Questionnaire TP AOD 2023-2024 à compléter et rendre sur teide

Binôme : COUX Julian – GATIMEL Loan

1 Préambule 1 point

. 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 ?
Réponse:

L'erreur d'exécution observée lors du test 5 du programme récursif est causé par la saturation de la RAM. Cette situation survient en raison d'un grand nombre d'appels récursifs générés par le programme. Bien que le but de la mémorisation soit de limiter le nombre d'opérations, le nombre d'appels récursifs est tel que la mémoire saturem. Le problème a été formulé de telle manière que de nombreuses valeurs distinctes devaient être calculées, ce qui a entraîné une utilisation excessive de la mémoire, dépassant la capacité de RAM disponible.

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

Important. Dans toute la suite, on demande des programmes qui allouent un espace mémoire $O(N + M)$.

2 Programme itératif en espace mémoire $O(N + M)$ (5 points)

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

L'algorithme vise à construire une matrice imaginaire de taille N*M. Nous organisons cette matrice de manière à mettre en ligne la plus courte des deux chaînes. Chaque élément de cette matrice représente la distance d'édition minimale jusqu'à cet indice spécifique. Pour optimiser l'utilisation de la mémoire, nous ne stockons que deux lignes de la matrice à la fois. On parcourt le second tableau de gauche à droite car pour calculer un élément on a besoin de la case à gauche, la case diagonale à gauche et celle du dessus. Le résultat final se trouve dans la dernière case de la dernière ligne de la matrice.

Analyse du coût théorique de ce programme en fonction de N et M en notation $\Theta(...)$

1. place mémoire allouée (ne pas compter les 2 séquences X et Y en mémoire via `mmap`) : $\Theta(2 * min(N, M))$
2. travail (nombre d'opérations) : $\Theta(N + M + (N * M))$
3. nombre de défauts de cache obligatoires (sur modèle CO, y compris sur X et Y): $\frac{N*M}{L} + \frac{M}{L} + \frac{N}{L} + O(1)$
4. nombre de défauts de cache si $Z \ll \min(N, M)$: M*N

3 Programme cache aware (3 points)

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

L'algorithme découpe la matrice "imaginaire" en blocs de taille KxK de manière à ce qu'ils tiennent dans le cache. Nous itérons à travers ces blocs en suivant un parcours de gauche à droite et de haut en bas. Tout d'abord, comme dans l'algorithme itératif, nous stockons deux lignes par bloc pour les calculs. De plus, pour chaque bloc, nous devons mémoriser la colonne de droite pour le bloc suivant mais aussi les dernières lignes de tous les blocs d'une même ligne pour la ligne de blocs en dessous. Le résultat final se trouve dans la dernière case de la dernière ligne du dernier bloc.

Analyse du coût théorique de ce programme en fonction de N et M en notation $\Theta(...)$

1. place mémoire (ne pas compter les 2 séquences initiales X et Y en mémoire via `mmap`) : $\Theta(3K + min(N, M))$
2. travail (nombre d'opérations) : $\Theta(\frac{N*M}{K} + N * M)$
3. nombre de défauts de cache obligatoires (sur modèle CO, y compris sur X et Y): $\frac{N*M}{K}$
4. nombre de défauts de cache si $Z \ll \min(N, M)$:

4 Programme cache oblivious (3 points)

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

On découpe récursivement en 2 notre matrice "imaginaire" sur la dimension la plus grande. Cette division se poursuit jusqu'à ce qu'un seuil prédéterminé soit atteint, ce seuil étant défini pour éviter d'engendrer des coûts excessifs. On applique l'algorithme itératif sur le bloc et on stock certaines valeurs pour initialiser correctement les autres (cf: cache

aware). Afin d'utiliser les bonnes données sauvegardées au bon moment, sur les bons appels récursifs, on devra garder des indices en mémoire pour les repérer. Le résultat se trouve dans la dernière case du dernier bloc.

