Le modèle PRAM

Note : Dans ce qui suit, pour simplifier la présentation des algorithmes et leur analyse, on va généralement supposer que n, la taille des données, est une puissance de 2 ($n = 2^k$, pour $k \geq 0$). Lorsque ce ne sera pas le cas, on le mentionnera explicitement.

1 Introduction : modèle RAM vs. PRAM

1.1 Le modèle RAM

Le modèle séquentiel classique de calcul est celui de la machine RAM — *Random Access Machine* —, parfois aussi appelé le modèle de *von Neumann*. Ce modèle architectural est une *abstraction* d'une architecture uni-processeur standard et suppose que la machine est formée de composantes ayant les caractéristiques suivantes :

- Une mémoire contenant M cases mémoire, où M est un nombre très grand mais fini. Chaque case possède une adresse unique et ne peut contenir qu'une unique valeur à la fois. Une caractéristique clé est qu'on suppose que le temps pour accéder au contenu d'une adresse A est constant (c'est-à-dire $\Theta(1)$) et ce peu importe la taille de la mémoire.

- Un processeur, qui opère sous le contrôle d'un algorithme séquentiel, donc qui exécute une instruction à la fois. Chaque instruction s'exécute de façon complète avant que la suivante ne débute son exécution.

  Plus précisément, on suppose que l'exécution d'*une* instruction est composée des trois phases suivantes (dans le style d'une machine RISC), phases réglées par la cadence d'une horloge :

  1. Lecture des données de la mémoire et transfert dans les registres du processeur ;
  2. Calcul sur les registres ;
  3. Écriture des résultats, c'est-à-dire transfert des registres vers la mémoire.

- Une unité d'accès mémoire, qui permet d'établir un chemin de communication direct entre le processeur et la mémoire.

La structure abstraite d'une machine RAM est donc telle que décrite à la figure 1.

L'analyse standard du temps d'exécution d'un algorithme pour une telle machine RAM suppose que l'exécution d'une instruction se fait en temps $\Theta(1)$. Formellement, cette approche d'analyse des algorithmes RAM est dite "*analyse uniforme*" et il est important de souligner qu'il s'agit, et ce de plusieurs façons, d'une abstraction :

- La plupart des machines modernes possèdent des pipelines d'exécution qui font qu'une instruction peut débuter son exécution *avant* que la précédente ne soit complétée.

- Le temps d'accès à une case mémoire varie grandement selon que l'adresse désirée est dans la cache (un cycle) ou non (jusqu'à plusieurs dizaines de cycles).[1]

[1] Par exemple, sur un Sun Enterprise 10000, le temps d'accès à la cache est d'environ 15 *ns* alors que le temps d'accès à la mémoire locale est de l'ordre de 600 *ns*. Soulignons aussi que sur une machine à mémoire distribuée, le temps d'accès à une adresse *non locale*, donc un accès par l'intermédiaire du réseau, peut être de l'ordre de plusieurs dizaines de μs, donc plusieurs *dizaines de milliers de ns*.

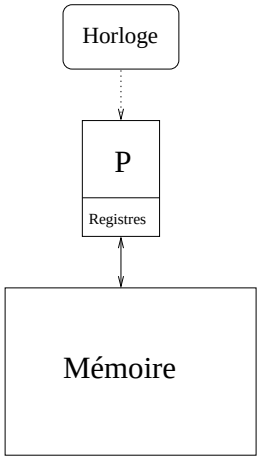


Figure 1: Structure abstraite d'une machine RAM

- Lorsqu'on regarde la mise en oeuvre effective d'une mémoire DRAM ou SRAM (au niveau des circuits), on constate que le temps d'accès effectif à une adresse dépend de la taille de la mémoire (par un facteur logarithmique). Il en est de même pour l'accès au banc de registres. Toutefois, ces facteurs n'influencent pas de façon significative le temps d'exécution d'un algorithme, c'est-à-dire qu'ils peuvent être ignorés et qu'on suppose donc que le temps d'accès est $\Theta(1)$.

Malgré l'abstraction qu'il introduit, le modèle RAM représente un modèle réaliste de calcul séquentiel, en ce sens que les temps d'exécution que l'on peut déduire d'un algorithme pour ce modèle peuvent permettre de prédire de façon relativement fiable le temps d'exécution sur de véritables machines. Plus précisément, le modèle peut nous permettre de *comparer* deux algorithmes différents : l'algorithme le plus lent pour le modèle RAM sera très certainement le plus lent sur une machine réelle arbitraire. Ceci est d'autant plus vrai qu'on s'intéresse particulièrement au comportement asymptotique des algorithmes : le fait qu'une instruction abstraite (par exemple, une affectation avec une expression complexe) nécessite plusieurs instructions de bas niveau *ne change par l'ordre de complexité* (pour toute constante c, on a $c \in \Theta(1)$).

1.2 Le modèle PRAM

> *En parallélisme, la recherche d'un modèle général se heurte à la variété des architectures à considérer. Comment définir le coût d'une communication? Finalement, il s'est avéré que le plus raisonnable était ... de l'ignorer. [LR03]*

Le modèle PRAM — *Parallel Random Access Machine* — est le modèle de calcul parallèle le plus couramment utilisé. Il a été développé avec comme objectif de définir un modèle de calcul qui jouerait, pour les architectures et algorithmes parallèles, le même rôle unificateur et, surtout, *simplificateur* que celui joué par le modèle RAM pour les algorithmes et machines séquentiels.

Plus précisément, l'objectif du modèle PRAM est de définir un modèle de calcul pour lequel on peut concevoir des algorithmes théoriques et abstraits, algorithmes dont les analyses asymptotiques permettent de "prédire" le comportement sur des machines parallèles, permettent de *classifier* les divers algorithmes parallèles et d'effectuer des analyses de complexité. Le modèle PRAM repose donc sur un ensemble de simplifications semblables à celles

du modèle RAM. Dans le cas du modèle PRAM, ces simplifications vont nous conduire à extraire, à déterminer le maximum de parallélisme possible contenu dans un algorithme.

Les composantes d'une machine PRAM sont donc les suivantes :

- Une collection de p processeurs $P_1, \ldots, P_p$, chaque processeur étant un processeur RAM standard.
- Une unité de mémoire globale, *partagée* entre les différents processeurs, qui permet à chaque processeur d'accéder à une case mémoire en temps $\Theta(1)$.

Il est important de souligner que les p processeurs ne sont pas reliés directement entre eux, mais communiquent strictement par l'intermédiaire de la mémoire.

L'exécution d'un algorithme procède alors, comme dans le modèle RAM, en trois phases, exécutées *conjointement* (c'est-à-dire de façon *synchrone*, sous le contrôle de l'horloge) par les divers processeurs :

1. Lecture d'une ou plusieurs cases mémoire et transfert dans les registres locaux au processeur.
2. Calcul sur les registres locaux, chacun des calculs se faisant en parallèle ... mais tous les processeurs exécutant la même instruction de façon synchrone.
3. Écriture de résultats (registres) dans la mémoire.

Le *synchronisme* dans l'exécution des instructions est une caractéristique importante et cruciale du modèle PRAM. Plus précisément, dans le modèle PRAM, les diverses phases d'un cycle d'exécution doivent être effectuées de façon synchrone par l'ensemble des processeurs. Ainsi, tous les accès en lecture sont tout d'abord faits de façon concurrente et synchrone. Ensuite, tous les processeurs exécutent de façon concurrente et synchrone les calculs sur les valeurs et registres locaux. Finalement, tous les processeurs écrivent leurs résultats en mémoire. Bien que pour certains algorithmes ce synchronisme ne soit pas tout à fait nécessaire, d'autres algorithmes ont absolument besoin de cette propriété d'exécution concurrente et synchrone pour assurer leur bon fonctionnement.

Une autre caractéristique du modèle strict est que tous les calculs se font strictement dans les registres locaux des processeurs. Les accès à la mémoire ne se font que pour charger des valeurs de la mémoire vers les registres, ou bien transférer le contenu de registres vers la mémoire. En d'autres mots, une machine PRAM est donc une machine de style *load–store architecture*, tout comme le sont les machines RISC modernes.

La structure abstraite d'une machine parallèle PRAM est donc telle que décrite à la figure 2.

Analyse uniforme et conflits d'accès

L'analyse d'un algorithme pour le modèle PRAM se fait donc, comme pour le modèle RAM, à l'aide d'une *analyse uniforme*, où l'on suppose que chaque cycle d'exécution (lecture, calcul, écriture) se fait en temps $\Theta(1)$. Ceci est possible dans la mesure où l'on suppose — propriété clé des machines PRAM — que tous les accès à la mémoire se font en temps $\Theta(1)$. Évidemment, la différence d'avec le modèle RAM, c'est que plusieurs opérations (p) peuvent toutefois s'effectuer en même temps (en parallèle). La propriété d'accès à la mémoire en temps constant peut donc être *irréaliste* (même en ignorant les contraintes liées à la mise en oeuvre physique d'une mémoire de grande taille) si on tient compte du fait que des *conflits* peuvent survenir si deux ou plusieurs processeurs tentent, à un même moment, de lire ou d'écrire *à une même adresse mémoire*. Différentes façons de permettre et traiter différents types de conflits sont présentées à la prochaine section.

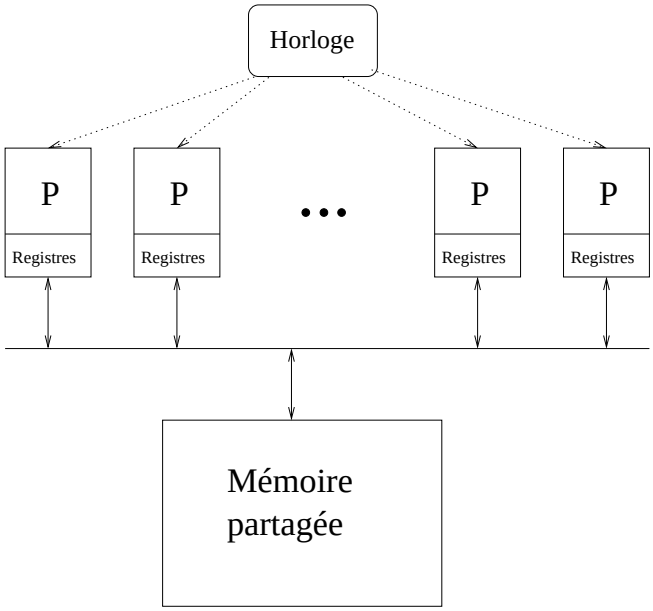


Figure 2: Structure abstraite d'une machine parallèle PRAM

2 Résolution des conflits d'accès et différents types de machines PRAM

2.1 Résolution des conflits d'accès

Deux types de conflits d'accès peuvent survenir :

1. Un *conflit de lecture* survient lorsque deux ou plusieurs processus tentent d'accéder, en lecture, à une même adresse. Deux façons de traiter de tels conflits sont généralement utilisées :
   (a) Approche *ER* (*Exclusive Read*) : un seul processeur à la fois peut accéder à une adresse donnée.
   (b) Approche *CR* (*Concurrent Read*) : plusieurs processeurs peuvent lire une même adresse.

   L'approche *CR* est la plus courante, car elle est généralement supportée de façon assez directe par la plupart des machines et unités de gestion de mémoire — à la limite, toutefois, il est certain qu'aucune machine réelle ne pourrait supporter un accès en temps $\Theta(1)$ à une même adresse par p processeurs en même temps si p était très grand.

2. Un *conflit d'écriture* survient lorsque, dans un cycle donné, deux ou plusieurs processeurs tentent d'écrire à une même adresse. Le traitement de tels conflits est évidemment plus complexe et peut se faire de différentes façons :
   (a) Approche *EW* (*Exclusive Write*) : un seul processeur à la fois peut effectuer une écriture à une adresse donnée.
   (b) Approche *CW* (*Concurrent Write*) : plusieurs processeur à la fois peuvent effectuer une écriture à une adresse donnée, mais l'effet (la sémantique) d'une telle écriture peut alors être défini de différentes façons :

- *CW* avec priorité : un niveau de priorité est associé à chacun des processeurs, et si plusieurs processeurs écrivent à une adresse dans un même cycle, la valeur résultante est celle produite par le processeur avec le plus haut niveau de priorité.
- *CW* égalitaire (appelé aussi *CW* commun) : tous les processeurs peuvent écrire en même temps, mais en autant qu'ils écrivent *la même valeur*.
- *CW* arbitraire : tous les processeurs peuvent écrire en même temps, et la valeur écrite sera choisie arbitrairement parmi celles ayant été écrites.
- *CW* par combinaison (*combining CW*) : lorsque plusieurs processeurs tentent d'écrire en même temps à une même adresse, on suppose qu'une opération de *combinaison* des valeurs est appliquée sur chacune de ces valeurs pour produire la valeur résultante. Les opérations de combinaison typiquement utilisées sont des opérations arithmétiques ou logiques, généralement associatives et commutatives, telles que "`+`", "`*`", "`ET`", "`OU`", "`MAX`", "`MIN`", etc.

2.2 Modèles PRAM les plus courants

Il existe différents modèles PRAM qui se distinguent par la façon dont les conflits d'accès à des adresses mémoires sont traités. Les plus couramment utilisés sont les suivants :

1. *EREW* : modèle, le plus restrictif, permettant strictement la lecture et l'écriture en mode exclusif.
2. *CREW* : modèle, intermédiaire, permettant la lecture concurrente mais ne supportant que l'écriture exclusive.
3. *CRCW* : modèle, le moins restrictif, permettant la lecture et l'écriture concurrente.

Un modèle *ERCW* pourrait aussi être considéré, mais ne serait pas particulièrement intéressant, puisqu'il ne correspondrait pas vraiment à un type de machine intéressante et réalisable (si on peut écrire de façon concurrente, on peut très certainement lire de façon concurrente).

3 Quelques exemples d'algorithmes

Dans ce qui suit (basé en partie sur la notation du livre de Miller et Boxer), on suppose que les processeurs sont identifiés par $P_1, \ldots, P_p$. On suppose aussi que le processeur P_i possède un certain nombre m_i de registres identifiés par $r_{i,1}, \ldots, r_{i,m_i}$.

3.1 Diffusion d'une donnée aux divers processeurs

Problème : On veut faire en sorte qu'une donnée, contenue dans le registre $r_{1,k}$ du processeur P_1, soit diffusée à l'ensemble des processeurs.

L'algorithme pour une machine PRAM avec n processeurs supportant la lecture concurrente (CR-PRAM) est présenté à l'algorithme 1. Le processeur P_1 écrit simplement la valeur dans l'adresse a, adresse que tous les processeurs peuvent ensuite lire en parallèle (*Concurrent Read*).

Par contre, l'algorithme pour une machine PRAM (avec n processeurs) ne supportant pas la lecture concurrente (ER-PRAM) aurait plutôt l'allure de celui présenté à l'algorithme 2.

La technique utilisée dans cet algorithme est celle du *dédoublement récursif* (*recursive doubling procedure*) : ici, à chaque étape de l'algorithme, le nombre de copies de la donnée initiale est doublé. Dans ce cas-ci, cette stratégie implique aussi qu'à chaque étape on double le nombre de processeurs qui possèdent la bonne valeur et en effectuent une copie. Les

```
PROCEDURE diffuser( r_{1,k}: Registre )
PRECONDITION
  Le processeur P_1 possède la valeur d dans son registre r_{1,k}
POSTCONDITION
  Tous les processeurs P_i ont la valeur d dans leur registre r_{i,k}
DEBUT
  P_1 : transfère le contenu de son registre r_{1,k} dans l'adresse a
  EN PARALLÈLE pour chacun des processeurs P_j (j = 1, ..., n) FAIRE
    r_{j,k} <- a
  FIN
FIN
```

Algorithme 1: Diffusion d'une valeur à l'ensemble des processeurs pour une machine PRAM-CR

```
PROCEDURE diffuser( r_{1,k}: Registre )
PRECONDITION
  Le processeur P_1 possède la valeur d dans son registre r_{1,k}
  La valeur n est une puissance de 2, c'est-à-dire, n = 2^m, pour m ≥ 0
POSTCONDITION
  Tous les processeurs P_i ont la valeur d dans leur registre r_{i,k}
DEBUT
  P_1 : a_1 <- r_{1,k}
  POUR i <- 1 A lg n FAIRE
    EN PARALLÈLE pour chacun des processeurs P_j (j = 1, ..., 2^{i-1}) FAIRE
      r_{j,k} <- a_j
      a_{j+2^{i-1}} <- r_{j,k}
    FIN
  FIN
  EN PARALLÈLE pour chacun des processeurs P_j (j = n/2+1, ..., n) FAIRE
    r_{j,k} <- a_j
  FIN
FIN
```

Algorithme 2: Diffusion d'une valeur à l'ensemble des processeurs pour une machine PRAM-ER

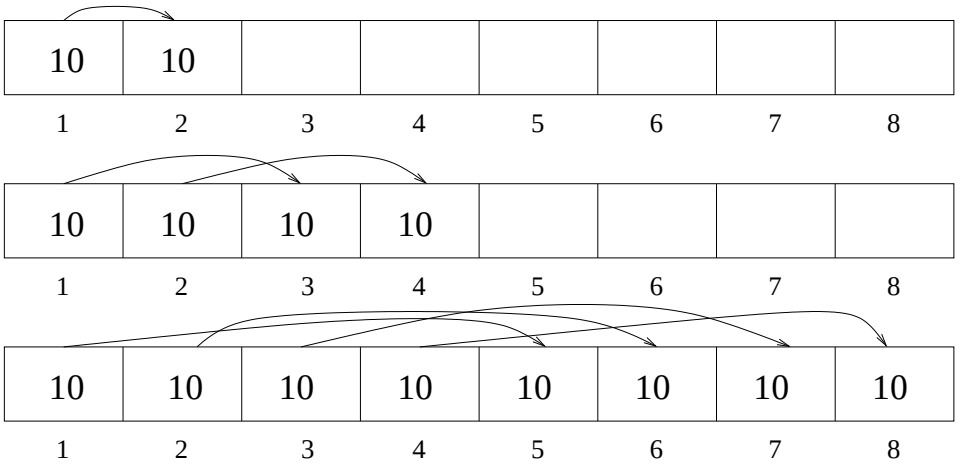


Figure 3: Illustration des copies effectuées par la diffusion avec dédoublement récursif pour l'algorithme PRAM-ER

copies/transferts qui sont effectués à chacune des itérations de la boucle `POUR`, et ce en parallèle, sont donc les suivants (à la $i^{ème}$ itération) :

$i = 1 : a_2 \leftarrow a_1$

$i = 2 : a_3 \leftarrow a_1,\ a_4 \leftarrow a_2$

$i = 3 : a_5 \leftarrow a_1,\ a_6 \leftarrow a_2,\ a_7 \leftarrow a_3,\ a_8 \leftarrow a_4$

...

