P-RAM – 2

1 Simulation d'un nouveau type de PRAM

Une PRAM CRTW (comme Concurrent Read Tolerant Write) est une PRAM avec un modèle légèrement différent du modèle CRCW :
- plusieurs lectures simultanées sont autorisées,
- si plusieurs processeurs veulent écrire une même valeur à une même adresse, cette écriture est autorisée
- si plusieurs processeurs veulent écrire des valeurs différentes à une même adresse, le conflit n'est pas résolu et aucune écriture n'a lieu.

▷ **Question 1** *Montrez qu'une exécution sur une PRAM CRTW à n processeurs qui s'effectue en temps t peut être simulée par une PRAM CRCW (en mode arbitraire) en temps $O(t)$.*

2 Compression de tableaux

Soit A un tableau de n entiers et B un tableau de n booléens. On veut calculer la compression du tableau A par le tableau B, c'est-à-dire un tableau C tel que

$$C[i] = \begin{cases} A[j] & \text{avec } j \text{ est le } i^{\text{ème}} \text{ bit non nul de } B \text{ s'il existe} \\ 0 & \text{sinon} \end{cases}$$

Par exemple :

$A=$	[	1	2	3	4	5	6	7	8	9	10]
$B=$	[	1	0	0	1	0	0	0	1	0	0]
$C=$	[	1	4	8	0	0	0	0	0	0	0]

▷ **Question 2** *Proposez un algorithme qui calcule une telle compression sur une PRAM EREW à n processeurs.*

3 Additionneur PRAM

On veut calculer la somme de deux nombres a et b de n bits, que l'on note $a = a_{n-1}a_{n-2}\dots a_1a_0$ et $b = b_{n-1}b_{n-2}\dots b_1b_0$. Pour ceci, on commence par calculer une fonction de propagation de retenue pour chaque bit :

$$\text{prop}_i = \begin{cases} s & \text{si } a_i = b_i = 0 & \text{signifie que la retenue est stoppée} \\ p & \text{si } (a_i = 1 \text{ et } b_i = 0) \text{ ou } (a_i = 0 \text{ et } b_i = 1) & \text{signifie que la retenue est propagée} \\ g & \text{si } a_i = b_i = 1 & \text{signifie qu'une retenue est générée} \end{cases}$$

On a par exemple :

$a=$	0	1	1	0	0	1	1	
$b=$	1	1	0	1	0	1	0	
prop=	p	g	p	p	s	g	p	s (retenue d'entrée)

Plus précisément, si on considère l'addition des bits a_i et b_i, la fonction de propagation prop_i signifie :
- si $\text{prop}_i = s$, la retenue venant des bits $i-1$ est arrêté, la retenue des bits i sera toujours 0,
- si $\text{prop}_i = g$, une retenue est générée, la retenue des bits i sera 1 quel que soit celle des bits $i-1$,
- si $\text{prop}_i = p$, la retenue des bits $i-1$ est transmise sans modification.

On peut aggréger l'action des fonctions de retenue en les composant, et par exemple, considérer l'effet de la retenue des bits $i-1$ sur celles des bits $i+1$: de la même façon, la retenue des bits $i-1$ peut être propagée (p), arrêtée (s), ou générée (g). Pour ceci on compose l'action des fonctions de propagation de retenue à l'aide d'un nouvel opérateur, noté $\otimes$. Formellement, $\text{prop}_i \otimes \text{prop}_{i-1}$ est la fonction de propagation de retenue correspondant à l'action de la retenue des bits (a_{i-1}, b_{i-1}) et (a_i, b_i) sur l'addition des bits (a_{i+1}, b_{i+1}).

▷ **Question 3** *Donnez la table de* $\otimes$.

▷ **Question 4** *Montrez que* $\otimes$ *est associatif.*

▷ **Question 5** *Proposez un algorithme qui calcule efficacement l'addition de deux nombres à* n *bits sur une PRAM EREW à* n *processeurs. Donnez sa complexité.*

4 Composantes connexes sur une P-RAM

On souhaite concevoir un algorithme CREW qui permette de calculer les composantes connexes d'un graphe $G = (V, E)$ dont les sommets sont numérotés de 1 à n. Plus précisément, on cherche un algorithme qui renvoie un tableau C de taille n tel que $C(i) = C(j) = k$ si et seulement si i et j sont dans la même composante connexe et k est le plus petit indice des sommets de cette composante.

