste* de u et *petite liste* de u. La liste $\textsc{gl}(u)$ est définie comme la liste des numéros de port des arêtes successives à emprunter pour aller de la racine r à u. Dans l'exemple de la figure 11(a),

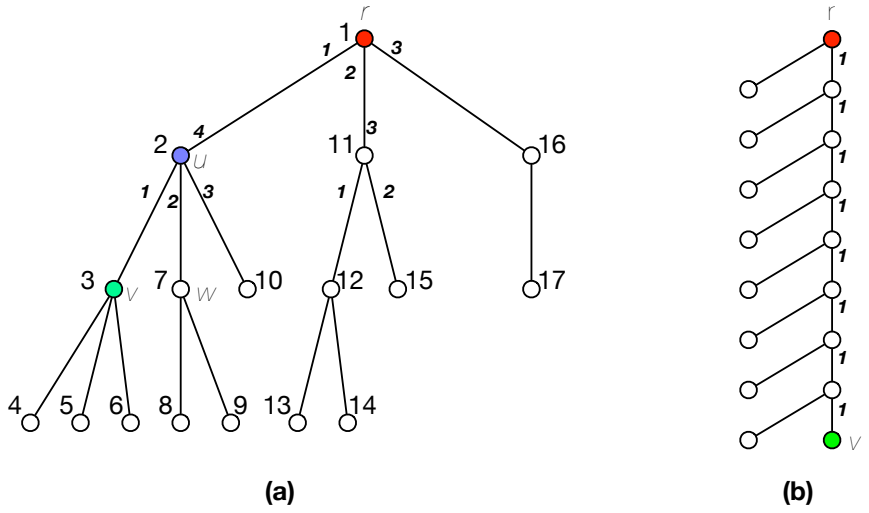

FIGURE 11 – *Routage compact dans les arbres*

on a $\text{GL}(v) = (1, 1)$ et $\text{GL}(w) = (1, 2)$. Notons que si un nœud v est un descendant d'un nœud u dans T alors $\text{GL}(u)$ est un préfixe de $\text{GL}(v)$. En conséquence, si u reçoit un message à destination d'un nœud v dont l'étiquette DFS est dans l'intervalle $[\text{DFS}(u), \text{DFS}(u) + N_0(u) - 1]$ alors, pour déterminer le numéro de port sur lequel renvoyer ce message, il suffit à u de lire le numéro de port en position $|\text{GL}(u)| + 1$ dans la liste $\text{GL}(v)$. Ainsi, N_0 et GL suffisent pour router. Ce routage n'est malheureusement pas compact car $\text{GL}(u)$ peut être très longue. C'est pas exemple le cas de $\text{GL}(v) = (1, 1, \ldots, 1)$ sur la figure 11(b), de longueur $\approx \frac{n}{2}$. Pour réduire $\text{GL}(u)$, il suffit d'observer que si u reçoit un message à destination d'un nœud v dont l'étiquette DFS est dans l'intervalle $[\text{DFS}(u), \text{DFS}(u) + N_1(u) - 1]$ alors u peut renvoyer le message sur le port 1. On définit donc $\text{PL}(u)$ à partir de $\text{GL}(u)$ en supprimant toutes les occurrences du port 1. Dans l'exemple de la figure 11(a), on a ainsi $\text{PL}(w) = (2)$, et, dans celui de la figure 11(b), on a $\text{PL}(v) = ()$, la liste vide. Nous allons voir dans la suite que les petites listes ont une taille qui peut être jusqu'à exponentiellement plus petite que celle des grandes listes.

Le schéma de routage est alors défini comme suit :

$$\begin{aligned} \texttt{nom}(u) &= (\text{DFS}(u), \text{PL}(u)) \\ \texttt{table}(u) &= (\text{DFS}(u), \text{PL}(u), N_0(u), N_1(u)) \end{aligned}$$

avec comme fonction de routage

$$\mathcal{R}(\texttt{nom}(v), \texttt{table}(u)) = \begin{cases} 0 & \text{si } \text{DFS}(v) = \text{DFS}(u) \\ 1 & \text{si } \text{DFS}(v) \in [\text{DFS}(u) + 1, \text{DFS}(u) + N_1(u)] \\ p & \text{si } \text{DFS}(v) \in [\text{DFS}(u) + N_1(u) + 1, \text{DFS}(u) + N_0(u) - 1] \\ & \quad \text{où } p \text{ est en position } |\text{PL}(u)| + 1 \text{ dans la liste } \text{PL}(v) \\ \deg(u) & \text{sinon} \end{cases}$$

Le fait que $\mathcal{R}$ soit correcte est par construction. Nous allons maintenant voir que, pour tout nœud u, $\texttt{nom}(u)$ et $\texttt{table}(u)$ se codent chacun sur $O(\log n)$ bits. Puisque $\text{DFS}(u)$, $N_0(u)$ et $N_1(u)$ sont des entiers entre 0 et n, il suffit de vérifier que $\text{PL}(u)$ se code sur $O(\log n)$ bits.

La liste $\text{PL}(u) = (p_1, \ldots, p_k)$ peut se coder sur $2\sum_{i=1}^{k} \lceil \log p_i \rceil$ en codant en binaire chacun des p_i à la suite les uns des autres, et en utilisant un tableau annexe pour indiquer les séparations

entre p_i et p_{i+1}, pour $i = 1, \ldots, k-1$. Donc $\text{PL}(u)$ peut se coder sur $O(k + \sum_{i=1}^{k} \log p_i)$ bits en majorant chaque partie entière $\lceil x \rceil \leq x + 1$. Ainsi, $\text{PL}(u)$ peut se coder sur $O(k + \log \prod_{i=1}^{k} p_i)$ bits. Pour $i = 1, \ldots, k$, notons

$$p_i = \texttt{port}(u_i, v_i)$$

où v_i est le $p_i$ème enfant de u_i, lorsque les enfants sont ordonnés par ordre décroissant de la taille de leurs sous-arbres. Observons ici que soit $u_{i+1} = v_i$, soit u_{i+1} est un descendant de v_i. Pour tout nœud w, notons $T(w)$ le sous arbre de T enraciné en w. Donc, en particulier

$$|T(u_{i+1})| \leq |T(v_i)| < |T(u_i)|$$

pour tout $i = 1, \ldots, k-1$. Plus précisément, puisque $p_i \geq 2$, $T(v_i)$ est au moins le deuxième plus grand sous-arbre parmi ceux enracinés aux enfants de u_i, donc

$$|T(v_i)| \leq \frac{1}{2} |T(u_i)| \leq \frac{1}{2} |T(v_{i-1})|$$

et donc $|T(v_i)| \leq n/2^i$. Il en découle que $k \leq \log n$. Pour borner $\log \prod_{i=1}^{k} p_i$, observons qu'en fait $|T(v_i)| \leq |T(u_i)|/p_i$ puisqu'au moins $p_i - 1$ sous-arbres parmi ceux enracinés aux enfants de u_i sont de tailles au moins $|T(v_i)|$. Donc $p_i \leq |T(u_i)|/|T(v_i)|$. En conséquence, comme $|T(u_{i+1})| \leq |T(v_i)|$, on obtient :

$$\log \prod_{i=1}^{k} p_i \leq \log \prod_{i=1}^{k} \frac{|T(u_i)|}{|T(v_i)|} \leq \log \prod_{i=1}^{k} \frac{|T(u_i)|}{|T(u_{i+1})|} \leq \log \Big(\frac{n}{|T(u_2)|} \cdot \frac{|T(u_2)|}{|T(u_3)|} \cdot \cdots \cdot \frac{|T(u_{k-1})|}{1} \Big) \leq \log n.$$