$i = \lg n : a_{\frac{n}{2}+1} \leftarrow a_1, a_{\frac{n}{2}+2} \leftarrow a_2, \ldots, a_n \leftarrow a_{\frac{n}{2}}$

La dernière boucle parallèle assure ensuite, tel que requis par la post-condition, que les registres des derniers processeurs sont mis à jour avec le contenu de l'adresse appropriée.

Graphiquement, les copies et transferts sont donc tels qu'illustrés dans la figure 3 (exemple pour $n = 8$).

L'algorithme possède les propriétés suivantes :

- Nombre de processeurs : $n/2$.
- Temps d'exécution : on a lg n itérations de la boucle `POUR`, chaque itération s'exécutant en temps $\Theta(1)$, donc $\Theta(\lg n)$ pour la boucle `POUR`. Quant à la deuxième boucle parallèle, elle s'exécute évidemment en temps $\Theta(1)$. Le temps total est donc $\Theta(\lg n)$.
- Coût : $\Theta(n \lg n)$.

L'algorithme peut aussi être facilement modifié pour fonctionner même si n n'est *pas* une puissance de 2, tel que cela est illustré à l'algorithme 3.

3.2 Recherche de l'élément minimum parmi un tableau de données

Problème : On désire obtenir l'élément minimum parmi un tableau de n éléments. Une première version de l'algorithme est celle présentée à l'algorithme 4 (valide pour un modèle CR ou ER).

```
PROCEDURE diffuser( r_{1,k}: Registre )
PRECONDITION
  Le processeur P_1 possède la valeur d dans son registre r_{1,k}
POSTCONDITION
  Tous les processeurs P_i ont la valeur d dans leur registre r_{i,k}
DEBUT
  P_1 : a_1 <- r_{1,k}
  POUR i <- 1 A ⌈lg n⌉ FAIRE
    EN PARALLÈLE pour chacun des processeurs P_j (j = 1, ..., 2^{i-1}) FAIRE
      r_{j,k} <- a_j
      SI j + 2^{i-1} ≤ n ALORS
        // On copie uniquement si on ne dépasse pas le nombre de processeurs.
        a_{j+2^{i-1}} <- r_{j,k}
      FIN
    FIN
  FIN
  EN PARALLÈLE pour chacun des processeurs P_j (j = n/2+1, ..., n) FAIRE
    r_{j,k} <- a_j
  FIN
FIN
```

Algorithme 3: Diffusion d'une valeur à l'ensemble des processeurs pour une machine PRAM-ER (lorsque n n'est pas une puissance de 2)

```
PROCEDURE trouverMin( X: sequence{Ty} ) leMin: Ty
PRECONDITION
  La séquence X = [x_1, ..., x_n] contient n éléments.
  Les éléments de type Ty peuvent être comparés entre eux.
POSTCONDITION
  leMin = la plus petite valeur parmi x_1, ..., x_n.
  X n'est pas modifié.
DEBUT
  // On copie le tableau X dans un tableau temporaire T.
  EN PARALLÈLE pour chacun des processeurs P_j (j = 1, ..., n) FAIRE
     T[j] <- X[j]
  FIN
  // On recherche le minimum, en modifiant T.
  POUR i <- 1 A lg n FAIRE
    EN PARALLÈLE pour chacun des processeurs P_j (j = 1, ..., 2^{lg n-i}) FAIRE
      r_{j,1} <- T[2j-1]
      r_{j,2} <- T[2j]
      T[j] <- min(r_{j,1}, r_{j,2})
    FIN
  FIN
  leMin <- T[1]
FIN
```

Algorithme 4: Algorithme PRAM pour la recherche du minimum

Par exemple, si $n = 8$ et $\texttt{X} = [4, 0, 6, 8, 1, 6, 7, 5]$ on aura alors les résultats intermédiaires suivants *à la fin* de l'itération i de la deuxième boucle `POUR` :

1. $\texttt{T} = [0, 6, 1, 5, 1, 6, 7, 5]$: les éléments aux positions $j = 1, 2, 3, 4$ ont été modifiés ($\texttt{T}[1] \leftarrow \texttt{min}(\texttt{T}[1], \texttt{T}[2]), \ldots, \texttt{T}[4] \leftarrow \texttt{min}(\texttt{T}[7], \texttt{T}[8])$).
2. $\texttt{T} = [0, 1, 1, 5, 1, 6, 7, 5]$: les éléments aux positions $j = 1, 2$ ont été modifiés ($\texttt{T}[1] \leftarrow \texttt{min}(\texttt{T}[1], \texttt{T}[2]), \texttt{T}[2] \leftarrow \texttt{min}(\texttt{T}[3], \texttt{T}[4])$).
3. $\texttt{T} = [0, 1, 1, 5, 1, 6, 7, 5]$ ($\texttt{T}[1] \leftarrow \texttt{min}(\texttt{T}[1], \texttt{T}[2])$).

Graphiquement, les diverses itérations peuvent être présentées telles qu'illustrées à la figure 4.

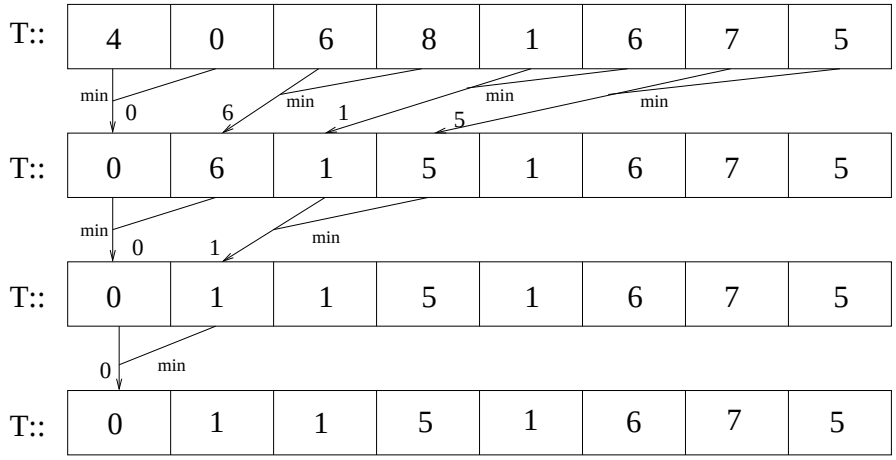


Figure 4: Illustration des transferts entre éléments pour le calcul ascendant du minimum

Les caractéristiques de cet algorithmes sont les suivantes :

- Nombre de processeurs : n.
- Temps : $\Theta(\lg\, n)$.
- Coût : $\Theta(n \lg\, n)$.

Notons que cet algorithme peut être interprété comme une façon d'obtenir un effet semblable à celui obtenu par un calcul récursif du minimum, mais sans avoir recours explicitement à la récursion, en utilisant plutôt une approche *ascendante* du calcul. À la première itération, qui correspond, en gros, au traitement des différents cas de base de la récursion,[2] les éléments sont comparés deux à deux ($n/2$ comparaisons et résultats). Les $n/4$ résultats de ces comparaisons sont ensuite comparés, deux à deux, pour obtenir $n/8$ autres résultats intermédiaires, etc., jusqu'à ce qu'un unique résultat soit produit. Il n'y a donc pas de récursion (ni d'allocations de bloc d'activations pour les arguments et variables locales des procédures, les valeurs intermédiaires étant conservées dans le tableau `T`), mais l'effet en termes de comparaisons effectuées et du résultat obtenu est semblable à ce qui se passe lors d'une approche récursive dichotomique (en supposant n une puissance de 2). Ceci est illustré à la figure 5.

[2] Plus précisément, c'est comme si on terminait la récursion lorsqu'il y a deux (2) éléments à comparer, ce qui correspondrait alors aux cas de base.

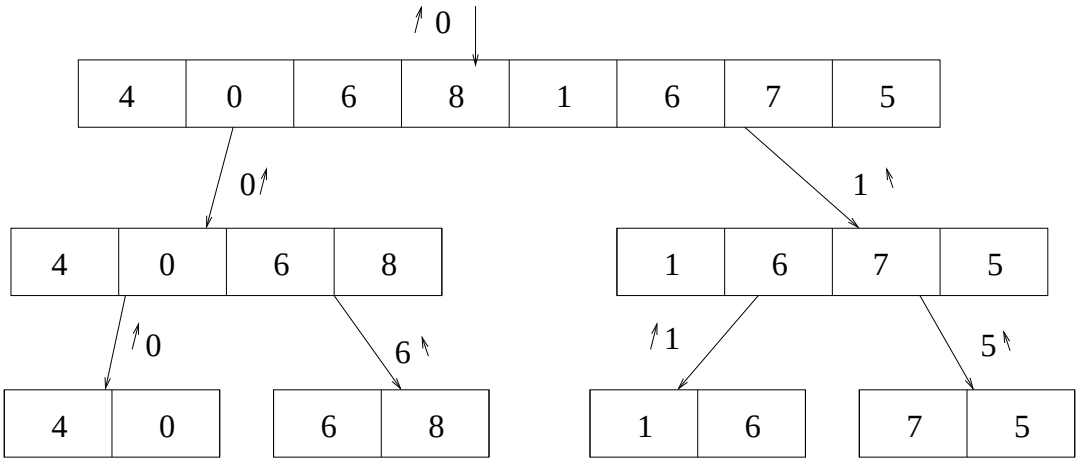


Figure 5: Arbre des appels pour une version récursive de recherche du minimum avec coupure (seuil) lorsqu'il ne reste que deux (2) éléments

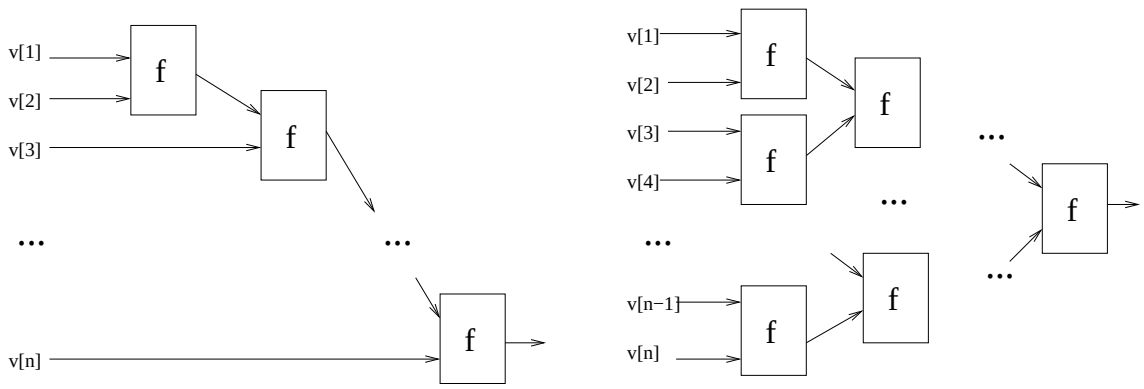
En d'autres mots, l'approche utilisée dans cet algorithme peut être interprétée comme une approche complémentaire (dual, symétrique) à une approche diviser-pour-régner récursive : alors que la stratégie diviser-pour-régner classique utilise une approche descendante, l'algorithme présenté utilise plutôt une approche *ascendante*. De même, alors que l'approche diviser-pour-régner débute avec *un* problème complexe qu'on décompose en sous-problèmes plus simples, l'approche utilisée ici débute avec plusieurs sous-problèmes simples, les solutions de ces sous-problèmes étant ensuite combinées, de façon répétitive, jusqu'à ce que le problème complet soit solutionné.

Notons que certains auteurs utilisent aussi l'appelation de *dédoublement* pour des algorithmes basés sur la stratégie utilisée dans la recherche du minimum. Plus précisément, Xavier et Iyengar parlent d'une approche de *growing by doubling* [XI98] :

> *If n entities ar to be* covered *for processing, at the initial step each processor covers a single entity and waits for consolidation. At the first step, the number of entities covered by each processor will be two, and at the second stage it will be 4, and at the third stage it is 8, and so on. At every step* the number of entities covered *is double the entities covered at the previous step.*

On rencontre aussi une stratégie équivalente, dans un contexte de parallélisme de données, sous le nom de "*β-réduction logarithmique*" [GUD96] :

> Une opération de type β est définie par la donnée d'une fonction f binaire et d'un unique vecteur v de taille l. Elle correspond à l'application de f sur les deux premiers éléments de v et puis, par la suite, l'application de f sur le résultat d'un calcul précédent et l'élément suivant du vecteur v. [...] si f est associative il est possible d'appliquer une variante permettant d'exploiter le parallélisme sous forme du schéma de *β-réduction logarithmique*. La fonction f associative permet de regrouper les arguments différemment et d'aboutir à une réduction en forme d'arbre comme indiqué sur la figure [6]. Le schéma β autorise donc le parallélisme pour les fonctions associatives.



(a) Schéma béta (b) Schéma béta−logarithmique

Figure 6: Schémas de réduction β et β-logarithmique

Note : Un point important à souligner concernant l'algorithme 4 est qu'il utilise certains "raccourcis" (abréviations) de notation. Par exemple, le modèle PRAM pur ne permet pas d'effectuer de façon directe un transfert de mémoire à mémoire, comme on le fait avec une instruction `T[`j`] <- X[`j`]` ou `T[`j`] <- min(`$r_{j,1}$`, `$r_{j,2}$`)`. De tels transferts devraient plutôt s'exprimer explicitement par un transfert dans un registre local du processeur puis par un calcul sur des registres ou par une écriture d'un registre dans la mémoire. En d'autres mots, de façon stricte, il aurait plutôt fallu écrire ce qui suit :

```
r_{j,3} <- X[j]
T[j] <- r_{j,3}
...
r_{j, 3} <- min(r_{j,1}, r_{j,2})
T[j] <- r_{j,3}
```

Toutefois, il est clair que la complexité du temps et celle du travail effectué sont les mêmes.

4 Algorithmes PRAM exprimés en MPD

En fait, et comme on le fera par la suite avec d'autres exemples, on peut souvent écrire des algorithmes dans la notation MPD, algorithmes qu'on peut ensuite interpréter comme des algorithmes PRAM si on établit la correspondance suivante lors de l'analyse de l'algorithme, comme on l'a fait précédemment : *le nombre de processeurs requis par l'algorithme est le nombre* maximum *de processus actifs requis à n'importe quel instant durant l'exécution de l'algorithme.*

4.1 Recherche de l'élément minimum d'un tableau

Un premier exemple est présenté à l'algorithme 5. L'analyse de cet algorithme, pour n une puissance de 2, permet de déterminer les propriétés suivantes :

- Nombre de processeurs requis : n.

  La copie de `X` dans `T` génère exactement n processus (n instances de `copier`). Quant à l'autre boucle, le plus grand nombre de processus actifs survient lorsque `i=1`, où l'on a alors $n/2$ instances du processus `min2`.

```
  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
```

Algorithme 5: Algorithme PRAM (notation MPD) pour la recherche du minimum

- Temps d'exécution : $\Theta(\lg\, n)$.

  Avec n processus, la copie s'effectue en temps $\Theta(1)$. Le temps requis pour un appel à `min2` est $\Theta(1)$. Chacune des itérations de la boucle `for` s'effectue donc en temps $\Theta(1)$ (processus exécutés en parallèle). Comme le nombre d'itérations est de $\lg\, n$, le temps total est donc $\Theta(\lg\, n)$.

- Coût : un total de n processeurs sont nécessaires, le tout durant un temps $\Theta(\lg\, n)$. Le coût est donc $\Theta(n \lg\, n)$. Ceci n'est évidemment pas optimal, puisqu'un algorithme séquentiel peut trouver le minimum en temps et en coût $\Theta(n)$.

- Travail : le travail requis pour la copie est $\Theta(n)$ (n processeurs exécutant $\Theta(1)$ opérations). Le nombre de processeurs requis pour la recherche du minimum est borné par $n/2$. Toutefois, le nombre de processeurs requis diminue au fur et à mesure que la boucle `for` progresse.

Le travail effectif peut être calculé comme suit :

$$\begin{aligned}
\sum_{i=1}^{\lg\ n} n/2^i &= n\sum_{i=1}^{\lg\ n}(\frac{1}{2})^i \\
&= n(-1+\sum_{i=0}^{\lg\ n}(\frac{1}{2})^i) \\
&= n(-1+\frac{(\frac{1}{2})^{\lg\ n+1}-1}{\frac{1}{2}-1}) \\
&= n[-1+(2-(\frac{1}{2})^{\lg\ n})] \\
&= n(1-\frac{1}{2^{\lg\ n}}) \\
&= n(1-\frac{1}{n}) \\
&= n-1
\end{aligned}$$

Le travail est donc $\Theta(n)$. En terme de travail plutôt que de coût, cet algorithme est donc optimal.