Définition 1. À toute étape de l'algorithme, on appellera pseudo-sommet étiqueté par i l'ensemble de sommets $j, k, l, \cdots \in V$ tels que $C(j) = C(k) = C(l) = \cdots = i$. On assimilera le pseudo-sommet i étiqueté par i au sommet étiqueté par i.

Un des invariants de l'algorithme est que le plus petit indice des sommets constituant un pseudo-sommet étiqueté par i est i et que les sommets appartenant à un pseudo-sommet sont dans la même composante connexe. Cette assertion est donc vraie si on initialise C par : pour tout $i \in V = [\![1, n]\!]$: $C(i) = i$. Ceci signifie que chaque processeur se considère au départ comme sommet de référence de sa composante connexe. L'objectif de l'algorithme est de modifier ce point de vue égocentrique.

Définition 2. Une arborescence k-cyclique ($k \geqslant 0$) est un graphe orienté faiblement connexe (c'est-à-dire tel que le graphe non orienté sous-jacent est connexe) tel que :
- tout sommet a un degré sortant égal à 1 et
- il existe exactement un circuit de longueur $k + 1$.

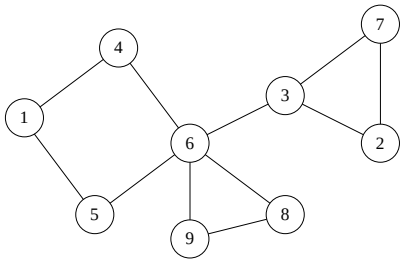
On appelle étoile une arborescence 0-cyclique dans laquelle toutes les arêtes sont incidentes à la racine et l'indice de la racine est le plus petit indice dans l'étoile.

L'invariant précédent est donc que le graphe orienté $(V, \{(i, C(i)) \mid i \in V\})$ est constitué d'étoiles. On peut donc identifier pseudo-sommets et étoiles, le centre de l'étoile étant l'indice du pseudo-sommet. Le calcul des composantes connexes s'effectue en enchaînant plusieurs fois de suite les deux fonctions suivantes :

```
GATHER()
  Pour tout i ∈ S en parallèle
      T(i) ← min{C(j) | {i,j} ∈ E, C(j) ≠ C(i)}
1:    {si l'ensemble est vide, on associe C(i)}
  Pour tout i ∈ S en parallèle
    T(i) ← min{T(j) | C(j) = i, T(j) ≠ i}
2:  {si l'ensemble est vide, on associe C(i)}

JUMP()
  Pour tout i ∈ S en parallèle
B(i) ← T(i)
  Pour j = 1 to log n
  Pour tout i ∈ S en parallèle
T(i) ← T(T(i))
  Pour tout i ∈ S en parallèle
C(i) ← min{B(T(i)), T(i)}
```

▷ **Question 6** *On considère le graphe suivant.*



Appliquer la fonction Gather *sur ce graphe, puis la fonction* Jump*, puis la fonction* Gather*, et ainsi de suite. Il sera instructif d'observer l'effet des opérations sur les graphes orientés* $(V, \{(i, T(i)) \mid i \in V\})$ *et* $(V, \{(i, C(i)) \mid i \in V\})$.

▷ **Question 7** *Montrer qu'après l'application de la fonction* Gather*, les composantes connexes contenant plusieurs pseudo-sommets induisent des arborescences* 1*-cycliques dans le graphe orienté* $(V, \{(i, T(i)) \mid i \in V\})$. *On notera également que le plus petit pseudo-sommet d'une arborescence* 1*-cyclique appartient au cycle.*

▷ **Question 8** *Montrer que la fonction* Jump *transforme une arborescence* 1*-cyclique en étoile (ou pseudo-sommet).*

▷ **Question 9** *Montrer qu'après* $\lceil \log n \rceil$ *enchaînements des fonctions* Gather *et* Jump*, les composantes connexes du graphe sont représentées par les pseudo-sommets induits par* C.