On en déduit donc le résultat suivant :

Théorème 11 *Il existe un schéma de routage de plus courts chemins dans les arbres utilisant des noms et des tables de $O(\log n)$ bits lorsque les ports peuvent être fixés par le concepteur du schéma.*

Que se passe-t-il dans le cas où les ports ne peuvent pas être fixés par le concepteur du schéma ? En ce cas, il est possible de faire presque aussi bien que le théorème précédent, à un facteur logarithmique près. Plus spécifiquement :

Théorème 12 *Il existe un schéma de routage de plus courts chemins dans les arbres utilisant des noms et des tables de $O(\log^2 n / \log \log n)$ bits lorsque les port sont fixés a priori.*

Preuve. La construction du schéma repose sur les mêmes idées que le schéma du théorème 11. Fixons $\ell \geq 1$. Pour tout nœud u, soit $T_1, \ldots, T_d$ les $d \geq 0$ sous-arbres enracinés aux d enfants $u_1, \ldots, u_d$ de u, avec $|T_1| \geq |T_2| \geq \cdots \geq |T_d|$. Définissons $N_i(u) = |T_i|$, $i = 1, \ldots, \ell$. (Si $\ell > d$, alors $N_i(u) = \perp$). On définit également la *très petite liste* de u, $\text{TPL}(u)$, comme $\text{GL}(u)$ où tous les ports relatifs aux arêtes conduisant aux racines $u_1, \ldots, u_\ell$ des arbres $T_1, \ldots, T_\ell$ sont supprimés. Enfin, soit $\text{COR}(u)$ le *tableau de correspondance* de u, c'est-à-dire $\text{COR}(u) = (q_0, q_1, \ldots, q_\ell)$ où q_0 est le numéro de port de l'arête incidente à u menant à son parent, et, pour $i \in \{1, \ldots, \ell\}$, q_i est le numéro de port de l'arête incidente à u menant à la racine u_i de T_i.

Le schéma de routage est alors défini par $\texttt{nom}(u) = (\text{DFS}(u), \text{TPL}(u), \text{COR}(u))$, et

$$\texttt{table}(u) = (\text{DFS}(u), \text{TPL}(u), N_0(u), N_1(u), \ldots, N_k(u)).$$

En posant $S_0 = 0$ et $S_i = \sum_{j=1}^{i} N_j(u)$ pour $i = 1, \ldots, \ell$, la fonction de routage est définie comme suit :

$$\mathcal{R}(\texttt{nom}(v), \texttt{table}(u)) = \begin{cases} 0 & \text{si } \textsc{dfs}(v) = \textsc{dfs}(u) \\ q_i & \text{si } \textsc{dfs}(v) \in [\textsc{dfs}(u) + S_{i-1}(u) + 1, \textsc{dfs}(u) + S_i(u)], i = 1, \ldots, \ell \\ p & \text{si } \textsc{dfs}(v) \in [\textsc{dfs}(u) + S_k(u) + 1, \textsc{dfs}(u) + N_0(u) - 1] \\ & \quad \text{où } p \text{ est en position } |\textsc{tpl}(u)| + 1 \text{ dans la liste } \textsc{tpl}(v) \\ q_0 & \text{sinon} \end{cases}$$

Soit $\textsc{tpl}(u) = (p_1, \ldots, p_k)$ où $p_i = \texttt{port}(x_i, y_i)$. Puisque p_i conduit à un sous-arbre $T(y_i)$ qui est au moins le $(\ell + 1)$ème sous-arbre le plus gros parmi ceux enracinés aux enfants de x_i, on a

$$|T(y_i)| < |T(u_i)|/(\ell + 1)$$

et donc $k \leq \frac{\log n}{\log(\ell+1)}$. Donc $\textsc{tpl}(u)$ se code sur au plus $O(\frac{\log^2 n}{\log(\ell+1)})$ bits car $p_i \leq n$ pour tout i. Par ailleurs, $\textsc{cor}(u)$ se code sur $O(\ell \log n)$ bits. Il en découle que $\texttt{nom}(u)$ est sur $O(\ell \log n + \frac{\log^2 n}{\log(\ell+1)})$ bits. Enfin $\texttt{table}(u)$ se code sur $O(\ell \log n)$ bits car elle contient ℓ entiers $N_i(u)$, $i = 1, \ldots, \ell$.

Le théorème découle de l'analyse ci-dessus, en choisissant $\ell = \lceil \frac{\log n}{\log \log n} \rceil$. □

Notons que la borne supérieure du théorème 12 est la meilleure borne qu'il est possible d'atteindre dans le modèle à ports fixés. En effet, pour tout schéma de routage dans ce modèle, il existe une famille infinie d'arbres $\{T_n, n \geq 1\}$ telle que le schéma vérifie $\max_{u \in T_n}(|\texttt{nom}(u)| + |\texttt{table}(u)|) = \Omega(\frac{\log^2 n}{\log \log n})$ bits.

4.1.3 Un schéma de routage compact d'élongation 3

4.2 Etiquetage informatif compact

4.2.1 Etiquetage d'adjacence et d'ancêtre

4.2.2 Etiquetage de distance

5 Le calcul distribué asynchrone

Cette section est consacrée à l'asynchronisme, c'est-à-dire au fait que les nœuds d'un système distribué peuvent avoir des horloges indépendantes, s'exécutant à des rythmes différents. Nous considérerons deux modèles : le modèle avec mémoire partagée, et le modèle par passage de messages. Dans les deux cas, nous verrons que l'asynchronisme peut limiter très significativement les performances d'un système distribué, voire empêcher la résolution de tâches élémentaires.

5.1 Mémoire partagée

Le modèle à mémoire partagée est une abstraction permettant de capturer conceptuellement le comportement de *threads* au sein d'un cœur d'un processeur multi-cœur, d'un cœur au sein d'un tel processeur, voire d'un processeur au sein d'une machine multi-processeur. (Plusieurs machines multi-processeur sont généralement interconnectés physiquement par un réseau de communication plutôt que par une mémoire partagée).

