Questionnaire TP AOD à compléter et rendre sur teide

Binôme : LI Maxime - AIT DRISS Salma

Préambule 1 point . Pourquoi le programme récursif avec mémoisation fourni génère-t-il une erreur d'exécution sur test 5 ?

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

Réponse: En effet, pour un programme recursif. lorsqu'il depasse un certain nombre d'appels recursifs, on se retrouve dans une surcharge de la pile d'éxecution ce qui ne permet pas d'enchainer l'appel recursif suivant et ainsi l'execution s'arrete apres s'etre planté

1 Programme itératif (4 points)

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

Réponse : Avec une analyse des dépendances, nous n'avons besoin de $\phi(i+1,j+1)$,$\phi(i,j+1)$ et $\phi(i+1,j)$ pour calculer $\phi(i,j)$. Donc il est possible de n'utiliser qu'une seule liste (tableau) de (N+1) éléments + une variable prev_value pour stocker $\phi(i+1,j+1)$ si l'on veut procéder par ligne. On va initialiser le tableau avec les valeurs de bordure correspondant à i=M. On applique l'equation de Bellman sur chaque case M fois. On parcours donc le tableau par ligne de la ligne M à la ligne 0 dans le sens décroissant. A la fin il suffit juste de prendre la dernière valeur du tableau pour avoir $\phi(0,0)$

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

   tab[N+1] ne rentre pas dans le cache donc il va falloir remettre tab[N+1] M fois en cache donc $\Theta\left(\frac{M*N}{Z}\right)$
   A cela s'ajoute l'accès au chaines de caractères qui doivent avoir un nombre de défaut de cache similaire.

2 Programme cache aware (4 points)

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

Réponse : le meme principe que la version iterative, sauf qu'ici on prend en consideration la taille du cache Z. On a un nbr de case nbr_case qui dépend de Z et ainsi on commence les calculs des nbr_case suivants du tableau tab jusqu'à atteindre la fin et pouvoir récuperer le dernier element du tableau col[M] qui est le résultat recherché. Sauf le calcul des nbr_case elements suivant dépendent de la dernière valeur du dernier element des nbr_case elements actuels pour chaque parcours. d'ou l'utilité du tableau col de M+1 elements qui permet de stocker ces valeurs.

A noté que au début nous avions fais le choix de partir sur un programme en blocking et avec un découpage en bloc $\sqrt{Z} * \sqrt{Z}$ Cependant nous nous somme rendu compte que cela augmentais de beaucoup les #Drefs et #Irefs et donc nous avons préférer séparer en bloc de Z*M.
Expérimentalement nbr_case fait Z/(sizeof(long) * 5) pour minimiser les caches MISS

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`) : $\Theta(M)$
2. travail (nombre d'opérations) : $\Theta(NM)$
3. nombre de défauts de cache obligatoires (sur modèle CO, y compris sur X et Y): $\Theta\left(\frac{2M+N}{L}\right)$
4. nombre de défauts de cache si $Z \ll \min(N, M)$:$\Theta\left(\frac{M*N*5*sizeof(long)}{Z^2}\right)$
   Accès à la bordure de taille M, $\frac{N}{Z/(5*sizeof(long))}$ fois :$\left(\frac{M*N*5*sizeof(long)}{Z}\right)$
   Accès au chaines de caractères supposé similaire
   On suppose que tab[nbr_case] reste dans le cache

s

3 Programme cache oblivious (2 points)

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

Réponse : A partir de la version cache aware, et en prenant un seuil pour le nombre de case qu'on peut calculer, on fait une recursivité : c'est à dire qu'on divise notre N en deux, et puis on appelle la fonction pour la moitié de la séquence et ensuite la deuxième moitié jusqu'à ce que la longueur du tableau soit inférieure au seuil. Toujours en utilisant notre tableau col de M+1 elements qui permet de stocker les valeurs.

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`) : $\Theta(M)$
2. travail (nombre d'opérations) : $\Theta(NM)$
3. nombre de défauts de cache obligatoires (sur modèle CO, y compris sur X et Y):M$\Theta\left(\frac{2M+N}{L}\right)$
4. nombre de défauts de cache si $Z \ll \min(N, M)$:$\Theta(\frac{M*N*2}{Z*seuil})$
   Dans le pire cas les tab ont une taille (seuil/2)+1. Donc on accède au pire cas $\frac{N}{seuil/2}$ fois à la bordure de taille M.
   On suppose que tab[seuil] reste dans le cache peut importe la situation.

