Le modèle PRAM (1)

Introduction

Le modèle RAM (Radomized Access Machine)

1. But. Définir un modèle, théorique, universel de calcul permettant d'évaluer et de comparer de manière fiable les algorithmes séquentiels.

2. Composants d'un calculateur RAM. Un mémoire accessible en temps constant ($\Theta(1)$), un processeur, composé de registres internes et d'une unité de calcul et une horloge.

3. Fonctionnement d'un calculateur RAM. Lors de l'exécution d'une *instruction*, le processeur effectue trois opérations successives, régées par l'horloge : lecture de données de la mémoire vers ses registres internes, calcul sur les données stockées dans ces registres, écriture de données présentes dans les registres vers la mémoire. L'ensemble de ces trois opérations se fait en temps $\Theta(1)$.

Le modèle PRAM (Parallel RAM)

1. But. Définir un modèle, théorique, universel de calcul permettant d'évaluer et de comparer de manière fiable les algorithmes parallèles.

2. Composants d'un calculateur PRAM. La seule différence entre un calculateur RAM et un calculateur PRAM réside dans le nombre de processeurs : un calculateur PRAM dispose de n processeurs, notés $P_1, \ldots, P_n$, similaires au processeur RAM. En particulier, ces processeurs ne peuvent pas communiquer directement entre eux.

3. Fonctionnement d'un calculateur PRAM. Lors de l'exécution d'une *instruction*, tous les processeurs effectuent en parallèle le même cycle d'opérations que dans le cas RAM. Le point important est le suivant : l'horloge dirige les n processeurs de sorte que, premièrement tous les processeurs transfèrent de la mémoire vers les registres, puis tous les processeurs débutent leurs calculs au même top d'horloge, puis tous les processeurs débutent leurs transferts vers la mémoire au même top d'horloge. L'ensemble de ces trois opérations se fait en temps $\Theta(1)$.

Conflit d'accès mémoire

1. Problème. Il est possible que plusieurs processeurs veuillent accéder à une même adresse mémoire soit tous en lecture, soit tous en écriture.

2. Modes d'accès mémoire. Pour l'accés mémoire, que ce soit en lecture, on définit deux modes principaux :
L'accès *exclusif* (EW pour Exclusive write, ER pour Exclusive read) autorise un seul processeur à demander l'accès à une adresse mémoire : c'est au programme de gérer cet aspect en interdisant les demandes d'accès simultanées.
L'accès *concurrent* (CW pour Concurrent write, CR pour Concurrent read) autorise plusieurs processeurs à demander l'accès à une même adresse mémoire. Si pour l'accès en lecture, cela ne pose aucun problème théorique*, pour l'accès en écriture, il faut préciser quelle sera la valeur écrite dans cette adresse.

(*) L'hypothèse de k processeurs lisant tous une même adresse mémoire en temps $\Theta(1)$ est discutable.

Écriture concurrente

1. Problème. Plusieurs processeurs ont le droit de demander l'accès à une même adresse mémoire pour y écrire une valeur. La question est alors de déterminer quelle valeur sera écrite à cette adresse.

2. Remarque. Il est important de noter que cela implique que certaines écritures demandées par des processeurs ne seront pas effectuées. Tout le monde avait le droit de demander l'écriture (CW) mais certains se sont vus ensuite refuser le droit d'effectuer cette écriture.

3. 4 stratégies.
a. Si chaque processeur dispose d'une priorité, on autorise uniquement le processeur de plus haute priorité à écrire sa valeur.
b. Une écriture n'est effectuée que si tous les processeurs demandent à écrire la même valeur.
c. Une valeur arbitraire est choisie.
d. Une combinaison de toutes les valeurs dont l'écriture est demandée est effectuée (AND, OR, addition, . . .).

Diffusion d'une donnée d'un processeur vers tous les autres processeurs

Le problème et une solution CR simple

1. Le problème. Le processeur P_1 contient une donnée d dans un registre $r_{1,k}$ et on veut que tous les processeurs contiennent cette donnée dans un de leurs registres.

