POLYTECHNIQUE MONTRÉAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2023)
3 crédits (3-1.5-4.5)

CORRIGÉ DU CONTRÔLE PÉRIODIQUE

DATE: Mercredi le 25 octobre 2023

HEURE: 9h30 à 11h20

DUREE: 1H50

NOTE: Aucune documentation permise sauf un aide-mémoire, 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) Un programme s'exécute sur un processeur 64 bits qui possède un cache L1 pour les données de 64Kio, avec chaque bloc en cache qui fait 64 octets. Ce cache a une fonction de correspondance directe et une stratégie de remplacement de *bloc le moins utilisé récemment* (LRU). Tel que montré dans le code qui suit, une boucle de calcul accède à répétition, 3 fois, un vecteur de n entiers de 64 bits pour faire différents calculs dans des registres. i) Jusqu'à quelle valeur de n est-ce que le vecteur peut entrer entièrement dans le cache de données L1? ii) Lorsque le vecteur entre entièrement en cache, quel sera le taux de succès en cache de données? iii) Lorsque la taille du vecteur dépasse nettement la taille du cache, quel sera le taux de succès en cache pour l'accès aux données? **(2 points)**

```
for(i = 0; i < 3; i++)
  for(j = 0; j < n; j++) sum = sum ^ v[j];
```

Le cache contient 64Kio ou 8Kio mots de 8 octets (64 bits). Chaque bloc en cache contient 8 mots de 8 octets. i) Jusqu'à n = 8192, le vecteur peut entrer entièrement en cache. Le taux de succès en cache peut se calculer avec le nombre de fautes et le nombre d'accès. Pour chaque bloc en cache, on a 1 faute (faute inévitable lors de l'accès au premier mot du bloc qui reste ensuite en cache). Par ailleurs, pour chaque bloc, nous avons 3 fois (boucle sur i) 8 accès (mots dans le même bloc). ii) Le taux de succès est donc de 23/24 ou 95.8%. Lorsque n est beaucoup plus grand, le bloc ne sera plus là pour les tours de boucle subséquents de i. Ainsi, lors de l'accès au premier mot d'un bloc, il y a une faute. Les 7 mots consécutifs dans le même bloc ne génèrent pas de faute. iii) Le taux de succès est alors de 7/8 ou 87.5%.

b) On veut configurer une grappe de calcul pour effectuer un travail. Deux configurations possibles sont offertes: a) une grappe de 32 ordinateurs utilisés pendant 1 heure, b) une grappe de 16 ordinateurs utilisés pendant 2 heures. Chaque ordinateur a une probabilité de 0.95 d'être fonctionnel pour toute la durée d'une heure de calcul. Un serveur de fichiers est utilisé tout au long des calculs. Son unité centrale de traitement a aussi une probabilité de 0.95 d'être fonctionnelle pour toute la durée d'une heure, alors que son unité de disque, en RAID 5, requiert 4 disques fonctionnels sur 5, chaque disque ayant une probabilité de 0.8 d'être fonctionnel pendant toute la durée d'une heure. Dans les deux cas, tous les ordinateurs de la grappe ainsi que le serveur de fichiers (unité centrale de traitement et unité de disque) doivent être fonctionnels pendant toute la durée du calcul. i) Quelle est la probabilité que la grappe de calcul a, excluant le serveur de fichiers, soit fonctionnelle pendant 1 heure? ii) La grappe de calcul b pendant 2 heures? iii) Quelle est la probabilité que le serveur de fichiers soit fonctionnel pendant 1 heure? iv) Quelle est finalement la configuration qui a le plus de chances de fonctionner pendant toute sa période afin de terminer son calcul? **(2 points)**