4 Expérimentation (10 points)

Description de la machine d'expérimentation:
Processeur: AMD Ryzen 5 3500U – Mémpire: 5845Mb (avec la commande free -m) – Système: 20.04.1-Ubuntu

4.1 (6 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 : **4194304 B, 64 B, 16-way associative**
Le tableau ci-dessous regroupe l'ensemble des résultats.

		récursif mémo			itératif		
N	M	#Irefs	#Drefs	#D1miss	#Irefs	#Drefs	#D1miss
1000	1000	220 179 405	122 122 160	4 923 998	94 770 131	43 371 935	148 014
2000	1000	439 392 554	243 447 074	11 020 820	189 252 639	86 653 271	290 067
4000	1000	879 104 548	487 438 911	23 220 432	378 217 387	173 215 855	574 186
2000	2000	878 992 864	487 915 856	19 893 684	378 171 277	173 193 813	574 052
4000	4000	3 512 565 876	1 950 374 258	79 989 226	1 511 627 897	692 417 321	2 262 424
6000	6000	7 901 714 126	4 387 807 868	180 310 732	3 400 623 621	1 557 747 165	5 076 743
8000	8000	14 045 466 858	7 799 645 606	320 850 929	6 045 158 449	2 769 183 345	9 015 058

		cache aware			cache oblivious		
N	M	#Irefs	#Drefs	#D1miss	#Irefs	#Drefs	#D1miss
1000	1000	93 052 510	43 568 166	7 480	92 335 906	45 662 603	8 299
2000	1000	185 819 028	87 045 504	9 051	184 382 228	91 232 137	10 651
4000	1000	371 351 796	174 000 092	12 295	368 474 604	182 371 117	15 678
2000	2000	371 315 986	173 986 264	12 103	368 442 088	182 359 493	15 603
4000	4000	1 484 237 266	695 602 212	30 020	1 472 727 604	729 086 201	43 288
6000	6000	3 338 843 617	1 564 804 474	59 181	3 306 769 408	1 636 413 965	63 739
8000	8000	5 935 425 105	2 781 793 094	100 319	5 889 591 244	2 915 871 329	154 986

Analyse expérimetale: *commenter les mesures expérimentales par rapport aux coûts théoriques précédents. Quel algotithme se comporte le mieux avec valgrind et les paramètres proposés, pourquoi ?*

Logiquement les algorithmes cache aware et cache oblivious se comporte le mieux car ils sont codée dans l'optique de réduire les cache MISS.
*On retrouve une certaine proportionnalité entre N*M et #Drefs ainsi que entre N*M et les #Irefs(Par exemple, les valeur de #Drefs et #Irefs sont similaire pour (M=1000, N=4000) et (M=2000, N=2000)). Ce qui montre la complexité en $\Theta(MN)$ pour le nombre de défaut de cache*

Les algorithmes cache-aware et cache-oblivious réduisent la taille de la structure de donnée sur laquelle on travaille le plus pour quelle rentre dans le cache et que l'on puisse évité les caches MISS, on va diviser par 90 le nombre de cache MISS au maximun. Cependant on à les bordures à stocker en plus, ce qui se traduit par une augmentation des #Drefs necessaires.

A noté que notre algorithme cache-aware en blocking, nous permettait de réduire de 60 000 (pour M=N=8000) le nombre de #D1miss mais augmentait approximativement de 500 000 le nombre de #Drefs et de #Irefs C'est pour celà que nous avons fais le choix de parcourir entièrement les M lignes en une fois.

4.2 (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 :
La fonction ***perfMesureCommand*** *du dossier srcperf [1] modifié pour avoir une moyenne, le minimun et le maximum sur 5 mesures.*
Par manque de place seul la moyenne sera affiché, étant donnée que la variabilité pour le temps écoulé et l'énergie reste assez faible.
La valeur du temps CPU quant à elle varie entre $8E-05s$ et $2.2E-04s$ pour chaque algorithme peut importe le N et M.