Finalement, soulignons que dans la mesure où le mode de passage par défaut des arguments en MPD est le mode de passage par valeur (*copy-in*), y compris pour les tableaux, l'algorithme 5 pourrait être simplifié comme suit :

```
procedure trouverMin( int X[*], int n ) returns int leMin
{
  for [i = 1 to lg(n)] {
    co [j = 1 to n/(2**i)]
      X[j] = min2( X[2*j-1], X[2*j] );
    oc
  }
  leMin = X[1];
}
```

Toutefois, l'utilisation d'un tableau auxiliaire `T` deviendra nécessaire dans la prochaine version de l'algorithme, où l'on tente d'obtenir un coût qui soit optimal.

Notons finalement que l'utilisation de la fonction `copier` dans l'algorithme 5 est nécessaire à cause de restrictions syntaxiques du langage MPD. Plus spécifiquement, seuls des appels (activations) de procédures ou fonctions peuvent apparaître dans un `co`. Le code suivant ne serait donc pas accepté par le compilateur, d'où la nécessité d'introduire une fonction auxiliaire `copier` :

```
co [i = 1 to n]
  T[i] = X[i]
oc
```

Pour plus de détails et pour une instruction *pseudo-MPD* permettant de simplifier la présentation d'algorithmes PRAM, cf. 7.4 (p. 33).

4.2 Recherche de l'élément minimum d'un tableau avec un coût optimal

Il est aussi possible d'obtenir un algorithme de style PRAM pour ce problème qui, même au niveau du coût, soit aussi de complexité $\Theta(n)$. En principe, deux stratégies sont envisageables

```
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
# POSTCONDITION
#   leMin = MINIMUM( i <= k <= j :: a[k] )
{
  # Recherche sequentielle du minimum de a[i:j].
  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];
}
```

Algorithme 6: Algorithme PRAM de coût optimal (notation MPD) pour la recherche du minimum

pour réduire le coût : réduire le temps d'exécution, ou réduire le nombre de processeurs. Étant donné la réduction du temps d'exécution déjà obtenue par rapport à l'algorithme séquentiel, une stratégie qui vise à réduire le nombre de processeurs semble plus prometteuse.

Dans l'algorithme 5, le coût total est $\Theta(n \lg n)$ parce que n processeurs sont utilisés. Pour obtenir un coût de $\Theta(n)$, équivalent au coût de l'algorithme séquentiel, tout en conservant un temp $\Theta(\lg n)$, il faudrait donc réduire le nombre de processeurs à $n/\lg n$. Ceci peut effectivement se faire en utilisant tout d'abord un tel nombre de processeurs pour trouver le minimum parmi des paquets de $\lg n$ éléments, de façon à obtenir $n/\lg n$ valeurs minimum intermédiaires. La stratégie de recherche du minimum utilisée dans la version non optimale de l'algorithme peut ensuite être utilisée sur ces $n/\lg n$ valeurs intermédiaires. L'algorithme, dans la notation MPD, est présenté à l'algorithme 6.

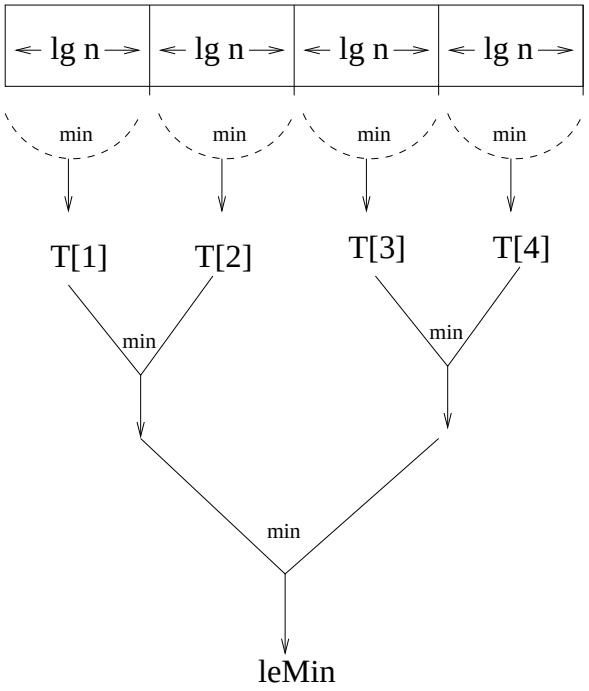


Figure 7: Illustration graphique pour $n = 16$ de l'approche de coût optimal pour la recherche du minimum

Intuitivement, le fonctionnement de cet algorithme peut être interprété comme l'utilisation d'une approche diviser-pour-régner *ascendante* avec utilisation d'un *seuil* pour terminer la récursion à partir d'un certain point (voir la section 2.7 du chapitre "Diviser-pour-régner"). Les sous-problèmes de base (les plus simples), qui sont traités séquentiellement (sans récursion, ou, dans le cas présent, sans parallélisme), sont donc ceux de taille $\lg n$. Graphiquement ceci peut être illustré tel que présenté à la figure 7 (pour le cas $n = 16$, donc $\lg n = 4$ et $n/\lg n = 4$).

L'analyse de cet algorithme nous donne les caractéristiques suivantes :

- Nombre de processeurs requis : `nbPaquets` $= n/\lg n$, puisque les deux instructions `co` ont une borne supérieure de `nbPaquets`.
- Temps d'exécution : $O(\lg n)$.

  Dans la première instruction `co`, les divers processus effectuent une recherche séquentielle du minimum sur `tailllePaquets` $= \lg n$ éléments, donc en temps $\Theta(\lg n)$.

Le temps requis pour un appel à `min2` est $\Theta(1)$. Chaque itération de la boucle `for` s'effectue donc en temps $\Theta(1)$ (avec au plus $n/\lg n$ processeurs). Le nombre d'itérations est $\lg$ `nbPaquets` $= \lg\ (n/\lg\ n)$. Pour une constante c appropriée (temps $\Theta(1)$ pour chaque itération), on a donc un temps total pour la boucle `for` défini comme suit :

$$\begin{aligned} \lg(n/\lg\ n) \times \Theta(1) &= \lg(n/\lg\ n) \times c \\ &= (\lg\ n - \lg^2\ n) \times c \\ &\leq (\lg\ n) \times c \\ &\in O(\lg\ n) \end{aligned}$$

Le temps d'exécution est donc $O(\lg\ n)$.

- Coût : un maximum de $n/\lg\ n$ processeurs sont nécessaires, le tout durant un temps $O(\lg\ n)$. Le coût est donc $O(n)$, ce qui est optimal relativement à l'algorithme séquentiel standard.

4.3 Réduction du coût d'un algorithme PRAM

La stratégie utilisée dans l'exemple précédent pour réduire le coût d'un algorithme PRAM est appelée *accelerated cascading* par certains auteurs, alors que d'autres auteurs parlent plutôt de *blocking technique*. De façon générale, cette stratégie, telle que décrite dans le livre de Keller *et al.* [KKT01], comporte trois éléments (qu'on peut aussi voir comme une forme de diviser-pour-régner) :

1. On utilise un algorithme optimal en coût pour réduire la taille du problème.
2. On utilise un algorithme rapide, mais non optimal, pour calculer la solution aux problèmes de taille réduite.
3. On utilise un algorithme optimal en coût pour calculer la solution au problème initial à partir des solutions aux problèmes réduits.

Pour l'exemple précédent de recherche du minimum, on a donc la correspondance suivante :

1. La réduction du problème initial en sous-problèmes plus simples est relativement directe, puisqu'il suffit de décomposer l'intervalle d'éléments à examiner en sous-intervalles.
2. L'algorithme non optimal consiste à trouver le minimum d'un intervalle réduit ($\lg\ n$) d'éléments de façon séquentielle. Ceci permet alors d'obtenir un nombre réduit d'éléments à examiner (comptant $n/\lg\ n$ éléments plutôt que n).
3. L'algorithme optimal en coût utilisé est celui du dédoublement, mais appliqué sur $n/\lg\ n$ éléments seulement.

Tel que mentionné précédemment, une telle approche peut être interprétée comme semblable à l'utilisation d'une approche diviser-pour-régner *ascendante* avec utilisation d'un *seuil* pour terminer la récursion lorsque le problème devient suffisamment simple pour être résolu à l'aide d'une méthode plus directe (ici, séquentielle plutôt que parallèle).

4.4 Limites de la présentation MPD d'algorithmes PRAM

Une caractéristique importante d'un algorithme PRAM est la suivante : l'exécution des différentes phases d'un algorithme PRAM se fait de façon *synchrone*. Plus précisément, ceci signifie que, à une étape donnée, toutes les lectures sont faites en parallèles (dans les registres),

puis tous les calculs sont faits en parallèles, suivis finalement des écritures. De façon stricte, ceci signifie donc que l'algorithme 5 *n'est pas correct lorsqu'exprimé en* MPD `:(`

Le problème est que, en théorie dans le modèle MPD, les diverses activations d'une instruction `co` peuvent s'exécuter en parallèle, donc *dans n'importe quel ordre.*[3] Dans l'exemple de l'algorithme 5, on a la boucle parallèle suivante :

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

Pour `n=8` et `i=2`, les diverses activations de l'instruction `co` pourraient donc, en théorie selon la sémantique du langage MPD, s'exécuter dans l'ordre suivant :

1. `T[2] = min2( T[3], T[4] );`
2. `T[1] = min2( T[1], T[2] );`

En d'autres mots, la valeur utilisée dans le deuxième appel à `min2` *ne serait pas la bonne*, la valeur calculée pour le premier appel ayant écrasé la valeur devant être utilisée par le deuxième appel.

```
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
{
  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];
}
```

Algorithme 7: Algorithme PRAM corrigé (notation MPD) pour la recherche du minimum

[3] En pratique, sur une machine uniprocesseur, les diverses activations d'une instruction `co` s'exécutent comme les diverses itérations d'une instruction `for` (en ordre croissant de l'index), ce qui ne crée aucun problème pour cet algorithme.

Une première façon de corriger ce problème est présentée à l'algorithme 7. Dans cette version, une variable locale `tmp` (un tableau) est utilisée pour recevoir les résultats intermédiaires du calcul des divers minimums. À la fin de la boucle, avant d'amorcer la prochaine itération, on recopie alors le contenu de ce tableau dans le tableau `T`.

```
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];
}
```

Algorithme 8: Autre algorithme PRAM correct (notation MPD) pour la recherche du minimum

Une autre version correcte d'un programme MPD de style PRAM pour la recherche du minimum est présentée à l'algorithme 8. Cet algorithme ne souffre pas du problème mentionné plus haut car l'entrée de `T` modifiée par chacun des appels à `min2` est l'une des entrées transmise en argument à `min2`. En d'autres mots, il n'y a pas d'interaction directe, pas d'interférences, entre chacun des appels à `min2`. Soit `X = [4, 0, 6, 8, 1, 6, 7, 5]` le tableau pour lequel on désire rechercher le minimum. Le contenu de `T` après chacune des itérations sera donc tel qu'illustré à la figure 8.

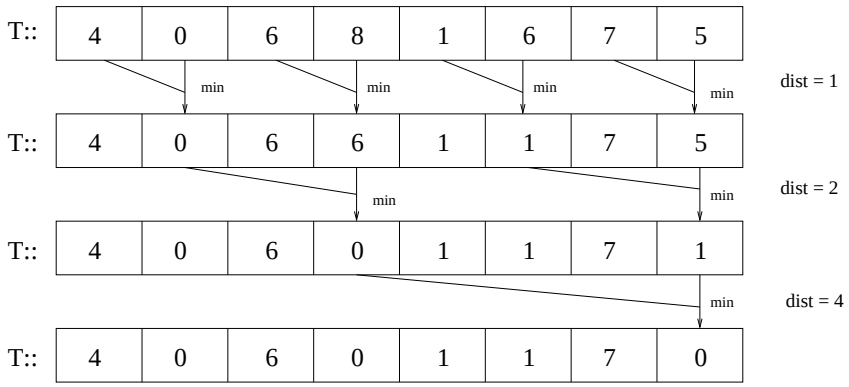


Figure 8: Illustration des transferts entre éléments pour le calcul ascendant du minimum

Soulignons que, comme pour l'algorithme 5, l'algorithme 8 pourrait être simplifié comme suit (le passage des arguments se fait, par défaut en MPD, par valeur) :

```
procedure trouverMin( int X[*], int n ) returns int leMin
{
  for [i = 1 to lg(n)] {
    int dist = 2**(i-1);
    co [j = 1 to (n/2)/dist]
      X[dist*2*j] = min2( X[dist*(2*j-1)], X[dist*2*j] );
    oc
  }
  leMin = X[n];
}
```

5 Multiplications de matrices

La stratégie utilisée dans les exemples précédents pour obtenir un algorithme parallèle efficace et de coût optimal peut être utilisée dans de nombreux algorithmes, par exemple, pour effectuer la multiplication de deux matrices $n \times n$, tel que vu dans la série d'exercices #7.

6 Calcul parallèle des préfixes

Le calcul parallèle des "préfixes" d'une série de valeurs est très une technique importante en programmation parallèle. Sur de nombreuses architectures parallèles, il existe souvent une (ou plusieurs) façon(s) efficace de calculer en parallèle les préfixes d'une série de valeurs. Or, plusieurs problèmes peuvent à leur tour être solutionnés de façon efficace lorsqu'on est capable de calculer les préfixes de façon efficace (par exemple, voir Section 6.4). En fait, sur certaines architectures parallèles développées à la fin des années 80 (par exemple, les *Connection Machines*, des machines SIMD conçues par la compagnie *Intelligent Thinking Machines* ;), l'opération de calcul parallèle des préfixes (appelée *scan*) était tellement fondamentale qu'elle était considérée comme une opération *primitive*.

6.1 Définition de la notion de préfixes

Soit $X = [x_1, \ldots, x_n]$ une série de valeurs provenant d'un ensemble de départ Y. Soit "$\otimes$" une opération binaire associative fermée relativement à Y (c'est-à-dire que $x, y \in Y \Rightarrow x \otimes y \in Y$).

Le résultat $x_1 \otimes x_2 \otimes \ldots \otimes x_k$ est appelé le $k$ième préfixe de X. Règle général, on s'intéresse plutôt au calcul des n préfixes x_1, $x_1 \otimes x_2$, $x_1 \otimes x_2 \otimes x_3$, $\ldots$, $x_1 \otimes x_2 \otimes \ldots \otimes x_{n-1}$, $x_1 \otimes x_2 \otimes \ldots \otimes x_n$. Lorsqu'effectué en parallèle, on parle alors du calcul parallèle des préfixes.

Typiquement, l'opérateur "$\otimes$" est un opérateur de base s'effectuant en temps $\Theta(1)$, par exemple, "$+$", "$\times$", "`ET`", "`OU`", "`MIN`","`MAX`", etc.

Pour un opérateur $\otimes$ s'effectuant en temps $\Theta(1)$, le calcul des préfixes sur une machine séquentielle peut s'effectuer en temps $\Theta(n)$ en parcourant — en anglais en *scannant*, ou en balayant (*sweep*) — les éléments, d'où le terme anglais parfois utilisé d'opération `scan` ou `sweep` pour le calcul des préfixes.

6.2 Approche diviser-pour-régner récursive pour le calcul des préfixes

Avant d'examiner une solution parallèle, examinons une première façon de calculer les préfixes qu'il serait relativement facile de paralléliser de façon naïve (appels récursifs en parallèle). L'algorithme (en MPD) est présenté, en partie, à l'algorithme 9.

```
# Operateur associatif utilise pour le calcul des prefixes relatifs a la somme.
procedure op0( int x, int y ) returns int r
{ r = x + y; }

procedure calculerPrefixes( int x[*], res int prefixes[*], int n )
# PRECONDITION
#   n >= 1 & SOME( k >= 0 :: n = 2**k )
# POSTCONDITION
#   ALL( 1 <= i <= n ::
#        prefixes[i] = x[1] `op0` x[2] `op0` ... `op0` x[i] )
#
# Note: x `op0` y = op0(x, y) (application infixe vs. prefixe)
{
  if (n == 1) {
     # Probleme trivial (cas de base).
     prefixes[1] = x[1];
  } else {
     # Probleme plus complexe (cas recursif)
     # Decomposition en deux sous-problemes.
     int mid = n/2;
     calculerPrefixes( x[1:mid],   prefixes[1:mid],   n/2 );
     calculerPrefixes( x[mid+1:n], prefixes[mid+1:n], n/2 );

     # Combinaison des solutions aux sous-problemes pour obtenir la
     # solution globale.

     ... A completer ...



  }
}
```

Algorithme 9: Calcul des préfixes à l'aide d'une approche récursive

Exercice : comment la combinaison des sous-solutions doit-elle être effectuée pour obtenir la solution globale (partie à compléter)?