i) La probabilité d'avoir les 32 noeuds fonctionnels pendant une heure est de $0.95^{32} = 0.193711484$. *La probabilité qu'un noeud soit fonctionnel pendant deux heures consécutives est de* $0.95^2 = 0.9025$. *ii) La probabilité d'avoir les 16 noeuds fonctionnels pendant deux heures est de* $0.9025^{16} = 0.193711484$. *La probabilité que l'unité de disque soit disponible pendant une heure est de* $0.8^5 + 5 \times 0.8^4 \times (1 - 0.8) = 0.73728$. *iii) La probabilité que le serveur de fichiers soit fonctionnel est donc de* $0.95 \times 0.73728 = 0.700416$. *La probabilité que le serveur de fichiers soit disponible pendant 2 heures est conséquemment de* $0.700416^2 = 0.490582573$. *iv) La première configuration a donc une probabilité d'être fonctionnelle de* $0.193711484 \times 0.700416 = 0.135678623$, *alors que la seconde a une probabilité de* $0.193711484 \times 0.490582573 = 0.095031478$. *La première configuration est donc préférable.*

c) Vous trouvez un vieil ordinateur 32 bits dans le recyclage qui est encore fonctionnel. Vous voulez déterminer la taille des blocs en cache pour les données à l'aide d'un petit programme qui accède à répétition les éléments d'un vecteur dans une boucle. Comment pourriez-vous le faire? Expliquez? **(1 point)**

On peut prendre un vecteur, accédé à répétition pour un nombre fixe (mais très grand) total d'accès, dont on fait varier la taille. On veut voir à partir de quelle taille le temps d'accès moyen augmente beaucoup, car on a dépassé la taille de la mémoire cache. On prend alors une taille bien supérieure à la taille de la mémoire cache. Ensuite, on accède des mots consécutifs, puis 1 mot sur 2, 1 mot sur 4, 1 mot sur 8... Le temps d'accès moyen croîtra, car on utilisera de moins en moins de mots pour chaque bloc chargé en mémoire cache. Lorsqu'on ne lira plus qu'un seul mot par bloc (que ce soit sur des blocs consécutifs, ou 1 bloc sur 2, 1 bloc sur 4...), le temps cessera de croître. Le point de transition nous indiquera la taille des blocs en cache. La grande difficulté est d'avoir des mesures exactes. On ne peut pas facilement mesurer le temps pour un seul accès et ainsi déterminer s'il a été fait en cache ou non.

Une autre possibilité serait d'utiliser des compteurs matériels du nombre de fautes de cache en mémoire de données, s'ils existent sur l'ordinateur en question. On pourrait accéder à un grand vecteur et s'attendre à une faute par bloc. Le nombre d'accès sur le nombre de fautes donnerait la taille d'un bloc de cache (en mots).

Question 2 (5 points)

a) Considérez le code suivant utilisant les *pthreads* qui s'exécute. Quelle en est la sortie? **(2 points)**

```
void *create_child_threads(void *arg) {
  int id = *((int *)arg);
  pthread_t t;
  fprintf(stderr, "T%d ", id);
  if(id < 100) {
    id = id * 10;
    for(int i = 1; i <= 2; i++) {
      id++;
      pthread_create(&t, NULL, create_child_threads, &id);
      pthread_join(t, NULL);
    }
  }
  pthread_exit(NULL);
}
int main(int argc, char **argv) {
  int id = 0;
  create_child_threads(&id);
  return(0);
}
```

T0 T1 T11 T111 T112 T12 T121 T122 T2 T21 T211 T212 T22 T221 T222

Il est à noter que l'ordre des éléments en sortie ne varie pas, car il y a un pthread_join tout de suite après le pthread_create, et donc jamais plus d'un fil d'exécution enfant en exécution.

b) La fonction suivante `DoParallelScan` est appelée avec comme arguments `y` un vecteur de 10 `float` tous à 0, `x` un vecteur de 10 `float` {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0}, et n un entier qui vaut 10. i) Quel est le contenu de `y` après la première passe du `parallel_scan`? ii) Après la deuxième passe? iii) Quelle est la valeur retournée par la fonction à la fin? **(2 points)**

```
class Body {
  T sum; T* const y; const T* const x;
  Body( T y_[], const T x_[] ) : sum(0), x(x_), y(y_) {}
  T get_sum() const {return sum;}
  template<typename Tag>void operator()(const blocked_range<int>& r, Tag) {
    T temp = sum;
    for( int i=r.begin(); i<r.end(); ++i ) {
      temp = temp + x[i] * x[i];
      if(Tag::is_final_scan()) y[i] = temp;
    }
    sum = temp;
  }
  Body(Body& b, split) : x(b.x), y(b.y), sum(0) {}
  void reverse_join(Body& a) { sum = a.sum + sum; }
  void assign( Body& b ) {sum = b.sum; }
};
float DoParallelScan(T y[], const T x[], int n) {
  Body body(y,x);
  parallel_scan(blocked_range<int>(0,n), body);
  return body.get_sum();
}
```

