POLYTECHNIQUE MONTRÉAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2024)
3 crédits (3-1.5-4.5)

CORRIGÉ DE L'EXAMEN FINAL

DATE: Mardi le 17 décembre 2024

HEURE: 9h30 à 12h00

DUREE: 2H30

NOTE: Aucune documentation permise sauf un aide-memoire, préparé par l'étudiant, qui consiste en une feuille de format lettre manuscrite recto-verso, calculatrice non programmable permise

Ce questionnaire comprend 4 questions pour 20 points

Question 1 (5 points)

a) Le programme OpenCL suivant reçoit en argument $n = 5$, le nombre d'éléments à traiter par un *work item*, et les matrices *a*, *b* et *c* de dimension 5×5, qui sont stockées en mémoire sous la forme d'un vecteur de 25 éléments. Les *work item* sont organisés en 2 dimensions et la taille du problème est de 5 dans chaque dimension. La taille des *work group* n'est pas précisée. Le contenu des matrices est de a = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5}, b = {0, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 9, 8, 7, 6}, c = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}. Quelle est l'opération effectuée par cette fonction OpenCL? Quelle sera la valeur en sortie de $c[12]$? **(2 points)**

```
__kernel void mm(const int n, __global float *a,
    __global float *b, __global float *c)
{ int k;
  int i = get_global_id(0);
  int j = get_global_id(1);
  for (k = 0; k < n; k++)
    c[i*n+j] += a[i * n + k] * b[k * n + j];
}
```

```
      | 1 2 3 4 5 |       | 0 9 8 7 6 |
      | 6 7 8 9 0 |       | 5 4 3 2 1 |
a =   | 1 2 3 4 5 |  b =  | 0 9 8 7 6 |
      | 6 7 8 9 0 |       | 5 4 3 2 1 |
      | 1 2 3 4 5 |       | 0 9 8 7 6 |
```

Ce programme effectue une multiplication de matrices, c = a x b. La valeur de c[12] est le troisième élément de la troisième rangée de la matrice c. Il est obtenu par le produit scalaire de la troisième rangée de a avec la troisième colonne de b. On obtient 1x8 + 2x3 + 3x8 + 4x3 + 5x8 = 90.

b) Le programme OpenCL précédent, de la question 1 a), doit être appliqué à des matrices de grande taille et on vous demande de l'optimiser. Proposez jusqu'à 4 améliorations possibles (pas nécessairement applicables simultanément) qui pourraient permettre à ce programme d'être plus rapide. Justifiez. **(2 points)**

*Une première amélioration simple est d'utiliser une variable temporaire pour faire la somme dans la boucle. Cette somme peut ensuite être replacée dans c[i*n+j]. Une seconde amélioration possible est de simplifier le calcul de l'indice i*n+k. En effet, on peut avoir une variable initialisée à i*n avant la boucle, et qui est ensuite incrémentée de 1 à chaque tour de boucle pour servir d'indice. L'indice k*n+j peut être simplifié d'une manière similaire avec une variable initialisée à j servant d'indice à b, et incrémentée de n à chaque tour de boucle. Une troisième amélioration est d'inverser l'assignation avec j sur global id 0 et i sur global id 1. En effet, on veut que des coeurs adjacents accèdent en même temps des valeurs adjacentes dans b ou c. Une quatrième amélioration est de prendre une copie dans la mémoire locale de la rangée de a pour tous les work item du même work group qui opèrent sur la même rangée. Ceci peut éviter un grand nombre d'accès à a en mémoire globale. Copier une partie de b en mémoire locale est beaucoup moins intéressant, puisque les work item adjacents travaillent sur des colonnes différentes de b.*

c) Plusieurs nouveaux processeurs présentent une intégration plus forte et supportent la mémoire virtuelle partagée entre le CPU et le GPU pour les applications. Quelles sont les conséquences d'une mémoire virtuelle partagée CPU-GPU sur la programmation d'applications qui utilisent le GPU, et sur leur performance? **(1 point)**

La mémoire virtuelle partagée permet d'utiliser les mêmes structures de données, possiblement contenant des pointeurs, sur les CPU et les GPU. Ceci peut éviter d'avoir à convertir le format des données en passant du CPU au GPU. Il est même possible d'opérer sur les mêmes données, de manière concurrente, à partir du CPU et du GPU, à condition de s'assurer de synchroniser correctement les accès concurrents, par exemple avec des opérations atomiques. Cependant, l'impact le plus important est probablement le fait qu'il n'est plus nécessaire de copier les données du CPU vers le GPU, pour amorcer le calcul sur GPU, et du GPU au CPU, pour rechercher les résultats. Ceci peut aussi diminuer la latence pour déléguer des calculs au GPU. Certains calculs moins longs qui ne pouvaient efficacement être délégués au GPU, le coût des transferts l'emportant sur le gain en performance, peuvent maintenant profiter d'une certaine accélération sur GPU.

