Rendu AOD

Mohamedou chirf M'hamedou , Mohamed Ahmed Mohamed Lmeine

October 2023

Questionnaire TP AOD 2023-2024

1 Préambule

. Le programme récursif avec mémoisation fourni alloue une mémoire de taille $N.M$. Il génère une erreur d'exécution sur le test 5 (c-dessous) . Pourquoi ?

Reponse :

- Le programme récursif avec mémoisation que nous utilisons alloue une mémoire de taille N.M, où N et M sont les paramètres de l'entrée pour notre fonction récursive.
- lors de l'exécution du test 5, nous rencontrons une erreur d'exécution : Cette erreur est due au fait que la mémoire utilisée pendant le test 5 est : $20236404 * 19944517 \approx 4.10^{14}$

qui dépasse la quantité de mémoire disponible sur le système que l'on peut trouver avec la commande :

```
free -h
```

- En exécutant la commande 'free -h'sur le pc de L'ensimag, nous observons que la quantité alloue par le systeme d'exploiatation est 28.10^9.
- Donc la maméoire utilise par le test 5 depasse la quantité allouee par le systéme d'exploiataion ce qui génere des errures .

2 Programme itératif en espace mémoire $O(N+M)$

2.1 Expliquation du méthode itératif :

Expliquer très brièvement (2 à 5 lignes max) le principe de votre code, la mémoire utilisée, le sens de parcours des tableaux.

- l'idée principale est d'allouée une table de taille N+M
- on remplir tout d'abord la ligne M et la colonne N dans la table
- Et à chaque itération on calucl la ligne M-i et colonne N -i et on le met dans la table
- l'idée en faite que pour calculer l'élement des indices i, j on a besion des élemnts : (i+1,j) , (i+1, j+1) , (i , j+1) qui sont sont stockées consécutivement dans le table et aprés calcul de (i,j) et on le me dans la possition (i+1, j+1) car l'élement (i+1 , j+1) ne sera jamains utilisées dans le calcul des autre élemntes .

2.2 Analyse du coût théorique de programme :

Dans le code source de la fonction itératif on a :

Listing 1: code c

```
for( int i = M ; i >=0 ; i-- )
{
    for( int j = N ; j >=0 ; j-- )
    {
        EditDistance_NW_Iter_i_j(&ctx, i , j);
    }
}
```

on a une fonction **EditDistance_NW_Iter**$_{ij}$

cette fonction sert à calcul l'élement (i,j) et le met dans sa position dans la table allouée

Donc le cout théorique de programme itératif ègale à : NM multiplié par le cout de

la fonction EditDistance_NW_Iter$_{ij}$

1. **place mémoire allouée (ne pas compter les 2 séquences X et Y en mémoire via `mmap`) :**

   la place mémoire allouée sans les 2 séquences X et Y est :$O(N+M)$**.**

2. **travail (nombre d'opérations) :**

   Le nombre d'opérations (coût arithmétique) est : $\Theta(NM)$

3. **nombre de défauts de cache obligatoires (sur modèle CO, y compris sur** X **et** Y**):**

   Réponse :

   Dans la fonction itératif on accéde aux élements (i,j+1) , (i+1,j) et (i+1,j+1) qui soant stockés consécutivement dans la table donc on

   le nombre de défaut de cache obligatoire dans la fonction est : $\Theta(\frac{N+M}{L})$

4. **nombre de défauts de cache si** $Z \ll \min(N, M)$ **:**

   Réponse :

   si $Z \ll \min(N, M)$ **:**

   ona le nombre de défauts de cache est : $\Theta(\frac{NM}{L})$

3 Programme cache aware

3.1 Expliquation de la Programme cache aware:

- **Dans le cas de cache aware on a chnager la structure en utilisant 2 tableaux pour stocker respectivement une ligne et une colonne courante.**
- **on a utiliser la méthode de Blocking pour calculer à chaque fois un block de taille K qui tient dans le cache.**
- **Pour minimiser les défauts de cache il faut que : 2K1 + 2K2 + (1) soit inférieur ou égal à Z.**
- **Afin de minimiser le nombre de défauts de cache on a choisi :** $K_1 = K_2 = \frac{Z}{4}$

3.2 Analyse du coût théorique de programme:

1. **place mémoire (ne pas compter les 2 séquences initiales** X **et** Y **en mémoire via** `mmap`**) :**

   Réponse : Le cout en espace sans les séquences X et Y est un $\Theta(N + M)$

2. **travail (nombre d'opérations) :**

   Réponse :

