Questionnaire TP AOD 2023-2024 a completer et rendre sur teide

Binome (JAAfOURAMouez – ZINE Ouadii) : .....

1 Preambule 1 point

. Le programme recursif avec memoisation fourni alloue une memoire de taille $N.M$. Il genere une erreur d'execution sur le test 5 (c-dessous) . Pourquoi ?
Reponse: Le programme echoue par une erreur "Out of Memory" car il alloue une memoire de taille $N.M$ pour la memoisation, ce qui depasse la capacite du systeme. Lorsque la memoire consommee devient trop elevee, le systeme d'exploitation met fin au processus pour preserver les ressources.

```
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 memoire $O(N + M)$.

2 Programme iteratif en espace memoire $O(N + M)$ (5 points)

Expliquer tres brievement (2 a 5 lignes max) le principe de votre code, la memoire utilisee, le sens de parcours des tableaux.
Reponse: Pour l'implementation iterative, on utilise deux tableaux : ColumnM de taille M et ColumnN de taille N, ainsi que des variables temporaires right-element et diagonal-element pour stocker les valeurs necessaires au calcul des prochains $\phi(i, j)$. On initialise ColumnM avec $\phi(i, N)$ pour $0 \leq i \leq M$ et ColumnN avec $\phi(M, j)$ pour $0 \leq j \leq N$, et on utilise ces valeurs pour calculer $\phi(0, 0)$ progressivement.

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

1. place memoire allouee (ne pas compter les 2 sequences X et Y en memoire via `mmap`) : $N + M$
2. travail (nombre d'operations) : $\Theta(N.M)$
3. nombre de defauts de cache obligatoires (sur modele CO, y compris sur X et Y):$\Theta(2\frac{N+M}{L})$
4. nombre de defauts de cache si $Z \ll \min(N, M)$: $\Theta(\frac{N+M}{L} + 2N + 3N.M)$

3 Programme cache aware (3 points)

Expliquer tres brievement (2 a 5 lignes max) le principe de votre code, la memoire utilisee, le sens de parcours des tableaux.
Reponse: Dans cette impementation de l'algorithme cache-aware pour calculer la distance d'edition, la matrice est subdivisee en blocs pour optimiser l'acces en memoire. Deux tableaux, ColumnM et ColumnN, stockent les couts intermediaires pour chaque ligne et colonne, et des variables temporaires (right-element et diagonal-element) permettent de conserver les valeurs necessaires pour les calculs successifs des eements $\phi(i, j)$. On remplit ColumnM et ColumnN en initialisant leurs valeurs depuis les indices finaux vers le debut, afin de minimiser les couts de decalage.

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

1. place memoire (ne pas compter les 2 sequences initiales X et Y en memoire via `mmap`) :$N + M$
2. travail (nombre d'operations) : $\Theta(N.M)$
3. nombre de defauts de cache obligatoires (sur modele CO, y compris sur X et Y):$\Theta(2\frac{N+M}{L})$
4. nombre de defauts de cache si $Z \ll \min(N, M)$: $\Theta(2\frac{N+M}{L})$

4 Programme cache oblivious (3 points)

Expliquer tres brievement (2 a 5 lignes max) le principe de votre code, la memoire utilisee, le sens de parcours des tableaux.
Reponse:La fonction EditDistance-NW-Oblivious calcule la distance d'edition entre deux sequences en utilisant une approche recursive pour diviser le calcul en blocs. La fonction principale initialise les tableaux columnM et columnN et appelle la fonction recursive EditDistance-NW-Oblivious-Rec. Cette derniere divise la tache en sous-blocs jusqu'a atteindre une taille limite definie (S). Pour les blocs de taille inferieure a S, la fonction Process-Block calcule directement la distance d'edition en modifiant les valeurs de columnM et columnN pour les indices concernes.

