DM d'Algorithmique Parallèle

Laure Danthony

Pour le 26 novembre 2001

Evaluation des expressions arithmétiques sur une PRAM CREW

Introduction

Le but de ce devoir est de réaliser l'évaluation d'une expression arithmétique de taille n en $O(\log n)$ avec $O(\frac{n}{\log n})$ processeurs sur une PRAM CREW. On construit tout d'abord un arbre d'évaluation pour de telles expressions (c'est l'objet de la première section), et ensuite on évalue cet arbre (deuxième section).

1 Construction de l'arbre d'évaluation

▶ QUESTION 1 : **Transformation d'une expression correctement parenthésée en expression simple** L'algorithme qui consiste à faire les transformations locales suivantes :

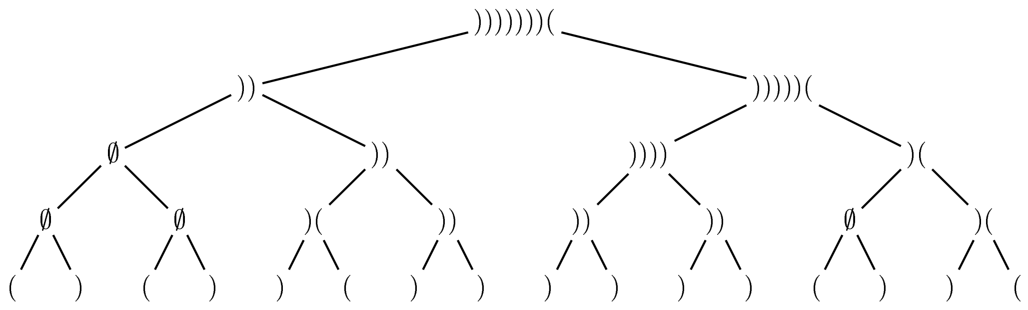
$$\begin{cases} (& \rightarrow & (((\\ + & \rightarrow &))+((\\ * & \rightarrow &)*(\\ ) & \rightarrow &))) \end{cases}$$

est d'une part clairement linéaire si l'on ne dispose que d'un unique processeur, et d'autre part, effectue bien le travail demandé si l'expression correctement parenthésée est encadrée par des parenthèses extérieures, par exemple sur l'expression correctement parenthésée : $(a + b * c)$ on obtient l'expression : $(((a)) + ((b) * (c)))$.

Ensuite, le théorème de Brent garantit l'existence d'un algorithme qui réalise la même chose en $O(\log n)$ avec $O(\frac{n}{\log n})$ processeurs.

▶ QUESTION 2 : **Dans le cas général** (*i.e.* dans le cas d'une suite de parenthèses quelconque et pas forcément dans le cas d'une expression bien parenthésée), une séquence réduite de parenthèses est de la forme $)^i(^j$ et lorsque l'on réunit deux de ces séquences on trouve une séquence du même type.

▶ QUESTION 3 : **Arbre de réduction de séquences de parenthèses :** Dessinons un arbre pour la séquence de parenthèses (()())))))())(, qui n'est pas une séquence "bien parenthésée" : chaque nœud de l'arbre représente la séquence irréductible des feuilles de son sous-arbre :



On suppose dans cette question que la longueur de la séquence est $n = 2^k$ avec $k|2^k$ (ici k=4 et la séquence vérifie ces hypothèses). Si on dispose de $\frac{2^k}{k}$ processeurs (ici 4), on peut construire cet arbre en parallèle en temps k : en effet, on découpe la séquence initiale en morceaux de même taille k. Ces processeurs en parallèle vont contruire le premier étage de l'arbre en temps $O(\frac{k}{2})$ (le processeur i s'occupe de ces deux premières parenthèses qu'il réduit éventuellement, puis des deux suivantes *etc.*). Le deuxième étage de l'arbre est construit en temps $O(\frac{k}{2})$, et ainsi de suite ... Finalement, on obtient une construction en $O(k) = O(\log n)$ soit bien ce qu'on voulait.

Une petite remarque tout de même : on considère que la réduction est en temps constant : en fait ce n'est pas vrai mais le temps est borné vu que l'on connaît la forme des séquences réduites.