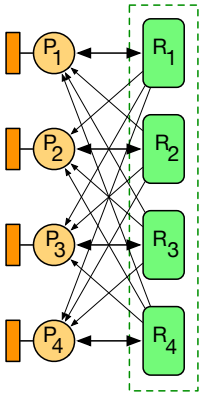

FIGURE 12 – *Un quadri-processeur à mémoire partagée*

5.1.1 Un modèle asynchrone

Nous considérons un ensemble de n processeurs $p_1, \ldots, p_n$, disposant chacun de sa mémoire privée (ce sont chacun des machines RAM), et ayant chacun un accès à une mémoire commune (voir la figure 12). Plus spécifiquement, la mémoire commune est découpée en n registres $R_1, \ldots, R_n$, de taille arbitrairement grande. Le processeur p_i à l'accès exclusif en écriture au registre R_i. L'écriture d'une valeur v par p_i dans R_i s'effectue au moyen de l'instruction "écrire(v)" exécutée par p_i. Nous allons considérer deux mode de lecture dans la suite. L'instruction "lire(R_j)" retourne au processeur p_i exécutant cette instruction le contenu du registre R_j. L'instruction "photographier" retourne au processeur exécutant cette instruction le contenu entier de la mémoire, c'est-à-dire de tous les registres $R_1, \ldots, R_n$. Chacune de ces instructions de lecture/écriture est atomique, c'est-à-dire s'exécute instantanément.

Dans ce chapitre, nous allons étudier les limitations causées par l'asynchronisme en terme d'incapacité à résoudre certaines tâches élémentaire. Nous ne nous intéresserons pas au temps nécessaire à l'exécution de ces tâches. Ainsi, nous ne limitons pas le volume de données pouvant être écrites en une étape par l'instruction écrire, ni nous ne limitons le volume des données pouvant être récupérées en une étape par l'instruction lire, ou l'instruction photographier. Notons qu'il est possible de simuler l'instruction photographier par une séquence d'instruction lire (voir algorithme 1).

Algorithm 1 Photo floue exécutée par le processeur p

```
1: for i ∈ {1, ..., n} do
2:     lire(R_i)
3: end for
```

Néanmoins, cette simulation n'est pas idéale car elle ne respecte pas l'atomicité. En particulier, si p_j et p_k exécutent tous deux l'algorithme 1, le commençant au même instant, et le terminant au même instant, il est tout à fait possible que p_j et p_k n'aient pas récupéré le même contenu mémoire. En effet, si chaque instruction read(R_i) est atomique, il est possible que des écritures d'autres processeurs viennent modifier le contenu d'un registre R_i entre la lecture de p_j et celle de p_k. Par exemple, pour $i \notin \{j, k\}$ on peut avoir une écriture de p_i intercalée entre l'instruction lire(R_i) de p_j et l'instruction lire(R_i) de p_k. L'algorithme 1 ne réalise donc pas une photographie atomique. Il existe néanmoins des façons de réaliser la photographie atomique.

Celle-ci ne sera toutefois pas présentée dans ce cours, et nous admettrons donc la capacité de réaliser une photographie atomique. [6]

Les pannes. L'exemple de la photo floue est une première illustration des problèmes que l'on peut rencontrer face à l'asynchronisme des processeurs. Ces problèmes s'accentuent encore en présence de *pannes*. Un processeur subissant une panne *crash* cesse simplement de fonctionner (il ne fait plus aucune lecture/écriture, et ne modifie pas son état interne), à jamais. Les pannes crash rentrent naturellement dans le cadre de l'asynchrone puisqu'un telle panne revient au cas extrême d'une horloge de vitesse nulle [7]. En particulier, la présence de pannes crash empêche les processeur d'attendre. En effet, si le code d'un processeur p inclut une instruction "attendre jusqu'à ce que q effectue l'action a", alors une panne crash de q avant d'avoir effectué a bloque p. En présence de pannes crash, le code ne doit pas contenir d'instructions "attendre" qui fassent références aux actions d'autres processeurs. On dit d'un processeur qui subit une panne crash qu'il est *fautif*.

5.1.2 Le consensus

En sus de l'accès à son registre privé et aux registres publics de la mémoire partagée, chaque processeur peut avoir accès à des ressources communes partagées, dont l'accès ne peut se faire simultanément par plus d'un processeur. (Un exemple typique d'une telle ressource est un compteur partagé par plusieurs *threads*). L'accès à une ressource critique peut typiquement se réaliser au moyen de verrous. Lorsqu'un processeur veut accéder à la ressource, il cherche à acquérir le verrou de cette ressource. Lorsqu'il l'obtient, il verrouille la ressource, et entre en section critique (la partie du code liée à cette ressource). Lorsqu'il termine sa section critique, le processeur relâche le verrou, et la ressource est maintenant à disposition des autres processeurs voulant accéder à la ressource.

La technique ci-dessus offre cependant un inconvénient majeur : si le processeur ayant accédé à la ressource, et ayant donc verrouillé celle-ci, tombe en panne et s'arrête, alors aucun autre processeur ne pourra accéder à la ressource, ce qui peut entrainer le blocage de tout le système !

Une autre façon de contrôler l'accès à une ressource critique consiste à s'assurer que tous les processeurs accédant à la ressource écrivent la même chose. De cette façon, n'importe quel processeur peut écrire, et les processeurs peuvent écrire les uns sur les autres, cela n'aura pas d'importance puisque tous écriront la même chose. Pour réaliser cela, il faut néanmoins que tous les processeurs se mettent d'accord sur la valeur à écrire, c'est-à-dire réalise un *consensus*.

La spécification du consensus ci-dessous inclut la notion de pannes. Rappelons que les pannes considérées dans ce cours sont uniquement des pannes crash.

Definition 2 (Spécifications du consensus.) *Chaque processeur p propose une valeur $v = val(p)$ dans un certain domaine de valeurs, et doit décider une valeur en respectant les trois règles suivantes :*

Terminaison : *chaque processeur non fautif doit décider ;*

Validité : *toute valeur décidée doit être une valeur proposée ;*

Accord : *tous les processeurs décidant une valeur doivent décider la même valeur.*

6. Notons que le même problème se pose de fait pour l'instruction lire. En effet, un registre peut consister en un nombre considérable de variables, lues les unes après les autres.

7. D'autres types de pannes pourraient être considérées (pannes *transitoires*, pannes *byzantines*, etc.), mais cela dépasserait le cadre du cours.

Avant d'étudier sous quelles conditions le consensus peut être résolu, nous allons considéré dans la section suivante l'exemple d'une tâche plus simple : l'*accord faible*.

5.1.3 Résolution du problème de l'accord faible

Les spécifications de l'*accord faible* sont les suivantes. Chaque processeur p propose une valeur $v = \mathit{val}(p)$ dans un certain domaine de valeurs, et doit décider une sortie en respectant les trois règles suivantes :

Terminaison : chaque processeur non fautif doit décider ;

Validité : toute valeur décidée doit être une valeur proposée ou $\perp$, et si aucun processeur n'est fautif, alors au moins un processeur doit décider une valeur différente de $\perp$;

Accord faible : tous les processeurs décidant une valeur différente de $\perp$ doivent décider la même valeur.

