Rapport de TP 4MMAOD : Génération d'ABR optimal

BIN MOHMAD SHAH Hariz (groupe SLE)
PRADA MEJIA Robinson (groupe SLE)

April 14, 2017

1 Principe de notre programme (1 point)

Nous avons choisi d'utiliser la méthode itérative pour implementer le programme de la création de l'arbre binaire de recherche optimal. La raison pour laquelle nous l'avons choisie est pour pouvoir d'éviter de répéter la même calcul plusieurs fois. nous utilisons la méthode memoïsation.

Le principe de notre programme est on calcul d'abord tous les sommes de fréquence parmis les combinations des noeuds possibles de l'arbre. On les stocke dans un tableau de deux dimension [i][j] avec i: le noeud initial et j: le noeud final.

Ensuite, nous calculons les coût optimaux pour tous les combinations possibles des noeuds et nous les stockons également dans un autre tableu qui est aussi en deux dimensions [i][j]. En même temps, à chaque fois nous prenons le coût minimum parmis tous les coûts possibles, nous sauvegardons le noeud correspondant.

En fin, nous utilisons tous les tableaux que nous avons créé dans une fonction qui est capable d'utiliser tous les informations dans ces tableaux afin de produire un arbre binaire de recherche optimal.

Ici, nous utilisons le parcours ligne par ligne parce que d'après de notre connaissance sur le cours de AOD, le cache est organisé bloc par bloc et pour chaque bloc, il contient le donnée à côté. Pourtant, notre algo pourrait être mieux si on utilise la méthode de cache-aware ou cache-oblivious afin d'utiliser le maximum le bloc de cache.

2 Analyse du coût théorique (2 points)

2.1 Nombre d'opérations en pire cas : $O(n^3)$

Justification : Dans notre programme, il y a 3 fonctions principaux.

- **void init_all(int n, FILE *freqFile)**: Cette fonction contient 2 boucle indépendant qui dépend de nombre d'element. Il y a seulement les opérations de la création de tableu. D'où, le coût pour cette fontion est de $O(n)$.

- **void create_matrices(int n)** : Cette fonction est la fonction de calcule les coûts optimaux qui contient plusieurs boucle qui correspondant à la somme

$$T(n_1, n_2, c_1) = \sum_{i=0}^{n_1} \sum_{j=1}^{n_2} k + \sum_{i=0}^{n_1} k + \sum_{i=0}^{n_1-1} k + \sum_{i=2}^{n_1} \sum_{j=0}^{n_1-i} k \sum_{k=0}^{n_2} C_{ij}$$

avec Cij le coût d'opération de comparaison le coût min et le coût courant.

Et k le coût des opérations costants.

En pire cas, le coût minimum est quand le dernière noeud est le noeud de l'arbre optimal. A chaque tour, il faut mettre à jour le valeur min dans le tableu. Comme ce n'est que l'affectation de nouvelle valeur, donc le cout de cette opération est aussi constant.

Alors en pire cas, Cij = k

$$T(n_1, n_2, c_1) = k(n^2 + n + n + kn^3)$$

Nous avons le nombre d'opérations en pire cas $O(n^3)$ pour cette fonction

- **void print_all(int n)**: Cette fonction contient 2 boucle indépendant qui dépend de nombre d'element. D'où, le coût pour cette fontion est aussi de O(n).

En faisant la somme de nombre d'opérations de trois fonctions, nous trouvons le nombre d'opérations est $O(n^3)$.

2.2 Place mémoire requise : $O(n^2)$

Justification : Dans notre programme, nous utilisons 3 tableaux (costs, weights, roots) de deux dimensions [i][j] avec i,j: de 0 jusqu'à n et 2 tableaux (p et keys) de une dimension de taille n. Nous alluons alors $3(n+1)^2+2n$ espace mémoire pour notre programme. Donc, c'est en $O(n^2)$.

