POLYTECHNIQUE MONTRÉAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2021)
3 crédits (3-1.5-4.5)

CORRIGÉ DU CONTRÔLE PÉRIODIQUE

DATE: Jeudi le 28 octobre 2021

HEURE: 13h45 à 15h35

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 ordinateur avec une mémoire centrale de 8GiO et une mémoire cache de données de 64KiO. La taille des blocs en cache est de 128 octets. La section de programme suivante s'exécute sur cet ordinateur. Chaque entier occupe 8 octets sur cet ordinateur 64 bits. Dans ce programme, les entiers du vecteur `buffer` sont accédés en boucle. Quel sera le taux de succès en mémoire cache, pour les accès au vecteur `buffer`, si la constante `STRIDE` vaut 1? Si elle vaut 2? **(2 points)**

```
// i, sum et buffer[1000000] sont des entiers, le cache est vide
for(i = 0; i < 1000000; i += STRIDE) sum += buffer[i];
```

Le vecteur `buffer` est beaucoup plus grand que la mémoire cache et ne peut donc y résider. Les mots de `buffer` doivent ainsi être chargés en mémoire cache avant d'être utilisés. Lors de l'accès à `buffer[0]`, une faute de cache survient et le bloc de 128 octets (16 entiers de 8 octets) qui contient cette valeur est chargé en cache. Ensuite, avec `STRIDE = 1`, les 15 accès suivants tombent dans la même ligne de cache et peuvent être complétés sans faute de cache. Il en sera de même pour tous les blocs suivants. Le taux de succès en cache est donc de 15/16, soit 93.75%. Dans le cas où `STRIDE = 2`, seulement 1 mot sur 2 du bloc sera utilisé, soit 8 des 16 entiers qu'il contient. Ceci donnera un taux de succès en cache de 7/8, soit 87.5%.

b) Une grappe de calcul parallèle contient 8 noeuds. Chaque noeud est constitué d'une carte mère, qui a une probabilité de disponibilité de 0.9, et d'un système de deux disques redondants en miroir. Un noeud est disponible si la carte mère et le système de disques sont disponibles. Le système de disques est disponible si au moins 1 des 2 disques est disponible. Chaque disque a une probabilité de disponibilité de 0.75. Quelle est la probabilité que les 8 noeuds soient disponibles simultanément? **(2 points)**

Le système de disques est disponible sauf si les deux disques sont en panne, une probabilité de $1-(1-0.75)^2 = 0.9375$. *On peut aussi sommer la probabilité que les deux disques soient fonctionnels* $1 \times 0.75^2 \times (1-0.75)^0 = 0.5625$ *et qu'exactement un disque le soit* $\binom{2}{1} \times 0.75^1 \times (1-0.75)^{2-1} = 0.375$, *pour un total de* $0.5625 + 0.375 = 0.9375$ *pour obtenir un résultat identique. Un noeud est disponible si ses deux composantes le sont, carte mère et système de disques,* $0.9 \times 0.9375 = 0.8437$. *La probabilité que les 8 noeuds soient disponibles simultanément est de* $0.8437^8 = 0.256868167$.

c) Un programme s'exécute sur un coeur et prend 100s. La fraction parallélisable de ce programme est de 0.95. Que deviendra le temps d'exécution de ce programme s'il s'exécute en parallèle sur 32 coeurs? Sur 64 coeurs? **(1 point)**

Sur les 100s, 5s sont séquentielles et ne changent pas avec le nombre de coeurs et 95s sont parallélisables. Le temps sur 32 coeurs devient 5s + 95s / 32 = 7.96875s. Avec 64 coeurs on a 5s + 95s / 64 = 6.484375s.

Question 2 (5 points)

a) Un ordinateur 64 bits à mémoire virtuelle utilise une table de pages à 3 niveaux et des pages de 64KiO. Chaque noeud dans la table de pages occupe une page. Les 9 bits les plus significatifs de l'espace adressable sont inutilisés. Quelle est la taille d'une entrée dans la table de page? Combien d'entrées

sont contenues dans un noeud de la table de pages? Comment se décompose l'adresse de 64 bits en bits inutilisés, index de niveau 0, index de niveau 1, index de niveau 2 et décalage dans la page? **(2 points)**

Chaque entrée dans la table de pages donne l'adresse physique de 64 bits, soit 8 octets, correspondant à l'adresse virtuelle de la page recherchée. Quelques bits additionnels donnent le statut de la page mais ils sont stockés dans les bits les moins significatifs de cette adresse de 64 bits. En effet, les derniers bits seraient autrement inutiles, toujours à zéro, puisque les pages sont toujours alignées sur un multiple de leur taille. Une page de 64KiO (2^{16} ou 65536 octets) peut donc contenir $65536/8 = 8192$ entrées. Il faut ainsi 13 bits pour indexer chaque niveau de la table de pages ($2^{13} = 8192$) et 16 bits pour le décalage dans la page (2^{16} = 65536 = 64KiO). Ceci donne donc 9 bits inutilisés, 13 bits index niveau 0, 13 bits index niveau 1, 13 bits index niveau 2 et 16 bits de décalage dans la page, pour un total de 64 bits.