Nous allons montrer que l'algorithme 2 réalise l'accord faible dans le modèle asynchrone. Afin de simplifier la présentation de l'algorithme, nous utilisons l'instruction "photographier".

Algorithm 2 Accord faible exécuté par le processeur p proposant $v = \mathit{val}(p)$

1: écrire $(\mathrm{Id}(p), v)$ ▷ 1ère écriture
2: photographier pour extraire $\mathit{vue} = \{(i_1, v_{i_1}), \ldots, (i_k, v_{i_k})\}$ ▷ 1ère lecture
3: écrire $(\mathrm{Id}(p), \mathit{vue})$ ▷ 2ème écriture
4: photographier pour extraire $\mathit{meta\text{-}vue} = \{(j_1, \mathit{vue}_{j_1}), \ldots, (j_\ell, \mathit{vue}_{j_\ell})\}$ ▷ 2ème lecture
5: $\mathit{vue\text{-}min} \leftarrow \bigcap_{r=1}^{\ell} \mathit{vue}_{j_r}$ ▷ vue minimum par inclusion
6: **if** pour tout i tel que $v_i \in \mathit{vue\text{-}min}$ on a $\mathit{vue}_i \in \mathit{meta\text{-}vue}$ **then**
7: décider la valeur minimum w dans $\mathit{vue\text{-}min}$
8: **else**
9: décider $\perp$
10: **end if**

Notons que $\bigcap_{r=1}^{\ell} \mathit{vue}_{j_r}$ est simplement la plus petite vue inclue dans $\mathit{meta\text{-}vue}(p)$. En effet, les vues des n processeurs obtenues par photographie sont imbriquées les unes dans les autres : la plus petite vue est celle du processeur ayant effectué sa première lecture le plus tôt, la deuxième vue la plus petite est celle du processeur ayant effectué sa première lecture le deuxième, etc.

La terminaison est satisfaite puisque le flot d'instructions de tout processeur p ne dépend pas des autres processeurs (tout processeur non fautif décide). Par ailleurs, la sortie de tout processeur est bien soit une valeur proposée w (w est proposée puisque w est dans une vue), soit $\perp$.

L'accord faible est également réalisé. En effet, supposons pas contradiction que deux processeurs p et p' différents décident $w \neq \perp$ et $w' \neq \perp$, respectivement, avec $w \neq w'$. Supposons alors, sans perte de généralité, que $w < w'$. Soit q et q' deux processeurs tels que $\mathit{vue\text{-}min}(p) = \mathit{vue}(q)$ et $\mathit{vue\text{-}min}(p') = \mathit{vue}(q')$. On a $w \notin \mathit{vue}(q')$ car sinon p' aurait décidé w. Donc $\mathit{vue}(q') \subset \mathit{vue}(q)$, et donc $\mathit{val}(q') \in \mathit{vue}(q) = \mathit{vue\text{-}min}(p)$. Or $\mathit{vue}(q') \notin \mathit{meta\text{-}vue}(p)$ car sinon on aurait $\mathit{vue\text{-}min}(p) = \mathit{vue}(q')$. Donc, p ne satisfait pas la condition de l'instruction 6, et décide $\perp$, contradiction.

Pour établir la validité, nous utilisons le résultat suivant :

Lemma 7 *During the execution of Algorithm 2, if some process p decides $\perp$, then there exists a process $q \neq p$ such that q performed its first write before the first read of p, and p performed its second read before the second write of q.*

Preuve. Assume first, for the purpose of contradiction, that no processes q performed their first writes before the first read of p. Then the view of process p is reduced to its own input, i.e., $\mathit{vue}(p) = \{(\mathrm{Id}(p), \mathit{val}(p))\}$. As a consequence, independently from what the other processes do, we get that p computes $\mathit{vue\text{-}min}(p) = \{(\mathrm{Id}(p), \mathit{val}(p))\}$, and decides $\mathit{val}(p)$, contradicting the fact that p decides $\perp$. Hence, some processes q performed their first writes before the first read of p. Let Q be the set of such processes.

Assume now, again for the purpose of contradiction, that, for every $q \in Q$, process p performed its second read after the second write of q. This implies that $\mathit{meta\text{-}vue}(p)$ includes $\mathit{vue}(q)$ for every $q \in Q$. Hence, $\mathit{vue\text{-}min}(p) \subseteq \mathit{vue}(p) \cap \big(\cap_{q \in Q} \mathit{vue}(q)\big)$. Since the views are totally ordered by inclusion, we get that $\mathit{vue\text{-}min}(p) \subseteq \mathit{vue}(q^*)$ for some $q^* \in Q \cup \{p\}$. Note that the first read of q^* must have been performed no later that the first read of p. Let $v_i \in \mathit{vue}(q^*)$. Process i has performed its first write before the first read of q^*, which implies that $i \in Q \cup \{p\}$. Therefore, p performed its second read not before the second write of i. Hence, $\mathit{vue}(i) \in \mathit{meta\text{-}vue}(p)$. This holds for every i such that $v_i \in \mathit{vue}(q^*)$, thus $\mathit{vue}(i) \in \mathit{meta\text{-}vue}(p)$ for every i such that $v_i \in \mathit{vue\text{-}min}(p)$. This leads p to decide a value $w \neq \perp$. This contradiction completes the proof. □

On dit du processeur q du lemme ci-dessus qu'il gêne p. Ainsi, le lemme 7 indique que si un processeur p décide $\perp$, alors ce processeur a été gêné par un autre processeur q. Pour montrer la validité, on note tout d'abord que si aucun processeur n'est fautif, alors au moins un processeur n'est pas gêné. En effet, si chaque processeur est gêné par un autre processeur, alors peut construire une séquence $p_1, \ldots, p_k$ avec $k \geq 2$, où p_{i+1} gêne p_i pour tout $i = 1, \ldots, k-1$, et p_1 gêne p_k. Donc, en particulier, la seconde écriture de p_{i+1} et après la seconde lecture de p_i. Ainsi, la seconde photographie de p_{i+1} et après la seconde photographie de p_i. De même, la seconde photographie de p_1 et après la seconde photographie de p_k, ce qui conduit à une contradiction. Donc, soit p un processeur non gêné. Le lemme 7 implique que p décide une valeurs différente de $\perp$. Ceci conclut la démonstration que l'algorithme 2 réalise la validité de l'accord faible.

5.1.4 Impossibilité de l'accord

Nous venons de voir dans la section précédente qu'il est possible de réaliser l'accord faible entre les processeurs. Qu'en est-il du consensus ? La situation est radicalement différente : il est impossible de réaliser le consensus dans un système asynchrone en présence de pannes !