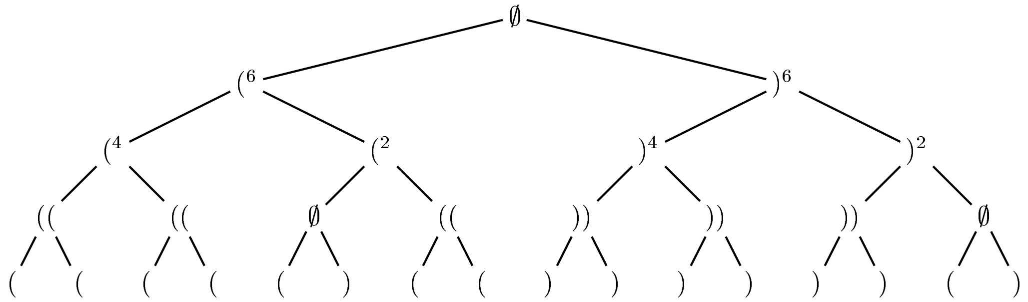
▶ QUESTION 4 : **Une fois que l'on dispose de cet arbre**, on va déduire de cet arbre un algorithme séquentiel qui trouve la parenthèse droite pour une parenthèse gauche donnée (c'est le *match*) : on suppose maintenant que la séquence étudiée est une séquence correctement parenthésée.

L'idée de l'algorithme décrit ici est que l'on va parcourir l'arbre en remontant une seule fois (depuis la parenthèse à matcher) puis le descendre une seule fois (vers sa parenthèse match). Pour cela, on va garder en mémoire une information qui sera :

- "en montant", $[(^i, j]$ où $(^i$ désigne le nœud courant du parcours, et j la cordonnée de la parenthèse à matcher dans le sous-arbre courant, *i.e.* dans le sous-arbre de racine $(^i$.
- "en descendant", $])^i, j[$ où $)^i$ désigne le nœud courant du parcours, et j la cordonnée de la parenthèse résultat du match dans le sous-arbre courant, *i.e.* dans le sous-arbre de racine $(^i$.

Ainsi, lorsque l'on décide de changer de sens, il faudra faire l'opération $j := i + 1 - j$[1]. On finit l'algorithme lorsque l'on atteint une feuille, qui est alors le match de la parenthèse considérée. Il ne reste plus qu'à savoir comment on décide de changer de sens :

Si on considère la séquence : (((((()(()))))))() (bien parenthésée), on cherche par exemple le match de la deuxième parenthèse ouvrante. On obtient par l'algorithme de la question précédente l'arbre :



Pour la deuxième parenthèse ouvrante, on a au départ la quantité $[((, 2]$ qui signifie que je suis dans le sous-arbre de racine ((qui contient la parenthèse que je veux matcher, et que la parenthèse est dans ce sous-arbre à l'indice 2. Ensuite, en remontant, vu que l'on est remonté par la "branche droite", on obtient la quantité $[(^4, 4]$, et en remontant encore, $[(^6, 4]$ (branche gauche). Comme on tombe alors sur le nœud $\emptyset$, on choisit alors de redescendre dans le sous-arbre droit en cherchant la quatrième parenthèse fermante en partant de la droite, soit la $6 + 1 - 4 = 3$-ème parenthèse fermante en partant de la gauche. On obtient donc successivement lors de la redescente les quantités $[)^6, 3]$, $[)^4, 3]$, et enfin $[)^2, 1]$ et comme ensuite on tombe sur un nœud, on a bien trouvé le match (en fait, on a trouvé le nœud qui contient cette parenthèse).

La formalisation de cet algorithme serait longue est fastidieuse, on va seulement ici se contenter de dire les cas où l'on décide de redescendre : on "redescent" (*i.e.* on décide de changer de sens de parcours de l'arbre') lorsque le label du nœud "fils droit" du nœud rencontré contient un nombre de parenthèses fermantes suffisant à fermer toutes les parenthèses ouvrantes précédent la parenthèse considérée (incluse). Avec les notations de l'algorithme informel, on descend si en étant $[(^j, k]$, on rencontre un "frère" qui contient au moins k parenthèses fermantes. Dans notre cas, dans l'état $[(^6, 4]$, au vu du frère droit $)^6$ on choisit[2] de redescendre

[1]une sombre histoire d'intervalle, voir l'exemple
[2]enfin, ici on n'avait guère le choix !

$(6 > 4)$.

Disons enfin que comme la profondeur de l'arbre est $O(\log n)$, cet algorithme séquentiel est bien en $O(\log n)$.