2.3 Nombre de défauts de cache sur le modèle CO : $O(n^3/L)$

$$\mathbf{Q(n,L,Z) = 6n/L + 3}n^2\mathbf{/L} + n^3\mathbf{/L}$$

Justification : On se déplace en pire cas quand le cache est, z « $3(n+1)^2$ + 2n. Au niveu de premier fonction **init_all**, il n'aura pas de problème de défaut de cache parce que ce n'est que allocation de la mémoire et l'affectation de valeur. Ensuite, dans la fonction **create_matrices**, il y aura des défauts de cache. Il y a 4 boucles dans cette fonctions:
1 - Pour la premier boucle Q(n,Z,L) = 2n/L.
2 - Pour la deuxième, Q(n,Z,L) = n/L
3 - Pour la troisième, Q(n,Z,L) = 2n/L
4 - Pour la quatrième, Q(n,Z,L) = $3n^2$/L + n^3/L
Alors, nous pensons donc obtenir Q(n,Z,L) = 5n/L + $3n^2$/L + n^3/L des défauts de cache pour cette fonction. Pour la dernier fonction qui est **print_all**, nous pensons qu'il y a des défaut de caches de n/L si z « n parce que dans cette fonction on parcours que le tableau des éléments un seul fois afin de les afficher à la sortie stdout. Au total Q(n,L,Z) = 6n/L + $3n^2$/L + n^3/L qui de $O(n^3/L)$).

3 Compte rendu d'expérimentation (2 points)

3.1 Conditions expérimentaless

Dans cette section sont décrits les conditions permettant la reproductibilité des mesures comme la description de la machine et la méthode utilisée pour mesurer le temps.

3.1.1 Description synthétique de la machine :

Pour les test efectués on a utilisé une machine de l'ensimag (ENSIPC137) avec les specifications suivantes:

- Processeur : Intel(R) Core(TM) i3 CPU M 370 @ 2.40GHz;
- Mémoire(MB) : total 15842, used 2073, shared buff/cache 38/2434, available 13437;
- Système d'exploitation : CentOS Linux release 7.2.1511;

Pendant les benchmarks la machine n'avait pas d'autres processus ou utilisateurs en cours d'exécution et chaque benchmark a été lancé manuellement.

3.1.2 Méthode utilisée pour les mesures de temps :

Nous avons utilisé la commande "time pour mésurer le temps d'éxécution de notre programme. L'unité de temps est en second.

3.2 Mesures expérimentales

Le tableau suivant a les temps d'exécution mesurés pour chacun des 6 benchmarks imposés (temps minimum, maximum et moyen sur 5 exécutions). Toutes les valeurs de temps sont exprimées en secondes.

	coût de l'arbre	temps min	temps max	temps moyen
benchmark1	74	0.003	0.003	0.003
benchmark2	131	0.003	0.003	0.003
benchmark3	4163722606	0.098	0.119	0.1118
benchmark4	280892076	0.119	0.315	0.2656
benchmark5	452533877	0.561	0.595	0.5834
benchmark6	823550785	1.687	1.754	1.72

Figure 1: Mesures des temps minimum, maximum et moyen de 5 exécutions pour les 6 benchmarks.

Benchmark 1
I refs: 204,055
I1 misses: 1,137
LLi misses: 1,121
I1 miss rate: 0.55%
LLi miss rate: 0.54%
D refs: 70,955 (53,009 rd + 17,946 wr)
D1 misses: 3,129 (2,537 rd + 592 wr)
LLd misses: 2,510 (1,979 rd + 531 wr)
D1 miss rate: 4.4% (4.7% + 3.2%)
LLd miss rate: 3.5% (3.7% + 2.9%)
LL refs: 4,266 (3,674 rd + 592 wr)
LL misses: 3,631 (3,100 rd + 531 wr)
LL miss rate: 1.3% (1.2% + 2.9%)

Benchmark 2
I refs: 230,878
I1 misses: 1,137
LLi misses: 1,121
I1 miss rate: 0.49%
LLi miss rate: 0.48%
D refs: 82,387 (61,122 rd + 21,265 wr)
D1 misses: 3,166 (2,540 rd + 626 wr)
LLd misses: 2,544 (1,979 rd + 565 wr)
D1 miss rate: 3.8% (4.1% + 2.9%)
LLd miss rate: 3.0% (3.2% + 2.6%)
LL refs: 4,303 (3,677 rd + 626 wr)
LL misses: 3,665 (3,100 rd + 565 wr)
LL miss rate: 1.1% (1.0% + 2.6%)