2. C'est facile dans le modèle CR. Il suffit que P_1 écrive v à une adresse libre a, puis que les $p - 1$ autres processeurs aillent lire cette adresse. Tout cela se fait en temps $\Theta(1)$.

```
PROCEDURE diffuser( r1,k: Registre )
PRECONDITION
  Le processeur P1 possede la valeur d dans son registre r1,k
POSTCONDITION
  Tous les processeurs Pi ont la valeur d dans leur registre ri,k
DEBUT
  P1 : transfere le contenu de son registre r1,k dans l'adresse a
  EN PARALLELE pour chacun des processeurs Pj (j = 1, ..., n) FAIRE
    rj,k <- a
  FIN
FIN
```

Le modèle ER (2)

1. Le problème. Dans ce modèle, on ne peut avoir plusieurs processeurs accédant en même temps à une adresse mémoire.

2. Quelques remarques. Si on ne met d que dans une seule adresse mémoire, il faudra que les processeurs aillent la lire un par un ce qui n'utilise pas du tout les capacités de travail en parallèle du modèle PRAM. Il faudra donc écrire d en plusieurs adresses mémoires.

Mais si on commence par écrire d dans $n-1$ adresses, puis on lit ces $n-1$ adresses en parallèle, on n'a là encore pas tiré profit du parallélisme de manière efficace.

Il va donc falloir trouver un moyen d'entrelacer les écritures de d en différentes adresses mémoires et les lectures de ces adresses.

Le modèle ER (2) avec 2^m processeurs

Pour diminuer le temps nécessaire à la diffusion, on peut utiliser la technique du *doublage récursif*

1. à la fin de chaque étape (lecture/calcul/écriture), chacun des k processeurs connaissant d l'écrit dans une nouvelle adresse (ce qui implique que d était déjà écrite dans au moins k adresses : pour recpecter cette condition on ajoute une étape au départ au cours de laquelle P_1 écrit d en mémoire)

2. ainsi à l'étape suivante, au moins $2k$ processeurs peuvent lire d.

Ainsi, lors de chaque étape, on double au moins le nombre de processeurs connaissant d et on peut diffuser d en au plus $\log(n) = m$ étapes.

L'algorithme suivant présente cette technique.

```
PROCEDURE diffuser( r1,k: Registre )
PRECONDITION
  Le processeur P1 possede la valeur d dans son registre r1,k
  La valeur n = 2^m (2 puissance m), pour m > 0.
POSTCONDITION
  Tous les processeurs Pi ont la valeur d dans leur registre ri,k
DEBUT
  P1 : a1 <- r1,k
  POUR i <- 1 A lg n FAIRE
    EN PARALLELE pour les processeurs Pj (j = 1, ..., 2^{i-1}) FAIRE
      rj,k <- aj, aj+2i-1 <- rj,k
    FIN
  FIN
  EN PARALLELE pour les processeurs Pj (j = n/2 + 1, ..., n) FAIRE
    rj,k <- aj
  FIN
FIN
```

Le modèle ER (3) avec 2^m processeurs

On peut remarquer que l'algorithme précédent peut être amélioré dans le sens où le processeur P_1 par exemple, lit $\log(n)$ fois la valeur d alors qu'il la connaît déjà au départ.

On peut alors proposer le modèle suivant, plus sophistiqué :
1. Au début d'une étape on a n_1 processeurs connaissant d (et qui n'auront pas besoin de la relire) et n_2 adresses mémoires contenant d.
2. Lors de cette étape, on a donc n_2 processeurs qui ne connaissaient pas d qui lisent cette valeur, puis les $n_1 + n_2$ processeurs qui connaissent maintetant d qui écriven cette valeur, ce qui donne $n_1 + 2n_2$ adresses mémoire qui contiennent d.
3. Au départ, on a $n_1 = 1$ et $n_2 = 0$.

Cependant, le nombre d'étapes de cet algorithme est plus difficile à évaluer, mais reste de l'ordre de $\Theta(\log(n))$. Les performances restent inchangées : temps $\Theta(\log(n))$, coût $\Theta(n \log(n))$ et travail $\Theta(n)$.

Recherche de l'élément minimum dans un tableau

Le problème et le principe du dédoublement

On se donne un tableau T (de taille une puissance de 2, $n = 2^m$) et on veut connaître son élément minimum. On cherche à minimiser le *temps d'exécution*.