▶ QUESTION 5 : **Fonction match pour toutes les parenthèses ouvrantes**. La méthode qui consiste à donner à chacun des $\frac{n}{\log n}$ processeurs ne fonctionne pas pour la complexité donnée, car elle est alors en $O(n)$. On va faire en sorte ici d'améliorer la complexité :

- On divise la séquence en k morceaux de taille égale, en parallèle on calcule la réduction de ces k morceaux. Cette étape est en $O(\log n)$ avec le nombre de processeurs dont on dispose.
- Pour chaque sous-séquence, on calcule le match de la première parenthèse ouvrante et de la dernière parenthèse ouvante (si elles existent). Là aussi, la complexité est en $\log n$.
- A l'aide de ces informations, on va pouvoir calculer le match pour les parenthèses des séquences réduites : on calcule tout d'abord pour chaque séquence réduite le match de la parenthèse (la plus à gauche, et aussi celle la plus à droite. On en déduit le match des parenthèses internes, puisque ce sont obligatoirement des parenthèses ouvrantes, et que l'on a réduit : on n'a plus que des parenthèses fermantes entre les deux matchs obtenus, et il y en a juste le compte.
- Ensuite, pour les parenthèses restantes, *i.e.* celles qui ont été réduites dans la première phase, on sait que leur match est dans la séquence. Calculer exhaustivement les matchs pour ces parenthèses-là n'augmentera pas le coût de l'algorithme.

On obtient donc un algorithme en $O(\log n)$ avec $O(\frac{n}{\log n})$ processeurs, c'est ce qu'on voulait.

▶ QUESTION 6 : **Construction de l'arbre d'évaluation** Etant donnée une expression simple, on calcule la fonction match sur toutes les parenthèses à l'aide des questions précédentes appliquées à la séquence où l'on a enlevé tout autre caractère que les parenthèses. Ceci se fait avec $O(\frac{n}{\log n})$ avec $O(\log n)$ processeurs. Ensuite, on peut trouver un algorithme qui utilise un processeur et qui crée l'arbre voulu en temps $O(n)$: on parcourt l'expression, lorsque l'on trouve une parenthèse, on construit récursivement le sous-arbre d'évaluation correspondant à la sous-séquence entre cette parenthèse et son match. Lorsque l'on trouve un opérateur, on joint les sous-arbres correpondant aux expressions simples de gauche et de droite. Cet algorithme est clairement linéaire et en appliquant le théorème de Brent, on peut donc obtenir l'arbre d'évaluation d'une expression de taille n en $O(\log n)$ en $O(\frac{n}{\log n})$ processeurs.

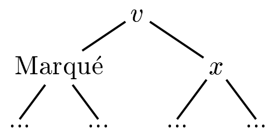
Ainsi, à ce stade, on a construit un arbre d'évaluation de l'expression en $O(\log n)$, où n est la taille de l'expression, et ceci avec $O(\frac{n}{\log n})$ processeurs. Il ne reste plus qu'à l'évaluer dans le même temps. Dans la suite, on l'évalue dans le même temps mais pour un nombre plus élevé de processeurs, un algorithme meilleur étant trop long à développer.

2 Evaluation de l'expression

L'idée est d'utiliser des techniques de saut de pointeurs pour parcourir l'arbre d'évaluation plus rapidement. A tout instant, si un nœud est marqué, on sait évaluer la sous-expression correspondant au sous-arbre enraciné en ce nœud : il suffit d'évaluer la valeur d'une expression homographique, ce qui se fait en temps constant.

▶ QUESTION 7 : **Evaluation de l'arbre de l'expression**

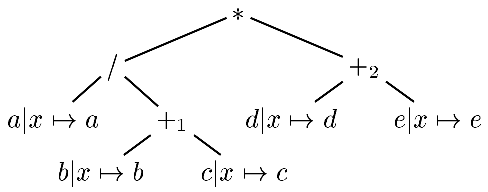
- Si $f(x) = \frac{ax+b}{cx+d}$ et $g(x) = \frac{a'x+b'}{c'x+d'}$, alors $h(x) = f(g(x)) = \frac{a''x+b''}{c''x+d''}$ où $a'' = aa' + bc'$, $b'' = ab' + d'b$, $c'' = ca' + dc'$, $d'' = dd' + b'c$. On peut donc calculer cette composée en temps constant (8 opérations $*$ et 4 opérations $+$).
- Dès que $cond(v)$ est marqué, on peut donc calculer sa valeur en temps constant, il suffit pour cela d'évaluer la composée de sa fonction et de celle de $cond(v)$ et d'évaluer le résultat. Ceci se fait en temps constant.
- **Activer** Pour tout nœud v en parallèle, si je suis dans la situation :



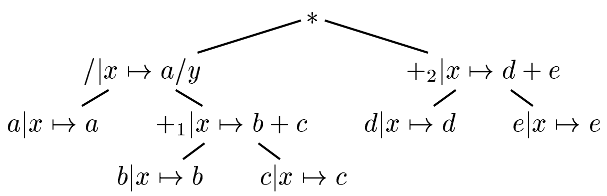
  Alors je m'active et je réalise $cond(v) \leftarrow x$ et la composée de ma fonction avec la fonction de mon fils marqué.
- **Marquer** Pour tout nœud en parallèle, si $cond(v)$ est marqué ou que les deux fils de v sont marqués, alors marquer v (on met un "drapeau" à v).
- **Sauter** Pour tout nœud v en parallèle, $cond(v) \leftarrow cond(cond(v))$.

