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

Binôme (BENABDELLAH Achraf – EL-ACHAB Aymane) : .....

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: En essayant le programme récursif sur le test 5, une certaine "Erreur 137" apparait. Une petite recherche permet de comprendre que cette erreur signifie une consommation excessive des ressources mémoires pour executer le code, ce qui est logique en considérant la taille des fichiers du test 5

```
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.
Afin de calculer $\varphi(0,0)$, le programme récursif stocke tous les $\varphi(i,j)$ dans une matrice. Notre programme n'utilisera qu'un seul tableau (une colonne de taille N+1 initialisée par les $\varphi(M,j)$) et une variable *tmp* (qui gardera l'information sur $\varphi(i+1,j+1)$ pour chaque itération). On écrasera toute donnée qui devient inutile. Le calcul se déroulera en progressant de colonne en colonne, jusqu'à atteindre la valeur de $\varphi(0,0)$.

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`) :
   Espace mémoire alloué : $O(N)$ où N est la plus petite taille des deux séquences.

2. travail (nombre d'opérations) :
   Boucle d'initialisation : $O(N)$
   Boucle Principale : $O(MN)$
   En total, on est à $O(MN)$ en terme d'opérations.

3. nombre de défauts de cache obligatoires (sur modèle CO, y compris sur X et Y):
   Si Z est suffisement grand, le nombre de défauts de cache est :
   - Initialisation de slidingCol : $\frac{2N}{L}$
   - nb de miss sur X : $\frac{M}{L}$, nb de miss sur Y : $O(1)$ (toujours dans le cache)
   - nb de miss sur slidingCol dans la Boucle: $O(1)$ (toujours dans le cache)
   – En total : $\frac{M+2N}{L} + O(1)$

4. nombre de défauts de cache si $Z \ll \min(N,M)$:
   Si Z $\ll \min(N,M) = N$, le nombre de défauts de cache est :
   - Initialisation de slidingCol : $\frac{2N}{L}$
   - nb de miss sur X : $M + O(1)$, nb de miss sur Y : $\frac{MN}{L}$
   - nb de miss sur slidingCol dans la Boucle: $\frac{MN}{L}$
   – En total : $M + 2\frac{MN+N}{L} + O(1)$

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 calculer $\varphi(0,0)$ en minimiseant les défaults de cache, notre programme calcule les valeurs d'un bloc carré de taille K^2 (choisi pour garantir qu'un block tient dans le cache) de la matrice. Mais on ne stocke que la valeur des colonnes dans un tableau de taille $O(N)$ en écrasant les valeurs inutiles. Pour garantir qu'il n'y a pas perte d'information, on stocke les éléments nécessaire pour le calcul du block suivant dans deux tableau: tmp de taille K et memLine de taille $O(M)$.

1. place mémoire allouée (ne pas compter les 2 séquences X et Y en mémoire via `mmap`) :
   Espace mémoire alloué : $O(M+N+K)$ où K est la taille d'un bloc.

2. travail (nombre d'opérations) :
   Boucle d'initialisation : $O(N+M)$
   Boucle Principale : $O(\frac{MN}{K^2}K^2) = O(MN)$
   En total, on est à $O(MN)$ en terme d'opérations.

3. nombre de défauts de cache obligatoires (sur modèle CO, y compris sur X et Y):
   Si Z est suffisement grand, le nombre de défauts de cache est :
   - Initialisation de slidingCol et memoLine: $2\frac{N+M}{L}$
   - nb de miss sur X : $O(1)$ (toujours dans le cache), nb de miss sur Y : $O(1)$ (toujours dans le cache)

- nb de miss sur tmp dans la Boucle: $\frac{K}{L}$
– En total : $2\frac{M+N}{L} + \frac{K}{L} + O(1)$

4. nombre de défauts de cache si $Z \ll \min(N, M)$:
Si Z $\ll \min(N, M) = N$, le nombre de défauts de cache est :
- Initialisation de slidingCol et memoLine: $2\frac{N+M}{L}$
- nb de miss sur X et memoLine dans la boucle : $\frac{MN}{K^2} \cdot \frac{K}{L}$
- nb de miss sur Y et slidingCol dans la boucle : $\frac{MN}{K^2} \cdot \frac{K^2}{L}$
- nb de miss sur tmp dans la boucle : $2\frac{MN}{K^2} \cdot \frac{K}{L}$
– En total : $2\frac{M+N}{L} + 2\frac{MN}{L} \cdot (1 + \frac{2}{K}) + O(1)$
On cherchera donc à maximiser K de sorte qu'un bloc de taille K^2 tienne en mémoire, d'où le choix $K = \sqrt{Z} - O(1)$

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.
Notre programme découpe le problème récursivement jusqu'à ce que la taille du block à calculer soit inférieure à un seuil S et applique le traitement itératif. S est choisi pour eviter un débordement de mémoire (Stack Overflow) et pour garantir que le block à traiter tient dans le cache Nous stockons les éléments des colonnes calculés dans un tableau de taille $O(N)$ en écrasant les valeurs inutiles. Les valeurs necessaires pour le calcul des blocks pendant le remontée recursive sont stockés dans deux tableaux de taille $O(M)$ chacun.