Deux observations importantes :
1. ce problème peut être résolu par un algorithme séquentiel efficace de type Diviser-pour-régner, qui induit donc un arbre des appels récursifs dans lequel le travail effectué en chaque nœud correspond à *comparer deux éléments du tableaux*, résultats des appels récursifs ;
2. les appels récursifs peuvent s'effectuer en parallèle car ils considèrent des sous-tableaux disjoints.

On peut donc résoudre ce problème en utilisant l'algorithme DPR classique dans lequel les deux appels récursifs sont lancés en parallèles : c'est une solution parallèle *récursive descendante* basée sur une simple modification d'un algorithme séquentiel.

```
procedure min2( int n1, int n2 ) returns int leMin
{ leMin = min( n1, n2 ); }
procedure trouverMin( int X[*], int i, int j, res int leMin) {
# PRECONDITION
#    j >= i & SOME( k >= 0 :: j-i+1 = 2**k )
# POSTCONDITION
#    leMin = MINIMUM( i <= l <= j :: X[i] )
  int m1, m2;
  if ( i == j-1 ) { leMin = min2(X[i],X[j]); }
  else {
    co
     trouverMin(X, i, i+((j-i+1)/2)-1, m1);
    //trouverMin(X, i+((j-i+1)/2), j, m2);
    oc
  }
  leMin = min2(m1, m2);
}
end
```

Complexité

1. Temps d'exécution. Le temps d'exécution est en $\Theta(\log(n))$ car :
1. comme les deux appels récursifs s'effectuent en parallèle, à tous moment on effectue en parallèle *tous* les appels récursifs d'un *même niveau.* étape 1 (1 processeur) : `trouverMin( X, 1, n)` ; étape 2 (2 processeurs) : `trouverMin( X, 1, n/2)` et `trouverMin( X, n/2+1, n)` ; étape 3 (4 processeurs) : `trouverMin( X, 1, n/4)`, `trouverMin( X, n/4+1, n/2)`, `trouverMin( X, n/2+1, 3n/4)` et `trouverMin( X, 3n/4+1, n)` ; ...
2. le travail effectué en chaque nœud est en $\Theta(1)$ (comparaison des résultats de deux appels récursifs).

2. Coût. Le nombre maximum de processeurs travaillant en même temps est donc le nombre de feuilles de l'arbres des appels récursifs, $n/2$ ici. On a donc un coût de $\Theta(n\log(n))$.

3. Travail. Chaque nœud de l'arbre des appels récursifs/parallèles effectue un travail en $\Theta(1)$ et on a $\Theta(n)$ nœuds, ce qui donne un travail en $\Theta(n)$.

Une version récursive ascendante (1)

On va maintenant décrire un autre algorithme pour ce problème, basé sur le même principe que l'algorithme décrit précédemment (suivre l'arbre des appels récursifs), mais non récursif.

Pour cela on va effectuer les mêmes calculs que l'algorithme précédent, non pas en descendant de la racine vers les feuilles de l'arbre, mais en remontant des feuilles vers la racine.

On va donc avoir une boucle principale dans laquelle chaque itération correspond au traitement parallèle d'un niveau de l'arbre des appels récursifs, en commençant par le dernier : cela implique donc que chaque processeur a pour travail de comparer deux éléments du tableau pour déterminer le plus petit des deux.

La structure de l'arbre permet d'affirmer qu'il faudra exactement $\log(n)$ itérations pour compléter la recherche.

Une version récursive ascendante (2)

La principale difficultés consiste donc à déterminer, pour chaque processeur, lors d'une itération de la boucle

1. où dans le tableau il doit aller chercher les deux éléments à comparer

2. et où il doit stocker le minimum calculé, étant entendu que cela doit être en mémoire car c'est la seule solution pour qu'un autre processeur puisse lire ce résultat.

Pour le point *1.*, à la première itération c'est facile : on divise le tableau en blocs de deux éléments et on assigne un bloc à chaque processeur (il en faut donc $n/2$).

Pour le point *2.* on peut décider que le processeur P_j place toujours le résultat en $X[j]$. Il faut noter ici que si on ne veut pas modifier le tableau X, il est préférable de travailler sur une copie de X, que l'on dénomme T.