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

1. place memoire (ne pas compter les 2 sequences initiales X et Y en memoire via `mmap`) : $N + M$
2. travail (nombre d'operations) : $\Theta(N.M)$
3. nombre de defauts de cache obligatoires (sur modele CO, y compris sur X et Y): $\Theta(2\frac{N+M}{L})$
4. nombre de defauts de cache si $Z \ll \min(N, M)$:

$$\begin{cases} \Theta(2\frac{N+M}{L}) & \text{si } S \leq L \\ \Theta(\frac{4N \cdot M}{S \cdot L}) & \text{sinon} \end{cases}$$

5 Reglage du seuil d'arret recursif du programme cache oblivious (1 point)

Comment faites-vous sur une machine donnee pour choisir ce seuil d'arret? Quelle valeur avez vous choisi pour les PC de l'Ensimag? (2 a 3 lignes)
Reponse: Le seuil d'arret pour un programme cache-oblivious est determine en tenant compte des caracteristiques specifiques de la machine cible, de l'architecture du cache, et des performances visees. Pour les PC de l'Ensimag, apres plusieurs essais et en utilisant une methode de diviser pour regner, nous avons trouve que S=128 minimise les defauts de cache, optimisant ainsi l'efficacite des acces memoire.

6 Experimentation (7 points)

Description de la machine d'experimentation: ENSIPC
Processeur: Intel(R) Core(TM) i5-8500 CPU @ 3.00GHz
Memoire: Total : 31Gi, Used : 1.2Gi,Free : 28Gi Shared : 12Mi Buffers/Cache : 1.1Gi, Available : 29Gi
Systeme: Ubuntu 22.04.4 LTS

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 parametres du cache LL de second niveau est : ... *mettre ici les parametres: soit ceux indiques ligne 3 du fichier cachegrind.out.(pid) genere par valgrind: soit ceux par defaut, soit ceux que vous avez specifies a la main* [1] *pour LL.*

Le tableau ci-dessous est un exemple, complete avec vos resultats et ensuite analyse.

		recursif memo			iteratif		
N	M	#Irefs	#Drefs	#D1miss	#Irefs	#Drefs	#D1miss
1000	1000	217,185,853	122,119,493	4,927,438	89,365,169	38,730,385	147,405
1000	1000	217,185,853	122,119,493	4,927,438	89,365,169	38,730,385	147,405
2000	1000	433,363,247	243,398,725	11,026,668	171,754,063	74,751,367	285,584
4000	1000	867,135,345	487,362,981	23,226,630	337,134,026	146,956,961	562,926
2000	2000	867,126,943	487,885,676	19,900,542	356,359,236	154,491,588	566,017
4000	4000	3,465,849,656	1,950,545,871	80,007,549	1,423,646,102	617,248,572	2,235,409
6000	6000	7,796,310,118	4,387,981,682	180,325,643	3,202,010,161	1,388,325,584	5,018,996
8000	8000	13,857,939,665	7,799,945,253	321,220,326	5,691,283,775	2,467,722,065	8,906,961

		cache aware			cache oblivious		
N	M	#Irefs	#Drefs	#D1miss	#Irefs	#Drefs	#D1miss
1000	1000	93,986,400	41,025,965	7,639	91,976,208	40,427,867	7,583
1000	1000	93,986,400	41,025,965	7,639	91,976,208	40,427,867	7,583
2000	1000	181,044,214	79,365,359	10,324	181,436,176	79,956,091	10,550
4000	1000	355,710,320	156,182,879	15,646	360,706,761	159,119,233	20,380
2000	2000	374,939,501	163,719,522	15,088	369,200,107	162,280,111	22,680
4000	4000	1,497,947,669	654,150,335	44,156	1,480,507,388	650,685,296	81,953
6000	6000	3,312,655,147	1,449,539,168	88,615	3,344,405,554	1,470,375,121	105,645
8000	8000	5,988,449,428	2,615,308,502	154,568	5,929,897,193	2,606,144,502	240,005

Important: analyse experimentale: ces mesures experimentales sont elles en accord avec les couts analyses theroiquement (justifier) ? Quel algorithme se comporte le mieux avec valgrind et les parametres proposes, pourquoi ?