Question 2 (5 points)

a) Le programme MPI suivant s'exécute sur 4 noeuds (MPI_Comm_size retourne 4). Donnez une sortie possible produite par ce programme? **(2 points)**

```
int main (int argc, char *argv[])
{ int i, rank, size, n = 40, m = 7, chunk, start;
  int in[40], h[7], hl[7];
  for(i = 0; i < n; i++) in[i] = i % 7;
  for(i = 0; i < m; i++) hl[i] = 0;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  chunk = n / size; start = chunk * rank;
  for(i = start; i < start + chunk; i++) hl[in[i]]++;
  printf("hl[%d] = %d\n", rank, hl[rank]);

  MPI_Reduce(hl, h, m, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  if(rank == 0) {
    printf("h = {");
    for(i = 0; i < m; i++) printf("%d ", h[i]);
    printf("}\n");
  }
  MPI_Finalize();
}
```

Le vecteur in contiendra in = {0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4}. Chaque processus fait un histogramme pour 1/4 de ce vecteur. Les 4 histogrammes sont ensuite réduits en un seul histogramme qui contiendra le nombre de chacune des valeurs (0 à 6). Une sortie possible sera comme suit.

```
hl[2] = 1
hl[3] = 2
hl[0] = 2
hl[1] = 1
h = {6 6 6 6 6 5 5 }
```

b) Le programme MPI suivant s'exécute sur 4 noeuds (MPI_Comm_size retourne 4). Donnez une sortie possible produite par ce programme? **(2 points)**

```
int main (int argc, char *argv[])
{
  int size, rank, i, in[4], o1[4], o2[4];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  for(i = 0; i < 4; i++) { in[i] = i * i + rank; o1[i] = o2[i] = 0; }
  MPI_Scatter(in, 1, MPI_INT, o1, 1, MPI_INT, 0, MPI_COMM_WORLD);
  for(i = 1; i < 4; i++) o1[i] = o1[0] + rank * i;
  MPI_Alltoall(o1, 1, MPI_INT, o2, 1, MPI_INT, MPI_COMM_WORLD);
  printf("%d: in={%d, %d, %d, %d};\n", rank, in[0], in[1], in[2], in[3]);
```

```
  printf("%d: o1={%d, %d, %d, %d};\n", rank, o1[0], o1[1], o1[2], o1[3]);
  printf("%d: o2={%d, %d, %d, %d};\n", rank, o2[0], o2[1], o2[2], o2[3]);
  MPI_Finalize();
}
```

```
1: in={1, 2, 5, 10};
1: o1={1, 2, 3, 4};
1: o2={0, 2, 6, 12};
2: in={2, 3, 6, 11};
2: o1={4, 6, 8, 10};
2: o2={0, 3, 8, 15};
3: in={3, 4, 7, 12};
3: o1={9, 12, 15, 18};
3: o2={0, 4, 10, 18};
0: in={0, 1, 4, 9};
0: o1={0, 0, 0, 0};
0: o2={0, 1, 4, 9};
```

c) Quelle est la différence de comportement entre les fonctions MPI_Send, MPI_Isend et MPI_Bsend? **(1 point)**

La fonction MPI_Send retourne après avoir initié un envoi et cessé d'utiliser le tampon qui lui a été passé, contenant les données à envoyer. Les données peuvent avoir été déjà envoyées, ou simplement avoir été copiées dans un tampon d'envoi. La fonction MPI_Isend initie l'envoi, mais le tampon qui lui a été passé est possiblement encore utilisé pour cet envoi, il faut vérifier le statut de l'envoi avant de modifier le tampon. La fonction MPI_Bsend copie les données à envoyer dans un autre tampon, afin de permettre à la fonction de retourner rapidement, avec le tampon passé en argument libre d'être réutilisé.

