Machines P-RAM

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

1 Séparation des modèles

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

1.1 Séparation CREW/CRCW : Calcul du maximum d'un tableau

▷ **Question 1.** Donner un algorithme PRAM CRCW permettant de calculer avec $O(n^2)$ processeurs en temps constant le maximum d'un tableau A composé de n éléments.

Réponse. Il suffit d'utiliser un processeur pour chaque comparaison et de leur faire écrire le résultat au même endroit, ce qui nous conduit donc a l'algorithme 1.

```
CALCUL_MAXIMUM(A, n)
1: Pour tout i ∈ [[1, n]] en parallèle
2:     m[i] ← TRUE
3: Pour tout i, j ∈ [[1, n]]^2 en parallèle
4:     Si A[i] < A[j] Alors m[i] ← FALSE
5: Pour tout i ∈ [[1, n]] en parallèle
6:     Si m[i] = TRUE Alors max← m[i]
7: Renvoyer max
```

ALGO. 1: Algorithme CRCW pour calculer le maximum d'un tableau.

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

▷ **Question 2.** Sur un modèle CREW, quelle est la complexité qu'on peut espérer ?

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

Une P-RAM CRCW est donc strictement plus puissante qu'une P-RAM CREW. Nous venons de séparer les modèles CRCW et CREW, en exhibant un algorithme pour lequel un facteur $O(\log p)$ sépare les temps d'exécution avec p processeurs d'un même problème. Ce facteur $O(\log p)$ est maximal. On peut même montrer le théorème suivant :

Théorème 1. *Tout algorithme sur une machine P-RAM CRCW à p processeurs ne peut pas être plus de $O(\log p)$ fois plus rapide que le meilleur algorithme P-RAM CREW à p processeurs pour le même problème. On a le même résultat entre les machines CREW et EREW et les machines CRCW et EREW.*

1.2 Séparation EREW/CREW : Recherche des racines dans une forêt

Soit $\mathcal{F}$ une forêt d'arbres binaires. Chaque nœud i d'un arbre est associé à un processeur $P(i)$ et possède un pointeur vers son père $pere(i)$. On va chercher des algorithmes EREW et CREW pour que chaque nœud connaisse la racine de son arbre (notée $racine(i)$), et ainsi prouver l'intérêt des lectures concurrentes.

▷ **Question 3.** Donner un algorithme PRAM CREW pour que chaque nœud détermine $racine(i)$. Démontrer que l'algorithme propose n'utilise que des lectures concurrentes et déterminer sa complexité.

Réponse. L'algorithme naturel pour chercher les racines utilise la technique de saut de pointeur, un nœud et ses ascendants ayant la racine. Une transformation directe du code nous conduit donc a l'algorithme 2.

```
CALCUL_RACINE
 1: Pour tout i en parallèle
 2:    Si pere(i) = NIL Alors racine(i) = i
 3: Tant que il existe un nœud i tel que pere(i) ≠ NIL
 4:    Pour tout i en parallèle
 5:       Si pere(i) ≠ NIL Alors
 6:          Si pere(pere(i)) = NIL Alors racine(i) = racine(pere(i))
 7:          pere(i) = pere(pere(i)).
```

ALGO. 2: Algorithme CREW pour le calcul des racines.

Cet algorithme est bien de type CREW puisque les seules écritures effectuées par le processeur i concernent des données qui lui sont propres($racine(i)$ et $pere(i)$). Par contre, cet algorithme n'est pas EREW puisque plusieurs processeurs peuvent avoir le même père (surtout à la fin !) et donc peuvent accéder simultanément à la même donnée en lisant $pere(i)$ par exemple.
En utilisant l'analyse de complexité effectuée en cours pour le calcul de la distance à la fin de la liste s'applique et on peut ainsi vérifier que tout les nœuds des arbres de la forêt connaissent leur racine en temps $O(\log d)$ où d est la profondeur maximale des arbres. □

▷ **Question 4.** Sur un modèle EREW, quelle est la complexité qu'on peut espérer ?

Réponse. Plaçons nous dans le pire des cas pour le modèle EREW, c'est à dire le cas où la forêt n'est constituée que d'un seul arbre, et considérons le nombre de processeurs qui peuvent apprendre qui est la racine a chaque étape. Dans le modèle EREW, le nombre de processeurs ayant accès à l'information ne peut que doubler à chaque étape (1 processeur peut au plus lire une case mémoire où est détenue l'information). Dans le cas d'une forêt à un seul arbre, un processeur au plus connaît la racine au départ, et il faut donc $O(\log n)$ étapes pour propager l'information à tous les processeurs. □

On vérifie donc ainsi que le modèle CREW est strictement plus puissant que le modèle EREW. Nous allons maintenant montrer comment simuler une lecture simultanée en $O(\log n)$ sur une EREW. La simulation des écritures concurentes en $O(\log n)$ étapes peut se faire en utilisant le même type de technique.

▷ **Question 5.** Comment simuler une lecture concurrente en $O(\log n)$ sur une EREW.

Réponse. La simulation de la lecture concurrente utilise le fait qu'on sait trier n nombres en temps $O(\log n)$ sur une EREW. On peut utiliser le même type de technique pour les écritures concurentes.

