Ecole Polytechnique

Cours : "Parallélisme"
Travaux dirigés
E. Goubault, S. Putot

TD 2

20 janvier 2014

1 Tri pair-impair

On suppose que l'on dispose d'un réseau linéaire de p processeurs $P_1, \ldots, P_p$, c'est à dire que les processeurs sont organisés sur une ligne, et que chacun peut communiquer avec son voisin de gauche et son voisin de droite s'il en a un. On veut trier n entiers, avec p qui divise n. Chaque processeur a au départ $\frac{n}{p}$ entiers, et on veut trouver un algorithme parallèle qui permette de faire en sorte que la sous-suite sur P_i soit ordonnée et inférieure à la sous-suite ordonnée P_j, pour tout $i < j$. C'est à dire tout simplement que l'on veut trier en parallèle les données réparties sur les processeurs P_i.

En s'inspirant du réseau de tri pair-impair vu en cours, écrire un tel algorithme en JAVA. Pour ce faire, on utilisera la classe `Buffer` suivante, pour coder les echanges de données entre chaque processus, chacun implémentant un comparateur du réseau de tri.

```
class Buffer {
    int val;
    Buffer() { val = -1; }

    synchronized void write(int v) {
        while (val!=-1) {
            try { wait(); } catch (InterruptedException e) {};
        };
        val = v;
        notifyAll();
    }

    synchronized int read() {
        while (val==-1) {
            try { wait(); } catch (InterruptedException e) {};
        };
        int n = val;
        val = -1;
        notifyAll();
        return n;
    }
}
```

Corrigé

```
class Buffer {
    int val;
    Buffer() { val = -1; }

    synchronized void write(int v) {
        while (val!=-1) {
            try { wait(); } catch (InterruptedException e) {};
        };
        val = v;
        notifyAll();
    }

    synchronized int read() {
        while (val==-1) {
            try { wait(); } catch (InterruptedException e) {};
        };
        int n = val;
        val = -1;
        notifyAll();
        return n;
    }
}


class Comparateur extends Thread {

    int proc;  // numero de processeur
    int val;   // valeur courante
    int nMax;  // nb de valeurs a trier
    int[] tab; // pour stocker les resultats finaux
    Buffer lg,ld,eg,ed; // lecture-ecriture, droite-gauche

    Comparateur(int p,int v,int n,int[] tt,Buffer llg,Buffer eeg,Buffer lld,Buffer eed) {
        proc = p; val = v; nMax = n; tab = tt;
        lg = llg; ld = lld; eg = eeg; ed = eed;
    }

    public void run() {
        int dv,gv; // valeurs lues a gauche et a droite

        for (int etape=0;etape<nMax;etape++) {
            if ((etape%2)==0) {
                if ((proc%2)==0) {
                    if (ed!=null) ed.write(val);
                    if (ld!=null) dv = ld.read(); else dv = val;
                    if (dv<val) { val = dv; }
                } else {
                    if (eg!=null) eg.write(val);
                    if (lg!=null) gv = lg.read(); else gv = val;
                    if (gv>val) { val = gv; }
```

```
                }
            } else {
                if ((proc%2)==0) {
                    if (eg!=null) eg.write(val);
                    if (lg!=null) gv = lg.read(); else gv = val;
                    if (gv>val) { val = gv; }
                } else {
                    if (ed!=null) ed.write(val);
                    if (ld!=null) dv = ld.read(); else dv = val;
                    if (dv<val) { val = dv; }
                }
            }
        }
        tab[proc] = val;
    }
}


class Tri {

    static final int nMax = 10;
    static int[] tab = { 62,49,22,67,11,6,51,17,29,37 };
    static int[] tabTrie = new int[nMax];

    public static void main(String[] args) {
        Comparateur[] cmp = new Comparateur[nMax];
        Buffer lg,eg,ld,ed;
        lg = null; eg = null ;
        for (int i=0;i<nMax-1;i++) { // creation des threads
            ld = new Buffer();
            ed = new Buffer();
            cmp[i]= new Comparateur(i,tab[i],nMax,tabTrie,lg,eg,ld,ed);
            lg = ed; eg = ld;
        };
        ld = null; ed = null; // creation du dernier thread
        cmp[nMax-1] = new Comparateur(nMax-1,tab[nMax-1],nMax,tabTrie,lg,eg,ld,ed);

        for (int i=0;i<nMax;i++) { // execution des threads
            cmp[i].start();
        };
        for (int i=0;i<nMax;i++) { // attente des resultats
            try { cmp[i].join(); } catch (InterruptedException e) {} ;
        };
        for (int i=0;i<nMax;i++) { // affichage resultats
            System.out.println(tabTrie[i]);
        };
    }
}
```