Théorème 13 *Il n'est pas possible de réaliser le consensus dans un système asynchrone en présence de pannes (crash).*

Preuve. Nous allons démontrer un résultat plus fort, en montrant que même le consensus *binaire* (les valeurs proposées sont 0 ou 1) est impossible. La preuve est par contradiction. Supposons l'existence d'un algorithme $\mathcal{A}$ permettant de résoudre le consensus binaire. Une *configuration* est définie par l'ensemble des états des n registres des n processeurs, et de leurs mémoires privées. La *valence* d'une configuration C est définie comme le sous-ensemble de $\{0, 1\}$ formé des valeurs pouvant être décidées par $\mathcal{A}$ à partir de la configuration C. Une configuration C est *bivalente* si sa valence est égale à $\{0, 1\}$. C'est-à-dire, partant de C, $\mathcal{A}$ peut décider 0, mais peut aussi

décider 1, selon son exécution (les processeurs sont asynchrones) et les pannes. Une configuration non bivalente est dite *monovalente*.

Observation 3 *Il existe une configuration initiale bivalente.*

En effet, soit C_0 la configuration initiale dans laquelle tous les processeurs proposent 0. Pour $i = 1, \ldots, n$, soit C_i la configuration initiale $(1, \ldots, 1, 0, \ldots, 0)$ consistant en $p_1, \ldots, p_i$ proposant 1, et $p_{i+1}, \ldots, p_n$ proposant 0. Supposons que toutes les configurations $C_0, C_1, \ldots, C_n$ soient monovalentes. Par définition du consensus, $\mathcal{A}$ doit nécessairement décider 0 pour C_0, et 1 pour C_n. Dit autrement, C_0 est 0-valente, et C_n est 1-valente. Soit $i \geq 1$ le plus petit indice tel que C_i est 1-valente. Considérons maintenant une exécution de $\mathcal{A}$ dans laquelle le processeur p_i ne participe pas (très lent, il ne fait aucune étape de calcul, lecture ou écriture). Cette exécution sera identique pour C_i et C_{i-1}. Or $\mathcal{A}$ décide 0 pour C_{i-1} et 1 pour C_i, contradiction. Donc, il existe une configuration C_j bivalente, $j \in \{0, 1, \ldots, n\}$, ce qui conclut la preuve de l'observation 3.

On dit que C et C' sont *similaires pour* p_i, noté $C \sim_i C'$, si C et C' sont identiques du point de vue de p_i. C'est-à-dire, dans C et C', l'état de chaque registre partagé est identique, et l'état interne de p_i est identique.

Observation 4 *Si C et C' sont deux configurations monovalentes avec $C \sim_i C'$, alors C et C' sont de même valence.*

En effet, il suffit de consider une execution de $\mathcal{A}$ dans laquelle seul le processeur p_i participe. Puisque C et C' sont identiques du point de vue de p_i, $\mathcal{A}$ décide la même valeurs pour ces deux configurations.

Si C est bivalente, et qu'une étape de p_i à partir de C résulte en une configuration C' monovalente, on dit que p_i est *critique* pour C. De façon non formelle, notons Cp_i "la" configuration résultat d'une étape de p_i à partir de C. En fait, cette configuration dépend de si p_i écrit ou lit (et où), ou calcule.

Observation 5 *Montrons que, pour toute configuration bivalente C, il existe au moins un processeur p_i qui n'est pas critique pour C.*

En effet, par contradiction, supposons que tous les processeurs sont critiques pour C. Puisque C est bivalente, il existe $i \neq j$ tels que Cp_i est 0-valente, et Cp_j est 1-valente. Si p_i et p_j accèdent des registres différents, ou si tous les deux lisent le même registre, alors $Cp_ip_j = Cp_jp_i$, ce qui est impossible car la première configuration est 0-valente alors que la seconde configuration est 1-valente. Donc p_i écrit dans le registre R, et p_j lit ou écrit le registre R. Dans les deux cas, $Cp_jp_i \sim_i Cp_i$. Or Cp_jp_i est 1-valente alors que Cp_i est 0-valente, ce qui est impossible pour deux configurations similaires pour p_i. Il existe donc au moins un processeur qui n'est pas critique pour C, ce qui conclut la preuve de l'observation 5.

On construit une exécution infinie $\mathcal{E}$ de $\mathcal{A}$ de la façon suivante. Soit C_0 une configuration initiale bivalente, et $p^{(0)}$ un processeur qui n'est pas critique pour C_0. L'exécution $\mathcal{E}$ laisse $p^{(0)}$ faire une étape, résultant en la configuration bivalente C_1. De même, soit $p^{(1)}$ un processeur qui n'est pas critique pour C_1. L'exécution $\mathcal{E}$ laisse $p^{(1)}$ faire une étape, résultant en la configuration bivalente C_2. Ainsi de suite, afin de construire une suite infinie $(C_i)_{i \geq 0}$ de configurations, toutes bivalentes. Dans cette exécution, $\mathcal{A}$ ne termine jamais, contradiction. L'algorithme $\mathcal{A}$ ne peut donc pas exister. □

Notez que la démonstration ci-dessus utilise des exécutions dans lesquelles tous les processeurs, sauf un, sont fautifs (pour montrer que deux configurations similaires pour un processeur ont la même valence). Il est en fait possible de montrer que, même en présence d'*une seule panne* (c'est-à-dire dans un modèle avec promesse stipulant qu'au plus un processeur peut tomber en panne), le consensus reste impossible. Ce résultat est souvent appelé "Théorème FLP" du nom de ses trois auteurs Michael Fischer, Nancy Lynch, et Michael Paterson (1985).

5.2 Passage de messages

Non documenté.

5.2.1 Election d'un leader

Non documenté.

5.2.2 L'algorithme de MST de Gallager, Humblet et Spira

Non documenté.

6 Algorithmes parallèles

(voir [2]).

6.1 Architectures parallèles

TBD

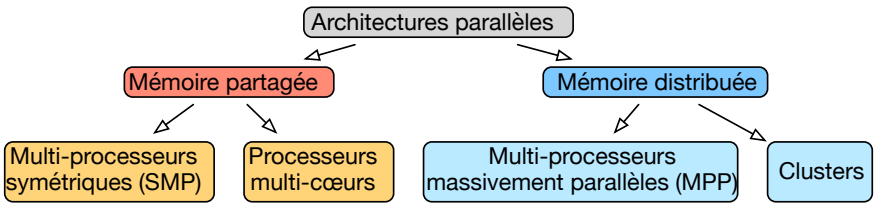

FIGURE 13 – *Une classification des architectures parallèles*

Mémoire partagée : Bus, crossbar, etc.

Mémoire distribuée : réseaux (grilles, hypercubes, etc.), switch multi-étages, etc.

Processeur vectoriel : pipeline d'instructions

6.2 Mesures de performances

TBD

— Facteur d'accélération (*speep up*) : $s(n) = T(1)/T(n)$
— Efficacité : $e(n) = s(n)/n$

Collection de machines RAM : $s(n) \leq n$ et donc $e(n) \in [0, 1]$.