▷ **Question 10** *Quelle est la complexité de l'algorithme ? Combien de processeurs sont utilisés ?*

5 Réponses aux exercices

▷ Question 1, page 1

Chaque écriture de type $mem[addr] \leftarrow value$ est remplacée par le code suivant sur le processeur i :

1. $save[addr] \leftarrow mem[addr]$ (tous les processeurs écrivent la même valeur, écritures consistentes)
2. $mem[addr] \leftarrow value$ (écritures concurrentes, une d'entre elles réussit)
3. si $mem[addr] \neq value$, alors $mem[addr] \leftarrow save[addr]$ (écritures consistentes)

▷ Question 2, page 1

1. Saut de pointeur classique sur B pour calculer les sommes partielles dans $B' = [1, 1, 1, 2, 2, 2, 2, 3, 3, 3]$.
2. Pour tout i en parallèle, si $B[i] = 1$ alors $C[B'[i]] \leftarrow A[i]$

▷ Question 3, page 1

$\text{prop}_{i-1}\text{prop}_i$	s	p	g
s	s	s	g
p	s	p	g
g	s	g	g

▷ Question 4, page 2

On peut par exemple remarquer que $s \otimes x = s$ et $g \otimes x = g$ pour tout x et $p \otimes x = x$. Ainsi, si $a = s$ ou $a = g$, $a \otimes (b \otimes c) = a$ et $(a \otimes b) \otimes c = a \otimes c = a$. Et si $a = p$, $a \otimes (b \otimes c) = b \otimes c = (a \otimes b) \otimes c$.

▷ Question 5, page 2

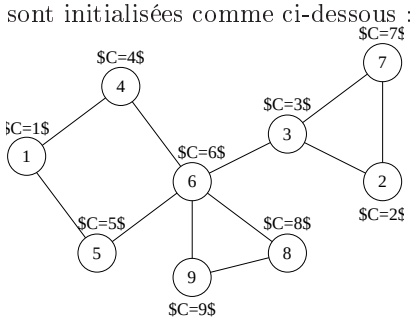
On utilise l'algo suivant :

1. Pour tout i en parallèle, calcul de prop_i.
2. Saut de pointeur avec l'opération associative $\otimes$.
3. Avec prop_i (= retenue propagée jusqu'à i), a_i et b_i, on calcule $c_i = \text{prop}_i XOR a_i XOR b_i$.

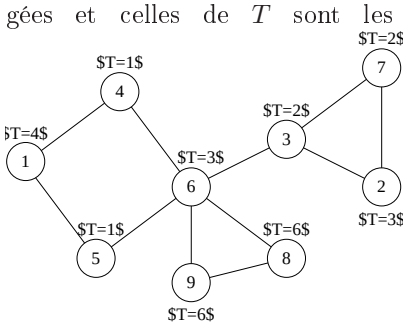
complexité : $O(\log n)$.

▷ Question 6, page 3

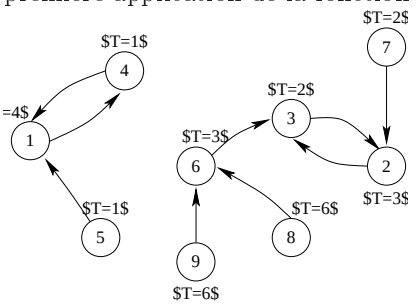
Avant le début de l'algorithme, les valeurs de C sont initialisées comme ci-dessous :



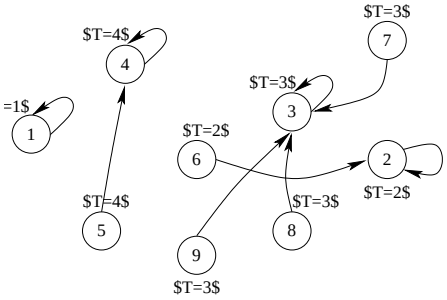
Après l'application de la fonction GATHER, les valeurs de C sont inchangées et celles de T sont les suivantes :



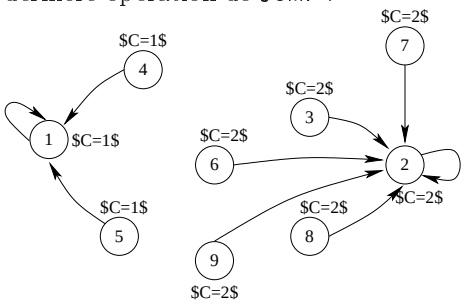
Graphe orienté $(V, \{(i, T(i)) \mid i \in V\})$ après la première application de la fonction GATHER :