L'algorithme consiste à enchaîner des séquences (Marquer, Activer, Sauter, Sauter) assez longtemps pour que la racine de l'arbre soit marquée. Alors on est en mesure d'évaluer l'expression toute entière.

On va détailler l'algorithme sur l'exemple suivant (situation au début de l'algorithme) : [3]



Toutes les feuilles sont marquées et $cond(v) = v$ pour tout nœud v. Au tour d'après, on obtient :



les nœuds $/$, $+_1$ et $+_2$ ont été activés, et on a calculé les fonctions composées correspondantes, et on a fait le saut de pointeurs vers les fils. Après encore un tour, on obtient finalement :

[3] J'ai numéroté les opérateurs $+$.

- $* : x \mapsto \frac{a}{b+c} * (d+e)$
  - $/$
    - $a|x \mapsto \frac{a}{b+c}$
    - $+|x \mapsto b+c$
      - $b|x \mapsto b$
      - $c|x \mapsto c$
  - $+|x \mapsto d+e$
    - $d|x \mapsto d$
    - $e|x \mapsto e$

Evidemment, les lettres a, b, c, d, e sont remplacées par des chiffres, et on calcule à chaque fois que l'on peut calculer.

Dans les deux questions suivantes, on va montrer que cet algorithme fonctionne bien dans le temps imparti. Pour cela, notons :
- $(I_2a)_k$ *la propriété : Soit x un nœud de l'arbre, après k étapes, $size(x/cond(x)) \geqslant 2^k$,*
- $(I_2b)_k$ *la propriété : Soit x un nœud de l'arbre, après k étapes, aucun des fils de $cond(x)$ n'est marqué,*
- $(I_2c)_k$ *la propriété : Soit x un nœud de l'arbre, après k étapes, $cond(x)$ est une feuille.*

▶ QUESTION 8 : **Montrons I_{1k} à l'aide de $(I_2)_{k-1}$ et de $(I_1)_{k-1}$.** Soit x un nœud de l'arbre.
- Si $size(x) \leqslant 2^{k-1}$ alors en utilisant $(I_1)_{k-1}$ on obtient immédiatement que x est marqué à l'étape k.
- Si $2^{k-1} < x \leqslant 2^k$ alors on va appliquer un des cas de la propriété $(I_2)_{k-1}$:
  - Si $(I_2a)_{k-1}$ alors comme après $k-1$ étapes $cond(x)$ est une feuille, à l'étape d'après x est marqué d'après l'algorithme de marquage.
  - Si $(I_2b)_{k-1}$ alors pour tout fils y de $cond(x)$, y n'est pas marqué à l'étape $k-1$ donc par la contraposée de $(I_1)_{k-1}$ on obtient $size(y) > 2^{k-1}$ donc $cond(x) > 2^k$ donc $size(x) > 2^k$, ce qui est impossible.
  - Si $(I_2c)_{k-1}$ alors à l'étape $k-1$, $size(x/cond(x) \geqslant 2^{k-1}$ donc $size(cond(x)) \geqslant 2^{k-1}$ donc par la contraposée de $(I_1)_{k-1}$, $cond(x)$ est marqué à l'étape $k-1$ donc d'après l'algorithme, x est marqué à l'étape k.

On a donc bien montré l'implication voulue.

▶ QUESTION 9 : **Montrons I_{2k} à l'aide de $(I_2)_{k-1}$ et de $(I_1)_{k-1}$.** On a trivialement la proposition : Si $(I_1)_{k-1}$ et $(I_2c)_{k-1}$ alors $(I_2c)_k$. Les autres cas semblent nettement plus difficiles.

Les questions 8 et 9 montrent par récurrence (les cas de base sont faciles à établir) en particulier pour tout $k \geqslant 1$, $(I_1)_k$. Avec $k = \lceil \log n \rceil$, on obtient la correction et la complexité de l'algorithme. On a donc bien obtenu une évaluation en $O(\log n)$ avec $O(n)$ processeurs.