6.3 Opérations arithmétiques élémentaires

En guise de premier exemple d'algorithme parallèle, considérons l'addition de deux entiers x et y codés en binaire chacun sur n bits. De la rapidité de cette opération élémentaire dépend évidemment les performances de la plupart des programmes utilisant des opérations arithmétiques. L'algorithme d'addition tel que nous le connaissons tous depuis l'école élémentaire est toutefois intrinsèquement séquentiel, du fait de la propagation de la retenue de la droite vers la gauche (c'est-à-dire des bits de poids faibles vers les bits de poids forts). Il n'est pas possible de connaitre le $i$ème bit z_i de $z = x + y$ sans connaitre x_i et y_i, ainsi que la retenue issue de l'addition de $x_{i-1} \dots x_0$ et $y_{i-1} \dots y_0$. Nous allons voir qu'il est toutefois possible d'*anticiper* la valeur de ce bit, et ainsi de paralléliser l'addition.

Afin de paralléliser l'addition, classifions les paires $(x_i, y_i) \in \{0, 1\}^2$, $i = 0, \dots, n-1$, en trois catégories : s, pour « stoppe », si $x_i + y_i = 0$; p, pour « propage », si $x_i + y_i = 1$; et g, pour « génère », si $x_i + y_i = 2$. Notons que si la paire i est de type s ou g, alors on peut commencer l'addition à partir de la position $i + 1$ sans attendre le résultat de la retenue issue de $x_{i-1} \dots x_0$ et $y_{i-1} \dots y_0$. En effet, si la paire i est de type s, alors cette retenue ne sera pas propagée après cette paire. Et si la paire i est de type g, alors il y a une retenue après cette paire quelque soit celle issue des bits de poids plus faibles. Nous allons voir comment utiliser cette remarque pour toutes les paires (x_i, y_i). Voir la figure 14 pour une illustration.

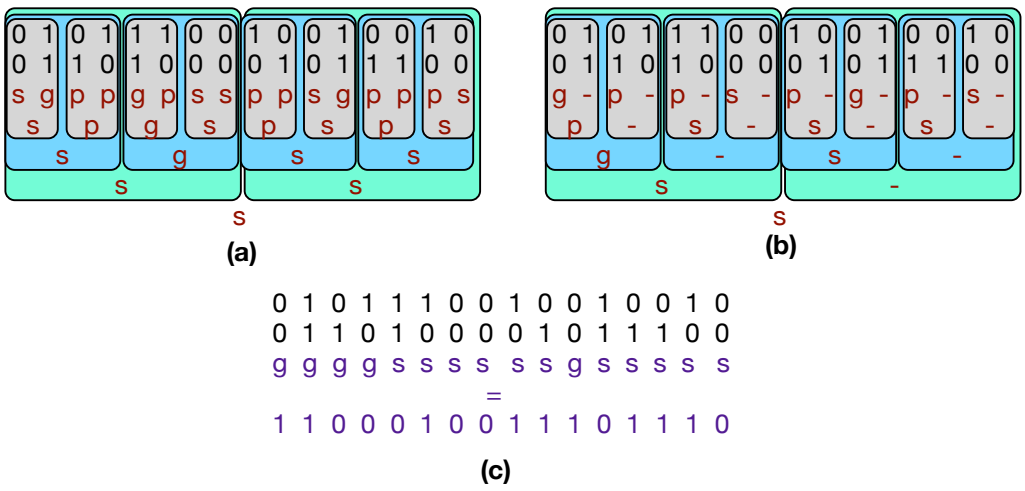

FIGURE 14 – *Illustration de l'algorithme parallèle d'addition*

A FINIR

6.4 Le tri parallèle

TBD

6.5 Opérations matricielles

TBD

6.6 Les environnements de programmation parallèle

6.6.1 Environnement de programmation

Du fait du très grand nombre d'environnements parallèles, et des différences énormes entre ces environnements, il n'existe pas à ce jour de langage de programmation universel pour le parallélisme. La gestion des threads au sein d'un processeur multi-cœur est par exemple très différente de celle de tâches au sein d'une ferme de processeurs. Ainsi, la famille de languages Cilk (Cilk, Cilk++, Cilk Plus, etc.) est principalement dédiée à la programmation de threads. OpenMP (pour « Open Multi-Processing ») est une interface de programmation [8] pour les plateformes parallèles à mémoire partagée, basée sur des langages séquentiels comme C, voire même Fortran. MPI (pour « Message Passing Interface ») est une standardization de procédures pour la programmation des plateformes à passage de messages. Récemment, la programmation des GPU (pour « Graphics Processing Unit ») a fait l'objet du développement d'un grand nombre d'interfaces de programmation, dont en particulier CUDA, mais dont la liste exhaustive couvrirait des pages de ce document.

L'environnement utilisé pour la programmation d'un système parallèle donné a évidemment un impact fort sur la façon de programmer efficacement pour ce système, et donc sur le développement d'algorithmes pertinents. Nous nous contenterons dans ce cours d'étudier un unique exemple, récent, d'environnement de programmation parallèle, dédié plus particulièrement à la gestion de grands centres de données : MapReduce. L'objectif de MapReduce est principalement la manipulation de données distribuées de très grande taille afin de faciliter le « Cloud computing ». Il a été originellement développé par Google. Il est relativement extrême, au sens où le format imposé à un programme utilisant MapReduce est très spécifique. MapReduce est typiquement utilisé pour pagerank, google-image, etc.

6.6.2 MapReduce

L'originalité de MapReduce par rapport aux précédents modèles que nous avons pu voir dans ce document est qu'il est essentiellement paramétré par le nombre N de données (et non pas par le nombre de processeurs ou de machines). On suppose que chaque machine ne peut contenir plus de $S = N^{1-\epsilon}$ données en mémoire locale, où $\epsilon > 0$. Chaque machine ne peut donc, à tout instant, traiter plus de $N^{1-\epsilon}$ données, ni envoyer ou recevoir plus de $N^{1-\epsilon}$ données. L'objectif est de pouvoir résoudre le problème donné le plus rapidement possible, en utilisant un nombre de machines $M = O(N^{\epsilon})$ de façon à être optimal en mémoire.

A titre d'exemple, étant donné un graphe pondéré de n sommets et $N = n^{1+c}$ arêtes, il a été montré que, pour tout $\alpha \in]0, c]$, la construction d'un arbre couvrant de poids minimum peut se faire en $\lceil c/\alpha \rceil$ phases de MapRedude avec $M = O(n^{1+\alpha})$ machines de mémoire $S = O(n^{c-\alpha})$, ce qui correspond au modèle, puisque $SM = N$.