Analyse du coût théorique de ce programme en fonction de N et M en notation $\Theta(...)$)

1. place mémoire (ne pas compter les 2 séquences initiales X et Y en mémoire via `mmap`) :
2. travail (nombre d'opérations) :
3. nombre de défauts de cache obligatoires (sur modèle CO, y compris sur X et Y):
4. nombre de défauts de cache si $Z \ll \min(N, M)$:

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)

Nous devons prendre en compte les caractéristiques de la machine sur laquelle nous sommes pour calculer, avec une approximation des performances de notre programme, à partir de quel seuil il est péférable d'exécuter un algorithme récursif. Comme nous l'avons vu dans la première question, le CPU ne peut pas gérer un trop grand nombre d'appels récursifs, nous devons donc le limiter avec ce seuil.

6 Expérimentation (7 points)

Description de la machine d'expérimentation:
Processeur: Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz – Mémoire: 31Gi – Système: Linux ensipc120 6.2.0-35-generic

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 : 9437184 B, 64 B, 18-way associative

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

		récursif mémo			itératif		
N	M	#Irefs	#Drefs	#D1miss	#Irefs	#Drefs	#D1miss
1000	1000	217,227,240	122,131,724	4,921,660	99,811,661	46,347,576	277,676
1000	1000	217,227,240	122,131,724	4,921,660	99,811,661	46,347,576	277,676
2000	1000	433,404,634	243,410,956	11,018,505	199,343,109	92,594,872	548,649
4000	1000	867,176,732	487,375,212	23,218,144	398,405,737	185,089,376	1,090,597
2000	2000	867,168,330	487,897,907	19,883,947	398,359,625	185,062,334	1,078,947
4000	4000	3,465,891,043	1,950,558,102	79,958,914	1,592,379,505	739,831,358	4,279,449
6000	6000	7,796,351,505	4,387,993,913	180,254,163	3,582,298,015	1,664,386,400	9,625,618
8000	8000	13,857,981,052	7,799,957,484	321,047,129	6,368,115,143	2,958,727,456	17,060,190

		cache aware			cache oblivious		
N	M	#Irefs	#Drefs	#D1miss	#Irefs	#Drefs	#D1miss
1000	1000	120,842,188	61,212,425	10,401			
1000	1000	120,842,188	61,212,425	10,401			
2000	1000	241,377,364	122,315,271	14,116			
4000	1000	482,438,689	244,517,297	21,399			
2000	2000	481,143,566	244,134,899	22,626			
4000	4000	2,031,548,705	1,034,057,776	308,315			
6000	6000	4,707,819,719	2,355,318,997	10,237,257			
8000	8000	7,584,501,746	3,874,372,337	17,636,390			

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 ?

Les mesures expérimentales sont plutôt en accord avec les coûts analysés théoriquement car on voit bien que le cache aware minimise les D1miss. Cependant au dela de N et M >= 6000 les défauts de cache sont plus grand que l'itératif. Ce n'est pas normal et cela s'explique car nous n'avons pas réussi à bien déterminer K lorsque M et N sont plus grand que la taille du cache.

L'algorithme qui se comporte le mieux est le cache aware quand N et M < 6000, car nous prenons en entrée la taille du cache et calculons K comme il faut.

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 : `time`

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].

		itératif			cache aware			cache oblivious		
N	M	temps cpu	temps écoulé	energie	temps cpu	temps écoulé	energie	temps cpu	temps écoulé	energie
10000	10000	0.751	0.763	6206209.8	0.892	0.891	19836826.6			
20000	20000	3.066	3.072	25689741	3.410	3.399	80583778			
30000	30000	6.837	6.843	58998835.8	7.639	7.633	196070372.2			
40000	40000	12.115	12.121	103654239	13.554	13.538	321885772.6			

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 ?

Nous avons mesuré l'énergie en utilisant un script python qui compare l'énergie du CPU0 avant et après l'exécution du programme. Nous faisons ça 5 fois d'affilé pour prendre la valeur moyenne. Comme précedemment, les résultats sont en désaccord avec nos estimations puisque le programme itératif est plus rapide et moins couteux en énergie que le programme cache-aware. C'est parce que les tests réalisés se font avec des valeurs ¿= 6000. Or, nous avons montré que au-dela de ce seuil, notre programme cache-aware n'est plus efficace car N et M dépassent la taille du cache. L'algorithme qui se comporte le miens avec ces résultats est le itératif. Cependant, si on prends en compte la taille des allocations mémoire, le programme cache-aware est largement meilleur.

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) : ... .

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!*

"Multi-Thread"