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

Binôme : JEANNESSON Tom, ABDELAZIZ Adame

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 (ci-dessous) . Pourquoi ?

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

Réponse:
L'erreur d'exécution lors du test 5 est probablement due à une insuffisance de mémoire. L'algorithme alloue une mémoire de taille N.M, où N et M sont les longueurs des deux séquences à aligner. Si ces séquences sont très longues, comme c'est le cas pour le test 5, la taille de la mémoire nécessaire peut dépasser la mémoire disponible, ce qui entraîne une erreur d'exécution. De plus, nous pouvons également rencontrer des erreur "Stack Overflow" si nous déscendons trop dans les appels récursifs, ce qui est surement le cas ici.

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.

Soit le graphe des dépendances définit comme une grille ϕ de dimension $M \times N$. Nous avons $\phi(M, N) = 0$ qui est la valeur initiale et nous souhaitons obtenir la valeur de $\phi(0, 0)$. Pour cela, nous calculons les colonnes de la grille une par une (de M jusqu'à 0) en descendant dans les lignes (de N jusqu'à 0). Nous avons donc un tableau Y_col de taille N qui sera mis à jour "in place" à chaque colonne calculée, avec une variable *prev_value* pour sauvegarder la valeur écrasée.

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`) :

   La place mémoire allouée est $\Theta(N)$, car nous avons besoin d'uniquement la colonne Y_{col}, que nous mettrons à jour "in place". Le coût des variables déclarées statiquement est négligeable.

2. travail (nombre d'opérations) :

   Nous parcourons une seule fois chaque case du tableau. Le travail est donc de $\Theta(N \times M)$.

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

   Nous avons $X.length = M$ et $Y.length = N$.
   Pour le calcul d'une colonne de taille N nous faisons $\lceil \frac{N}{L} \rceil$ défauts de câche. Nous posons $X > Y$.
   Pour Y nous avons le même nombre de défauts de câche que la colonne.
   Pour X nous avons $\lceil \frac{M}{L} \rceil$.
   Finalement : $Q_{obligatoires}(M, N, Z, L) = \lceil 2\frac{N}{L} + \frac{M}{L} \rceil$

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

   Dans ce cas-là, nous supposons qu'à chaque nouveau calcul de colonne (M fois), nous avons des défauts de câches sur X et Y_{col}, soit $2\frac{M \times N}{L}$ en plus des $\frac{M}{L}$ défauts obligatoires sur X.
   Nous avons donc: $Q(M, N, Z, L) = \lceil 2\frac{M \times N}{L} + \frac{M}{L} \rceil$

3 Programme câche 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'idée est de découper la grille en blocs de dimension $(K \times K)$ tels que $4K << Z$. Nous nécessitons $4K << Z$ car nous devons être en capacité de stocker dans le câche une ligne de longueur K que l'on nomme K_{row}, K éléments de X, K éléments de Y, et une section de Y_{col} de longueur K. À l'intérieur d'un bloc, le sens et la méthode de calcul des valeurs sont les mêmes que pour l'implémentation itérative. Une fois un bloc calculé, nous traitons le bloc du dessous en utilisant K_{row} qui a enregistré les valeurs de la dernière ligne du bloc précédent. S'il n'y en a pas, nous remontons et nous passons à la colonne de blocs suivante, ainsi, nous travaillons uniquement sur des morceaux colonnes de tailles K ce qui réduit considérablement les défauts de câche.

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`) :

   Nous avons encore la colonne de taille N, même si l'on travaille dessus bloc par bloc, et en plus de ça nous avons K_{row} de taille K. Ce qui nous donne $\Theta(N + K)$

2. travail (nombre d'opérations) :

   Idem que pour l'implémentation itérative : $\Theta(N \times M)$

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

   On doit charger, X, Y, Y_{col}, et K_{row}.
   Donc: $Q_{obligatoires}(M, N, Z, L) = \lceil 2\frac{N}{L} + \frac{M}{L} + \frac{K}{L} \rceil$

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

   Pour chaque bloc de taille $K \times K$ on a $2\frac{K}{L}$ défaut de câche : sur Y_{col} et Y ($4K << Z$ un bloc rentre dans le câche).
   Or, nous avons $\frac{M \times N}{K^2}$ blocs.
   Nous avons donc $\frac{M \times N}{K^2} \times 2\frac{K}{L} = 2\frac{M \times N}{K \times L}$ défauts de câche, en plus des $\frac{M}{L} + \frac{K}{L}$ défauts obligatoires sur X et K_{row}.
   Donc: $Q(N, M, Z, L, K) = \lceil 2\frac{N \times M}{K \times L} + \frac{M}{L} + \frac{K}{L} \rceil$

4 Programme câche 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.

Le principe est le même que le câche aware à la différence où nous allons couper les blocs récursivement jusqu'à un certain seuil S. Pour ce faire, nous découpons récursivement la grille verticalement jusqu'à atteindre le seuil S, puis nous faisons de même horizontalement. Nous obtenons alors un bloc de taille $S \times S$, avec $S << Z$. Nous résolvons ensuite ce bloc de la même façon que pour l'implémentation câche aware. Et nous traitons les blocs dans le même ordre que pour le câche aware.