Question 3 (5 points)

a) Vous compilez un programme avec les options de profilage de `gprof` et l'exécutez ensuite. Ce programme ne comporte qu'un seul fil d'exécution, qui est interrompu par `gprof` à chaque 1ms afin de noter la fonction dans laquelle se trouve le compteur de programme. De plus, à chaque appel, `gprof` note le décompte du nombre d'appels de chaque provenance (fonction appelante). L'information donnée dans le tableau qui suit est ainsi produite pendant l'exécution. Calculez pour chaque fonction le temps passé dans la fonction elle-même (self) et le temps passé dans la fonction elle-même plus ses enfants (self+childs), comme le ferait l'outil `gprof`. **(2 points)**

Fonction	échantillons	appels	appelants
main	4	1	
f1	1	3	main: 3
f2	3	5	main: 2, f1: 3
f3	8	8	f1: 5, f2: 3
f4	9	6	f1: 1, f2: 2, f3: 3
f5	7	7	f2: 4, f4: 3
f6	10	3	f3: 3

Le temps passé dans chaque fonction elle-même (self) est donné par le nombre des échantillons, chacun comptant pour 1ms. Pour le temps passé dans chaque fonction incluant les fonctions appelées (self+childs), il faut imputer le temps des fonctions appelées aux fonctions appelantes, au prorata des appels. On commence par les fonctions feuilles (f5 et f6) où le temps self+childs est le temps de chaque fonction elle-même (self). La fonction f6 n'a qu'un seul appelant et son temps self+childs pourra donc être imputé directement à f3. Le temps de f5 (7ms) doit être imputé 4/7 à f2 (4ms) et 3/7 à f4 (3ms). Le temps de f4 (12ms) doit être imputé 1/6 à f1 (2ms), 2/6 à f2 (4ms) et 3/6 à f3 (6ms). Le temps de f3 (24ms) doit être imputé 5/8 à f1 (15ms) et 3/8 à f2 (9ms). Le temps de f2 (20ms) doit être imputé 2/5 à main (8ms) et 3/5 à f1 (12ms). Finalement, le temps de f1 (30ms) doit être imputé entièrement à main.

Le temps total self+childs de main est donc de 42ms, ce qui correspond effectivement au temps total d'exécution du programme (somme des temps self de chaque fonction). Le résultat complet est fourni dans le tableau qui suit.

```
Fonction  self  self+childs appels   appelants
main       4ms   4ms + 8ms (f2) + 30ms (f1) = 42ms
f1         1ms   1ms + 2ms (f4) + 15ms (f3) + 12ms (f2) = 30ms
f2         3ms   3ms + 4ms (f5) + 4ms (f4) + 9ms (f3) = 20ms
f3         8ms   8ms + 10ms (f3) + 6ms (f4) = 24ms
f4         9ms   9ms + 3ms (f5) = 12ms
f5         7ms   7ms
f6        10ms   10ms
```

b) Un outil de mesure comme Heap Profile mémorise le contenu de la pile d'appels (la fonction courante et celles dans le chemin d'appel) ainsi que le nombre d'octets alloués, à chaque appel pour allouer de la mémoire (e.g. malloc et new). Voici les échantillons récoltés. Calculez le nombre d'octets alloués dans chaque fonction elle-même (self) et dans chaque fonction elle-même plus ses enfants (self+childs). **(2 points)**

```
(main), 32 octets
(main, f1, f2, f3), 96 octets
(main), 64 octets
(main, f2), 48 octets
(main, f3, f4, f5), 24 octets
(main, f1, f3, f5), 40 octets
(main, f3, f5, f6), 48 octets
(main, f1, f5), 16 octets
(main, f2, f4, f6), 56 octets
(main, f1, f3, f5), 72 octets
```

L'exécution complète a alloué 496 octets. Pour le compte d'octets self de chaque fonction, on somme le nombre d'octets, pour tous les appels d'allocation où elle arrive en sommet de pile. Pour le compte d'octets self+childs de chaque fonction, on somme le nombre d'octets, pour tous les appels d'allocation où elle est présente dans la pile d'appels.

```
fonction   self            self+child
main       32+64=96        32+96+64+48+24+40+48+16+56+72=496
f1          0              96+40+16+72=224
f2         48              96+48+56=200
f3         96              96+24+40+48+72=280
f4          0              24+56=80
f5         24+40+16+72=152 24+40+48+16+72=200
f6         48+56=104       48+56=104
```

c) Vous suspectez un problème dans votre programme, relié à l'accès à une matrice qui occasionne un trop grand nombre de fautes de cache, et qui en ralentit l'exécution. Quel serait un bon outil pour diagnostiquer un tel problème? Expliquez comment fonctionne l'outil en question. Comment interagit-il avec l'exécution de votre programme pour prendre des mesures, quel traitement fait-il sur ces mesures, et quelle présentation vous fait-il des résultats afin de vous aider à comprendre d'où vient le problème? **(1 point)**

Un outil comme perf ou oprofile utilise les compteurs de performance en matériel disponibles sur la plupart des processeurs. Ces compteurs peuvent être incrémentés selon divers événements comme les fautes de cache L1, L2 ou L3, les délais de branchements ou les blocages dans le pipeline. Ils peuvent ensuite générer une interruption lorsque le compteur arrive à une certaine valeur (e.g. passent de 1111...111 à 0000...000). L'outil active le compteur de performance souhaité, dans ce cas-ci les fautes de cache normalement sur L1, et initialise le compteur à l'intervalle souhaité pour l'interruption (e.g. -100000 pour retomber à 0000...000 après 100000 fautes de cache).