Benchmark 3
I refs: 94,490,943
I1 misses: 1,162
LLi misses: 1,147
I1 miss rate: 0.00%
LLi miss rate: 0.00%
D refs: 44,340,387 (39,635,856 rd + 4,704,531 wr)
D1 misses: 2,703,645 (2,155,202 rd + 548,443 wr)
LLd misses: 103,909 (4,508 rd + 99,401 wr)
D1 miss rate: 6.0% (5.4% + 11.6%)
LLd miss rate: 0.2% (0.0% + 2.1%)
LL refs: 2,704,807 (2,156,364 rd + 548,443 wr)
LL misses: 105,056 (5,655 rd + 99,401 wr)
LL miss rate: 0.0% (0.0% + 2.1%)

Benchmark 4
I refs: 369,487,124
I1 misses: 1,162
LLi misses: 1,155
I1 miss rate: 0.00%
LLi miss rate: 0.00%
Drefs:174,083,538 (156,595,480 rd + 17,488,058 wr)
D1misses:11,167,466 (8,941,242 rd + 2,226,224 wr)
LLd misses: 552,492 (163,555 rd + 388,937 wr)
D1 miss rate: 6.4% (5.7% + 12.7%)
LLd miss rate: 0.3% (0.1% + 2.2%)
LL refs: 11,168,628 (8,942,404 rd + 2,226,224 wr)
LL misses: 553,647 (164,710 rd + 388,937 wr)
LL miss rate: 0.1% (0.0% + 2.2%)

Benchmark 5
I refs: 823,682,729
I1 misses: 1,145
LLi misses: 1,134
I1 miss rate: 0.00%
LLi miss rate: 0.00%
Drefs:388,558,162 (350,394,209 rd + 38,163,953 wr)
D1misses:25,386,244 (20,356,028 rd + 5,030,216 wr)
LLd misses:1,329,585 (463,764 rd + 865,821 wr)
D1 miss rate: 6.5% (5.8% + 13.1%)
LLd miss rate: 0.3% (0.1% + 2.2%)
LL refs: 25,387,389 (20,357,173 rd + 5,030,216 wr)
LL misses: 1,330,719 (464,898 rd + 865,821 wr)
LL miss rate: 0.1% (0.0% + 2.2%)

Benchmark 6
I refs: 2,273,374,881 I1 misses: 1,160
LLi misses: 1,153
I1 miss rate: 0.00%
LLi miss rate: 0.00%
Drefs:1,072,898,451(968,896,024rd+104,002,427wr)
D1misses:71,226,856(57,216,226 rd+14,010,630 wr)
LLd misses:3,687,241(1,307,044 rd+2,380,197 wr)
D1 miss rate: 6.6% (5.9% + 13.4%)
LLd miss rate: 0.3% (0.1% + 2.2%)
LL refs: 71,228,016 (57,217,386 rd + 14,010,630 wr)
LL misses: 3,688,394 (1,308,197 rd + 2,380,197 wr)
LL miss rate: 0.1% (0.0% + 2.2%)

Les données ci-dessus ont les resultats d'exécution du cachegrind mesurés pour chacun des 6 benchmarks imposés.

3.3 Analyse des résultats expérimentaux

Les temps obtenus pendant les benchmarks (Figure 1) et les valeurs de défaut du cache correspondent aux valeurs théoriques calculées dans la dernière section. Pour certaines de ces valeurs, les valeurs attendues sont plus petites puisque les calculées sont pour les pire cas scénario. Dans les résultats du cachegrind, pour le benchmark le plus significatif (6) avec n = 5000 échantillons, la valeur du cache défauts passe jusqu'à 71 000 000 environ. Cela indique qque pour ce défaut du cache, on a obtenu beaucoup moins que l'attendu $O(2n^2)$.

Pour le nombre d'opérations attendues pour le même benchmark, il est possible de remarquer que les valeurs correspondent beaucoup mieux. Les valeurs à comparer sont $O(n^3)$ (theorique) avec 2,273,374,881 (réel). A partir de l'anterieur on peut concluire que les calculs ne sont pas exactement precis mais qui nous donnent une bonne approximation des resultats à attendre dans la partie experimentale.