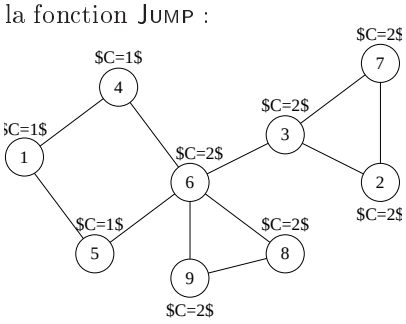
Après le saut de pointeur de la fonction JUMP :



Le graphe $(V, \{(i, C(i)) \mid i \in V\})$ après de la dernière opération de JUMP :

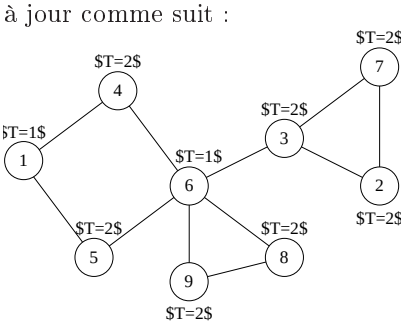


Il ne reste plus que deux pseudo-sommets. On se retrouve dans la situation suivante à la fin de la fonction JUMP :

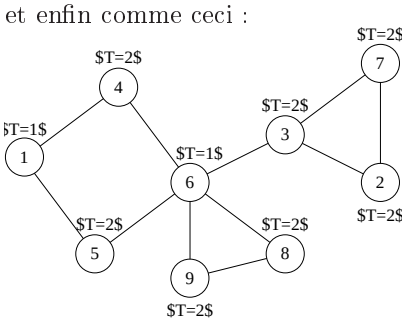


(fin de la première itération GATHER + JUMP)

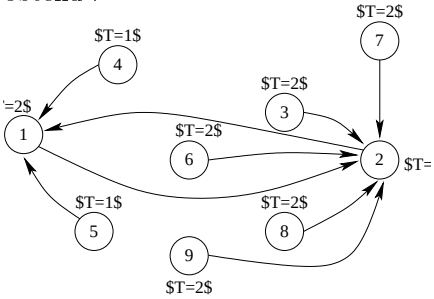
Après la première étape de GATHER, T est mis à jour comme suit :



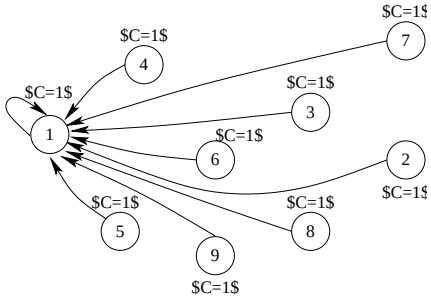
et enfin comme ceci :



Voici le graphe orienté $(V, \{(i, T(i)) \mid i \in V\})$ obtenu :



Les pseudo-sommets 1 et 2 sont donc fusionnés à la fin de l'appel à JUMP :



On a bien calculé les composantes connexes du graphe.

▷ Question 7, page 3

Tout d'abord, il est clair que lorsqu'une composante connexe ne contient qu'un seul pseudo-sommet, l'étoile correspondante est transférée dans T sans modification.

Si une composante connexe contient plusieurs pseudo-sommets, par contre, T décrira un ensemble d'arborescences 1-cycliques contenues dans cette composante. En effet, tout pseudo-sommet de cette composante contient au moins un sommet adjacent à un sommet d'un autre pseudo-sommet. GATHER fait pointer le représentant de chaque pseudo-sommet – *via* T – vers le représentant d'un autre pseudo-sommet, tout en laissant pointer les autres sommets vers leur pseudo-sommet initial. En clair, si deux groupes se touchent, leurs représentants se retrouvent liés dans le graphe orienté induit par T et les autres sommets continuent de pointer vers leur représentant respectif. Chaque composante de ce graphe orienté doit contenir au moins une boucle car le degré sortant de chaque sommet vaut 1. Il y a au plus une boucle car sinon il y aurait un pseudo-sommet avec deux valeurs pour T. Enfin la boucle en question ne peut être que de longueur 2, sinon (si elle était de longueur 1) i et $T(i)$ seraient identiques ou (si elle était de longueur supérieure à 2) il existerait un sommet i sur la boucle tel que $T(i)$ n'est pas le plus petit indice des pseudo-sommets adjacents au pseudo-sommet i.