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`) :

   Nous avons $\Theta(M + N)$. À la différence du câche aware, nous avons une taille M pour K_{row} (devenu X_{row}) pour une grande simplicité d'implémentation.

2. travail (nombre d'opérations) :

   Idem que pour l'implémentation itérative : $\Theta(N \times M)$

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

   On doit charger, X, Y, Y_{col}, et X_{row}.
   Donc: $Q_{obligatoires}(M, N, Z, L) = \lceil 2\frac{N}{L} + 2\frac{M}{L} \rceil$

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

   Pour chaque bloc de taille $S \times S$ on a $2\frac{S}{L}$ défaut de câche : sur Y_{col} et Y.
   Or, nous avons $\frac{M \times N}{S^2}$ blocs.
   Nous avons donc $\frac{M \times N}{S^2} \times 2\frac{S}{L} = 2\frac{M \times N}{S \times L}$ défauts de câche, en plus des $2\frac{M}{L}$ défauts obligatoires sur X et X_{row}.
   Donc: $Q(N, M, Z, L, S) = \lceil 2\frac{N \times M}{S \times L} + 2\frac{M}{L} \rceil$

5 Réglage du seuil d'arrêt récursif du programme câche 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)
Pour l'implémentation câche oblivious, nous devons prendre un seuil S assez petit pour être sur que $S << Z$ sur toutes les machines. Nous avons choisi $S = 10$ pour notre implémentation. De plus, S ne doit pas être trop petit non plus sous peine d'avoir un nombre de défauts de câche similaire au programme itératif.

6 Expérimentation (7 points)

Description de la machine d'expérimentation:
Processeur: Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz 2.30 GHz – Mémoire: 16.0 GB – Système: Ubuntu 20.04

6.1 (3 points) Avec `valgrind --tool =câchegrind --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 câche LL de second niveau est : 8388608 B, 64 B, 16-way associative
(mettre ici les paramètres: soit ceux indiqués ligne 3 du fichier câchegrind.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	220,150,001	122,115,188	4,928,086	123,135,658	56,601,925	148,401
2000	1000	439,363,150	243,440,102	11,025,864	245,402,062	112,420,463	291,145
4000	1000	879,075,144	487,431,939	23,225,467	491,792,399	225,434,273	576,755
2000	2000	878,963,460	487,908,884	19,905,251	491,865,846	226,267,270	572,938
4000	4000	3,512,536,472	1,950,367,286	80,018,484	1,966,739,964	904,970,234	2,262,543
6000	6000	7,901,684,722	4,387,800,896	180,361,363	4,424,829,262	2,036,169,665	5,074,518
8000	8000	14,045,437,454	7,799,638,634	320,952,312	7,865,888,915	3,619,668,399	9,009,195

		câche aware			câche oblivious		
N	M	#Irefs	#Drefs	#D1miss	#Irefs	#Drefs	#D1miss
1000	1000	149,295,957	72,667,317	7,391	143,494,996	67,345,662	28,324
2000	1000	297,721,825	144,550,927	9,090	286,121,300	133,908,089	50,518
4000	1000	596,431,090	289,694,881	12,552	573,231,429	268,409,675	93,138
2000	2000	597,162,687	290,853,938	13,008	573,277,391	269,236,978	91,061
4000	4000	2,389,240,013	1,163,967,646	32,310	2,292,354,800	1,076,847,784	404,372
6000	6000	5,376,415,767	2,619,390,245	68,497	5,203,697,311	2,450,907,852	942,014
8000	8000	9,558,451,412	4,656,927,883	119,366	9,168,044,617	4,307,059,888	1,399,045

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

On remarque que:

- Le programme récursif est le pire en termes de nombre d'instructions, en termes d'accès mémoire, et en termes de défauts de câche.

- Le programme itératif se comporte beaucoup mieux en termes de défauts de câche que le récursif, même sans réelle optimisation de câche en raison de l'énorme réduction d'espace mémoire (on passe de $\Theta(M \times N)$ à $\Theta(N)$). C'est le plus efficace en nombre d'instructions assembleur en raison de sa simplicité (pas de découpage) et sa complexité optimale.

- Le programme câche aware est moins bon que l'implémentation itérative en termes de nombre d'instructions et d'accès mémoire et est le meilleur en termes de défauts de câche (c'est bien le but de cette implémentation).

- Le programme câche oblivious est similaire au câche aware en termes de nombre d'instructions et d'accès mémoire et est le deuxième meilleur en termes de défauts de câche (il est légèrement moins optimisé que le câche aware).

Ces mesures sont en accord avec les coûts analysés théoriques. Le câche aware est bien le meilleur en termes de défauts de câche grâce à la taille de câche Z connue. Il est suivi par le câche oblivious avec un facteur 10 de défauts de câche en plus, ce qui est tout de même raisonnable en comparaison avec l'itératif qui lui a un facteur 100 de défaut de câche en plus, par rapport au câche aware. Enfin, le récursif a lui 1000 fois plus de défauts de câche que le câche aware.