À partir de ces deux choix, à la k^{eme} itération les éléments à considérer sont dans les 2^{m-k+1} premières cases de T et il suffit d'associer à P_j les cases $2j-1$ et $2j$.

```
procedure copier( int x ) returns int r { r = x; }
procedure min2( int n1, int n2 ) returns int leMin
{ leMin = min( n1, n2 ); }
procedure trouverMin( int X[*], int n ) returns int leMin {
# PRECONDITION
#    n >= 1 & SOME( k >= 0 :: n = 2**k )
# POSTCONDITION
#    leMin = MINIMUM( 1 <= i <= n :: X[i] )
  int T[n]; # Copie du tableau.
  co [i = 1 to n]
    T[i] = copier( X[i] );
  oc

  for [i = 1 to lg(n)] {
    co [j = 1 to n/(2**i)]
      T[j] = min2( T[2*j-1], T[2*j] );
    oc
  }
  leMin = T[1];
}
end
```

Une version récursive ascendante (3)

Les performances de cet algorithmes sont les mêmes que pour l'algorithme DPR parallèle : temps en $\Theta(\log(n))$, coût en $\Theta(n \log(n))$ et travail en $\Theta(n)$.

Ce point est important : ces deux algorithmes fonc exactement le même travail. Le second est simplement une version itérative ascendante.

Cette similarité est basée sur le fait bien connu que l'on peut transformer un algorithme récursif en algorithme itératif de même complexité [parfois au prix de l'introduction d'une structure de données auxiliaire permettant de conserver des résultats partiels, mais ça n'est pas le cas ici] : il suffit de remplacer les `co ... oc` par une boucle `for`.

La technique utilisée ici est celle du *dédoublement* : lors de chaque itération, la zone du tableau couverte par un processeur (celle dont il calcule le minimum) est deux fois plus grande que celle couverte à l'étape précédente.

Optimisation en coût (1)

1. Non optimalité en coût. L'algorithme précédent est optimal en travail mais non en coût. En effet, la complexité optimale séquentielle est en $\Theta(n)$, mais le coût est en $\Theta(n \log(n))$.

2. Les deux choix possibles. Pour le rendre optimal en coût, on peut soit réduire le temps d'exécution, soit réduire le nombre de processeurs [(rappel : le coût est donné par le nombre de processeur - $\Theta(n)$ ici - fois le temps d'exécution - $\Theta(\log(n))$ -)].

3. Réduire le temps d'exécution. Il semble difficile de réduire le temps d'exécution, notamment car c'est le paramètre que l'on a essayé d'optimiser en mettant au point l'algorithme, et que $\Theta(\log(n))$ est le meilleur résultat que l'on a pu obtenir. On va donc essayer de réduire le nombre de processeurs nécessaires.

Optimisation en coût (2)

1. Nombre de processeurs maximum. Si on se base sur un maintien du coût d'exécution à $\Theta(\log(n))$, il faut donc réduire le nombre de processeurs à $n/\log(n)$, pour obtenir un coût optimal en $\Theta(n)$.

2. Contrainte de l'arbre des appels récursifs/parrallèles. De par sa nature, l'arbre des appels nous impose de ne pas effectuer d'appels récursifs après le niveau faisant travailler en parallèle $n/log(n)$ processeurs, ce qui correspond aux sous-problèmes consistant à trouver le minimum de sous-tableaux de taille $\log(n)$.

3. Principe. On n'a donc pas le choix, une fois arrivé à un problème de taille $\log(n)$, il ne faut plus demander l'utilisation d'un nouveau processeur, et donc résoudre le problème en séquentiel, ce qui se fait en temps $\Theta(\log(n))$.