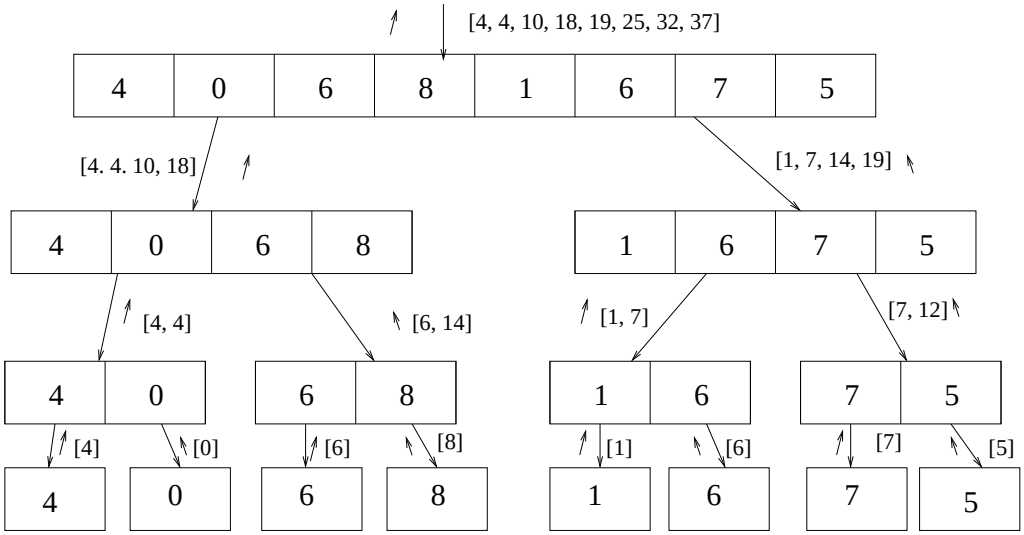


Figure 9: Arbre des appels pour la version récursive de calcul des préfixes pour huit éléments

L'arbre des appels récursifs pour $X = [4, 0, 6, 8, 1, 6, 7, 5]$ et $\otimes = +$ serait donc tel qu'illustré à la figure 9, les séquences apparaissant près des flèches d'appel dénotant les résultats produits par la procédure appelée.

6.3 Calcul parallèle (en MPD) des préfixes

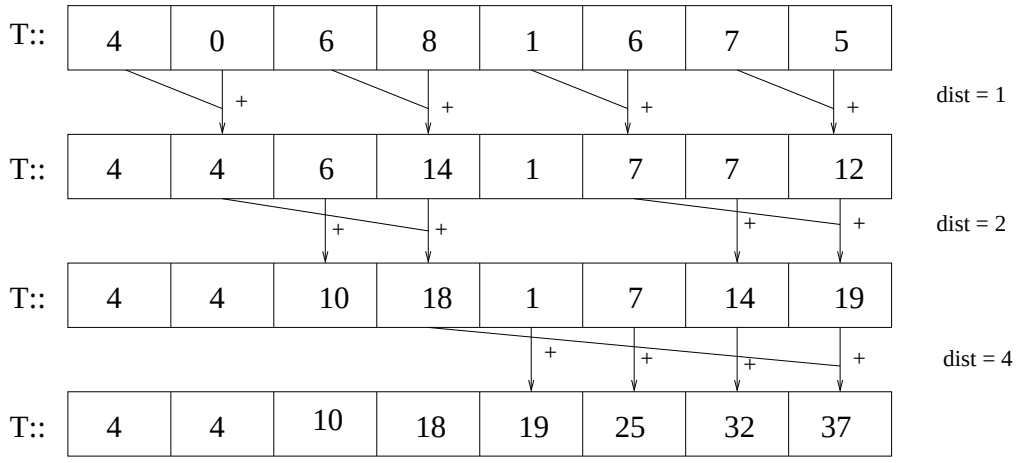


Figure 10: Illustration des calculs effectués dans l'algorithme 10

La stratégie utilisée dans la version récursive présentée à l'algorithme 9 peut être adaptée de façon à produire un algorithme de style PRAM, tout comme on l'a fait, précédemment, pour calculer efficacement le minimum d'une série de valeurs : voir algorithme 10. Graphiquement, les transferts et calculs effectués aux diverses itérations de l'algorithme seraient tels que présentés à la figure 10.

```
procedure copier( int x ) returns int r
{ r = x; }

procedure op0( int x, int y ) returns int r
{ r = x + y; }

procedure combinerPrefixes( ref int prefixes[*], int k, int dist )
{
  co [i = k+1 to k+dist]
    prefixes[i] = op0( prefixes[i], prefixes[k] );
  oc
}

procedure calculerPrefixes( int x[*], res int prefixes[*], int n )
{
  # Definition des cas de base.
  co [i = 1 to n]
    prefixes[i] = copier( x[i] );
  oc

  # Combinaison repetitive des sous-solutions.
  for [i = 1 to lg(n)] {
    int dist = 2**(i-1);
    co [j = dist to n by 2*dist]
      combinerPrefixes( prefixes, j, dist )
    oc
  }
}
```

Algorithme 10: Calcul parallèle des préfixes à la PRAM

Les caractéristiques de cet algorithme sont les suivantes :

- Nombre de processeurs : n (à cause du `co` qui effectue la copie initiale).
- Temps : $\Theta(\lg\ n)$.
- Coût : $\Theta(n\ \lg\ n)$.

Cet algorithme, bien que plus rapide que l'algorithme séquentiel, n'est donc pas optimal. Il est toutefois possible, comme on l'a fait pour le calcul parallèle du minimum (série précédente de notes de cours) ou pour la multiplication de matrices (série d'exercices #8), de réduire le coût en réduisant le nombre de processeurs à $\Theta(n/\lg\ n)$, ce qui peut être fait en faisant traiter des paquets de $\lg\ n$ éléments par chacun des processeurs (exercice en classe ou devoir?).

6.4 Application : Sous-séquence de somme maximum

Nous allons maintenant illustrer comment le calcul parallèle des préfixes peut être utilisé pour résoudre efficacement d'autres problèmes.

Problème : Soit une séquence de valeurs $x = [x_1, \ldots, x_n]$. On désire déterminer les indices u et v, $u \leq v$, tels que la somme des éléments de la sous-séquence $[x_u, \ldots, x_v]$ soit maximale, et ce parmi toutes les sous-séquences possibles de x.

Pour que le problème soit intéressant, on peut supposer que les éléments x_i ne sont pas tous de même signe : s'ils sont tous positifs, alors la solution triviale est de prendre l'ensemble de la séquence ($u = 1, v = n$) ; par contre, s'ils sont tous négatifs, alors la séquence vide est la solution triviale.

Il est possible de résoudre ce problème à l'aide d'un algorithme séquentiel de complexité $\Theta(n)$ (voir [MB00, p. 144]).

Une solution parallèle pour le modèle PRAM peut être obtenue en effectuant deux calculs parallèles de préfixes distincts. Une solution pour une version simplifiée du problème (recherche de la somme maximum, sans identifier de façon explicite les bornes u et v de la sous-séquence correspondante) est présentée à l'algorithme 11.

Les grandes lignes de cet algorithme sont les suivantes, que nous illustrerons avec la séquence $\texttt{x} = [-3, 5, 2, -1, -4, 8, 10, -2]$:

- Dans un premier temps, on calcule en parallèle les préfixes des sommes des éléments de la séquence `x`. On obtient alors $\texttt{sommes} = [-3, 2, 4, 3, -1, 7, 17, 15]$. Cette séquence nous indique donc, pour une position donnée, la somme de *tous* les éléments à gauche de cette position (en incluant aussi l'élément lui-même).

  Notons que, dans cette version du calcul parallèle des préfixes, la procédure `calculerPrefixes` a été paramétrisée de façon à la rendre *générique* par rapport à l'opérateur binaire devant être utilisé. L'en-tête de cette procédure est donc maintenant le suivant :

  ```
  procedure calculerPrefixes( int x[*], res int prefixes[*], int n, cap OpBinaire op0 )
  ```

  Le dernier argument est de type `cap OpBinaire`, lequel type est déclaré comme suit, c'est-à-dire, dénote une opération arbitraire du type indiqué (avec la signature indiquée) :

  ```
  optype OpBinaire = (int, int) returns int
  ```

  Le mot-clé `cap` dénote une `cap`*ability*. Plus spécifiquement, cela indique une forme de référence à une opération (`op`) ou procédure (`procedure`). Le dernier argument devant être fourni lors d'un appel à `calculerPrefixes` doit donc être une fonction recevant deux arguments entiers et retournant un résultat entier. Pour le calcul parallèle des préfixes des sommes, la fonction `plus2` (haut de l'algorithme 11) est donc du type approprié.

```
# Type pour des operateurs binaires.
optype OpBinaire = (int, int) returns int;

# Les deux operateurs binaires utilises par l'algorithme
# dans le calcul des prefixes/postfixes paralleles.
procedure plus2( int x, int y ) returns int r { r = x + y; }
procedure max2 ( int x, int y ) returns int r { r = max(x, y); }
...

procedure calculerPrefixes( int x[*], res int prefixes[*], int n, cap OpBinaire op0 )
{ ... code inchange ... }

procedure inverser( int x[*], res int y[*], int n )
{ co [i = 1 to n]  y[n-i+1] = copier(x[i]);  oc }

procedure calculerPostfixes( int x[*], res int postfs[*], int n, cap OpBinaire op0 )
{
  int prefs[n];

  # Pour un operateur binaire commutatif, les postfixes peuvent
  # etre obtenus en inversant les prefixes pour la sequence inverse.
  inverser( x, x, n );
  calculerPrefixes( x, prefs, n, op0 );
  inverser( prefs, postfs, n );
}

procedure maxPrefixeDroite( int s, int m, int x ) returns int r
{ r = m - s + x;  }

procedure trouverMaxSomme( int x[*], int n ) returns int s
{
  int sommes[n], maxs[n], b[n];

  # On calcule les prefixes des sommes des elements.
  calculerPrefixes ( x,      sommes,  n, plus2 );

  # On calcule les postfixes des maximums des sommes.
  calculerPostfixes( sommes, maxs,    n, max2  );

  # On determine, pour chaque position, le maximum pouvant etre
  # obtenu a partir de cette position.
  co [i = 1 to n]
    b[i] = maxPrefixeDroite( sommes[i], maxs[i], x[i] );
  oc
  # On obtient le maximum parmi les valeurs obtenues.
  s = trouverMax( b, n );
}
```

Algorithme 11: Recherche parallèle de la sous-séquence de somme maximum à l'aide de préfixes calculés en parallèles

- L'étape suivante consiste à calculer en parallèle les postfixes pour l'opérateur `MAX` relativement aux préfixes des sommes déjà calculées. Ceci permettra alors de déterminer, pour une position donnée, le maximum pouvant être obtenu pour les positions à droite. Dans notre exemple, on aura `maxs` $= [17, 17, 17, 17, 17, 17, 17, 15]$.

  Plus spécifiquement pour la notion de postfixes, soit $x = [x_1, \ldots, x_n]$ une série de valeurs. Soit $\otimes$ une opération binaire associative. Le calcul des n postfixes consiste à calculer $x_1 \otimes x_2 \otimes \ldots \otimes x_n,\ x_2 \otimes x_3 \otimes \ldots \otimes x_n,\ \ldots,\ x_{n-1} \otimes x_n, x_n$. Or, dans le cas d'un opérateur binaire *commutatif*, ceci est équivalent à calculer les préfixes de $x' = [x_n, \ldots, x_1]$, puis à inverser le résultat obtenu. C'est effectivement ce qui est fait dans la procédure `calculerPostfixes` (`x` peut être réutilisé, puisqu'étant passé par valeur, les changements effectués sur `x` seront strictement locaux) :

```
procedure calculerPostfixes( int x[*], res int postfs[*], int n, cap OpBinaire op0 )
{
  int prefs[n];

  inverser( x, x, n );
  calculerPrefixes( x, prefs, n, op0 );
  inverser( prefs, postfs, n );
}
```

  Ici aussi on remarque que la procédure a été paramétrisée avec un argument indiquant l'opération binaire à utiliser. Dans le contexte de notre algorithme, cet argument est associé à la procédure `max2` (dans l'appel effectué dans `trouverMaxSomme`).

- Connaissant, pour chaque position, la somme de la sous-séquence allant de cette position jusqu'à la fin, de même que les sommes équivalentes pour les séquences débutant à des positions strictement à droite, on peut maintenant déterminer, pour chacune des positions, le maximum pouvant être obtenu parmi toutes les sous-séquences débutant à cette position.

  Notons par s_i la somme de la sous-séquence débutant à la position i (i.e., `sommes[i]`) et par m_i le maximum pour les positions à droite (i.e., `maxs[i]`). La valeur b_i suivante va alors indiquer, pour chacune des positions i, la somme maximale pouvant être obtenue pour n'importe quelle sous-séquence débutant à cette position :

$$b_i = m_i - s_i + x_i$$

  C'est ce qui est calculé dans l'instruction `co` à la fin de la procédure `trouverMaxSomme`. Dans notre exemple, on obtient alors `b` $= [17, 20, 15, 13, 14, 18, 10, -2]$.

- Plus précisément, on a pour une position `i` donnée, les équivalences suivantes :

$$\begin{aligned} \texttt{sommes[i]} &= \texttt{x[1]} + \ldots + \texttt{x[i]} \\ \texttt{maxs[i]} &= \texttt{max}\{\texttt{sommes[i]}, \ldots, \texttt{sommes[n]}\} \end{aligned}$$

  On aura donc `b[i]` défini comme suit :

$$\begin{aligned} \texttt{b[i]} &= \texttt{maxs[i]} - \texttt{sommes[i]} + \texttt{x[i]} \\ &= \texttt{max}\{\texttt{sommes[i]}, \ldots, \texttt{sommes[n]}\} - \texttt{sommes[i]} + \texttt{x[i]} \\ &= \texttt{max}\{\texttt{x[1]} + \ldots + \texttt{x[i]}, \ldots, \texttt{x[1]} + \ldots + \texttt{x[n]}\} - (\texttt{x[1]} + \ldots + \texttt{x[i]}) + \texttt{x[i]} \\ &= \texttt{max}\{\texttt{x[1]} + \ldots + \texttt{x[i]}, \ldots, \texttt{x[1]} + \ldots + \texttt{x[n]}\} - (\texttt{x[1]} + \ldots + \texttt{x[i-1]}) \\ &= \texttt{max}\{\texttt{x[i]}, \texttt{x[i]} + \texttt{x[i+1]}, \ldots, \texttt{x[i]} + \ldots + \texttt{x[n]}\} \end{aligned}$$

  On voit donc bien que `b[i]` nous indique alors la somme maximum parmi toutes les séquences débutant à la position `i`.

Une fois chacune des valeurs `b[i]` calculées, il suffit alors de trouver la valeur maximum parmi celles-ci, ce qui est réalisé par l'appel à `trouverMax` à la fin de la procédure.

Dans notre exemple, la valeur retournée sera donc 20, qui correspond à la sous-séquence $[5, 2, -1, -4, 8, 10]$. Il est intéressant de souligner que cette séquence compte des éléments positifs et d'autres négatifs.

Analyse de l'algorithme

Pour analyser cet algorithme, nous allons tout d'abord supposer que les algorithmes utilisés pour le calcul parallèle des préfixes et pour la recherche du maximum sont ceux présentés précédemment, donc qui fonctionnent en temps $\Theta(\lg\ n)$, mais qui ne sont pas optimaux en coût. On a alors les caractéristiques suivantes :

- Nombre de processeurs : n.
- Temps : $\Theta(\lg\ n)$.

  On a vu précédemment que le calcul parallèle des préfixes peut se faire en temps $\Theta(\lg\ n)$. Il en est de même pour le calcul parallèle des postfixes (la procédure `inverser` est $\Theta(1)$ avec n processeurs). Idem pour la recherche du maximum avec `trouverMax` (puisqu'on sait comment trouver le minimum en temps logarithmique). Finalement, le calcul des maximum locaux (`maxPrefixeDroite`) peut lui aussi se faire en $\Theta(1)$.
- Coût : $\Theta(n\ \lg\ n)$.

Cet algorithme n'est donc pas optimal, puisque la version séquentielle peut se faire en temps $\Theta(n)$. Toutefois, comme on l'a vu ou mentionné précédemment, il est possible d'obtenir, tant pour le calcul parallèle des préfixes que pour la recherche du maximum, des algorithmes en temps $\Theta(\lg\ n)$ qui soient malgré tout optimaux en termes de coûts ($\Theta(n)$). On peut donc en conclure qu'il existe un algorithme de temps $\Theta(\lg\ n)$ et de coût optimal $\Theta(n)$ pour la recherche de la sous-séquence de somme maximale.

7 Sauts de pointeurs sur des listes chaînées

La technique du dédoublement peut aussi être utilisée lorsqu'on manipule des structures de données dynamiques avec pointeurs. Dans ce cas, on parle alors d'algorithmes avec *sauts de pointeurs* (*pointer jumping*).

Soulignons que ces algorithmes ont initialement été découverts, dans les années 80, par des chercheurs travaillant sur des machines parallèles SIMD avec un très grand nombre de processeurs (Connection Machines), donc des machines reflétant assez bien certaines des caractéristiques du modèle PRAM (plus précisément, l'exécution *synchrone* des instructions sur des données différentes par un grand nombre de processeurs). Soulignons aussi que les concepteurs de ces algorithmes admettent avoir eux-mêmes été très "*étonnés*", initialement, de réaliser que de telles structures de données, qui semblaient intrinsèquement séquentielles, pouvaient conduire à des algorithmes de temps logarithmique.

7.1 Rang des éléments d'une liste chaînée

Le problème à traiter est le suivant. Soit une liste chaînée séquentiellement. On désire déterminer le rang relatif de chacun des éléments de la liste, c'est-à-dire la distance de chacun des noeuds par rapport au dernier noeud de la liste — le nombre de pointeurs à traverser avant d'atteindre le dernier noeud. Par exemple, soit la liste chaînée suivante, contenant les quatre noeuds indiqués avec les valeurs associées 40, 30, 20 et 10 :

```
Noeud(40, _) --> Noeud(30, _) --> Noeud(20, _) --> Noeud(10, _)
```

À la fin de l'exécution, on désire que le deuxième attribut de chaque noeud soit remplacé par le *rang* de l'item dans la liste, le rang étant interprété comme la distance du noeud par rapport au dernier élément de la liste :

```
Noeud(40, 3) --> Noeud(30, 2) --> Noeud(20, 1) --> Noeud(10, 0)
```

Un algorithme séquentiel pour ce problème serait évidemment de complexité linéaire $\Theta(n)$.