Reponse: L'analyse experimentale des performances, en utilisant les mesures de Valgrind, est globalement en accord avec les couts theoriques analyses pour les 3 algorithmes. En effet, les resultats montrent que l'augmentation theorique des defauts lorsque la memoire necessaire excede la capacite de cache pour l'algorithme iteratif. Pour cache aware et cache oblivious on peut constater qu'on a pas de grande difference au niveau des defauts de cache en passant de $N + M \leq Z$ a $N + M \geq Z$. Algorithme Cache-aware comme Meilleur Choix(l'algorithme cache-aware optimise le mouvement des donnees en exploitant la hierarchie de la memoire cache, ce qui se traduit par de meilleures performances sous Valgrind et les parametres specifies. La strategie de decoupage en blocs de cet algorithme lui permet de reduire les deplacements de cache de maniere efficace, rendant son execution plus rapide et plus efficace en termes de gestion des acces memoire)

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

L'algorithme cache-aware semble se comporter le mieux avec Valgrind et les parametres proposes. Cela s'explique par sa gestion efficace des acces memoire via la subdivision en blocs, qui optimise les deplacements de cache en fonction de la structure memoire de la machine. Il maintient ainsi un faible nombre de defauts de cache (D1miss) par rapport aux autres algorithmes, ce qui ameliore les performances globales.

6.2 (3 points) Sans valgrind, par execution 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 ecoule, le temps CPU et l'energie consommee avec : `time`

L'energie consommee sur le processeur peut etre estimee en regardant le compteur RAPL d'energie (en microJoule) pour chaque core avant et apres l'execution et en faisant la difference. 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 resultat fiable/reproductible (si variailite), il est preferable de faire chaque mesure 5 fois et de reporter l'intervalle de confiance [min, moyenne, max].

		iteratif			cache aware			cache oblivious		
N	M	temps cpu	temps ecoule	energie	temps cpu	temps ecoule	energie	temps cpu	temps ecoule	energie
10000	10000	0.9296	0.9608	5.330e-06	0,9238	0,9274	6.013e-06	0.9302	0,9384	6.650e-06
20000	20000	3.7372	3.7454	6.707e-06	3,7582	3,7602	1.926e-05	3.7290	3.7572	2.082e-05
30000	30000	8.4370	8.4598	5.031e-05	8,4578	8,4598	5.237e-05	8.4402	8.4491	6.27e-05
40000	40000	15.0651	15.0851	15.498e-05	15.0611	15.0842	7.760e-05	15.0733	15.0848	9.387e-05

Important: analyse experimentale: ces mesures experimentales sont elles en accord avec les couts analyses theroiquement (justifier) ? Quel algorithme se comporte le mieux avec valgrind et les parametres proposes, pourquoi ?

Reponse: L'algorithme iteratif consomme moins d'energie et s'execute plus rapidement que les versions cache-aware et cache-oblivious, ce qui est en accord avec les attentes theoriques. En reduisant la memoire utilisee, il minimise les mouvements de donnees, mais peut potentiellement augmenter le temps d'execution et la consommation d'energie si la memoire est encore reduite davantage.En termes de performance sous Valgrind avec les paramÃ¨tres specifies, l'algorithme iteratif se distingue. Son utilisation optimale de la memoire permet de maintenir un bon compromis entre consommation d'energie et rapidite d'execution. Bien qu'il ne soit pas explicitement optimise pour l'hierarchie de cache comme les algorithmes cache-aware et cache-oblivious, il conserve un profil efficace grace a une gestion simplifiee des donnees, ce qui lui permet de surpasser les autres en termes de consommation d'energie et de temps de calcul.

6.3 (1 point) Extrapolation: estimation de la duree et de l'energie pour la commande :

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

A partir des resultats precedents, le programme *iteratif* est le plus performant pour la commande ci dessus (test 5); les ressources pour l'execution seraient environ: *regression lineaire*

- Temps cpu (en s) : 50,935 s $\approx$ 14h15min
- Energie (en kWh) : 0.5326 kWh .

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

Reponse: Parallelisme