2 Récurrences linéaires

Le problème qui nous préoccupe dans cette section est de calculer efficacement, en parallèle, des suites récurrentes linéaires d'ordre $m \geq 1$, de la forme (pour $i \geq 0$):

$$\begin{array}{lcl} y_0 & = & a_0^0 \\ \cdots & \cdots & \cdots \\ y_{m-1} & = & a_0^{m-1} \\ y_{m+i} & = & a_m^{m+i} y_{m+i-1} + \cdots + a_1^{m+i} y_i + a_0^{m+i} \end{array}$$

Comme applications visées, on a par exemple:

- l'évaluation d'un polynôme $p(x) = a_0 + a_1 x + \cdots + a_k x^k$ par la méthode de Horner: on fait $y_0 = a_k$, puis $y_{i+1} = xy_i + a_{k-i-1}$ (récurrence linéaire d'ordre 1); alors $p(x) = y_k$.
- la résolution de systèmes linéaires par bandes: soit par exemple

$$A = \begin{pmatrix} a_{1,1} & 0 & 0 & 0 & \cdots & \cdots & 0 \\ a_{2,1} & a_{2,2} & 0 & 0 & \cdots & \cdots & 0 \\ a_{3,1} & a_{3,2} & a_{3,3} & 0 & & & \vdots \\ 0 & a_{4,2} & a_{4,3} & a_{4,4} & & & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ & & & 0 & a_{n,n-2} & a_{n,n-1} & a_{n,n} \end{pmatrix}$$

La solution de l'équation $Ax = b$ peut être trouvée par la récurrence suivante: $x_1 = \frac{b_1}{a_{1,1}}$, $x_2 = \frac{1}{a_{2,2}}(b_2 - a_{2,1}x_1)$,

$$x_i = \left(\sum_{j=i-m}^{i-1} -\frac{a_{i,j}}{a_{i,i}} x \right) + \frac{b_i}{a_{i,i}}$$

pour $3 \leq i \leq n$.

2.1 Récurrences d'ordre 1

On se place dans le cas de récurrences linéaires d'ordre 1 ($m = 1$): c'est à dire,

$$\begin{array}{lcl} y_0 & = & a_0^0 \\ y_{i+1} & = & a_1^{i+1} y_i + a_0^{i+1} \end{array}$$

et on veut calculer y_i pour les indices $0 \leq i \leq n-1$ (pour un n donné, que l'on pourra supposer être une puissance de 2).

Question On suppose que a_1^i est toujours égal à un. Comment calculer efficacement la suite des y_i, $0 \leq i \leq n-1$ sur une machine PRAM EREW, CREW, CRCW? Quelle sera l'efficacité de l'algorithme?

Corrigé: Quand les a_1^i sont tous égaux à un, il s'agit de calculer toutes les sommes partielles des a_0^j, pour $j = 0, \ldots, i$, sur le processeur P_i. Il s'agit donc d'utiliser la technique de saut de pointeur: sur une PRAM EREW, CREW et CRCW on a un calcul en temps $O(logn)$ utilisant un total de $O(n)$ opérations sur n processeurs. L'efficacité est donc de l'ordre de $O(\frac{1}{logn})$. Par le théorème de simulation de Brent, on aurait pu simuler cet algorithme avec $O(\frac{n}{log(n)})$ processeurs, et ainsi atteindre une efficacité de 1.

Question On se place maintenant dans le cas d'une récurrence linéaire d'ordre 1 générale, c'est à dire que les a_1^i sont maintenant quelconques. Calculer y_i en fonction de y_{i-2}.

Corrigé:

$$y_i = a_1^i a_1^{i-1} y_{i-2} + a_1^i a_0^{i-1} + a_0^i$$

Question On suppose maintenant $n = 2^k$. A l'aide de la question précédente, comment modifier l'algorithme de la question 1 pour calculer efficacement la suite des y_i, $0 \leq i \leq n-1$ sur une machine PRAM CREW, en utilisant la technique de saut de pointeur? On donnera un pseudo-algorithme sous la même forme que les algorithmes donnés dans le polycopié (chapitre PRAM). Quelle est l'efficacité de cet algorithme? Justifier brièvement.