À première vue, il peut sembler impossible de faire mieux étant donné la nature intrinsèquement séquentielle des liens qui relient les noeuds de la liste. Toutefois, il est malgré tout possible d'obtenir un algorithme qui fonctionne en temps $\Theta(\lg\ n)$. Encore une fois, il s'agit ici d'appliquer une forme de dédoublement, mais adaptée aux pointeurs utilisés pour les structures de liste. Le "dédoublement" se fait alors en termes d'accès aux pointeurs, par une étape qu'on appelle "*saut de pointeurs*.

Ainsi, supposons que chaque noeud possède un champ auxiliaire `suivantSaut` utilisé pour effectuer de tels sauts de pointeurs — on pourrait utiliser directement le pointeur `suivant`, sans utiliser de pointeur auxiliaire, mais nous allons éviter de le faire simplement pour conserver intacte la structure initiale de la liste. L'étape cruciale du saut de pointeurs pour un noeud `pt` consistera alors à effectuer, de façon répétitive, la mise à jour de ce champ de la façon suivante :

```
if (pt^.suivantSaut != null) {
  ... Traitement approprié avant cette étape de saut ...
  # On effectue le saut de pointeur.
  pt^.suivantSaut = pt^.suivantSaut^.suivantSaut;
}
```

Les types utilisés pour les noeuds et pointeurs vers des noeuds dans notre algorithme seront les suivants :

```
type PtNoeud = ptr Noeud;
type Noeud = rec(
                int valeur;
                PtNoeud suivant;
                int rang;
                PtNoeud suivantSaut;
            );
```

Chaque noeud possède un champ `valeur` (utilisé simplement pour distinguer les noeuds, donc ne jouant aucun rôle dans l'algorithme), un champ `suivant` qui détermine l'ordre effectif (et séquentiel) des noeuds de la liste, et un champ `rang`, qui est celui qu'on cherche à définir par l'algorithme. Quant au champ `suivantSaut`, il s'agit du champ auxiliaire utilisé et modifié durant l'étape de saut de pointeurs (on ne veut pas modifier le champ `suivant`).

L'algorithme 12 présente le code MPD permettant de déterminer le rang des éléments d'une liste chaînée par saut de pointeurs.

Pour qu'un tel algorithme basé sur le saut de pointeurs avec structures chaînées dynamiques (listes, arbres, graphes) puissent fonctionner en parallèle, il faut pouvoir faire en sorte que tous les noeuds de la collection puissent être traités en parallèle, en utilisant du parallélisme de données. Dans l'algorithme 12, qui traite une liste de `n` noeuds, on suppose donc qu'on a `n` processeurs et que c'est le tableau `noeuds` qui indique le noeud devant être traité par le `i`ème processeur — le processeur `i` traite donc l'élément `noeuds[i]`.[4]

[4]Dans un *programme* parallèle, on tenterait évidemment d'optimiser le code en associant plusieurs noeuds à un processeur donné.

```
procedure initNoeud( PtNoeud pt )
{
  # Initialisation des rangs: initialement tous a 1, sauf le dernier de la liste.
  if (pt^.suivant == null) {
    pt^.rang = 0;
  } else {
    pt^.rang = 1;
  }
  # Initialement, tous les suivants pour le saut pointent au vrai suivant.
  pt^.suivantSaut = pt^.suivant;
}

procedure sauterPointeur( PtNoeud pt )
{
  if (pt^.suivantSaut != null) {
     pt^.rang = pt^.rang + pt^.suivantSaut^.rang;
     pt^.suivantSaut = pt^.suivantSaut^.suivantSaut;
  }
}

procedure determinerRang( PtNoeud noeuds[*], int n )
# PRECONDITION
#   NUMBER( 1 <= i <= n & noeuds[i] == null :: i ) = 1
# POSTCONDITION
#   ALL( 1 <= i <= n :: noeuds[i]^.rang = nombre de pointeurs a
#                                         traverser pour arriver au dernier noeud )
#   ALL( 1 <= i <= n :: noeuds[i]^.suivantSaut = null )
{
 # On initialise les rangs.
 co [i = 1 to n]
    initNoeud( noeuds[i] );
 oc

 # On effectue les sauts de pointeurs.
 for [j = 1 to lg(n)] {
   co [i = 1 to n]
     sauterPointeur( noeuds[i] )
   oc
 }
}

#
# Programme principal.
PtNoeud noeuds[n];
PtNoeud tete = generer( noeuds, n );

determinerRang( noeuds, n );
```

Algorithme 12: Détermination du rang des éléments d'une liste chaînée par saut de pointeurs

Dans l'algorithme 12, cette association entre un élément de la liste chaînée — dont la tête de liste est donnée par le pointeur `tete` — et un processeur se fait par l'intermédiaire du tableau `noeuds` et est réalisée par la procédure `generer`. Après l'appel à `generer`, le tableau `noeuds` et la variable `tete` pourraient alors ressembler à ce qui est présenté à la figure 11 : l'élément `noeuds[i]`, traité par le `i`ème processeur, ne correspond donc pas nécessairement au $i$ème élément de la liste chaînée.

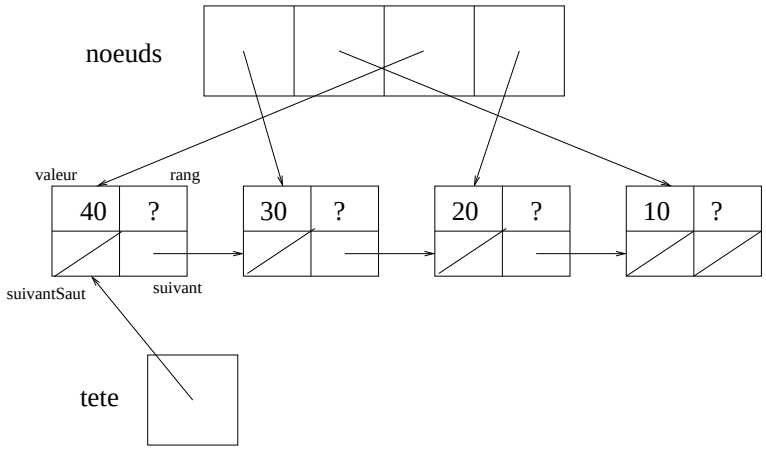


Figure 11: Une liste chaînée avec quatre éléments après la création de la liste avec la procédure `generer` et le tableau `noeuds` associé

Une fois la liste créée et le tableau `noeuds` défini, la procédure `determinerRang` de l'algorithme 12 procède alors comme suit :

- On initialise tout d'abord, en parallèle, le champ `rang` de chacun des noeuds. Initialement, le dernier noeud de la liste possède un rang nul, alors que tous les autres ont un rang égal à 1. On initialise aussi le champ `suivantSaut` pour qu'il point de façon directe à l'élément suivant de la liste.

  La liste de la Figure 11 *après* l'initialisation par `initNoeud` estprésentée à la figure 12.

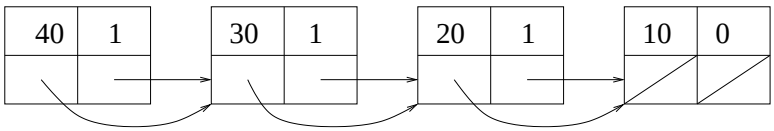


Figure 12: Une liste chaînée avec quatre éléments après l'initialisation des divers champs

- On effectue ensuite un nombre logarithmique d'étapes de saut de pointeur (`sauterPointeur`). Au cours de chacune de ces étapes, on effectue une mise à jour du champ `suivantSaut` par le biais d'un saut de pointeur, de même que la mise à jour du champ `rang` : la rang d'un noeud est son rang courant additionné du rang de son successeur défini par le saut de pointeur. L'état de la liste après l'exécution est présenté à la Figure 13.

  Notons que cet algorithme fonctionne correctement peu importe la valeur de `n`, c'est-à-dire pas uniquement avec `n` une puissance de 2, et ce en autant que l'expression entière `lg(n)` soit inteprétée comme $\lceil \texttt{lg(n)} \rceil$ — c'est-à-dire arrondissement vers le haut, par exemple, `lg(5)` = 3, `lg(9)` = 4. C'est la condition `pt^.suivantSaut != null` dans le `if` de la procédure `sauterPointeur` qui assure que les processeurs qui n'ont plus de

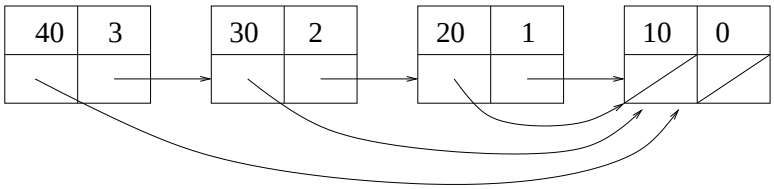


Figure 13: La liste chaînée de la Figure 12 après le calcul du rang via le saut de pointeurs

suivant resteront simplement inactifs, c'est-à-dire ne tenteront plus de modifier `rang` ou `suivantSaut`.

Les caractéristiques de cet algorithme sont alors les suivantes, en supposant que chaque noeud de la liste est traité *par un processeur distinct* (parallélisme de données) :

- Nombre de processeurs : n.
- Temps : $\Theta(\lg\, n)$.
- Coût : $\Theta(n \lg\, n)$.

L'algorithme n'est donc pas de coût optimal. Bien qu'il soit possible d'obtenir un algorithme optimal [KKT01], la solution est relativement complexe puisqu'associer un nombre logarithmique d'éléments à chaque processeur dans le cas d'une liste chaînée n'est pas triviale, les éléments n'étant pas contigus.

7.2 Mais ... il y a un problème :(

Malheureusement, l'algorithme 12 interprété dans le contexte strict du langage MPD, n'est pas correct, encore une fois parce que le modèle d'exécution MPD *n'est pas un modèle synchrone* d'exécution.

Ainsi, soit deux processus P_1 et P_2 qui exécutent tous deux la séquence d'instructions $s_1; s_2$. Dans le modèle d'exécution MPD, il est possible (en théorie du moins) que l'exécution d'instructions provenant de ces deux processus soit entrelacée (*interleaved*). En d'autres mots, il est possible que P_1 exécute sa première instruction s_1, puis que l'autre processus exécute sa propre première instruction suivie immédiatement de sa seconde instructon s_2, et ce avant même que P_1 n'exécute sa deuxième instruction s_2.

Dans le cas de l'algorithme 12, le modèle MPD d'exécution permettant un tel entrelacement peut alors conduire à un résultat incorrect. (Exercice!)

Alors que le modèle d'exécution PRAM est un modèle *synchrone*, de style *SIMD* (*Single Instruction, Multiple Data*), le modèle MPD est plutôt de style *SPMD* (*Single Program, Multiple Data*). Dans le modèle SPMD de programmation parallèle, les divers processus/processeurs exécutent tous *un même programme*, mais pas nécessairement la même instruction à un instant donné. Lorsque cela est nécessaire pour assurer le bon fonctionnement d'un programme SPMD, les différents processus peuvent évidemment se synchroniser entre eux, généralement à l'aide d'une instruction d'attente appelée une "*barrière de synchronisation*".

7.3 Barrières de synchronisation de style SPMD en MPD

L'algorithme 13 présente une `resource` MPD définissant une telle barrière de synchronisation entre processus (adaptée de [AO93]). Le nombre de processus impliqués dans la synchronisation est spécifié au moment de la création de la barrière (argument `nbProcs`). Une telle barrière est créée à l'aide de l'opération `create`, telle qu'illustré dans le segment de code

```
resource Barriere
  # Operation pour mise en attente a la barriere.
  op attendre()

  # Le nombre de processus impliques dans l'attente a la barriere est
  # specifie au moment de la creation de la ressource (parametre nbProcs).

body Barriere( int nbProcs )
  int nbArrives        = 0;
  int semAttenteActif  = 0;

  sem semMutex        = 1;
  sem semAttente[0:1] = ([2] 0);

  procedure prochainSemAttente( int n1 ) returns int r
  # Permet d'alterner entre semAttente[0] et semAttente[1].
  { r = 1 - n1; }

  proc attendre()
  {
    P(semMutex);
    nbArrives += 1;

    if (nbArrives < nbProcs) {
      # Il reste encore d'autres processus qui ne sont pas arrives
      #   => Mise en attente sur le semaphore semAttente actif.

      int actif = semAttenteActif;
      V(semMutex);
      P(semAttente[actif]);

    } else {
      # Tous les processus sont maintenant arrives :
      #   => on les reactive.

      nbArrives = 0;
      for [i = 1 to nbProcs-1] {
        V(semAttente[semAttenteActif]);
      }
      semAttenteActif = prochainSemAttente(semAttenteActif);
      V(semMutex)
    }
  }
end
```

Algorithme 13: Ressource MPD définissant une barrière pour synchronisation entre `nbProcs` processus

MPD qui suit (ici, `N` spécifie le nombre de processus qui seront impliqués dans l'utilisation de la barrière `b`) :

```
import Barriere;
...
cap Barriere b = create Barriere(N);
...
```

L'utilisation des deux sémaphores `semAttente` est nécessaire pour assurer que des processus qui réutilisent la barrière ne recevront pas des signaux qui proviennent d'une utilisation précédente de la barrière.[5]

```
resource TesBarriere
  import Barriere;

  const int NBPROCS = 4;
body TesBarriere()
  cap Barriere b = create Barriere(NBPROCS);

  process p[i = 1 to NBPROCS] {
    writes( "p[", i, "]:: Debut\n" );
    b.attendre();
    writes( "p[", i, "]:: Apres barriere #1, avant barriere #2\n" );
    b.attendre();
    writes( "p[", i, "]:: Apres barriere #2\n" );
  }

end
```

Algorithme 14: Exemple d'utilisation d'une barrière pour synchronisation entre processus

Un exemple simple d'utilisation d'une telle barrière est donnée à l'algorithme 14. Un résultat possible d'exécution de ce programme serait alors le suivant (l'ordre des impressions entre chacune des barrières pourrait varier) :

```
p[1]:: Debut
p[2]:: Debut
p[3]:: Debut
p[4]:: Debut
p[4]:: Apres barriere #1, avant barriere #2
p[1]:: Apres barriere #1, avant barriere #2
p[2]:: Apres barriere #1, avant barriere #2
p[3]:: Apres barriere #1, avant barriere #2
p[3]:: Apres barriere #2
p[4]:: Apres barriere #2
p[1]:: Apres barriere #2
p[2]:: Apres barriere #2
```

L'algorithme 12 pourrait donc être modifié pour utiliser une telle barrière et toujours produire un résultat correct, peu importe l'ordre dans lequel s'exécutent les divers processus. (Exercice!)

[5] Une telle situation pourrait survenir si un changement de contexte se produisait juste après que le $n-1^{ième}$ aurait relaché le sémaphore `semMutex`, mais *avant* d'avoir accédé au sémaphore `semAttente[semAttenteActif]`. Si le $n^{ième}$ processus arrive alors à la barrière puis envoie les $n-1$ signaux au sémaphore d'attente, il faut être certain que ce seront les processus bloqués sur l'utilisation précédente de la barrière qui sont réactivés.

7.4 Calcul parallèle des postfixes sur une liste chaînée

Le calcul du rang des éléments d'une liste chaînée par sauts de pointeurs, tel qu'effectué dans l'algorithme 12, peut être effectué par l'intermédiaire d'un calcul parallèle de postfixes sur une telle liste chaînée. C'est ce que nous allons tenter de montrer dans la section qui suit.

Rappelons que pour une séquence $x = [x_1, \ldots, x_n]$ et une opération binaire associative $\otimes$, un calcul des postfixes sur X avec $\otimes$ consiste à produire la séquence X' ayant la propriété suivante :

$$X' = [x_1 \otimes x_2 \otimes \ldots \otimes x_n, x_2 \otimes x_3 \otimes \ldots \otimes x_n, \ldots, x_{n-1} \otimes x_n, x_n]$$

En d'autres mots, on produit X' tel que pour $1 \leq i \leq n$, $x'_i = x_i \otimes x_{i+1} \otimes \ldots \otimes x_{n-1} \otimes x_n$.

Par exemple, soit une séquence de longueur 4 définie par les valeurs $[1, 1, 1, 0]$. Un calcul des postfixes avec l'opérateur + sur cette séquence produira alors la séquence suivante :

$$[1 + 1 + 1 + 0, 1 + 1 + 0, 1 + 0, 0] = [3, 2, 1, 0]$$

Comme on peut le constater en comparant avec les figures 12 et 13, cette séquence correspond bien au calcul des rangs des éléments, tel que vu précédemment.

L'algorithme 15 effectue un tel calcul du rang des éléments d'une liste chaînée effectué par un calcul parallèle des postfixes (procédure `calculerPostfixes` appelée par `determinerRang`).