b) Le programme TBB suivant s'exécute. Quelle sera la sortie affichée sur `cout`? Expliquez! **(2 points)**

```
struct Comp {
  float value;
  Comp() : value(0) {}
  Comp(Comp& c, split) { value = 0; }
  void operator()(const blocked_range<float*>& r) {
    float temp = value;
    for(float* a=r.begin(); a!=r.end(); ++a) temp += *a;
    value = temp;
  }
  void join(Comp& rhs) {value += rhs.value;}
};

int main(int argc, char *argv[]) {
  Comp comp; float array[1000];
  for(int i = 0; i < 1000; i++) array[i] = i % 10;
  parallel_reduce(blocked_range<float*>(array, array+1000),comp);
  cout << "Value: " << comp.value << "\n";
}
```

Le vecteur `array` contient: `{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4...}`. Le programme TBB utilise `parallel_reduce` pour calculer la somme partielle en parallèle dans différentes sections de `array` avec `operator()` de la classe `Comp`. Ensuite, la somme partielle dans chaque section est cumulée dans la phase de réduction avec la méthode `join` de la classe `Comp`. Le résultat placé dans le champ `value` de l'objet `comp` de la fonction `main` est donc la somme de tous les éléments du vecteur `array`. Puisque ce vecteur contient 100 fois la séquence de 0 à 9, la somme est donc de $100 \times (0+1+2+3+4+5+6+7+8+9) = 4500$. Le programme affiche ainsi la ligne `Value: 4500`.

c) Que permet de spécifier la fonction `pthread_attr_setstacksize` de la librairie pthread? Qu'est-ce qui détermine la taille requise pour cette valeur? **(1 point)**

Cette fonction permet de spécifier la taille de la pile pour les threads à créer. Cette taille doit être suffisante pour placer sur la pile toutes les variables locales et tous les registres sauvés pour les appels imbriqués du thread dans le pire cas. Si l'arbre d'appel complet est connu, il faut regarder, pour chaque feuille, la taille requise pour ce chemin d'appel et prendre la valeur maximale trouvée.

Question 3 (5 points)

a) Un programme multi-thread s'exécute sur 2 coeurs en parallèle. Afin d'aller plus vite, il n'utilise pas de verrou. Un thread veut mettre à jour des valeurs, `data1_a` et `data1_b` puis `data2_a` et `data2_b`, et indiquer lorsqu'elles sont valides en mettant à 1 `valid1` et `valid2`, respectivement. Le programme proposé suit. Doit-on ajouter des barrières mémoire pour avoir un comportement correct (i.e., ne pas lire les données avant qu'elles n'aient été mises à jour)? Si oui, modifiez le programme de chacun des deux coeurs en ajoutant les barrières mémoire minimales requises. **(2 points)**

```
(initialement toutes les variables sont à 0)
Coeur 0                          Coeur 1

data1_a = 16;                    if(valid1) {
data1_b = 14;                      new1 = data1_a + data1_b;
data2_a = 5;                     }
data2_b = 7;                     if(valid2) {
valid1 = 1;                        new2 = data2_a - data2_b;
valid2 = 1;                      }
```

Il faut insérer une barrière d'écriture après avoir mis à jour les valeurs et avant de mettre valid1 et valid2 à 1. Ceci assure que toutes les valeurs ont bien été écrites en mémoire centrale avant de ne confirmer le tout par le changement à 1 de valid1 et valid2. Une seule barrière mémoire suffit pour les deux, puisque les accès aux data1 et data2 sont regroupés. Du côté du coeur 1, il faut s'assurer que les mises à jour sont bien arrivées avec une barrière de lecture, après avoir lu que valid1 est à 1, mais avant de lire data1. La même chose est requise après avoir vu valid2 à 1, avant de lire data2. Il n'y a pas de garantie d'ordre d'arrivée entre valid1 et valid2, et il se peut que l'un soit à 1 et l'autre à 0. Il faut donc une barrière de lecture pour chacune des deux conditions, si on assume que les deux sont indépendantes.

```
(initialement toutes les variables sont à 0)
Coeur 0                          Coeur 1

data1_a = 16;                    if(valid1) {
data1_b = 14;                      cmm_smp_rmb();
data2_a = 5;                       new1 = data1_a + data1_b;
data2_b = 7;                     }
cmm_smp_wmb();                   if(valid2) {
valid1 = 1;                        cmm_smp_rmb();
valid2 = 1;                        new2 = data2_a - data2_b;
                                 }
```

b) Sur certains processeurs, les écritures à des adresses différentes vers la mémoire centrale peuvent être réordonnancées (Partial Store Order). Expliquez quels mécanismes matériels sur un processeur peuvent expliquer que de tels réordonnancements sont possibles? **(2 points)**

