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

Binôme (BOUIHI Hamza – VERILLON Matisse) : .....

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: Etant donnée que les séquences du test 5 sont très longues, la mémoire qui est demandé par ce programme récursif pour allouer une mémoire de taille N x M est beaucoup trop élevé ce qui est a l'origine de l'erreur d'exécution, une OOM (out of memory).

```
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.

On sait que $\phi(M, N) = 0$ et que $\phi(M, j-1) = 2 \cdot \beta_{y_{j-1}} + \phi(M, j)$ donc on peut construire la dernière ligne de la matrice (du dernier élément au premier) si on connait le dernier élément de celle-ci. On se rend compte qu'en utilisant les autres propriétés de $\phi(i, j)$ on peut faire ça à chaque ligne et on peut également passer de L_i à L_{i-1} (grâce à un tampon) dans le but de remonter jusqu'à $\phi(0, 0)$ qui est la distance d'édition.

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`) : $\Theta(N)$
2. travail (nombre d'opérations) : $\Theta(M \cdot N)$
3. nombre de défauts de cache obligatoires (sur modèle CO, y compris sur X et Y): $\Theta\left(\frac{N \cdot M}{L}\right)$,$\Theta\left(\frac{M}{L}\right)$,$\Theta\left(\frac{N}{L}\right)$
4. nombre de défauts de cache si $Z \ll \min(N, M)$: $\Theta\left(M \cdot L\right)$,$\Theta\left(\frac{M}{L}\right)$,$\Theta\left(\frac{M \cdot N}{L}\right)$

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.

Afin de rendre le programme cache aware, on va diviser le tableau en bloc de taille K (inspiration TD2), on va ensuite calculer de manière itérative au sein des blocs. Cela nécessite donc de modifier notre approche afin de retenir la ligne (taille N+1) dont chaque portion de longueur K est modifiée au sein des blocs. De plus, il est nécessaire de conserver la colonne la plus à gauche d'un bloc et l'élément haut gauche du bloc en bas à droite afin de calculer le suivant.

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`) : $\Theta(N + K)$ avec $K = \sqrt{Z}$
2. travail (nombre d'opérations) : $\Theta(M \cdot N)$
3. nombre de défauts de cache obligatoires (sur modèle CO, y compris sur X et Y):$\Theta\left(\frac{M \cdot N}{L}\right)$,$\Theta\left(\frac{M}{L}\right)$,$\Theta\left(\frac{N}{L}\right)$
4. nombre de défauts de cache si $Z \ll \min(N, M)$: $\Theta\left(\frac{M \cdot N}{L}\right)$,$\Theta\left(\frac{M \cdot N}{L \cdot \sqrt{Z}}\right)$,$\Theta\left(\frac{M \cdot N}{L \cdot \sqrt{Z}}\right)$

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.

Pour ce programme, on doit le rendre cahce oblivious donc indépendant des paramètres du cache, on va donc utiliser du blocking (inspiration TD2) afin de diviser récursivement de la matrice en colonnes jusqu'à ce qu'elles rentrent dans le cache puis à procéder de manière itérative dans ces colonnes afin d'éviter des problèmes de récursion comme dans la question 1. Pour cela, on utilise une fonction récursive blockingREC afin de diviser le tableau jusqu'à des colonnes de taille convenable lorsqu'elles sont supérieurs à un seuil et lorsque la condition d'arrêt est vérifiée, calcule itérativement la colonne (sous-colonne par sous-colonne).

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)

Réponse: On aurait choisi ce seuil afin que la succession d'appels récursifs ne fasse pas déborder la pile et provoque un stack overflow. On peut le trouver en recherchant empiriquement par dichotomie, en affinant vers la valeur donnant de meilleurs résultats.

6 Expérimentation (7 points)

Description de la machine d'expérimentation:
Processeur: Intel Core i5-7500 CPU @ 3.40GHz × 4 – Mémoire: 32.0 GiB – Système: Ubuntu 22.04.3 LTS, X11

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.

		récursif mémo			itératif		
N	M	#Irefs	#Drefs	#D1miss	#Irefs	#Drefs	#D1miss
1000	1000	217,204,783	122,122,987	4,935,111	105,877,004	49,490,264	147,913
2000	1000	433,382,177	243,402,219	11,045,942	210,790,502	98,178,570	290,756
4000	1000	867,154,275	487,366,475	23,273,379	422,018,944	196,956,808	576,456
2000	2000	867,145,873	487,889,170	19,941,850	422,861,561	197,818,498	572,450
4000	4000	3,465,868,586	1,950,549,365	80,219,305	1,690,722,823	791,130,424	2,262,056
6000	6000	7,796,329,048	4,387,985,176	180,813,797	3,803,765,413	1,779,974,748	5,074,029
8000	8000	13,857,958,595	7,799,948,747	322,133,239	6,761,675,090	3,164,038,584	9,020,104

		cache aware			cache oblivious		
N	M	#Irefs	#Drefs	#D1miss	#Irefs	#Drefs	#D1miss
1000	1000	121,318,665	58,747,444	9,461			
2000	1000	241,645,733	116,678,894	13,706			
4000	1000	483,700,803	233,943,123	22,075			
2000	2000	484,561,832	234,814,094	26,713			
4000	4000	1,937,262,188	938,966,540	103,468			
6000	6000	4,358,286,478	2,112,497,116	154,953			
8000	8000	7,747,609,582	3,755,257,784	347,820			

[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.

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

En effet, ces mesures sont cohérentes. Le nombre de défauts de cache de l'algorithme cache aware est bien moins important que celui de l'itératif. De plus, si l'on regarde l'évolution du défaut de cache par rapport au produit $M \cdot N$, il évolue linéairement en fonction de $M \cdot N$ dans le cas de l'algorithme itératif (en cohérence avec les résultats théoriques). En revanche, ce n'est pas vrai dans le cas de l'algorithme cache aware (Toujours en cohérence avec la théorie).

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].

		itératif			*cache aware*			*cache oblivious*		
N	*M*	*temps cpu*	*temps écoulé*	*energie*	*temps cpu*	*temps écoulé*	*energie*	*temps cpu*	*temps écoulé*	*energie*
10000	*10000*	*1.22124*	*1.27092*	*6.3356e-06*	*1.32752*	*1.32775*	*5.65342e-06*			
20000	*20000*	*4.94093*	*4.94152*	*2.34858e-05*	*5.31888*	*5.32065*	*2.38417e-05*			
30000	*30000*	*11.0882*	*11.0979*	*5.55166e-05*	*12.298*	*12.2989*	*5.78109e-05*			
40000	*40000*	*18.756*	*18.8458*	*8.54012e-05*	*22.2578*	*22.6901*	*0.000108162*			

Important: analyse expérimentale: *ces mesures expérimentales sont elles en accord avec les coûts analysés théoriquement (justifier) ?*

Réponse: *Les résultats sont bien cohérents, l'algorithme itératif est plus rapide et moiins couteux en énergie car il y a moins d'opération que dans l'algorithme cache Aware. On sacrifie de la performance énergétique et temporelle au profit d'une optimisation de l'utilisation du cache. En revanche les performances temporelles restent acceptable car une optimisation de l'utilisation du cache permet également un gain de temps.*

Quel algorithme se comporte le mieux avec valgrind et les paramètres proposés, pourquoi ?

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 cache aware 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!

Réponse: *On pourrait implémenter un algorithme utilisant le principe du parallélisme.*