		itératif			*cache aware*			*cache oblivious*		
N	*M*	*temps cpu (s)*	*temps écoulé (s)*	*energie (kWh)*	*temps cpu (s)*	*temps écoulé (s)*	*energie (kWh)*	*temps cpu (s)*	*temps écoulé (s)*	*energie (kWh)*
10000	*10000*	*1,19E-04*	*0,888867*	*3,291E-06*	*8,80E-05*	*0,866501*	*3,069E-06*	*1,02E-04*	*0,908296*	*3,284E-06*
20000	*20000*	*1,50E-04*	*3,42402*	*1,374E-05*	*1,37E-04*	*3,41026*	*1,410E-05*	*1,64E-04*	*3,57344*	*1,475E-05*
30000	*30000*	*1,42E-04*	*7,70926*	*3,604E-05*	*1,70E-04*	*7,62185*	*2,978E-05*	*1,24E-04*	*7,94682*	*3,152E-05*
40000	*40000*	*1,28E-04*	*14,2525*	*6,162E-05*	*1,41E-04*	*13,5558*	*5,175E-05*	*1,13E-04*	*13,9124*	*5,530E-05*

[1] `/matieres/4MMAOD6/2022-10-TP-AOD-ADN-Docs-fournis/tp-ADN-distance/srcperf/`

4.3 (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
```

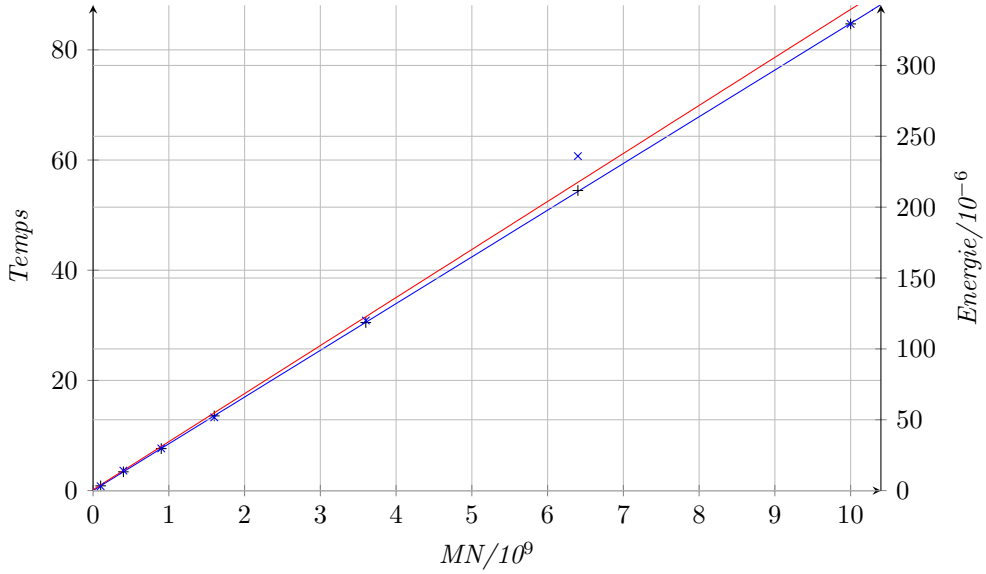
A partir des résultats précédents, le programme cache aware *est le plus performant pour la commande ci dessus (test 5);*
Dans la première partie nous avons calculé que le nombre d'opérations pour le cache aware était en $\Theta(MN)$, donc il suffit de réaliser une régression linéaire pour avoir une fonction de la forme:

$$\begin{aligned} Energie &= a*M*N+b \\ Temps &= c*M*N+d \end{aligned}$$

Modélisation avec Regressi

$$Temps = a \cdot MN + b$$

$$Energie = c \cdot MN + d$$



Ecart-type données-modèle Temps=96,86 10^{-3} Coeff. corrélation=1,0000
Ecart-type données-modèle Energie=9,654 10^{-6} Coeff. corrélation=0,99753
Intervalle de confiance à 95%
a=(8,480 ±0,028)10^{-9}
b=(17 ±131)10^{-3}
c=(33,9 ±2,7)10^{-15}
d=(647 ±1,301E4)10^{-9}

les ressources pour l'exécution seraient donc après calcul:

- *Temps cpu :* $(3,423 \pm 0.012)10^6 s$ *Soit 40 jours*
- *Energie :* $(13.00 \pm 1.09) kWh$.

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! *Utiliser un algorithme de type Branch and Bound pour pouvoir au moins une approximation (borne inférieur) à l'issue de la minute et utiliser en multi threading avec un meilleur matériel.*