Après la première passe, rien n'a été écrit dans y qui reste donc à 0. Après la seconde passe (is_final_scan), on se retrouve avec la somme courante des carrés, soit: {0.0, 1.0, 5.0, 14.0, 30.0, 55.0, 91.0, 140.0, 204.0, 285.0}. La valeur retournée à la fin est 285.0.

c) Vous réalisez un programme avec la librairie `pthreads`. Au moment de créer chaque nouveau fil d'exécution, il faut spécifier la taille à utiliser pour sa pile d'exécution. Comment peut-on estimer l'espace à réserver pour la pile d'un fil d'exécution? **(1 point)**

On peut initialiser la pile à zéro, ou avec un autre patron spécifique de bits, prendre une très grande pile, et à la fin vérifier la case la plus loin dans la pile qui n'est plus à zéro. Comme méthode alternative, on peut faire une analyse statique du programme pour déterminer l'arbre d'appel, et la taille sur la pile prise par chaque appel imbriqué, afin de mesurer la taille requise pour la pile dans le chemin d'appel qui requiert le plus d'espace. On peut aussi se fier sur les pages de garde placées à la fin de chaque pile et attendre qu'une erreur de segmentation soit générée, si on dépasse la fin de la pile.

Question 3 (5 points)

a) Un ordinateur adressable à l'octet, semblable au x86-64, utilise des adresses de 64 bits et des pages de 4Kio. Les 16 bits les plus significatifs de l'adresse virtuelle sont inutilisés, et des tables de pages à

4 niveaux sont utilisées. Chaque noeud dans l'arbre qui constitue la table de pages occupe une page. Un processus commence avec 2Mio, utilisés pour son code exécutable et le monceau au tout début de son espace adressable, et 2Mio, utilisés à la toute fin de son espace adressable pour la pile. i) Quelle est la taille d'une entrée dans un noeud de la table de page? ii) Combien d'entrées a-t-on dans chaque noeud de la table de pages? iii) Comment se décomposent en différents champs les 64 bits d'une adresse virtuelle afin d'accéder l'arbre qui constitue la table de pages? iv) Combien de noeuds existent présentement dans la table de pages pour ce processus qui utilise 2Mio au début et 2Mio à la fin de son espace adressable? **(2 points)**

Chaque entrée donne soit l'adresse physique du prochain noeud ou, au niveau des feuilles, l'adresse physique recherchée. Ces adresses font 64 bits mais plusieurs bits sont inutilisés (e.g. les 12 bits les moins significatifs qui sont implicitement à 0) et peuvent servir de bits de statut pour la page. i) Chaque entrée fait donc 64 bits ou 8 octets. ii) Une page de 4Kio contient donc 512 entrées de 8 octets. iii) L'adresse de 64 bits se décompose, du plus significatif au moins significatif, en 16 bits inutilisés, 4 champs de 9 bits pour indexer les 512 entrées dans le noeud de la table de pages au niveau correspondant, et 12 bits qui spécifient l'octet dans la page de 4Kio. Chaque noeud feuille de la table de pages couvre 512 pages de 4Kio, soit 2Mio. Nous avons donc 2 noeuds feuille (au début et à la fin de l'espace adressable), chacun avec 1 parent et 1 grand-parent, pour un total de 6 noeuds pour ces 3 niveaux. iv) Au premier niveau, il y a une racine commune, ce qui ajoute un 7ème noeud.