Corrigé: Moralement, on organise n processeurs P_i, $0 \leq i \leq n-1$ en liste chaînée, P_{i+1} pointant vers P_i. Au début, on associe la valeur $p[i]$ à chaque P_i, avec $p[0] = a_0^0$ et $p[i] = 0$ si $i > 0$, ainsi que les coefficients $a[i] = a_i^1$ si $i > 0$ (0 si $i = 0$) et $b[i] = a_i^0$ si $i \geq 0$. Ensuite on itère le pseudo-code suivant:

```
forall proc i s. t. next[i] != nil in parallel {
  p[i] =_i a[i]*p[next[i]] + b[i];
  a[i] =_i a[i]*a[next[i]];
  b[i] =_i a[i]*b[next[i]] + b[i];
  next[i] = next[next[i]];
}
```

Alors au bout de k itérations, $p[n-1]$ vaut y_{n-1}. On peut le prouver par récurrence, en montrant qu'à l'étape $1 \leq e \leq k$, les y_i, $0 \leq i < 2^e$ sont calculés. Par exemple pour $k = 2$ on a les étapes suivantes:

- Etape 1: $p[0] = a_0^0$, $p[1] = a_1^1 a_0^0 + a_0^1$, $p[2] = a_0^2$, $p[3] = a_0^3$ et $a[1] = 0$, $a[2] = a_1^2 a_1^1$, $a[3] = a_1^3 a_1^2$. Enfin, $b[1] = a_1^1 a_0^0 + a_0^1$, $b[2] = a_1^2 a_0^1 + a_0^2$, $b[3] = a_1^3 a_0^2 + a_0^3$ et $next[1] = nil$, $next[2] = 0$, $next[3] = 1$.
- Etape 2: $p[2] = a_1^2 a_1^1 a_0^0 + a_1^2 a_0^1 + a_0^2 = y_2$,

  $$\begin{array}{rcl} p[3] & = & a_1^3 a_1^2 (a_1^1 a_0^0 + a_0^1) + a_1^3 a_0^2 + a_0^3 \\ & = & a_1^3 a_1^2 a_1^1 a_0^0 + a_1^3 a_1^2 a_0^1 + a_1^3 a_0^2 + a_0^3 \\ & = & y_3 \end{array}$$

  On a aussi: $a[2] = 0$, $a[3] = 0$, $b[2] = a_1^2 a_1^1 a_0^0 + + a_1^2 a_0^1 + a_0^2$, $b[3] = a_1^3 a_1^2 (a_1^1 a_0^0 + a_0^1) + a_1^3 a_0^2 + a_0^3$, et tous les $next[i]$ sont à nil.

Cet algorithme prend un temps $O(log(n))$ et utilise $O(n)$ opérations arithmétiques, et n processeurs. L'efficacité est donc de l'ordre de $O(\frac{1}{log(n)})$. Par le théorème de simulation de Brent, on aurait pu transformer cet algorithme afin qu'il utilise seulement $O(\frac{n}{log(n)})$ processeurs, et on aurait pu ainsi atteindre une efficacité de 1.

2.2 Récurrence linéaire d'ordre supérieur

On se place maintenant dans le cas général $m \geq 1$, et on suppose disposer d'un algorithme PRAM permettant de faire la multiplication de deux matrices carrées $m \times m$ en temps $O(log(m))$, et utilisant $O(M(m))$ opérations (on pourra supposer ici $M(m) = O(m^{2.376})$) sur $O(m^3)$ processeurs.

Question: Comment améliorer l'algorithme de la question 3 afin de calculer y_i, $0 \leq i \leq n-1$, sur une PRAM, dans le cas d'une récurrence d'ordre m? Quelle est l'efficacité de l'algorithme sur une PRAM CREW? Justifier brièvement.

Correction:
Il suffit de poser:

- $b_0 = \begin{pmatrix} a_0^0 \\ a_0^1 \\ \dots \\ a_0^{m-1} \end{pmatrix}$, $b_i = \begin{pmatrix} 0 \\ 0 \\ \dots \\ a_0^{m+i-1} \end{pmatrix}$ (pour $i \geq 1$), $Y_i = \begin{pmatrix} y_i \\ y_{i+1} \\ \dots \\ y_{i+m-1} \end{pmatrix}$

- $A_i = \begin{pmatrix} 0 & 1 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \dots & & & \\ 0 & 0 & \dots & 1 \\ a_1^{m+i} & a_2^{m+i} & \dots & a_m^{m+i} \end{pmatrix}$

Ainsi, la récurrence d'ordre m revient à la récurrence linéaire d'ordre un, dans un espace m-dimensionnel:

$$\begin{aligned} Y_0 &= b_0 \\ Y_i &= A_i Y_{i-1} + b_i \quad 1 \leq i < n \end{aligned}$$

On utilise alors exactement le même algorithme qu'à la question précédente. Mais maintenant, les multiplications n'ont plus un coût unitaire mais en $O(log(m))$. Donc le temps d'exécution parallèle sera de $O(log(n)log(m))$. Le nombre d'opérations arithmétiques (donc la complexité séquentielle) est maintenant non plus de l'ordre de n, mais de nm. On a besoin maintenant de l'ordre de nm^3 processeurs. Ainsi, l'efficacité est de l'ordre de $\frac{nm}{log(n)log(m)nm^3} = \frac{1}{log(n)}\frac{1}{m^2 logm}$.