Notons tout d'abord que l'algorithme présenté *n'est pas du MPD valide* — disons qu'il s'agit d'un algorithme en *pseudo-MPD* :

- Des restrictions syntaxiques en MPD font que, dans une instruction `co`, seules des invocations de fonction ou de procédures peuvent apparaître, et non pas du code arbitraire comme c'est le cas dans l'algorithme 15.[6] Lorsque nous utiliserons de tels segments de code arbitraires à l'intérieur d'un `co`, nous l'indiquerons en utilisant le mot-clé `CO` plutôt que `co`.
- Une instruction `co`, comme nous l'avons expliqué précédemment, ne conduit pas à une exécution synchrone, dans le style PRAM, des diverses activations du `co`. Pour obtenir un résultat correct équivalent à celui du modèle PRAM, il faut donc parfois, comme on l'a vu précédemment, introduire des barrières, ce qui alourdit l'algorithme. Dans l'algorithme 15 ainsi que dans d'autres algorithmes que nous présenterons, nous utiliserons plutôt, lorsque nécessaire, une pseudo-instruction d'exécution parallèle *synchrone à la PRAM* : `CO-SYNC`.[7]

Finalement, notons que plutôt que de créer une copie de la séquence sur laquelle le calcul des postfixes s'effectue, nous avons supposé que les noeuds de la liste chaînée contiennent deux champs spéciaux utilisés par le calcul des postfixes, faisant en sorte que les champs de base de la liste chaînée (`valeur` et `suivant`) ne sont pas modifiés. Ces deux champs sont les suivants :

- `suivantSaut` : champ utilisé pour les sauts de pointeur, pour éviter de modifier le pointeur `suivant` qui définit la structure initiale de la liste.
- `tmp` : champ utilisé pour conserver les résultats du calcul parallèle des postfixes. À la fin de l'exécution, le champ `tmp` d'un noeud `pt` correspondant au $j$ème élément de la séquence contiendra alors la valeur $x_j \otimes x_{j+1} \otimes \ldots \otimes x_{n-1} \otimes x_n$. Rappelons qu'il n'y pas nécessairement de lien entre l'index `i` du tableau `noeuds` et la position j de l'élément

[6] En fait, le terme `co` signifie *co-invocation*. Donc, si jamais le compilateur MPD vous indique un message d'erreur du style "`"foo.mpd", line 152: fatal: invalid co-invocation`", c'est que vous avez inclus dans un `co` autre chose qu'une invocation de fonction ou procédure.

[7] Notons que ces algorithmes en pseudo-MPD ont été dérivés, à la main, de véritables programmes MPD fonctionnant correctement. La transformation a consisté à remplacer les instructions `co` par des instructions `CO-SYNC`, à inclure textuellement (*inlining*) le code des procédures dans le `CO/CO-SYNC` et à modifier le code, par substitution textuelle, pour éliminer les barrières et les copies locales.

```
type PtNoeud = ptr Noeud;
type Noeud = rec(
                  int valeur;
                  PtNoeud suivant;
                  PtNoeud suivantSaut;
                  int tmp;
                );

optype OpBinaire = (int, int) returns int;

procedure calculerPostfixes( PtNoeud noeuds[*], int n, cap OpBinaire op0  )
{
 # On initialise les pointeurs de saut.
 CO [i = 1 to n]
    noeuds[i]^.suivantSaut = noeuds[i]^.suivant;
 OC

 # On effectue les sauts de pointeurs en appliquant l'operateur op0.
 for [i = 1 to lg(n)] {
   CO-SYNC [i = 1 to n]
     if (noeuds[i]^.suivantSaut != null) {
       noeuds[i]^.tmp = op0( noeuds[i]^.tmp, noeuds[i]^.suivantSaut^.tmp );
       noeuds[i]^.suivantSaut = noeuds[i]^.suivantSaut^.suivantSaut;
     }
   OC
 }
}

procedure plus( int x, int y ) returns int z
{ z = x + y; }

procedure determinerRang( PtNoeud noeuds[*], int n )
{
  CO [i = 1 to n]
    if (noeuds[i]^.suivant == null) {
      noeuds[i]^.tmp = 0;
    } else {
      noeuds[i]^.tmp = 1;
    }
  OC
  calculerPostfixes( noeuds, n, plus );
}
```

Algorithme 15: Détermination du rang des éléments d'une liste chaînée par calcul parallèle des postfixes (pseudo-MPD)

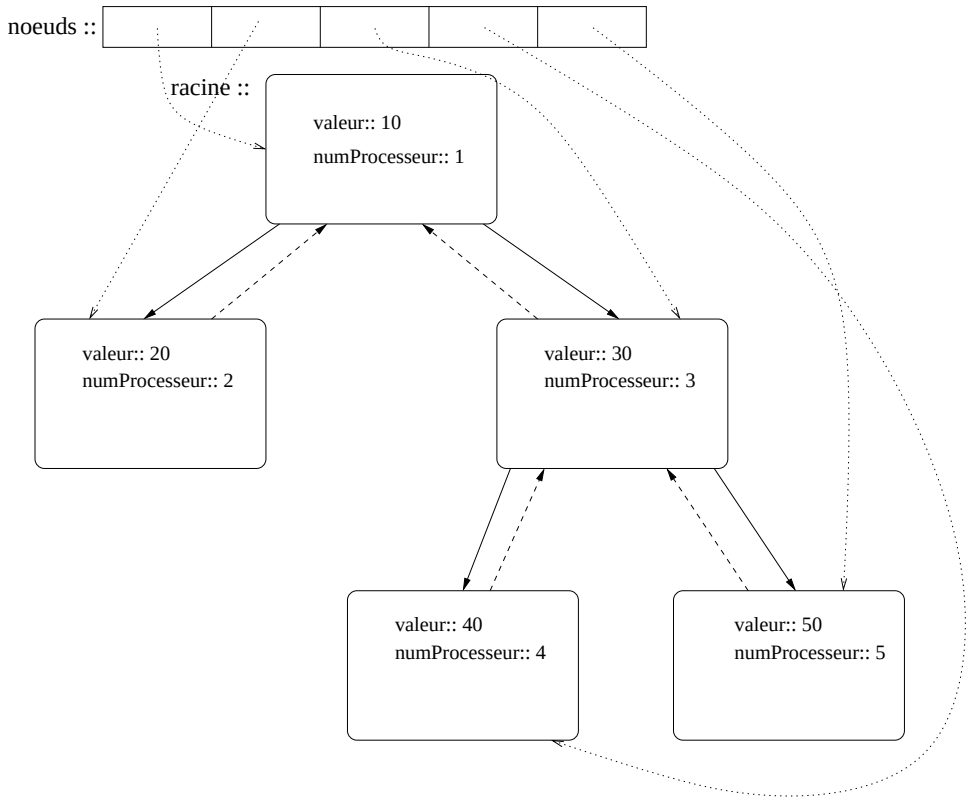


Figure 14: Arbre binaire `racine` avec ses cinq (5) noeuds

dans la séquence : lorsque le pointeur vers le $j$ème élément de la séquence est conservé à la position `i` du tableau `noeuds`, c'est pour modéliser le fait que ce $j$ème élément est traité par le processeur no. `i`.

Évidemment, une fois définie une procédure générale de calcul parallèle des postfixes sur des listes chaînées, il devient dès lors possible d'effectuer, en temps *logarithmique*, diverses opérations sur de telles listes, par exemple, recherche du maximum, somme des éléments de la liste, etc. (Exercice?)

8 Sauts de pointeurs et arbres binaires

8.1 Le problème : calcul de la profondeur des noeuds d'un arbre

Soit un arbre binaire tel que celui présenté à la Figure 14, dénoté par `racine`. Chaque noeud de l'arbre possède des pointeurs vers ses sous-arbres (enfants) `gauche` et `droit` (`null` si absent) ainsi qu'un pointeur vers son `parent` (`null` pour la racine). Un champ `valeur` est aussi présent, mais n'est pas vraiment utilisé dans notre exemple et ne sert qu'à mieux distinguer les divers noeuds.

Comme dans l'exemple du calcul du rang des éléments d'une liste chaînée, on va développer un algorithme où les noeuds de l'arbre seront traités par des processeurs indépendants (parallélisme de données). On suppose donc que l'arbre est accessible par l'intermédiaire de sa `racine` ainsi que par l'intermédiaire d'un tableau `noeuds` contenant des pointeurs vers les divers noeuds de l'arbre. On suppose aussi que chaque noeud de l'arbre possède un champ `numProcesseur` indiquant le numéro du processeur traitant ce noeud, c'est-à-dire, l'index `i`

tel que `noeuds[i]` réfère au noeud concerné. Les types et variables pour représenter des arbres de ce type sont donc les suivants :

```
# Types pour les noeuds de l'arbre binaire (allocation dynamique).
type PtNoeud = ptr Noeud;
type Noeud = rec(
                  int valeur;               # Valeur associee au noeud.
                  PtNoeud gauche, droit;    # Enfants gauche et droit.
                  PtNoeud parent;           # Parent (null pour la racine).
                  int numProcesseur;        # Numero du processeur.
                  int profondeur;           # Pour algorithme a venir.
             );

# Nombre total de noeuds de l'arbre.
int n;

# Tableau contenant les divers noeuds de l'arbre.
PtNoeud noeuds[n];
   # INVARIANT
   #   ALL( i: nat SUCH THAT 1 <= i <= n :: noeuds[i]^.numProcesseur == i )
```

Le problème que l'on désire résoudre est de déterminer, pour chacun des noeuds, sa profondeur (son niveau) dans l'arbre — étant entendu que la racine est de profondeur 0 — et d'assigner la valeur ainsi calculée au champ `profondeur` du noeud. Pour un arbre comptant n noeuds, un algorithme séquentiel pour résoudre ce problème serait évidemment $\Theta(n)$, puisque chaque noeud doit être visité. Nous allons présenter un algorithme qui, peu importe la structure de l'arbre, donc peu importe sa profondeur, fonctionnera en temps $\Theta(\lg\, n)$.

La technique utilisée, qui peut aussi être employée pour résoudre rapidement (en temps logarithmique) de nombreux autres problèmes sur des arbres (binaires ou non), est celle du *circuit eulérien*.

8.2 La technique du circuit eulérien

Un *circuit eulérien* sur un graphe orienté est un circuit (un chemin qui forme un *cycle*) qui traverse exactement chaque arc une fois, mais en visitant possiblement un sommet plusieurs fois. Étant donné un arbre et sa racine tel que celui présenté à la Figure 14, il est toujours possible de construire un circuit eulérien à partir de la racine en utilisant, à l'aller, les arcs liant un parent à ses enfants (pointeur `gauche` ou `droit`), et au retour l'arc liant l'enfant à son parent. Par exemple, un circuit eulérien possible pour l'arbre de la Figure 14 serait le suivant :

- (`noeuds[1]`) On traverse l'arc reliant la racine à son premier enfant (pointeur `gauche`).
- (`noeuds[2]`) On traverse l'arc retournant à la racine (pointeur `parent`).
- (`noeuds[1]`) On traverse l'arc reliant la racine à son deuxième enfant (pointeur `droit`).
- (`noeuds[3]`) On traverse l'arc vers le premier enfant (pointeur `gauche`).
- (`noeuds[4]`) On retourne au parent.
- (`noeuds[3]`) On traverse l'arc vers le deuxième enfant (pointeur `droit`).
- (`noeuds[5]`) On retourne au parent.
- (`noeuds[3]`) On retourne à la racine.

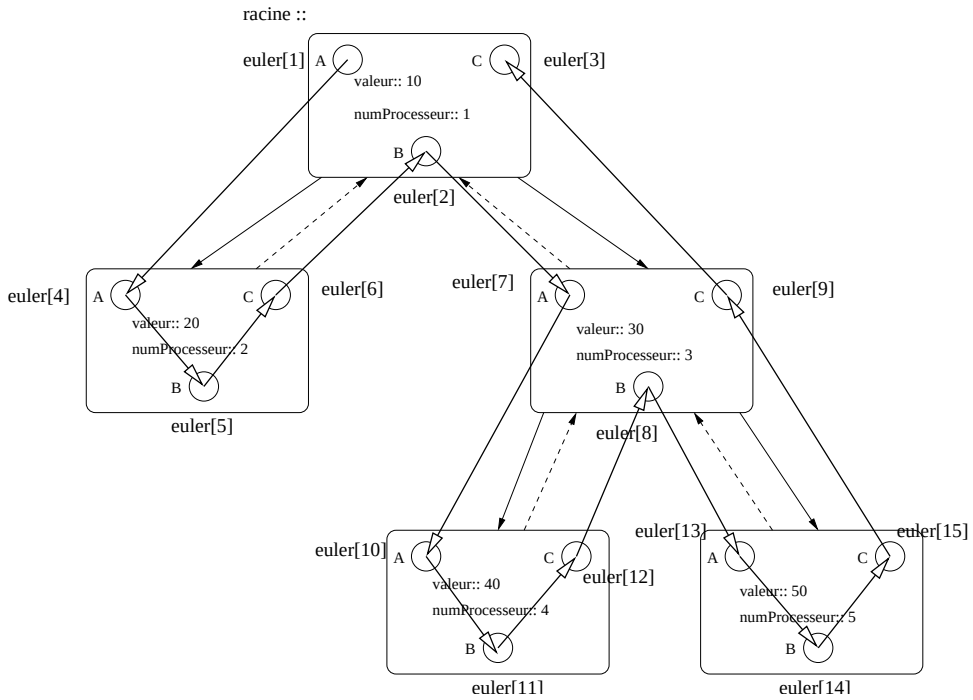


Figure 15: Arbre binaire `racine` et son circuit eulérien

Une façon intéressante d'associer un circuit eulérien à un arbre est illustrée à la Figure 15 (adaptée de [CLR94, Figure 30.4]). Comme on va vouloir effectuer un parcours parallèle (de style parallélisme de données) du circuit eulérien, on va associer plusieurs noeuds du circuit eulérien, c'est-à-dire plusieurs processeurs, à chacun des noeuds de l'arbre. Plus précisément, on aura qu'à chaque noeud de l'arbre seront associés trois (3) noeuds distincts utilisés définir le circuit eulérien approprié, le champ `suivant` de ces noeuds étant défini comme suit :

`A` : Réfère au noeud `A` de l'enfant gauche, s'il existe, sinon réfère à son propre noeud `B`.

`B` : Réfère au noeud `A` de l'enfant droit, s'il existe, sinon réfère à son propre noeud `C`.

`C` : Réfère au noeud `B` ou `C` de son parent, selon qu'il s'agit d'un enfant gauche ou droit, à l'exception de la racine pour lequel `C` est `NULL`.

Une façon d'associer ainsi trois (3) noeuds de circuit eulérien, et donc trois (3) processeurs en présence de parallélisme de données, à chacun des noeuds de l'arbre est donnée par les déclarations présentées dans l'extrait de code MPD 1.

- Chaque noeud du circuit eulérien est représenté par un enregistrement de type `NoeudCircuitEulerien`. Notons que contrairement aux noeuds de l'arbre, il s'agit ici de noeuds alloués de façon *statique* ; les "pointeurs" entre les noeuds du circuit sont donc représentés simplement par des `index` (entiers) dans le tableau, un pointeur `NULL` étant indiqué par un index invalide (`-1`). C'est par l'intermédiaire des champs `suivant` de ces noeuds que sera constitué le circuit eulérien. Les champs `suivantSaut` et `tmp` seront utilisés ultérieurement pour effectuer des sauts de pointeur, comme pour les listes chaînées dynamiques de la section 7.

- Les `n` noeuds de l'arbre sont conservés dans un tableau `noeuds`. Étant donné un index `i` dans ce tableau, on veut pouvoir retrouver de façon directe les trois (3) noeuds du circuit eulérien associé. Il suffit pour ce faire d'utiliser un tableau (`euler`) de taille `N = 3*n`

```
# Constante et types pour les noeuds qui representent le circuit eulerien.
# (allocation statique)
type Index = int;
const Index NULL = -1;
type NoeudCircuitEulerien = rec (
      Index suivant;            # Le suivant dans le circuit initial.
      Index suivantSaut;        # Le suivant durant les sauts de pointeurs.
      int tmp;                  # Valeur temporaire (modifiee durant les
                                  sauts de pointeur).
      );

# Variable pour les divers noeuds du circuit eulerien.
const int N = 3*n;
NoeudCircuitEulerien euler[N];

# Fonctions indiquant le noeud A, B ou C associe a un noeud de l'arbre
# (plus precisement, au noeud traite par le processeur i).
procedure A( int i ) returns int r { r = 3*i-2; }
procedure B( int i ) returns int r { r = 3*i-1; }
procedure C( int i ) returns int r { r = 3*i; }
```

Extrait de code MPD 1: Types, variables et fonctions auxiliaires pour les circuits eulériens sur des arbres binaires

tel que les trois noeuds pour l'élément `noeuds[i]` sont conservés aux positions `3*i-2`, `3*i-1` et `3*i` (puisque l'indice de base est `1`) du tableau `euler` — c'est le rôle des fonctions `A`, `B` et `C` de spécifier ces associations.

Dans le circuit eulérien pour l'arbre `racine` présenté à la Figure 15, les divers noeuds `A`, `B` et `C` du circuit ont été annotés (à l'extérieur de la boîte) par le noeud du tableau `euler` associé (par exemple, `euler[1]` pour l'élément `A` de la racine, `euler[2]` pour son élément `B`, etc.). Quant à la Figure 16, elle présente le contenu du tableau `euler` (en ignorant les champs `suivantSaut` et `tmp`) correspondant à ce même circuit eulérien.

À l'aide de ces définitions, l'algorithme 16 (p. 40) permet alors de construire un circuit eulérien sur un arbre dont les divers noeuds sont spécifiés par un tableau `noeuds` (il n'est même pas nécessaire de connaître explicitement la racine) : les fonctions `suivantPourA`, `suivantPourB` et `suivantPourC` correspondent aux règles décrites précédemment pour la construction du circuit eulérien. Le corps de la fonction `construireCircuitEulerien` consiste alors simplement à définir le champ `suivant` de chacun des noeuds du circuit, ce qui peut se faire de façon complètement parallèle. La construction du circuit eulérien d'un arbre binaire peut donc s'effectuer ... en temps constant (temps $\Theta(1)$), et ce avec un nombre linéaire de processeurs ($\Theta(n)$).

8.3 Calcul des profondeurs des noeuds avec circuit eulérien et calcul parallèle des préfixes

L'algorithme 17 (p. 41) décrit comment effectuer le calcul de la profondeur des différents noeuds d'un arbre à l'aide de la construction d'un circuit eulérien puis en parcourant ce circuit via un calcul des préfixes. L'algorithme commence par initialiser le champ `tmp` de chacun des noeuds à une valeur spécifique à chacun des types de noeuds (`A`, `B` ou `C`).[8] L'état de l'arbre et de son circuit après cette initialisation est décrit à la Figure 17

[8] On remarquera que deux pseudo-instructions `CO` ont été utilisées. Dans les deux cas, le programme MPD correspondant utilisait un `co` avec un appel à une fonction identité (`copier`).