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`
(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].

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

Valeurs minimales sur 5 tests:

		itératif			*câche aware*			*câche oblivious*		
N	*M*	*temps cpu*	*temps écoulé*	*energie*	*temps cpu*	*temps écoulé*	*energie*	*temps cpu*	*temps écoulé*	*energie*
10000	*10000*	*1,31*	*1,32*	*2,14E+07*	*1,58*	*1,59*	*2,51E+07*	*1,49*	*1,5*	*2,39E+07*
20000	*20000*	*5,28*	*5,28*	*8,38E+07*	*6,41*	*6,42*	*1,06E+08*	*6,05*	*6,06*	*9,68E+07*
30000	*30000*	*11,91*	*11,92*	*1,90E+08*	*14,77*	*15,02*	*2,84E+08*	*14,38*	*14,38*	*2,28E+08*
40000	*40000*	*21,23*	*21,24*	*3,37E+08*	*29,38*	*29,44*	*5,37E+08*	*24,33*	*24,34*	*3,87E+08*

Valeurs Maximales sur 5 tests:

		itératif			*câche aware*			*câche oblivious*		
N	*M*	*temps cpu*	*temps écoulé*	*energie*	*temps cpu*	*temps écoulé*	*energie*	*temps cpu*	*temps écoulé*	*energie*
10000	*10000*	*1,51*	*1,54*	*1,29E+08*	*1,82*	*1,87*	*1,77E+08*	*1,52*	*1,58*	*1,21E+08*
20000	*20000*	*6,15*	*6,28*	*2,33E+08*	*7,39*	*7,58*	*3,18E+08*	*6,08*	*6,09*	*2,28E+08*
30000	*30000*	*13,92*	*14,29*	*2,35E+08*	*16,68*	*16,69*	*3,22E+08*	*14,59*	*14,66*	*2,52E+08*
40000	*40000*	*24,8*	*25,33*	*4,13E+08*	*29,57*	*31,74*	*6,12E+08*	*24,42*	*24,45*	*3,92E+08*

Valeurs Moyennes sur 5 tests:

		itératif			*câche aware*			*câche oblivious*		
N	*M*	*temps cpu*	*temps écoulé*	*energie*	*temps cpu*	*temps écoulé*	*energie*	*temps cpu*	*temps écoulé*	*energie*
10000	*10000*	*1,432*	*1,448*	*4,49E+07*	*1,762*	*1,794*	*6,05E+07*	*1,506*	*1,526*	*4,43E+07*
20000	*20000*	*5,794*	*5,852*	*1,22E+08*	*7,166*	*7,248*	*1,68E+08*	*6,066*	*6,074*	*1,24E+08*
30000	*30000*	*13,108*	*13,316*	*2,16E+08*	*16,226*	*16,284*	*3,09E+08*	*14,438*	*14,456*	*2,34E+08*
40000	*40000*	*23,334*	*23,444*	*3,83E+08*	*29,476*	*29,956*	*5,64E+08*	*24,39*	*24,404*	*3,90E+08*

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 ?*

On remarque que:

- Le programme itératif est le meilleur, que ce soit en temps ou en énergie consommée. Ceci est le cas car il ne perd pas de temps à faire des calculs pour optimiser la gestion de la mémoire, ce qui lui permet d'aller plus vite et de consommer moins.

- Le câche aware est le moins bon, que ce soit en temps ou en énergie consommée. On peut justifier ça par le fait qu'en réalité la quantité de défauts de câche n'est pas si couteuse pour le processeur. En revanche, dans le cas où l'on augmenterait encore la taille de N et M, étant donné que l'on a 100 fois moins de défaut de câche que l'itératif, il se pourrait que cette différence se fasse réellement ressentir à ce moment-là.

- Le câche oblivious est le deuxième meilleur, que ce soit en temps ou en énergie consommée. Ceci est le cas car il passe moins de temps à optimiser son empreinte que le câche aware, ce qui lui permet d'être plus économe.

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/câche aware/ câche 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)*

Si nous nous basons sur les précédentes mesure, l'itératif serait le plus efficace. Cependant, comme évoqué précédemment, si nous faisons cela, nous ne prenons pas en compte la gigantesque augmentation de défaut de câche que ferait l'itératif sur de plus gros exemples.
Nous pensons donc que le modèle câche aware serait le plus efficace.

La méthode de calcul utilisée est une régression linéaire appliquée par rapport à $M \times N$ (en abscisse).

- Temps cpu (en s) : $\approx 10^7$ s
- Energie (en kWh) : $\approx \frac{10^14}{3.6\times10^6} = 2.7 \times 10^8 kWh$
- Prix (en euros) : $\approx$ 10 millions d'euros, en se basant sur le prix du kWh fournis par EDF

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!*

Calculer les valeurs proches de **la diagonale** de la grille, a de fortes chances de nous donner la valeur correcte ou une bonne approximation.