b) Un fil d'exécution sur le coeur 0 place des valeurs mises à jour dans les variables `data_1_a` et `data_1_b`, et change ensuite `data_ready_1` à 1 pour indiquer que ces variables sont prêtes. Il fait la même chose avec les variables `data_2_a` et `data_2_b`, et change ensuite `data_ready_2` à 1. Initialement, toutes les variables sont à 0. Avec la librairie standard du langage C, des barrières mémoire doivent être ajoutées pour s'assurer que ces valeurs ne sont utilisées sur le coeur 1 que si elles ont été mises à jour. A cette fin, une barrière mémoire a été ajoutée sur le fil d'exécution du coeur 0. Doit-on aussi ajouter des barrières mémoire sur le fil d'exécution du coeur 1? i) A quel endroit? ii) Quelles sont les valeurs finales possibles pour `new1` et `new2` si les bonnes barrières mémoire ont été ajoutées? **(2 points)**

```
Coeur 0                          Coeur 1

data_1_a = 25;           if(data_1_ready) {
data_1_b = 7;              new1 = data_1_a + data_1_b;
data_2_a = 22;           }
data_2_b = 11;
cmm_smp_wmb();           if(data_2_ready) {
data_1_ready = 1;          new2 = data_2_a + data_2_b;
data_2_ready = 1;        }
```

Oui, il faut avoir des barrières mémoire des deux côtés. i) Il faut placer la barrière mémoire pour s'assurer que si `data_ready_1` est à 1, alors nous devons nous assurer que toutes les mises à jour des autres variables nous arrivent, d'où une barrière mémoire de lecture entre le moment où on voit que `data_ready_1` est à 1 et l'accès aux autres variables. La même chose s'applique pour `data_ready_2`. ii) Si `data_ready_1` est à 0 au moment où le coeur 1 fait son test, rien n'est exécuté et `new1` reste à 0. Autrement, `new1` verra les 2 valeurs mises à jour et deviendra $25+7=32$. De la même manière, `new2` sera soit 0, soit $22+11=33$.

```
(initialement toutes les variables sont à 0)
```

```
Coeur 0                          Coeur 1

data_1_a = 25;          if(data_1_ready) {
data_1_b = 7;             cmm_smp_rmb();
data_2_a = 22;            new1 = data_1_a + data_1_b;
data_2_b = 11;          }
cmm_smp_wmb();          if(data_2_ready) {
data_1_ready = 1;         cmm_smp_rmb();
data_2_ready = 1;         new2 = data_2_a + data_2_b;
                        }
```

c) La plupart des circuits de mémoire cache utilisent les protocoles MESI ou MOESI pour en assurer la cohérence. Quel état les différencie? Donnez un exemple concret de comportement différent possible, selon qu'une cache utilise l'un ou l'autre protocole. **(1 point)**

La différence est le mode Owned, le bloc est modifié par rapport à la copie en mémoire centrale et d'autres cache peuvent en avoir une copie. Avec MESI, si un cache veut avoir une copie d'un bloc modifié, cela doit passer par une mise à jour de la mémoire centrale. Avec MOESI, une copie d'un bloc modifié peut être envoyée à un autre cache avant que la mise à jour n'ait été envoyée en mémoire centrale.

Question 4 (5 points)

a) Le programme OpenMP suivant est exécuté. Donnez une version possible des lignes qui seront générées en sortie. Expliquez si l'ordre entre les lignes, ou leur contenu, peuvent changer d'une exécution à l'autre et comment. **(2 points)**

```
int main(int argc, char **argv)
{ int t = -1, nbt = -1;
  printf("a) thread %d / %d\n", t, nbt);
  omp_set_num_threads(4);
  #pragma omp parallel private(t, nbt)
  { nbt = omp_get_num_threads();
    t = omp_get_thread_num();
    printf("b) thread %d / %d\n", t, nbt);
    #pragma omp for schedule(static,2)
    for(int i = 0; i < 7; i++)
      printf("c) thread %d / %d, i = %d\n", t, nbt, i);
    #pragma omp single
    printf("d) thread %d / %d\n", t, nbt);
    #pragma omp masked
    printf("e) thread %d / %d\n", t, nbt);
  }
  printf("f) thread %d / %d\n", t, nbt);
}
```