- Dans une première étape chaque processeur écrit dans une case mémoire le couple $< x_i, i >$, où i est son numéro de processeur et x_i est l'adresse de la case mémoire à laquelle il souhaite accéder.
- Dans la deuxième étape, on effectue un tri des couples $< x_i, i >$ selon la première composante, ce qui «regroupe» les processeurs qui désirent accéder à la même donnée. On obtient ainsi une liste triée y dont les composantes sont des couples de la forme $y_i =< x_k, \sigma(i) >$ où σ est une permutation de $[1, n]$.
- Dans la troisième étape, le processeur P_i regarde les cases $< x_k, \sigma(i) >$ et $< x_{k'}, \sigma(i+1) >$. Si $x_k = x_{k'}$, c'est a dire si $P_{\sigma(i)}$ et $P_{\sigma(i+1)}$ cherchent à accéder à la même donnée, alors $suivant(\sigma(i)) = \sigma(i+1)$. Dans le cas contraire, on pose $suivant(\sigma(i)) = NIL$. Il est facile de vérifier que cette étape n'utilise que des lectures indépendantes (si $< x_k, \sigma(i) >$ et $< x_{k'}, \sigma(i+1) >$ sont lus en deux temps) et qu'on obtient ainsi l listes chaînées contenant chacune les processeurs qui désirent accéder à une même donnée.
  La dernière étape de l'algorithme est une simple adaptation de l'algorithme de saut de pointeur vu en

cours pour propager l'information à lire dans toute la liste et qui peut s'exprimer par l'algorithme 3, dont la preuve de la correction est immédiate.

```
SIMULE
1: Pour tout i en parallèle
2:    Si suivant(i) = NIL Alors val(i) = Lecture Sinon val(i) = NaN
3: Tant que il existe un nœud i tel que suivant(i) ≠ NIL
4:    Pour tout i en parallèle
5:       Si suivant(i) ≠ NIL Alors
6:          val(i) = val(suivant(i))
7:          suivant(i) = suivant(suivant(i))
```

ALGO. 3: Dernière étape de la simulation d'une lecture concurrente sur une EREW.

□

▷ **Question 6.** Donner un algorithme efficace CRCW pour calculer (en le moins de temps possible) le produit de deux matrices booléennes (dans ce cadre, l'addition est remplacée par le *ou* et la multiplication par le *et*).

Réponse. Ce n'est pas très compliqué. On va utiliser $O(n^3)$ processeurs et réaliser le produit de matrice en temps $O(1)$, ce qui conduit bien à un algorithme efficace optimal en temps. L'astuce est qu'il est facile de faire le *ou* de n nombres en temps $O(1)$ sur une CRCW. En effet, le résultat est 1 dès que l'une des opérandes est 1 et il suffit donc que chaque processeur écrive 1 dans une case prédéfinie s'il possède un 1 et ne fasse rien sinon. Formellement l'algorithme peut être décrit comme suit.
Le processeur $P_{i,j,k}$ lit les valeurs $a_{i,k}$ et $b_{k,j}$, effectue le *et* des deux valeurs et stocke le résultat dans $c_{i,j,k}$.
Le processeur $P_{i,j,k}$ écrit 1 dans la case $c_{i,j}$ si $c_{i,j,k} = 1$ et ne fait rien sinon. □

2 Théorème de Brent, travail et efficacité

▷ **Question 7.** Soit $\mathcal{A}$ un algorithme comportant un nombre total de m opérations et qui s'exécute en temps t sur une P-RAM (avec un nombre de processeurs indéterminé). Montrer que l'on peur simuler $\mathcal{A}$ en temps $O\left(\frac{m}{p} + t\right)$ sur une P-RAM de même type avec p processeurs (théorème de Brent).

Réponse. A l'étape i, $\mathcal{A}$ effectue $m(i)$ opérations, avec $\sum_{i=1}^{n} m(i) = m$. On simule l'étape i avec p processeurs en temps $\left\lceil \frac{m(i)}{p} \right\rceil \leqslant \frac{m(i)}{p} + 1$. On obtient le résultat en sommant sur les étapes. □

Le théorème de Brent permet de prédire les performances quand on réduit le nombre de processeurs.

▷ **Question 8.** Proposez un algorithme pour le calcul du maximum sur une P-RAM EREW et calculez sa complexité. Est-il possible d'utiliser moins de processeur tout en conservant la meme complexité ?

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

```
CALCUL_MAXIMUM(A, n)
1: Pour k = m − 1 à 0
2:    Pour tout j ∈ ⟦0, 2^k − 1⟧ en parallèle
3:       A[j] ← max(A[2j], A[2j + 1])
```

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

Donnons quelques définitions usuelles : soit P un problème de taille n à résoudre, et soit $T_{seq}(n)$ le temps du meilleur algorithme séquentiel (connu) pour résoudre P. Soit maintenant un algorithme parallèle P-RAM qui résout P en temps $T_{par}(p)$ avec p processeurs.

Définition 1. Le facteur d'accélération (*speed-up* en anglais) est défini comme

$$S_p = \frac{T_{seq}(n)}{T_{par}(p)}$$

et l'efficacité comme

$$e_p = \frac{T_{seq}(n)}{p.T_{par}(p)}.$$

Définition 2. Le travail (*work*) de l'algorithme est

$$W_p = p.T_{par}(p).$$

Intuitivement, le travail est une surface rectangulaire de taille $T_{par}(p)$, le temps d'exécution, multiplié par p, le nombre de processeurs, et est minimal si à chaque étape de calcul tous les processeurs sont utilisés à faire des choses utiles, i.e. des opérations présentes dans la version séquentielle.

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

▷ **Question 9.** Soit $\mathcal{A}$ un algorithme qui s'exécute en temps t sur une P-RAM avec p processeurs. Montrer que l'on peut simuler $\mathcal{A}$ sur une P-RAM de même type avec $p' \leqslant p$ processeurs, en temps $O\left(\frac{t.p}{p'}\right)$.

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

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