▷ Question 8, page 3

Cette étape fusionne tous les sommets d'une même arborescence 1-cyclique en une étoile indexée par le sommet de plus petit indice en utilisant la technique de saut de pointeur. En effet, étant donné la configuration d'une étoile, à l'issue des sauts de pointeur, chaque sommet a pour valeur de T l'une des anciennes valeurs d'un des sommets de la boucle. La dernière étape permet donc d'assigner à tous les sommets la plus petite valeur des sommets de l'arborescence à laquelle ils appartiennent.

▷ Question 9, page 3

Il suffit de montrer que le nombre de pseudo-sommets diminue au moins de moitié à chaque étape. Concentrons-nous sur les représentants des pseudo-sommets et intéressons-nous au graphe induit par T sur ces sommets. Dans un tel graphe, deux pseudo-sommets i et j sont connectés si, et seulement si, il existe deux sommets k et l connectés dans le graphe initial et tels que $C(k) = i$ et $C(l) = j$. Dans l'exemple précédent, cela revient à avoir un graphe composé des pseudo-sommets 1 et 2 reliés par une arête après la première application de GATHER. La fonction JUMP fusionne tous les pseudo-sommets ainsi reliés en un seul pseudo-sommet. Ainsi le nombre de pseudo-sommets dans une même composante connexe diminue au moins de moitié à chaque étape et $\lceil \log n \rceil$ enchaînements de GATHER et JUMP suffisent à calculer les composantes connexes.

▷ Question 10, page 3

Premièrement, on peut remarquer que la boucle séquentielle de la fonction JUMP implique que le temps de calcul total est au moins $O(\log^2 n)$, et ce quel que soit le nombre de processeurs. On va d'abord montrer que ce temps peut être atteint avec $O(n^2)$ processeurs.

On peut déjà remarquer qu'avec autant de processeurs, la première et la dernière boucle de JUMP prennent un temps $O(1)$ et le saut de pointeur un temps $O(\log n)$. En fait, $O(n)$ processeurs suffisent pour arriver à un tel temps de calcul de la fonction JUMP. Si on veut arriver à un temps total de l'ordre de $O(\log^2 n)$, on va donc devoir montrer que la fonction GATHER peut s'exécuter en temps $O(\log n)$.

Le calcul du maximum de n valeurs peut se faire en temps $O(1)$ avec n^2 processeurs. Cependant, ce sont les maxima de plusieurs ensembles que l'on veut calculer et il faut raffiner donc l'approche

La première boucle se fait clairement en temps $O(1)$ avec $O(n^2)$ processeurs (en réalité avec $O(|E|)$ processeurs) sur une CREW. Les deux boucles suivantes se font en temps $O(\log n)$ avec $O(n^2)$ processeurs en divisant pour régner.

Le calcul des composantes connexes s'effectue donc bien en temps $O(\log^2 |V|)$ avec $O(|V| + |E|)$ processeurs. Cependant, la fonction JUMP gaspille des ressources (le «diviser pour régner» est trop gourmand en ressources). Les calculs de minima peuvent également être optimisés en terme de ressources. Le théorème de Brent nous permet donc de diminuer le nombre de processeurs à $O\left(\frac{n^2}{\log n}\right)$ (en réalité à $O\left(\frac{|E|}{\log |V|} + |V|\right)$ processeurs) sans modifier le temps de calcul.

Pour tout $i \in S$ **en parallèle**
$T(i) \leftarrow \min\{C(j) \mid \{i,j\} \in E, C(j) \neq C(i)\}$
1: {si l'ensemble est vide, on associe $C(i)$}
2:La boucle précédente peut être transformée en le code suivant
3:**Pour tout** $i,j \in S$ **en parallèle Si** $\{i,j\} \in E$ **And** $C(i) \neq C(j)$
$Temp(i,j) \leftarrow C(j)$ **Sinon**
$Temp(i,j) \leftarrow \infty$ **Pour tout** $i \in S$ **en parallèle**
$Temp(i,1) \leftarrow \min\{Temp(i,j) \mid j \in S\}$ **Pour tout** $i \in S$ **en parallèle Si** $Temp(i,1) = \infty$
$T(i) \leftarrow C(i)$ **Sinon**
$T(i) \leftarrow Temp(i,1)$

Algorithm 1: Précision sur la mise en œuvre des calculs de maxima.