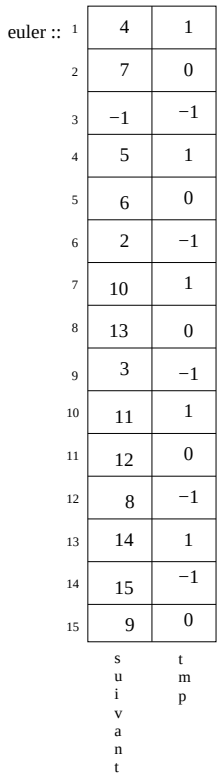


Figure 16: Arbre binaire `racine` et son circuit eulérien représenté par le tableau `euler`

```
# Fonctions pour determiner le processeur du circuit eulerien associe
# au processeur qui traite un noeud i de l'arbre.

# Les fonctions pour determiner le suivant du circuit eulerien,
# selon le type du processeur (A, B ou C).

procedure suivantPourA( ref PtNoeud noeuds[*], int i ) returns Index suivant
{
  if ( noeuds[i]^.gauche != null ) {
     suivant = A(noeuds[i]^.gauche^.numProcesseur);
   } else {
     suivant = B(i);
  }
}

procedure suivantPourB( ref PtNoeud noeuds[*], int i ) returns Index suivant
{
  if ( noeuds[i]^.droit != null ) {
     suivant = A(noeuds[i]^.droit^.numProcesseur);
   } else {
     suivant = C(i);
  }
}

procedure suivantPourC( ref PtNoeud noeuds[*], int i ) returns Index suivant
{
  if (noeuds[i]^.parent == null) {
    suivant = NULL;
  } else if ( noeuds[i]^.parent^.gauche == noeuds[i] ) {
    suivant = B(noeuds[i]^.parent^.numProcesseur);
  } else {
    suivant = C(noeuds[i]^.parent^.numProcesseur);
  }
}

# La procedure de construction d'un circuit eulerien pour un arbre.
# ENTREE
#   noeuds : les divers noeuds pour l'arbre
#   n      : le nombre de noeuds de l'arbre
# SORTIE
#   euler  : la liste chainee pour les divers noeuds (A, B et C)
procedure construireCircuitEulerien(  ref PtNoeud noeuds[*],
                                      ref NoeudCircuitEulerien euler[*],
                                      int n )
{
  co [i = 1 to n]  euler[A(i)].suivant = suivantPourA(noeuds, i);
  // [j = 1 to n]  euler[B(j)].suivant = suivantPourB(noeuds, j);
  // [k = 1 to n]  euler[C(k)].suivant = suivantPourC(noeuds, k);
  oc
}
```

Algorithme 16: Algorithme de construction d'un circuit eulérien sur un arbre binaire

```
procedure calculerProfondeurs( ref PtNoeud noeuds[*], int n )
{
  const int N = 3*n;
  NoeudCircuitEulerien euler[N];

  # On commence par construire le circuit eulerien.
  construireCircuitEulerien( noeuds, euler, n );

  # On initialise le champ tmp des divers processeurs, selon le type (A, B ou C).
  CO [i = 1 to n]  euler[A(i)].tmp =  1;
  // [j = 1 to n]  euler[B(j)].tmp =  0;
  // [k = 1 to n]  euler[C(k)].tmp = -1;
  OC

  # On effectue un calcul des prefixes sur le circuit eulerien avec l'addition.
  calculerPrefixes( euler, N, plus );

  # Le resultat pour chaque noeud est dans le processeur C qui lui est associe.
  CO [i = 1 to n]
    noeuds[i]^.profondeur = euler[C(i)].tmp;
  OC
}
```

Algorithme 17: Algorithme pour calculer des profondeurs des noeuds dans un arbre binaire via parcours d'un circuit eulérien

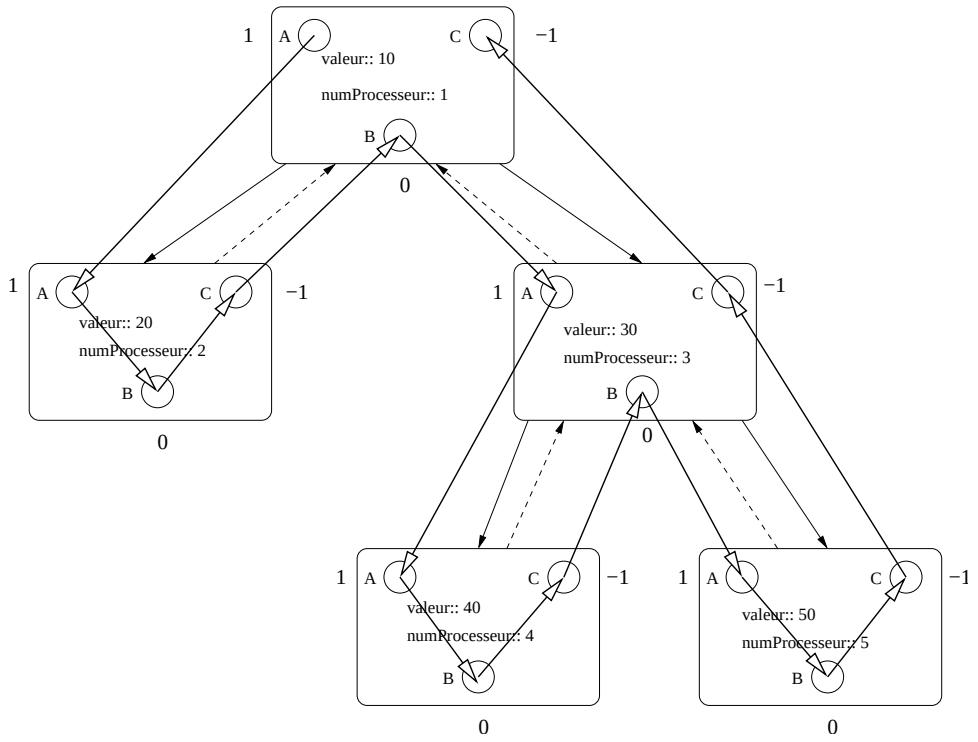


Figure 17: Arbre binaire `racine` et son circuit eulérien après initialisation pour le calcul de la profondeur

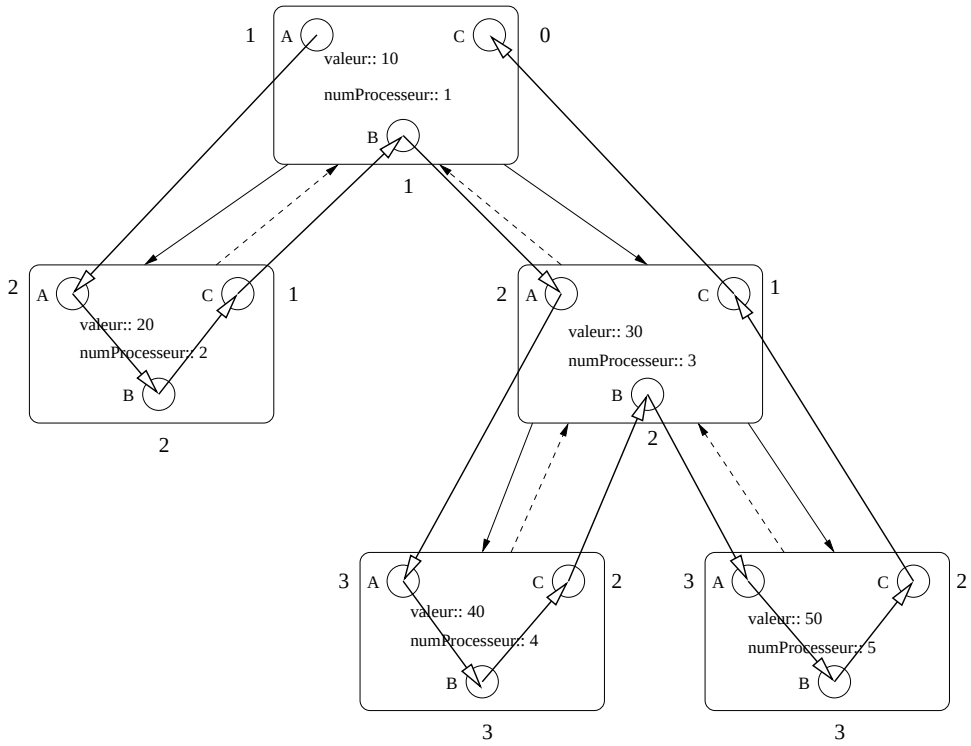


Figure 18: Arbre binaire `racine` et son circuit eulérien après parcours pour le calcul de la profondeur

Un circuit eulérien étant en essence une séquence, il est donc possible, comme on l'a vu précédemment pour les listes chaînées (algorithme 15), d'effectuer un calcul parallèle des préfixes sur cette séquence à l'aide de la technique des sauts de pointeurs.[9] L'étape suivante pour le calcul des profondeurs consiste alors simplement à effectuer un tel calcul parallèle des préfixes sur la séquence composant le circuit eulérien. L'état après ce calcul des préfixes est alors celui présenté à la Figure 18. Comme on peut alors le constater — à expliquer (exercice?) — la valeur associée au noeud `C` de n'importe quel noeud de l'arbre représente alors sa profondeur. L'algorithme 17 se termine donc en assignant cette valeur au champ `profondeur` du noeud de l'arbre pour obtenir l'effet désiré.

Quant à la complexité de cet algorithme, il est facile de voir qu'il s'agit, à cause du calcul parallèle des préfixes, d'un algorithme fonctionnant en temps *logarithmique*, donc $\Theta(\lg n)$.

9 Machines PRAM EREW vs. CRCW : la recherche du minimum parmi une série de n éléments

On a vu dans une section précédente qu'on pouvait, à l'aide d'une machine PRAM, trouver le minimum parmi une série de n éléments en temps $\Theta(\lg n)$. Cet algorithme fonctionnait en émulant le comportement ascendant d'une approche récursive et faisait en sorte qu'aucune écriture concurrente ne s'effectuait dans une même case mémoire. Il en était de même pour les lectures. L'algorithme présenté était donc approprié pour une machine PRAM de type

[9] En fait, dans l'algorithme 15, nous avons plutôt effectuer un calcul parallèle des postfixes. Toutefois, le principe est exactement le même pour le calcul parallèles des préfixes, ne demandant que quelques modifications mineures du code.

EREW.

Dans ce qui suit, nous allons montrer à l'aide d'un exemple simple que l'utilisation d'une machine PRAM CRCW peut conduire, pour certains problèmes, à un algorithme *plus rapide*.

```
procedure copier( int x ) returns int r
{ r = x; }

procedure comparerEtEcrire( int Xi, int Xj, ref int Ti, ref int Tj )
{ if (Xi < Xj) { Tj = 0; } else { Ti = 0; } }

procedure identifierMin( int Ti, int Xi, res int leMin )
{ if (Ti == 1) { leMin = Xi; } }

procedure trouverMin( int X[*], int n ) returns int leMin
{
  int T[n]; # Tableau pour les resultats des comparaisons.

  # On initialise chacune des entrees de T a 1 (gagnant jusqu'a
  # preuve du contraire).
  co [i = 1 to n]
    T[i] = copier( 1 );
  oc

  # En parallele, on effectue toutes les comparaisons possibles,
  # en mettant a 0 une entree de T des qu'elle perd une comparaison,
  # c'est-a-dire, lorsqu'elle est plus grande qu'une autre.
  co [i = 1 to n, j = 1 to n st i < j]
    comparerEtEcrire( X[i], X[j], T[i], T[j] );
  oc
  # A ce point, on aura que la seule entree encore a 1 sera
  # celle n'ayant "perdu" aucune comparaison, c'est-a-dire, celle
  # associee a l'element minimum du tableau.

  # Il reste maintenant a identifier cet element minimum.
  co [i = 1 to n]
    identifierMin( T[i], X[i], leMin );
  oc
}
```

Algorithme 18: Algorithme pour recherche du minimum, en temps *constant*, parmi une série de n éléments sur une machine PRAM CRCW égalitaire

L'algorithme 18 effectue la recherche du minimum dans un tableau `X` de n éléments. Pour simplifier l'algorithme et assurer qu'il fonctionne sur n'importe quelle machine PRAM CRCW, nous allons supposer que *tous* les éléments de `X` sont différents les uns des autres. Sous cette condition, l'algorithme fonctionnera correctement pour n'importe quelle type de machine PRAM CRCW — c'est-à-dire, peu importe la technique utilisée pour résoudre les conflits d'écriture (priorité, arbitraire, commun, combinaison) — car lorsque plusieurs processeurs doivent écrire de façon concurrente une valeur dans une case mémoire, il s'agira toujours de la même valeur (la valeur 0, indiquant que l'élément vient de perdre au niveau des comparaisons).

L'intuition derrière cet algorithme est la suivante : un tableau auxiliaire `T` est utilisé pour déterminer le résultat de toutes les comparaisons possibles entre les divers éléments. Initialement, $T[i] = 1$ pour tout i. En parallèle, on effectue ensuite *toutes les comparaisons possibles* entre les diverses paires d'éléments `X[`i`]` et `X[`j`]` (sans répétition, c'est-à-dire pour

$i < j$). Lorsqu'un élément `X[`i`]` est identifié comme étant supérieur à un élément `X[`j`]`, il "perd" alors la comparaison (puisqu'on cherche à identifier le minimum) et l'entrée de la matrice `T` correspondant à cet élément (i ou j, selon le cas) est mise à 0 (procédure `comparerEtEcrire`). Après ces diverses comparaisons, puisqu'on a supposé que tous les éléments étaient distincts, on est alors assuré qu'un seul élément de `T` est encore à 1, à savoir l'élément minimum qui n'aura "perdu" aucune des comparaisons. Il suffit alors d'identifier la position de cet élément pour déterminer le minimum (procédure `identifierMin`).

L'analyse de cet algorithme conduit au résultat, quelque peu surprenant, que la recherche du minimum parmi une série de n éléments distincts *peut se faire en temps constant*, c'est-à-dire en temps $\Theta(1)$! Évidemment, il y a toutefois un prix important à payer pour obtenir un tel résultat, à savoir que $n(n-1)/2$ processeurs doivent être utilisés, ce qui conduit donc à un algorithme dont le coût est loin d'être optimal — $\Theta(n^2)$ plutôt que $\Theta(n)$.

10 Simulation d'un algorithme PRAM CRCW par un algorithme PRAM EREW

On a vu dans l'exemple précédent qu'un algorithme PRAM CRCW peut être asymptotiquement plus rapide qu'un algorithme EREW pour un même problème. Il est clair aussi qu'un algorithme EREW est automatiquement valide pour le modèle CRCW. On peut donc en conclure que le modèle PRAM CRCW est *plus puissant* que le modèle EREW. Toutefois, on peut aussi montrer que pour n'importe quel problème et pour un nombre de processeurs donné (disons n), un algorithme CRCW ne sera plus rapide *qu'au plus* par un facteur polylogarithmique (plus précisément, jamais plus de $\Theta(\lg^k n)$ fois plus rapide, pour un $k \geq 1$) qu'un algorithme EREW équivalent.

Pour prouver cette propriété, il suffit de montrer que chaque étape d'un algorithme CRCW peut être *simulée* par un calcul EREW en un temps d'au plus $\Theta(\lg^k n)$, pour un $k \geq 1$. Nous allons montrer cette propriété dans le cas de l'écriture concurrente *égalitaire* (*common CW*), la propriété pour la lecture concurrente pouvant être prouvée par une approche similaire.

Pour effectuer la simulation des écritures concurrentes, on utilise un tableau auxiliaire A de taille n (= nombre de processeurs). Lorsqu'un processeur CRCW p_i désire effectuer une écriture d'une valeur v à l'adresse a, le processeur EREW p_i correspondant écrit tout d'abord le couple (a, v) à la position i du tableau A ($A[i]$ `<-` (a, v)). Puisque chaque processeur n'écrit qu'à une unique position de A, toutes les écritures se font clairement dans des positions différentes les unes des autres (EW).

L'étape suivante de la simulation consiste alors à trier les divers couples (a, v) de A en fonction de la première coordonnée du couple (c'est-à-dire selon l'adresse où l'écriture CRCW doit être effectuée), opération qui peut se faire en temps $\Theta(\lg^2 n)$.[10] Ce tri a alors pour effet que les toutes les valeurs à écrire au même endroit (identiques, puisqu'il s'agit du modèle CW égalitaire) se retrouvent à des positions contiguës du tableau.

Avant d'expliquer la dernière étape, illustrons les étapes décrites jusqu'à présent à l'aide d'un bref exemple. Supposons que les six écritures suivantes sont effectuées (par six processeurs), où $P_k : a_j \leftarrow v$ dénote l'écriture de la valeur v à l'adresse a_j par le processeur CW P_k :

[10] C'est-à-dire qu'il existe un algorithme, que nous ne verrons pas ici, qui permet d'effectuer un tri en temps $\Theta(\lg^2 n)$. En fait, il a aussi été montré que le tri en parallèle pouvait se faire en temps $\Theta(\lg n)$ avec $\Theta(n)$ processeurs. Malheureusement, l'algorithme résultant est nettement plus complexe. Pour plus de détails, voir la description de l'algorithme de Cole pour une machine PRAM CREW dans [LR03].