   Le nombre d'opérations (coût arithmétique) est : $\Theta(N * M)$

3. **nombre de défauts de cache obligatoires (sur modèle CO, y compris sur** X **et** Y**):**

   Réponse: le nombre de défauts de caches si le cache est trés grand : $\Theta(\frac{(N+M)}{L})$

4. **nombre de défauts de cache si** $Z \ll \min(N, M)$ **: Réponse :**

   si $Z \ll \min(N, M)$ **:**

   ona le nombre de défauts de cache est : $\Theta(\frac{NM}{L})$

4 Programme cache oblivious:

4.1 Expliquation de la Programme cache aware:

- **On découpe récursivement suiavnt la dimensionla plus grans jusqu'à un seuil S.**
- **initalemnton a fixé un seui par défaut égla e à 20 .**
- **on a utiliser la fonction CalculBloc qui si on arrive à la seuil S .**

4.2 Analyse du coût théorique de programme:

1. **place mémoire (ne pas compter les 2 séquences initiales** X **et** Y **en mémoire via** `mmap`**) :**

   Réponse : Le cout en espace sans les séquences X et Y est un $\Theta(N + M)$

2. **travail (nombre d'opérations) :**

   Réponse :

   Le nombre d'opérations (coût arithmétique) est : $\Theta(N * M)$

3. **nombre de défauts de cache obligatoires (sur modèle CO, y compris sur** X **et** Y**):**

   Réponse: le nombre de défauts de caches si le cache est trés grand : $\Theta(N + ML)$

4. **nombre de défauts de cache si $Z \ll \min(N, M)$: Réponse :**

   si $Z \ll \min(N, M)$:

   ona le nombre de défauts de cache est : $\Theta(\frac{NM}{L})$

5 Réglage du seuil d'arrêt récursif du programme cache oblivious (1 point)

Comment faites-vous sur une machine donnée pour choisir ce seuil d'arrêt? Quelle valeur avez vous choisi pour les PC de l'Ensimag? (2 à 3 lignes)

6 Expérimentation (7 points)

Description de la machine d'expérimentation:
Processeur: Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz ,architecture x86$_6$4

Caches :

- **L1d: 64 KiB (2 instances)**
- **L1i: 64 KiB (2 instances)**
- **L2: 512 KiB (2 instances)**
- **L3: 4 MiB (1 instance)**

Système: linux

6.1 (3 points) Avec `valgrind --tool =cachegrind --D1=4096,4,64`

```
distanceEdition ba52_recent_omicron.fasta 153 N wuhan_hu_1.fasta 116 M
```

en prenant pour N et M les valeurs dans le tableau ci-dessous.

Les paramètres du cache LL de second niveau est : ... ***mettre ici les paramètres: soit ceux indiqués ligne 3 du fichier cachegrind.out.(pid) généré par valgrind: soit ceux par défaut, soit ceux que vous avez spécifiés à la main*** [1] ***pour LL.***

Le tableau ci-dessous est un exemple, complété avec vos résultats et ensuite analysé.

[1] par exemple: `valgrind --tool=cachegrind --D1=4096,4,64 --LL=65536,16,256 ...` mais ce n'est pas demandé car cela allonge le temps de simulation.

		récursif mémo			itératif		
N	M	#Irefs	#Drefs	#D1miss	#Irefs	#Drefs	#D1miss
1000	1000	217,159,532	122,112,157	3,666,488	200,547,095	119,366,597	2,645
2000	1000	433,336,926	243,391,389	9,656,458	400,095,943	237,886,705	2,826
4000	1000	867,109,024	487,355,645	21,782,312	800,595,077	476,328,539	3,128
2000	2000	867,100,622	487,878,340	17,022,218	801,360,771	477,141,309	2,987
4000	4000	3,465,823,335	1,950,538,535	73,698,297	3,204,271,102	1,908,065,726	1,946,876
6000	6000	7,796,283,797	4,387,974,346	170,433,509	7,208,892,362	4,292,823,762	5,080,286
8000	8000	13,857,913,344	7,799,937,917	307,280,238	12,815,027,344	7,631,218,208	9,020,903

		cache aware			cache oblivious		
N	M	#Irefs	#Drefs	#D1miss	#Irefs	#Drefs	#D1miss
1000	1000	92,408,536	47,864,478	2645	93,432,452	48,510,155	2,647
2000	1000	183,889,388	94,929,569	2,828	185,937,868	96,220,032	2,843
4000	1000	368,253,990	190,462,835	28,736	372,354,938	193,046,210	36,767
2000	2000	369,016,288	191,279,606	3,346	373,230,672	193,938,209	3,468
4000	4000	1,475,266,432	764,865,522	102,460	1,492,365,810	775,656,267	129,722
6000	6000	3,318,891,864	1,720,788,066	259,611	3,411,733,724	1,779,848,217	439,184
8000	8000	5,899,735,346	3,058,890,008	457,735	5,968,606,024	3,102,355,277	584,897

Important: analyse expérimentale: ces mesures expérimentales sont elles en accord avec les coûts analysés théroiquement (justifier) ? Quel algorithme se comporte le mieux avec valgrind et les paramètres proposés, pourquoi ?