Ainsi, lorsqu'une interruption survient à chaque 100000 fautes de cache, la fonction d'interruption échantillonne le compteur de programme et l'ajoute à un histogramme. Le surcoût engendré par un tel outil est très faible. Les compteurs de performance font leur travail sans générer de surcoût. Le seul surcoût est associé aux interruptions qui arrivent à un intervalle ajustable. On parle généralement d'un surcoût de l'ordre de 1%.

A la fin de l'exécution du programme, on peut donc voir comment se distribuent les valeurs de compteur de programme échantillonnées. Ceci donne les adresses des instructions, numéros de lignes de code source, ou noms de fonctions, où on retrouve le plus d'échantillons. Ceci informe donc le programmeur de la prévalence des fautes de cache, et des sections de programme qui en causent le plus.

L'outil Cachegrind de Valgrind pourrait être utilisé, mais ce n'est pas un bon outil dans un tel cas. En effet, son facteur de ralentissement est très grand. De plus, il se peut qu'il ne simule pas exactement le cache de notre ordinateur, si tous les bons paramètres n'ont pas été entrés. Cachegrind est plus intéressant lorsqu'on se demande quelle serait la performance pour différentes configurations de cache pour lesquelles nous ne possédons pas le matériel correspondant, par exemple un nouvel ordinateur à concevoir ou à acheter.

Question 4 (5 points)

a) Un programme de tri parallèle doit trier N nombres en ordre croissant, sur M processus qui se trouvent sur autant de noeuds dans une grappe d'ordinateurs. Les N nombres ont déjà été répartis également en M fichiers d'entrée, un pour chacun des M noeuds. Pour commencer, i) chaque processus lit ses nombres et les trie en ordre croissant. Ensuite, ii) chaque processus prend 10 nombres dans sa liste de nombres triés, en position 0, M/9, 2M/9... M-1, et envoie cette sous-liste au processus racine (rank == 0). iii) Le processus racine fusionne en ordre croissant les sous-listes des M processus afin de produire une liste qui contient 10M nombres. iv) Le processus racine extrait ensuite les nombres qui divisent cette liste en M intervalles afin de les utiliser comme pivots, en position 0, 10, 20, ... M-1, et il envoie cette liste de pivots à chaque processus. v) Chaque processus envoie alors ses nombres aux autres processus, en envoyant au processus de rang i tout nombre qui tombe dans l'intervalle i, selon les pivots reçus du processus racine. vi) Chaque processus trie les nombres reçus et écrit le résultat dans son fichier de sortie. La liste complète triée est alors disponible en lisant un à la suite de l'autre les M fichiers de sortie, un par noeud. Est-ce que cet algorithme produira effectivement un fichier trié à la fin? Combien d'opérations de comparaison, en fonction de N et M, est-ce que chaque étape prendra sur chaque processus? Pour simplifier la question, on ne considère pas les envois de message, les accès en mémoire, ou les lectures et écritures, seulement les comparaisons. **(2 points)**

Effectivement, cet algorithme permet de faire un tri. On divise le domaine des nombres en intervalles et on distribue les nombres entre ces intervalles. Après avoir trié les nombres dans chaque intervalle, il suffit de juxtaposer les intervalles triés pour obtenir la liste complète triée. La partie la plus critique est le choix des pivots, pour s'assurer que chaque intervalle contient à peu près la même quantité de nombres à trier, afin d'équilibrer la charge entre les M processus.

Le tri, sur un processeur, de n nombres prend dans le cas général $nlog_2(n)$ comparaisons. Dans certains cas, il est possible de faire mieux, par exemple si ces nombres sont des entiers dans un intervalle restreint. Alors, un tri par casiers permet d'avoir une complexité linéaire. Faire un tri avec un algorithme inefficace, par exemple avec une complexité quadratique, n'est pas représentatif de ce qui peut être fait et ne serait jamais considéré pour le tri en parallèle d'un grand nombre d'éléments. Par exemple, si on a 2^{32} éléments, la complexité $nlog_2(n)$ donne 32×2^{32}, alors qu'une complexité quadratique donnera $2^{32} \times 2^{32} = 2^{64}$. Ceci donne un ratio de $2^{64}/(32 \times 2^{32}) = 2^{27} = 134217728$ entre les deux!