Écritures CRCW :

- $P_1 : a_3 \leftarrow 15$;
- $P_2 : a_2 \leftarrow 10$;
- $P_3 : a_1 \leftarrow 20$;
- $P_4 : a_2 \leftarrow 10$;
- $P_5 : a_3 \leftarrow 15$;
- $P_6 : a_2 \leftarrow 10$;

Après la première étape :

- $A[1] = (a_3, 15)$;
- $A[2] = (a_2, 10)$;
- $A[3] = (a_1, 20)$;
- $A[4] = (a_2, 10)$;
- $A[5] = (a_3, 15)$;
- $A[6] = (a_2, 10)$;

Après le tri :

- $A[1] = (a_1, 20)$;
- $A[2] = (a_2, 10)$;
- $A[3] = (a_2, 10)$;
- $A[4] = (a_2, 10)$;
- $A[5] = (a_3, 15)$;
- $A[6] = (a_3, 15)$;

La dernière phase de la simulation consiste finalement à écrire dans chacune des adresses la valeur appropriée. Il faut évidemment faire en sorte d'assurer qu'une seule écriture se fasse pour chaque adresse (machine EW). Pour ce faire, il suffit que l'écriture se fasse uniquement à partir *de la première occurrence* d'un couple ayant une adresse donnée dans le tableau A.

Cette opération peut être effectuée simplement en vérifiant, en parallèle pour chacune des entrées de A, si l'élément *précédent* de A réfère ou non à une adresse différente. En d'autres mots (en notation *pseudo*-MPD), il suffit d'exécuter de façon concurrente le fragment d'algorithme suivant (on suppose que `adresse()` retourne la première composante du couple, alors que `valeur()` retourne la seconde) :

```
co [i = 1 to n]
  if ( i == 1 | (adresse(A[i]) != adresse(A[i-1])) ) {
    ecrire valeur(A[i]) dans adresse(A[i])
  }
oc
```

On remarque que si un processeur P_i ($i > 1$) contient un couple tel que `adresse(A[i]) est égal à adresse(A[i-1])`, alors ce processeur ne fera aucune écriture, ce qui assure donc l'unicité des écritures pour une adresse donnée.

11 Simulation entre algorithmes PRAM CRCW

Soit un algorithme pour le modèle PRAM CRCW égalitaire. Intuitivement, il est facile de voir que cet algorithme fonctionnera correctement pour le modèle CRCW arbitraire : les valeurs écrites par l'algorithme CRCW égalitaire étant toutes égales entre elles, le choix d'une valeur quelconque par le modèle CRCW arbitraire produira donc toujours la même valeur.

De même, soit un algorithme pour le modèle PRAM CRCW arbitraire. Cet algorithme fonctionnera correctement pour le modèle CRCW avec priorité : le choix de la valeur produite par le processeur de plus grande priorité peut être vu simplement comme un choix arbitraire *biaisé* par le niveau de priorité. On peut donc en conclure que, d'une certaine façon, le modèle CRCW avec priorité est le plus puissant parmi les modèles CRCW égalitaire, arbitraire et avec priorité.

Si aucune limite n'est imposée au nombre de processeurs et à la quantité d'espace mémoire utilisée, on peut aussi montrer qu'un algorithme CRCW avec priorité peut être simulé sans perte asymptotique de vitesse (donc simulation d'un algorithme de temps $T(n)$ en temps $\Theta(1) \times T(n)$) par un algorithme pour les modèles CRCW égalitaire ou arbitraire. Toutefois, une telle simulation n'est possible que si on augmente le nombre de processeurs par un facteur logarithmique, entraînant de ce fait une augmentation du coût total.

Plus précisément, il est possible de montrer (mais nous ne le ferons pas ici) qu'une étape d'écriture d'un algorithme PRAM CRCW avec priorité pour n processeurs peut être simulée en temps $\Theta(1)$ en utilisant une machine PRAM CRCW égalitaire avec $\Theta(n \lg n)$ processeurs.

Ce résultat s'obtient en montrant que le problème du "*prisonnier le plus à gauche*"[11] pour n éléments peut être résolu en temps $\Theta(1)$ par un algorithme CRCW égalitaire si chacun des n processeurs utilise lg n processeurs auxiliaires.

Toujours pour les modèles CRCW égalitaire et avec priorité, il est aussi possible de montrer qu'une étape d'écriture d'un algorithme CRCW avec priorité pour n processeurs peut être simulée par un algorithme CRCW égalitaire avec n processeurs en temps $O(\frac{\lg\ n}{\lg\ lg\ n})$.

De nombreux autres résultats d'équivalences et de simulations entre les divers modèles peuvent être prouvés. Tous ces résultats permettent de conclure qu'au pire, à un facteur logarithmique près (en temps ou en coût), tous les modèles PRAM sont fondamentalement équivalents entre eux.

12 Simulation de machines PRAM sur des machines réelles

Lorsqu'on conçoit un algorithme PRAM, on travaille avec un modèle idéal de machine, sans limite de ressources. Ainsi, dans de nombreux exemples vus dans les sections qui précèdent, on a supposé que le nombre de processeurs était du même ordre de grandeur que la taille des données. Ainsi, dans plusieurs algorithmes, le nombre de processeurs était d'ordre $\Theta(n)$, n indiquant la taille des données d'entrée. En fait, on a même vu un algorithme (la recherche du minimum en temps constant sur une machine PRAM CRCW, algorithme 18) où le nombre de processeurs était quadratique ($\Theta(n^2)$).

Il est clair que pour de grandes valeurs de n, il n'est pas réaliste de penser utiliser, sur une vraie machine, autant de processeurs. Bien que certaines machines modernes puissent avoir de nombreux processeurs (par ex., plusieurs centaines de processeurs, parfois quelques milliers), on est évidemment loin d'un nombre arbitraire n de processeurs.

Une question qu'on peut alors se poser est la suivante : est-il possible de simuler, de façon relativement efficace, un algorithme PRAM pour p processeurs sur une machine réelle comportant uniquement p' processeurs (avec $p' < p$)?

12.1 Théorème de Brent

Un théorème (pour plus de détails, voir [CLR94, Section 30.3]) nous donne une première indication sur l'efficacité possible d'une simulation d'une machine à p processeurs avec uniquement p' processeurs.

Théorème 1 (Théorème de Brent (bis)) *Si un algorithme PRAM A à p processeurs s'exécute dans un temps t, alors pour $p' < p$, il existe pour le même problème un algorithme A' à p' processeurs qui s'exécute en temps $O(\frac{p}{p'} \times t)$.*

La preuve repose essentiellement sur l'idée suivante : chacune des étapes de l'algorithme A pour p processeurs peut être simulée en plusieurs étapes successives par les p' processeurs ($p' < p$). Plus précisément, une façon simple et naïve d'organiser une telle simulation est simplement que les p' processeurs soient utilisés autant de fois que nécessaire pour simuler, à chacune des étapes de l'algorithme A, le travail des p processeurs. En d'autres mots, si l'algorithme A nécessite t étapes, numérotées $1, 2, \ldots, t$, alors une étape arbitraire i sera simulée en temps $O(\lceil \frac{p}{p'} \rceil)$ par la machine à p' processeurs.[12] L'algorithme A' nécessitera

[11] Soit un groupe de n processeurs, chacun conservant dans l'un de ses registres une valeur indiquant si le processeur est "*vivant*" ou "*mort*". Le problème du *prisonnier le plus à gauche* consiste à déterminer, pour chaque processeur *vivant*, s'il est ou non le processeur avec le plus petit index (donc celui le plus à gauche) parmi tous les processeurs vivants.

[12] De façon stricte, il faut utiliser la partie entière supérieure (plafond) car, dans le cas général, p n'est pas nécessairement un multiple de p'. Toutefois, ceci ne fait pas de différence asymptotiquement. Notons aussi qu'on utilise une borne asymptotique puisque du travail additionnel d'ordonnancement des p' processus doit aussi être fait pour effectuer chacune des étapes de la simulation.

donc $\lceil \frac{p}{p'} \rceil \times t$ étapes, plus des surcoûts additionnels pour gérer la simulation, d'où la borne asymptotique indiquée.

Un point important à souligner d'une telle simulation est que bien que le temps d'exécution puisse augmenter, le coût total reste inchangé, puisque la simulation conduit à une augmentation du temps équivalente à celle de la réduction dans le nombre de processeurs. Toutefois, comme on le verra à la prochaine section, une hypothèse sous-jacente importante pour obtenir une simulation efficace est de préserver le temps $\Theta(1)$ d'accès à la mémoire, ce qui n'est pas toujours possible sur des machines avec un grand nombre de processeurs.

12.2 La machine SB-PRAM et le langage de programmation `Fork`

Divers chercheurs ont développé des machines parallèles physiques (en matériel) basées sur le modèle PRAM. Par exemple, à la fin des années 90, une machine appelée SB-PRAM a été réalisée par des chercheurs de l'université de Saarbrücken (Allemagne) — SB-PRAM pour SaarBrücken University PRAM. Le premier prototype fonctionnel de cette machine comptait 512 processeurs. En 2001, un nouveau prototype, avec 2048 processeurs et une mémoire partagée de 2 GB était en développement. Cette machine réalise le modèle de programmation PRAM le plus puissant, à savoir une machine PRAM-CRCW avec mécanisme de priorité pour la résolution des conflits. Elle fonctionne sous le principe de simulation par des processeurs virtuels et, au niveau architectural, utilise les techniques suivantes pour obtenir une simulation efficace [KKT01, PBB+02] :

- La machine physique sous-jacente est une machine à mémoire distribuée. Pour éviter les points de congestion (*hot spots*) lors des transferts dans le réseau, un réseau pipeliné de type *combining butterfly network* est utilisé. Un tel type de réseau supporte aussi de façon efficace les opérations de combinaison requises par le modèle CW (les noeuds du réseau peuvent effectuer eux-même les opérations de combinaison, sans accéder à la mémoire).
- Une fonction de hachage est utilisée pour distribuer les adresses entre les divers modules de mémoire, toujours pour éviter les points de congestion et assurer une simulation plus efficace (meilleure répartition du travail entre les processeurs).
- Les délais de latence pour les accès à la mémoire sont dissimulés par l'utilisation de fils d'exécution multiples (*multi-threading*) plutôt que par l'utilisation de caches.

Un langage de niveau machine, style assembleur, est évidemment disponible pour cette machine, langage basé sur le langage des machines RISC Sparcle (Université de Berkeley). Par contre, un langage de haut niveau est aussi disponible. Ce langage est appelé `Fork` et est basé sur le langage `C`, avec des extensions pour la gestion des variables privées et partagées, la gestion statique et dynamique du parallélisme, la création de groupes de processus, etc. `Fork` supporte divers styles de programmation parallèle, tels que le parallélisme de données, le parallélisme récursif, les pipelines, les sacs de tâches, etc. Une bibliothèque d'algorithmes et structures de données — appelée PAD = *Parallel Algorithms and Data structures* — est aussi disponible et définit des dictionnaires avec accès parallèles, des algorithmes pour calculer le rang d'une liste, calculer les préfixes, effectuer des fusions et des tris, construire et parcourir un circuit eulérien d'un arbre, déterminer les composantes connexes ou un arbre de recouvrement minimal d'un graphe, etc.

13 Conclusion : Avantages et limites du modèle PRAM

Le modèle PRAM est l'un des modèles les plus populaires pour l'étude et l'analyse des algorithmes parallèles — à peu près *tous* les livres existants qui traitent de conception et d'analyse

d'algorithmes parallèles se restreignent essentiellement à la présentation de ce modèle. Il existe donc un corpus très développé de méthodes et techniques de conception d'algorithmes parallèles pour ce modèle.

La popularité de ce modèle s'explique en grande partie par le fait que de nombreux détails de synchronisation et communication peuvent être ignorés lorsqu'on conçoit un algorithme parallèle pour le modèle PRAM : ceci est possible à cause de la nature *synchrone* du modèle d'exécution sous-jacent (exécution synchrone des lectures, calculs, écritures). De plus, un modèle de programmation basé sur l'utilisation d'une mémoire partagée par l'ensemble des processeurs est plus facile à utiliser qu'un modèle basé strictement sur l'échange de messages, puisque cela correspond mieux au modèle séquentiel et impératif habituel.

Malheureusement, il est important de réaliser qu'aucune machine réelle ne peut supporter, de façon directe, le modèle PRAM. Ainsi, on pourrait croire qu'une machine multi-processeurs avec mémoire partagée, où les communications se font par l'intermédiaire d'un *bus* (plutôt que d'un réseau) soit une approximation intéressante d'une machine PRAM. Toutefois, en pratique on a pu constater, dans les années 80–90, que de telles machines, pour des raisons physiques (une seule communication à la fois sur le bus, augmentation des conflits d'accès lorsqu'il y a plusieurs processeurs, augmentation du bruit et des délais lorsqu'on augmente la taille du bus) pouvaient difficilement supporter un grand nombre de processeurs. En pratique, on considère qu'au plus une trentaine (30) de processeurs peuvent fonctionner efficacement avec une telle architecture. Au-delà de cette limite, les performances obtenues peuvent décliner de façon significative avec l'ajout de processeurs, plutôt qu'augmenter.

Pour obtenir des machines comptant un plus grand nombre de processeurs, il faut plutôt utiliser une architecture de type *multi-ordinateurs*, où les processeurs sont reliés entre eux par l'intermédiaire d'un réseau (plutôt que d'un bus) et où chaque processeur est intimement associé à une partie physique de la mémoire (mémoire distribuée). En fait, les machines modernes utilisent de plus en plus une approche *hybride*, et hiérarchique, à savoir que la machine dans son ensemble est composée d'un groupe de machines reliées par un réseau, chacune de ces machines étant elle-même une machine multi-processeurs — on parle alors de grappes de multi-processeurs = *multi-processors cluster*. Au niveau *physique*, les processeurs d'un noeud (multi-processeurs) donné partagent alors une mémoire commune (par l'intermédiaire d'un bus), alors que les échanges entre processeurs provenant de noeuds différents se font par l'intermédiaire du réseau, donc par l'intermédiare d'échange de messages. On obtient alors une machine de type NUMA — *Non Uniform Memory Acccess* — donc une machine où les temps d'accès à la mémoire ne sont pas uniformes et dépendent de la position mémoire requises (locale ou non locale).

Soulignons aussi que, au niveau du modèle *logique*, de plus en plus de machines modernes supportent un espace d'adressage mémoire global et commun. En d'autres mots, au niveau du modèle logique et abstrait, le modèle supporté est celui d'une mémoire partagée, même si ce n'est pas le cas au niveau physique. Dans ce cas, c'est alors le système d'exécution (*run-time system*) ou certains composants de la machine physique elle-même (par ex., unités de gestion de mémoire) qui prennent en charge la transformation (le *mapping*) des accès à la mémoire partagée en des opérations d'accès au réseau de communication. En d'autres mots, l'architecture supporte donc une forme de *mémoire partagée virtuelle*, réalisée sur une *mémoire physique distribuée*.

Bien qu'il soit possible de simuler, comme on l'a vu à la Section 12, un algorithme PRAM sur une machine réelle, une telle simulation peut toutefois entraîner une décélération importante. Ceci est d'autant plus vrai lorsque, comme c'est le cas des grappes de multi-processeurs, le temps d'accès à la mémoire n'est alors plus de temps $\Theta(1)$. Par exemple, un accès en lecture à une adresse mémoire située sur un autre noeud demandera au moins deux transferts réseau : un premier transfert pour envoyer la requête au processeur où est conservée l'adresse qu'on désire accéder et un autre transfert pour retourner le résultat. Or, sur les machines modernes, un accès réseau est nettement plus lent qu'un accès mémoire, et ce *de plusieurs*

ordres de grandeur (voir le chapitre sur "Métriques de performance pour les algorithmes et programmes parallèles").

Malgré les limites du modèle PRAM, il n'en reste pas moins que ce modèle représente un modèle théorique important et intéressant, une forme de "machine abstraite et idéale" pour la conception et l'analyse d'algorithmes parallèles basés sur la programmation concurrente avec variables partagées. En d'autres mots, s'il n'est pas possible de développer un algorithme asymptotiquement efficace pour le modèle PRAM, alors on peut fortement douter qu'il sera possible de développer un programme parallèle qui sera rapide et efficace sur une vraie machine parallèle.

Références

[AO93] G.R. Andrews and R.A. Olsson. *The SR Programming Language*. Benjamin/Cummings Publishing, 1993. [QA76.73S68A54].

[CLR94] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction à l'algorithmique*. Dunod, 1994. [QA76.6C65614].

[GUD96] M. Gengler, S. Ubéda, and F. Desprez. *Initiation au parallélisme — Concepts, architectures et algorithmes*. Masson, 1996. [QA76.58G45].

[JaJ92] J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992. [QA76.58J34].

[KKT01] J. Keller, C. Kessler, and J. Traff. *Practical PRAM Programming*. John Wiley & Sons, Inc., 2001.

[LR03] A. Legrand and Y. Robert. *Algorithmique parallèle*. Dunod, 2003. [QA76.642L44].

[MB00] R. Miller and L. Boxer. *Algorithms Sequential & Parallel*. Prentice-Hall, 2000. [QA76.9A43M55].

[NN04] R. Neapolitan and K. Naimipour. *Foundations of Algorithms Using C++ Pseudocode (Third edition)*. Jones and Bartlett Publishers, 2004.

[PBB+02] W.J. Paul, P. Bach, M. Bosch, J. Fischer, C. Lichteneau, and J. Rohrig. Real PRAM programming. In *Euro-Par 2002 —- Parallel Processing*, pages 522–531. Springer-Verlag, LNCS-2400, 2002.

[XI98] C. Xavier and C. Iyengar. *Introduction to Parallel Algorithms*. John Wiley & Sons, 1998.

Cette partie des notes de cours est basée principalement sur les références suivantes :

- [NN04, Section 11.2]
- [MB00, Chapitres 5, 6, 7, 8 et 9]
- [CLR94, Sections 30.1 et 30.2]
- [JaJ92, Section 10.2]
- [KKT01, Chapitres 1, 2 et 8]
- [XI98, Chapitre 3]
- [LR03, Chapitre 1]
- [GUD96, Chapitre 6]