Réponse:

- **Dans le test expérimental, on remarque que le programme récursif avec mémoisation présente un nombre de défauts de cache sur D1 très important par rapport aux autres méthodes.**
- **Par exemple, dans le test avec N=1000 et M=1000, le programme récursif avec mémoisation a 3.10^6 défauts de miss sur D, alors que les autres programmes présentent 2.10^3 défauts de miss, ce qui est bien compatible avec la théorie.**
- **D'aprés le résulat exprimantale l'algorithme se comporte le mieux avec valgrind est : la méthode cache aware , et pour la cahce obilivous on a fixe le seuil S à 20 ce qui peut jusitfier ce résultat .**

6.2 (3 points) Sans valgrind, par exécution de la commande :

```
distanceEdition  GCA_024498555.1_ASM2449855v1_genomic.fna  77328790  M
                 GCF_000001735.4_TAIR10.1_genomic.fna      30808129  N
```

**On mesure le temps écoulé, le temps CPU et l'énergie consommée avec : *[présicer ici comment vous avez fait la mesure:* `time` *ou* `/usr/bin/time`

ou `getimeofday` *ou* `getrusage` *ou...*
L'énergie consommée sur le processeur peut être estimée en regardant le compteur RAPL d'énergie (en microJoule) pour chaque core avant et après l'exécution et en faisant la différence. Le compteur du core K *est dans le fichier* `/sys/class/powercap/intel-rapl/intel-rapl:K/energy_uj` *.*
Par exemple, pour le cœur 0: `/sys/class/powercap/intel-rapl/intel-rapl:0/energy_uj`

Nota bene: pour avoir un résultat fiable/reproductible (si variailité), il est préférable de faire chaque mesure 5 fois et de reporter l'intervalle de confiance [min, moyenne, max].

Réponse : On mesure le temps écoulé, le temps CPU et l'énergie consommée avec : time.

		itératif			*cache aware*			*cache oblivious*		
N	***M***	*temps cpu*	*temps écoulé*	*energie*	*temps cpu*	*temps écoulé*	*energie*	*temps cpu*	*temps écoulé*	*e*
10000	*10000*	*0m1.6435s*	*0m.1647s*		*0m0.809 s*	*0m0.811 s*		*0m0.809 s*	*0m0.810 s*	
20000	*20000*	*0m6.606 s*	*0m6.608 s*		*0m3.211 s*	*0m3.21 s*		*0m3.219 s*	*0m3.223 s*	
30000	*30000*	*0m14.870 s*	*0m14.872 s*		*0m7.232 s*	*0m.7.234 s*		*0m7.467s*	*0m7.468 s*	
40000	*40000*	*0m26.448s*	*0m26.455 s*		*0m12.99 s*	*0m13.001 s*		*0m13.092 s*	*0m13.096s*	

Important: analyse expérimentale: *ces mesures expérimentales sont elles en accord avec les coûts analysés théroiquement (justifier) ? Quel algorithme se comporte le mieux avec valgrind et les paramètres proposés, pourquoi ?*

Réponse : D'après les résultats expérimentaux, on constate une cohérence entre les résultats théoriques et expérimentaux

6.3 (1 point) Extrapolation: estimation de la durée et de l'énergie pour la commande :

```
distanceEdition  GCA_024498555.1_ASM2449855v1_genomic.fna  77328790  20236404
                 GCF_000001735.4_TAIR10.1_genomic.fna      30808129  19944517
```

A partir des résultats précédents, le programme **préciser itératif/cache aware/ cache oblivious** ***est le plus performant pour la commande ci dessus (test 5); les ressources pour l'exécution seraient environ:*** **(préciser la méthode de calcul utilisée)**

- ***Temps cpu (en s) : ...***
- ***Energie (en kWh) : ... .***

Réponse :
D'après toute l'étude précédente, on peut conclure que le programme le plus performant est celui qui utilise la méthode cache-oblivious

Question subsidiaire: comment feriez-vous pour avoir un programme s'exécutant en moins de 1 minute ? **donner le principe en moins d'une ligne, même 1 mot précis suffit!**

Réponse :

Parallélisation