4. L'arbre ainsi obtenu comporte au plus $\log(n)$ niveaux, dans lesquels les nœuds internes effectuent un travail en $\Theta(1)$, et les $n/\log(n)$ feuilles un travail (en parallèle) en $\Theta(\log(n))$. Le temps d'exécution est donc de $\Theta(\log(n))$, ce qui donne un coût en $\Theta(n)$, qui est optimal.

```
procedure inf( int i, int taille ) returns int r
{ r = (i-1) * taille + 1; }

procedure sup( int i, int taille ) returns int r
{ r = i * taille; }

procedure min2( int n1, int n2 ) returns int leMin
{ leMin = min( n1, n2 ); }

procedure trouverMinSeq( int a[*], int i, int j ) returns int leMin {
# Recherche sequentielle du minimum de a[i:j].
# POSTCONDITION
#    leMin = MINIMUM( i <= k <= j :: a[k] )
  leMin = high(int);
  for [k = i to j] { leMin = min( leMin, a[k] ); }
}
```

```
procedure trouverMin( int X[*], int n ) returns int leMin {
  int taillePaquets = lg(n);            # Taille des paquets.
  int nbPaquets     = n/taillePaquets; # Nombre de paquets/processeurs.
  int T[nbPaquets];                     # Copie de travail.
  # On calcule le minimum des differents paquets de lg(n) elements.
  co [i = 1 to nbPaquets]
    T[i] = trouverMinSeq(X,inf(i, taillePaquets),sup(i, taillePaquets));
  oc
  # On applique la strategie du dedoublement.
  # sur les (nbPaquets) resultats obtenus.
  for [i = 1 to lg(nbPaquets)] {
    co [j = 1 to nbPaquets/(2**i)]
      T[j] = min2( T[2*j-1], T[2*j] );
    oc
  }
  leMin = T[1];
}
```

Conclusion

1. Le point important à relever dans l'exemple que nous venons de détailler réside dans le lien très fort entre la mise au point de l'algorithme parallèle, son analyse et l'arbre des appels récursifs de l'algorithme séquentiel, qui peut maintenant être vu comme un arbre des appels récursifs/parrallèles.

2. La mise au point de la structure de l'algorithme s'est basé sur deux points : une stratégie DPR et une transformation de l'algorithme récursif en algorithme itératif (deux grands classiques), puis l'observation que le traitement des problèmes d'un même niveau peut se dérouler en parallèle (indéoendance des problèmes).

3. L'analyse était calquée sur la structure de l'arbre comme pour un algo DPR séquentiel, mais les équations de récurrences du temps d'exécution prenaient en compte le parallélisme $[T(n) = T(n/2) + \Theta(1)]$.

4. L'optimisation en coût est basée sur la technique d'élaguage de sous-arbres déj'a vue pour améliorer l'efficacité des algorithmes DPR séquentiels.

Problème de synchronisation

Les lignes suivantes de l'algorithme précédent peuvent poser un problème :

```
co [j = 1 to nbPaquets/(2**i)] T[j] = min2( T[2*j-1], T[2*j] );  oc
```

En effet, il est possible qu'en parallèle on ait alors le processeur P_1 qui veuille effectuer, au cours d'une instruction,

`lire T[1], lire T[2] (*), calcul, écrire T[1]`

et le processeur P_2

`lire T[3], lire T[4], calcul, écrire T[2] (+)`

Si le modèle PRAM n'est pas complètement respecté (lectures, puis calcul, puis écritures), comme en MPD, il est possible que l'opération `(+)` soit effectuée avant `(*)`, ce qui va fausser le résultat.

Les deux algorithmes suivants présentent des solutions à ce problème : le premier recopie utilise un tableau auxiliaire `tmp`, le second s'arrange pour que toute écriture s'effectue dans une partie de T qui ne sera pas suscpetible d'être lue.

```
procedure trouverMin( int X[*], int n ) returns int leMin {
  int T[n];    # Copie de travail du tableau X.
  # On copie X dans T.
  co [i = 1 to n] T[i] = copier( X[i] ); oc
  # On effectue la recherche (ascendante) du minimum.
  for [i = 1 to lg(n)] {
    int tmp[n/(2**i)];    # Copie de travail temporaire (locale).
    # On calcule les divers minimums dans tmp.
    co [j = 1 to n/(2**i)] tmp[j] = min2( T[2*j-1], T[2*j] ); oc
  # On recopie tmp dans T.
  co [j = 1 to n/(2**i)] T[j] = copier( tmp[j] ); oc
  }
  leMin = T[1];
}
```

```
procedure trouverMin( int X[*], int n ) returns int leMin {
  int T[n]; # Copie du tableau.
  co [i = 1 to n]
    T[i] = copier( X[i] );
  oc

  for [i = 1 to lg(n)] {
    int dist = 2**(i-1);
    co [j = dist to n by 2*dist]
      T[j+dist] = min2( T[j], T[j+dist] );
    oc
  }
  leMin = T[n];
}
```