MapReduce est un environnement de programmation pour plateformes parallèles. Il suppose un système d'exploitation distribué gérant des données encodées sous forme de paires (c, v). Le premier élément de cette paire est appelé *clé*, et le second *valeur*. A titre d'exemple, une matrice M $n \times n$ peut être stockée sous la forme de paires clé-valeurs $((i, j), v)$ avec $v = M_{i,j}$, $1 \leq i, j \leq n$. De même, un ensemble $\mathcal{E}$ de fichiers html peut être stocké sous la forme $(@, F)$ où @ est l'adresse html d'un fichier, et F son contenu. La répartition des paires clé-valeur sur les processeurs est à la charge du système d'exploitation, et l'utilisateur ne contrôle pas cette

8. API, pour « Application programming interface » en anglais.

répartition. En particulier, l'équilibrage de la charge (i.e., du nombre de paires clé-valeur) entre les processeurs est à la charge du système.

MapReduce propose essentiellement deux fonctions permettant à l'utilisateur de manipuler des paires clé-valeur : la fonction *map* et la fonction *reduce*. La fonction map permet de créer un *ensemble de paires* clé-valeur à partir de toute paire clé-valeur. La fonction reduce permet de créer *une paire* clé-valeur à partir de toute paire clé-valeur dont la valeur regroupe l'ensemble des valeurs de *même clé* produite par map. On a donc :

Fonction map : $(c, v) \rightarrow \{(c_1, v_1), \ldots, (c_k, v_k)\}$
Fonction reduce : $(c, \{v_1, \ldots, v_k\}) \rightarrow (c', v)$

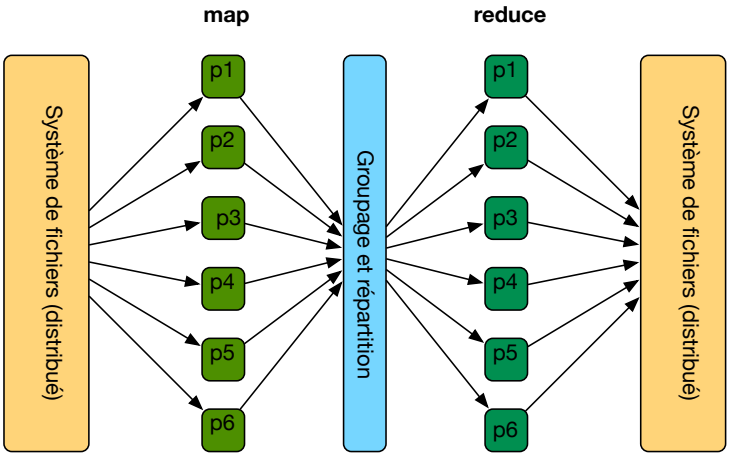

FIGURE 15 – *Le schéma général de MapReduce*

La figure 15 illustre le fonctionnement d'un programme MapReduce. L'utilisateur programme la fonction map, sous la contrainte de format imposée par MapRedude, c'est-à-dire la fonction map doit, pour chaque paire clé-valeur (c, v), créer un ensemble $\{(c_1, v_1), \ldots, (c_k, v_k)\}$ de paires clé-valeur. A la suite de l'exécution de la fonction map sur chacun des processeurs, le système *regroupe* toutes les valeurs produites par map partageant la même clé, créant ainsi un ensemble de paires $(c, \{v_1, \ldots, v_k\})$. Ces paires sont ensuite réparties par le système entre les processeurs. La fonction reduce, programmée par l'utilisateur, peut alors être appliquée sur les paires $(c, \{v_1, \ldots, v_k\})$, afin de retourner des paires clé-valeurs. Il convient de noter que le type des clés produites par map (et par reduce) peut être différent de celui des clés en entrées. Il en est de même du type des valeurs manipulées par map et par reduce. Notez également qu'une phase de MapReduce produit des paires clé-valeur à partir de paires clé-valeurs. La sortie d'une phase de MapReduce peut donc être utilisée en entrée d'une nouvelle phase de MapReduce.

Le facteur d'accélération et l'efficacité d'un programme MapReduce dépend beaucoup du nombre de clés que le système doit traiter lors de la phase de groupage et de répartition. L'utilisateur veillera donc à programmer des fonctions map ne générant pas un trop grand nombre de clés.

6.6.3 Exemples

Exemple 1 : Construction d'un index. L'objectif est de construire un index associant à chaque mot m d'un ensemble $\mathcal{M}$ de mots-clés, l'ensemble des documents html d'un corpus de

fichiers $\mathcal{F}$ contenant le mot clé m. On suppose que $\mathcal{F}$ est stocké sous forme de paires $(@, F)$ où $F \in \mathcal{F}$ est le contenu d'un document html, et @ son adresse html, distribuées sur les processeurs (machines) $p_1, \ldots, p_n$.

Algorithm 3 Fonction map exécutée sur la paire $(@, F)$

1: **for** $m \in F$ **do**
2: créer $(m, @)$
3: **end for**

La fonction map crée donc, pour chaque paire $(@, F)$, l'ensemble des paires $\{(m, @), m \in F\}$. Le système exécute ensuite l'opération de groupage et répartition des paires clé-valeur créées par la fonction map. Chaque processeur obtient ainsi un ensemble de paires $(m, \{@_1, \ldots, @_k\})$ pour un certain sous-ensemble de mots m. La paire de clé m contient comme valeur l'ensemble des adresses de tous les fichiers html contenant le mot m. La fonction reduce est ici simplement l'identité. Le système de fichiers récupère ainsi un ensemble de paires $(m, \{@_1, \ldots, @_k\})$ telles que, pour tout mot clé $m \in \mathcal{M}$, les pages contenant m sont aux adresses $@_1, \ldots, @_k$.

Exemple 2 : Calcul du nombre d'occurrences de mots. L'objectif est de compter le nombre d'occurence de chaque mot m d'un ensemble $\mathcal{M}$ de mots-clés dans les documents d'un corpus de fichiers $\mathcal{F}$ (par exemple les œuvres de Molière). Comme dans le premier exemple, on suppose que $\mathcal{F}$ est stocké sous forme de paires $(@, F)$ où $F \in \mathcal{F}$ est le contenu d'un document html, et @ son adresse html.

Algorithm 4 Fonction map exécutée sur la paire $(@, F)$

1: **for** $m \in F$ **do**
2: créer $(m, 1)$
3: **end for**

Le système exécute ensuite l'opération de groupage et répartition des paires clé-valeur créées par la fonction map. Chaque processeur obtient ainsi un ensemble de paires $(m, \{1, 1, \ldots, 1\})$.

Algorithm 5 Fonction reduce exécutée sur la paire $(m, (c_1, \ldots, c_k))$

1: $s \leftarrow 0$
2: **for** $c \in \{c_1, \ldots, c_k\}$ **do**
3: $s \leftarrow s + c$
4: **end for**
5: créer (m, s)

Le problème de cette implémentation est le très grand nombre de paires clé-valeur crées par map. Une implémentation alternative de map est la suivante, comptant "manuellement" l'occurence de chaque mot $m \in \mathcal{M}$ dans chaque fichier $F \in \mathcal{F}$:

Algorithm 6 Fonction map exécutée sur la paire $(@, F)$

for $m \in \mathcal{M}$ **do**
    $c[m] \leftarrow 0$
end for
for $m \in F$ **do**
    $c[m] \leftarrow c[m] + 1$
end for
for $m \in \mathcal{M}$ **do**
    if $c[m] \neq 0$ **then**
        créer $(m, c[m])$
    end if
end for

A la suite de quoi, on peut utiliser la même fonction reduce que celle décrite dans l'algorithme 5 ci-dessus.