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`) :
Espace mémoire alloué : $O(2M + N)$.

2. travail (nombre d'opérations) :
Boucle d'initialisation : $O(N + M)$
Boucle Principale : $O(\frac{MN}{K^2}K^2) = O(MN)$
En total, on est à $O(MN)$ en terme d'opérations.

3. nombre de défauts de cache obligatoires (sur modèle CO, y compris sur X et Y):
Si Z est suffisement grand, le nombre de défauts de cache est :
– En total : $\frac{3M+2N}{L} + O(1)$

4. nombre de défauts de cache si $Z \ll \min(N, M)$:
Lorsque on franchit le seuil S, le nombre de défauts du programme itératif est de l'ordre de $O\left(\frac{2S}{L} + \frac{2S^2}{L}\right)$. Au pire des cas, le nombre de blocs sera de l'ordre de $O\left(\log\left(\frac{M}{S}\right)\right)$. Au total, donc $Q(M, N, L, Z)$ lorsque $Z \ll \min(M, N) = N$ est : $O\left(\log\left(\frac{M}{S}\right) \cdot \frac{S^2}{L}\right)$.

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) On choisit le seuil d'arrêt dans l'algorithme cache-oblivious empiriquement. Il représente le point où l'algorithme cesse de diviser le problème en sous-problèmes plus petits et bascule vers un algorithme itératif de base. Pour les PC de l'Ensimag, nous avons pris S = 200

6 Expérimentation (7 points)

Description de la machine d'expérimentation: *ENSIPC306*
Processeur: Intel(R) Core(TM) i5-10500 CPU @ 3.10GHz – Mémoire: 32GiB – Système: 22.04.1-Ubuntu

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 : 12582912 B, 64 B, 24-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	217,185,326	122,119,390	4,935,698	117,682,603	69,099,897	150,823
2000	1000	433,362,720	243,398,622	11,044,306	234,419,435	137,401,125	296,405
4000	1000	867,134,818	487,362,878	23,270,011	469,294,537	275,405,199	587,611
2000	2000	867,126,190	487,885,543	19,939,882	470,136,235	276,268,970	583,611
4000	4000	3,465,848,903	1,950,545,738	80,217,738	1,879,916,371	1,104,986,826	2,306,764
6000	6000	7,796,309,365	4,387,981,549	180,809,120	4,229,527,314	2,486,211,263	5,174,114
8000	8000	13,857,938,912	7,799,945,120	322,132,951	7,518,771,848	4,419,745,069	9,183,090

		cache aware			cache oblivious		
N	M	#Irefs	#Drefs	#D1miss	#Irefs	#Drefs	#D1miss
1000	1000	137,962,533	77,267,636	9,887	137,645,143	77,132,617	18,724
2000	1000	274,978,523	153,736,353	15,801	274,343,515	153,466,311	26,266
4000	1000	550,411,130	308,074,800	20,077	549,141,621	307,535,313	30,769
2000	2000	551,229,013	308,932,822	27,273	549,977,314	308,401,076	23,563
4000	4000	2,204,117,335	1,235,567,952	67,808	2,199,265,473	1,233,518,484	65,105
6000	6000	4,958,852,729	2,779,963,390	134,752	4,946,142,756	2,774,118,683	242,628
8000	8000	8,815,237,966	4,941,921,914	358,866	8,796,141,389	4,933,879,097	257,432

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 résultats empiriques montrent que le nombre de cache misses décroit énormément avec les nouvelles implémentations. Le programme cache oblivious est généralement le meilleur, puisqu'il est conçu de sorte qu'il soit optimal quelque soit le niveau de cache.

6.2 (3 points) Sans valgrind

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

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

		itératif			cache aware			cache oblivious		
N	M	temps cpu	temps écoulé	energie	temps cpu	temps écoulé	energie	temps cpu	temps écoulé	energie
10000	10000	1.20799	1.20795	4.63446e-06	1.26864	1.26863	5.00099e-06	1.28422	1.28364	5.32164e-06
20000	20000	4.89200	4.89131	1.99841e-05	5.13932	5.13926	2.05629e-05	5.17938	5.17575	2.07550e-05
30000	30000	11.0281	11.0222	4.60374e-05	11.5283	11.5278	4.78964e-05	11.6675	11.6637	4.87354e-05
40000	40000	19.6529	19.6525	8.19819e-05	20.6400	20.6394	8.91589e-05	20.8331	20.8282	8.76766e-05

Important: analyse expérimentale: ces mesures expérimentales sont elles en accord avec les coûts analysés théroiquement (justifier) ? Les résultats experiementaux montrent que les trois programmes sont à peu près équivalent en terme de travail. Certes, le nombre de cache misses est beaucoup plus inférieur pour les programmes cache aware et cache oblivious, mais on ajoute un coût d'opérations, et peut-être le compilateur optimise mieux le programme itératif.

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) : 3500000
- Energie (en kWh) : 17.5 .

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!* Il faut essayer de prendre compte des dépendences entre les calculs et écrire un programme qui calcule les blocs en parallèle en applicant le principe du parallélisme.