Sur les architectures superscalaires, deux instructions ou plus peuvent s'exécuter en parallèle. Il est possible qu'une instruction bloque en raison d'une faute de cache ou d'une dépendance, alors que

l'autre, possiblement postérieure dans le code, s'exécute sans bloquer et termine en premier, causant un réordonnancement. Un autre mécanisme qui peut causer un réordonnancement est les queues d'écritures. S'il y a deux queues d'écritures ou plus, il se peut qu'une queue soit plus congestionnée et cause un plus grand délai, ce qui permet aux écritures dans l'autre queue des les dépasser, causant là aussi un réordonnancement.

c) Quelle est la différence entre les protocoles de cohérence de cache MESI et MOESI? En quoi consiste l'état qui diffère entre les deux? **(1 point)**

La différence entre ces deux protocoles est l'existence de l'état Owned *dans le protocole MOESI. Cet état signifie que le bloc est possiblement partagé mais que la copie en mémoire centrale n'a possiblement pas encore été mise à jour. Avec MESI, avant qu'un bloc ne puisse être partagé, il faut que la copie en mémoire centrale soit mise à jour.*

Question 4 (5 points)

a) Le programme OpenMP suivant est exécuté. Donnez les lignes qui seront générées en sortie. Pour chaque ligne, dites si le nombre affiché peut changer d'une exécution à l'autre. Laquelle des 3 boucles sera la plus rapide? La plus lente? Pourquoi **(2 points)**

```
int main(int argc, char **argv)
{ int i, sum = 0;
  omp_set_num_threads(4);
  #pragma omp parallel for
  for(i = 0; i < 1000000; i++) sum += 1;
  printf("sum = %d\n", sum);
  sum = 0;
  #pragma omp parallel for
  for(i = 0; i < 1000000; i++) {
    #pragma omp atomic update
    sum += 1;
  }
  printf("atomic sum = %d\n", sum);
  sum = 0;
  #pragma omp parallel for
  for(i = 0; i < 1000000; i++) {
    #pragma omp critical
    sum += 1;
  }
  printf("critical sum = %d\n", sum);
}
```

Ce programme génère la sortie suivante. Pour la première boucle, la variable sum est accédée en parallèle par 4 threads et sa valeur peut changer car plusieurs incréments peuvent être sautés, lorsque plus d'un thread prennent la valeur de `sum` *en même temps, avant de l'incrémenter chacun séparément. Pour les deux dernières boucles, l'opération atomique ou la section critique assurent qu'un thread complète son incrément avant qu'un autre thread puisse prendre la valeur de sum. La première*

boucle sera la plus rapide, car il n'y a pas d'opération de synchronisation. L'opération atomique est normalement plus rapide qu'une section critique, car celle-ci utilise une opération atomique à la base et fait des opérations en plus.

```
sum = 552599
atomic sum = 1000000
critical sum = 1000000
```

b) Le programme OpenMP suivant est exécuté. Donnez les 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)\n");
  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,1)
    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) est toujours la première et la ligne f) la dernière. Pour chaque thread, les lignes b) et c) sont toujours dans le même ordre. L'ordre entre les lignes b) et c) peut varier d'un thread à l'autre. La ligne d) arrive après toutes les lignes b) et c) en raison de la barrière implicite à la fin du pragma omp for. La ligne d) n'apparaît que pour un seul thread, n'importe lequel (présumément le premier qui passe par cette section). La ligne e) arrive après la ligne d), en raison de la barrière implicite à la fin du pragma omp single. Le contenu de la ligne d) peut changer d'une exécution à l'autre, car le thread choisi pour exécuter la section single *peut changer. Le contenu des autres lignes ne change pas.*

```
a)
b) thread 0 / 4
c) thread 0 / 4, i = 0
c) thread 0 / 4, i = 4
b) thread 3 / 4
c) thread 3 / 4, i = 3
b) thread 2 / 4
```

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

c) Un processeur contient 32 coeurs. Vous désirez exécuter un programme parallèle multi-thread sur ce processeur. Le programme effectue surtout des calculs et fait un peu d'entrées-sorties. Combien de threads suggérez-vous d'utiliser? Pourquoi? **(1 point)**

Pour un programme qui ne fait que des calculs, il est habituellement plus efficace d'avoir un thread par coeur. Ainsi, tous les coeurs sont occupés, et il n'y a pas de temps perdu dans les changements de contexte entre les threads, même si cette perte n'est pas très grande. Lorsqu'il y a des entrées-sorties, les threads peuvent être en attente et il est plus efficace d'avoir 2 threads par coeur, afin de ne pas laisser un coeur inutilisé, lorsqu'un thread est bloqué en attente d'entrée-sortie. Il est donc suggéré dans ce cas d'utiliser 2 threads par coeur, soit 64 threads.

Le professeur: Michel Dagenais