Exemple 3 : statistique. L'objectif est de calculer le revenu moyen par personne dans chaque ville française en 2012. En donnée, on a deux fichiers F_1 et F_2, tous deux indexés par les numéros de sécurité sociale. $F_1[N]$ stocke des informations personnelles sur la personne de numéro de sécurité sociale N :

$$F_1[N] = (\text{nom}, \text{prénoms}, \text{ville de résidence})$$

Le fichier F_2 stocke les salaires en euros par années :

$$F_2[N] = ((2011, 39.230), (2012, 41.380), (2013, 42.020))$$

Ces fichiers sont stockés sur le système sous la forme de paires $(N, (F_1, F_2))$ where $F_i = F_i[N]$, $i = 1, 2$.

Algorithm 7 Fonction map exécutée sur la paire $(N, (F_1, F_2))$

extraire le ville v de résidence de F_1
extraire le salaire 2012 s de F_2
créer (v, s)

Algorithm 8 Fonction reduce exécutée sur la paire $(v, \{s_1, s_2, \ldots, s_k\})$

créer $(v, \frac{1}{k}\sum_{i=1}^{k} s_i)$

Exemple 4 : calcul matriciel. L'objectif est de calculer le produit $C = AB$ de deux matrices $N \times N$, A et B, en utilisant `map-reduce`, pour $N \gg n$. (Rappelons que $C_{i,j} = \sum_{k=1}^{N} A_{i,k}B_{k,j}$). Chaque matrice X est initialement stockée sous forme de paires clé-valeur ("X", $(i, j, X_{i,j})$) qui sont distribuées sur les machines d'un système distribué (où "X" denote le caractère X). Il est possible d'effectuer le produit de deux matrices en une itération de MapReduce, comme suit :

Algorithm 9 Fonction map exécutée sur la paire $(\text{“}X\text{”}, (i, j, x))$

1: **if** X=A **then**
2: créer $((i, k), (\text{“}A\text{”}, j, x)$, pour $k = 1, \ldots, N$
3: **else**
4: créer $((k, j), (\text{“}B\text{”}, i, x)$, pour $k = 1, \ldots, N$
5: **end if**

Cette fonction map crée N copies de chaque élément de A, et N copies de chaque élément de B, de façon à ce que, après groupage, la clé (i, j) regroupe la ligne i de A avec la colonne j de B. Précisément, la clé (i, j) regroupe les éléments de l'ensemble

$$E_{i,j} = \{(\text{“}A\text{”}, k, A_{i,k}\}), k = 1, \ldots, N\} \cup \{(\text{“}B\text{”}, k, B_{k,i}\}), k = 1, \ldots, N\}$$

Le produit s'effectue alors par la fonction reduce ci-après :

Algorithm 10 Fonction reduce exécuté sur la paire $((i, j), E_{i,j})$

1: créer $((i, j), \sum_{k=1}^{N} A_{i,k}B_{k,j})$

Le défaut de cette solution est que la fonction reduce doit gérer des paires clé-valeur, chacune contenant $2N$ éléments matriciels. Pour N grand, il peut donc être possible que $E_{i,j}$ ne puisse pas résider en mémoire interne, mais nécessite un stockage sur un disque externe. Or, extraire les éléments $A_{i,k}$ et $B_{k,j}$ de $E_{i,j}$ nécessite au préalable de trier l'ensemble $E_{i,j}$. Ce tri peut s'avérer extrêmement coûteux si $E_{i,j}$ ne réside pas en mémoire interne. Cette solution nécessite donc que chaque processeur effectue $\Omega(N^2/n)$ tris coûteux. Il est possible de contourner partiellement ce problème en effectuant deux itérations de MapReduce. La première itération procède comme suit :

Algorithm 11 Fonction map exécutée sur la paire $(\text{“}X\text{”}, (i, j, x))$

1: **if** X=A **then**
2: créer $(j, (\text{“}A\text{”}, i, x)$
3: **else**
4: créer $(i, (\text{“}B\text{”}, j, x)$
5: **end if**

Cette fonction appliquée sur les deux matrices ne génère que N^2 clés au total. Après groupage, les entrées de la fonction reduce sont donc de la forme (k, F_k) avec

$$F_k = \{(\text{“}A\text{”}, i, a_i), i = 1, \ldots, N\} \cup \{(\text{“}B\text{”}, j, b_j), j = 1, \ldots, N\}$$

où $a_i = A_{i,k}$ et $b_j = B_{k,j}$.

Algorithm 12 Fonction reduce exécutée sur la paire (k, F_k)

1: créer toutes les N^2 paires $((i, j), a_i b_j)$, $i, j = 1, \ldots, N$

On retrouve le même problème que celui évoqué pour la première solution, à savoir le fait de devoir trier l'ensemble F_k pour faire correspondre les éléments $A_{i,k}$ et $B_{k,j}$. Néanmoins il n'y a que N ensembles F_k au lieu de N^2 ensembles $E_{i,j}$, ce qui représente un gain considérable, en diminuant drastiquement le nombre de tris coûteux à effectuer.

Une seconde opération de MapReduce permet de créer les paires clé-valeur $((i,j), C_{i,j})$ à partir des paires clé-valeurs obtenues en sortie de la fonction reduce ci-dessus. La fonction map est l'identité :

Algorithm 13 Fonction map exécutée sur la paire $((i,j), v)$

1: créer $((i,j), v)$

La fonction reduce effectue alors la somme sur k des produits $A_{i,k}B_{k,j}$:

Algorithm 14 Fonction reduce exécuté sur la paire $((i,j), \{v_1, \ldots, v_N\})$

1: créer $((i,j), \sum_{k=1}^{N} v_k)$

Remarquons qu'effectuer la somme $\sum_{k=1}^{N} v_k$ ne nécessite aucune opération de tri des valeurs $v_k = A_{i,k}B_{k,j}$.

Références

[1] Hagit Attiya and Jennifer Welch. Distributed Computing : Fundamentals, Simulations, and Advanced Topics. John Wiley and Sons, Inc. 2006.

[2] Frank Thomson Leighton. Introduction to Parallel Algorithms and Architectures : Arrays, Trees, Hypercubes. Morgan Kaufmann, 1991.

[3] Nancy Lynch. Distributed Algorithms. Morgan Kaufmann, 1996.

[4] David Peleg. Distributed Computing : A Locality-Sensitive Approach. SIAM Monographs on Discrete Maths and Applications, 2000.