Pour la première étape i), chaque processus trie N/M nombres à l'aide de $N/M \times log_2(N/M)$ comparaisons. ii) La seconde étape ne demande pas de comparaisons. En effet, on prend 10 nombres, en accédant un vecteur à 10 positions fixes. Pour la troisième étape, iii) le noeud racine trie 10M nombres alors que les autres processus ne font rien. Ceci lui demande $10M \times log_2(10M)$. Il est possible de faire un peu mieux, puisque les listes dans chaque processus étaient déjà triées, par exemple en utilisant une queue de priorité pour fusionner les listes qui arrivent des différents noeuds, ce qui donne $10M \times log_2(M)$. L'étape iv) se fait aussi sur le noeud racine. Là encore, on ne fait qu'accéder un vecteur à des positions fixes, sans nécessiter de comparaisons. Pour l'étape v), chaque processus doit classer ses N/M nombres dans le bon intervalle. Puisque ses nombres sont déjà triés, il commence à comparer les premiers nombres avec le premier pivot, et les envoie au premier noeud tant qu'ils sont en-deça de ce pivot.

Lorsque les nombres deviennent plus grands que le premier pivot, il les compare au second pivot et les envoie au second noeud. Ainsi, chaque nombre sera comparé avec un pivot, sauf M fois où une seconde comparaison sera nécessaire, lorsque le nombre dépasse le pivot. Le nombre total de comparaisons sera donc de $N/M+M$. Pour l'étape vi), on suppose que chaque noeud reçoit à peu près N/M nombres. Chaque processus trie les nombres reçus en $N/M \times log_2(N/M)$ comparaisons. Là encore, il serait possible de tenir compte du fait que les nombres envoyés par chaque processus étaient déjà triés et faire une fusion en utilisant une queue de priorité. Cela réduirait un peu le nombre de comparaisons requises à $N/M \times log_2(M)$.

b) Un programme OpenMP s'exécute sur m processeurs. Il doit mettre une matrice de taille $n \times n$ à la puissance 250. Comment pourriez-vous réaliser cette opération? Ne composez pas un programme, mais expliquez les opérations arithmétiques qu'un tel programme réaliserait. Combien d'opérations d'addition et de multiplication est-ce que chacun des m processeurs devra effectuer? **(2 points)**

Pour mettre une matrice A à la puissance 250, on peut calculer: $A^2 = A \times A$, $A^4 = A^2 \times A^2$, A^8, A^{16}, A^{32}, A^{64}, A^{128}, $A^{192} = A^{128} \times A^{64}$, $A^{224} = A^{192} \times A^{32}$, $A^{240} = A^{224} \times A^{16}$, $A^{248} = A^{240} \times A^8$, $A^{250} = A^{248} \times A^2$. Ceci demande 12 multiplications de matrices. Chaque multiplication de matrices demande n^3 multiplications et autant d'additions. Ceci se parallélise bien sur m processeurs et chaque processeur fait donc $12 \times n^3/m$ multiplications et autant d'additions.

c) La solution d'un ensemble d'équations représenté par $b = Ax$ peut se faire par la méthode itérative de Jacobi avec l'équation $x_{k+1} = D^{-1}(b - Ox_k)$, où on calcule x à l'itération k+1 à partir de x à l'itération k, et où D est la diagonale de A et O est A - D. Si b et x sont de dimension n et A de $n \times n$, combien d'additions/soustractions et de multiplications, en fonction de n, est-ce que le calcul d'une itération requiert? **(1 point)**

Pour la multiplication matrice fois vecteur Ox_k, il faut $n \times (n-1) = n^2 - n$ multiplications et autant d'additions. En effet, la diagonale de O est nulle, ce qui fait une opération de moins à chaque rangée, d'où le $(n-1)$. La soustraction $(b - Ox_k)$ demande n soustractions. La multiplication par D^{-1} demande n multiplications, car D^{-1} est une matrice diagonale. Le total est donc de $n^2 - n + n = n^2$ multiplications et $n^2 - n + n = n^2$ additions ou soustractions. Si on traite spécialement le cas où on additionne le premier élément d'un calcul avec 0, on pourrait réduire de n additions le premier calcul.

Le professeur: Michel Dagenais