La ligne a) arrive toujours en premier. les lignes b) et c) entre les fils d'exécution peuvent être mélangées mais restent en séquence pour un même fil. La ligne d) peut être imprimée par n'importe

lequel fil, mais par un seul. Il y a une barrière implicite à la fin du pragma single, si bien que e) est toujours après les b), c) et le d). Le f) vient après le e). Puisque la clause schedule(static,2) est spécifiée, les valeurs de i associées à chaque fil d'exécution ne changent pas d'une exécution à l'autre.

```
a) thread -1 / -1
b) thread 0 / 4
c) thread 0 / 4, i = 0
c) thread 0 / 4, i = 1
b) thread 1 / 4
c) thread 1 / 4, i = 2
c) thread 1 / 4, i = 3
b) thread 3 / 4
c) thread 3 / 4, i = 6
b) thread 2 / 4
c) thread 2 / 4, i = 4
c) thread 2 / 4, i = 5
d) thread 1 / 4
e) thread 0 / 4
f) thread -1 / -1
```

b) Dans le programme OpenMP suivant, les éléments des vecteurs b et c sont initialisés à 0.0, et la matrice a est initialisée avec des données à traiter. Le programme sera exécuté sur un ordinateur 64 bits avec les blocs de cache L1 d'une taille de 128 octets. La valeur de nb est un paramètre de configuration que l'usager doit choisir. On veut optimiser la performance de ce programme tout en s'assurant d'un comportement correct (toujours le même comportement attendu, pas de course non déterministe). i) Quelle valeur de nb recommandez-vous et pourquoi? ii) Est-ce que le programme comporte une course, et si oui laquelle? iii) Serait-il avantageux de changer l'ordre des boucles, j sur NCA d'abord et i sur NRA imbriquée ensuite? iv) Suggérez une optimisation possible. **(2 points)**

```
double a[NRA][NCA], b[NRA], c[NCA];
int nb;

int main(int argc, char **argv)
{
  int i, j;
  omp_set_num_threads(8);
  #pragma omp parallel for schedule(static,nb) shared(a,b,c) private(i,j)
  for (i = 0; i < NRA; i++) {
    for (j = 0; j < NCA; j++) {
      b[i] += a[i][j];
      c[j] += a[i][j];
} } }
```

Les cases consécutives de b seront accédées par des fils d'exécution différents si nb=1, ce qui causera un faux partage et risque de nuire sérieusement à la performance. i) Si on prend n = 128 octets/bloc / 8octets/mot = 16 mots/bloc, les différents fils prendront des cases de b dans des blocs différents et cela évitera le faux partage. ii) Il y a une course dans ce programme au niveau du calcul de c. En effet,

différents fils traitent différentes valeurs de i mais peuvent accéder simultanément les mêmes valeurs de j, créant une course sur c[j]. iii) L'ordre des boucles est présentement bon, car on accède l'indice le plus à droite dans la boucle la plus imbriquée (mots consécutifs en mémoire accédés par le même fil). iv) Une optimisation possible serait de faire le calcul de b[i] dans une variable temporaire dans la boucle, permettant à ce calcul de se faire dans un registre. On pourrait aussi demander une réduction sur c[j], de manière à avoir une copie locale de c pour chaque fil d'exécution. Ces copies seraient ensuite sommées par la réduction, ce qui réduirait les fautes de cache sur c et enlèverait la course.

c) Le calcul matriciel suivant est effectué en parallèle à l'aide de OpenMP. Est-ce que i, j, k est l'ordre d'imbrication le plus efficace pour les trois boucles? Sinon, quel est-il? Expliquez. **(1 point)**

```
#pragma omp parallel for shared(a,b,c) private(i,j,k)
for(i=0; i<NRA; i++)
  for(j=0; j<NCB; j++)
    for(k=0; k<NCA; k++)
      c[i][j] += a[i][k] * b[k][j];
```

En raison de c[i][j], on veut mettre i avant j, a[i][k] implique de mettre i avant k, et b[k][j] implique k avant j. Un ordre qui satisfait tout cela est i, ensuite k, ensuite j, c'est à dire échanger les boucles sur j et k.

Le professeur: Michel Dagenais