POLYTECHNIQUE MONTRÉAL

Département de génie informatique et génie logiciel

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

CORRIGÉ DU CONTRÔLE PÉRIODIQUE

DATE: Mercredi le 23 octobre 2024

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 ordinateur multi-coeur 32 bits, dont la mémoire est adressable à l'octet, possède un cache de données de 16Kio, avec des ensembles de 2 blocs, pour chacun de ses 8 coeurs. Chaque bloc en cache contient 64 octets. Ce cache est initialement vide. i) Combien de blocs et d'ensembles contient le cache de chaque coeur de processeur? ii) Comment se décomposent les 32 bits d'adresse en: décalage dans le bloc, numéro d'ensemble et étiquette? Les adresses suivantes sont accédées en séquence, en lecture (R) ou en écriture (W), sur les processeurs spécifiés (P0 à P7). iii) Donnez l'état (M, E, S ou I) des blocs non vides du cache après chaque accès. **(2 points)**

```
P0 W 0x32B 7;P3 W 0x32B 5;P7 W 0x32B 9;P1 R 0x32B;P2 R 0x43B;
```

Pour des blocs de 64 octets, les 6 bits d'adresse les moins significatifs représentent le décalage dans le bloc. Le cache contient $2^{14}/2^6 = 2^8 = 256$ *blocs de 64 octets soit* $256/2 = 128$ *ensembles de 2 blocs. Il faut donc 7 bits pour représenter l'ensemble en cache et le reste,* $32-6-7 = 19$ *constitue l'étiquette du bloc (qui distingue un bloc des autres qui peuvent se retrouver dans le même ensemble). Pour l'adresse 0x0000032B, cela donne 0b0000 0000 0000 0000 0000 0011 0010 1011 ou 0b0000 0000 0000 0000 000|0 0011 00|10 1011, soit étiquette 0, ensemble 0x0A, décalage 0x2B. Pour l'adresse 0x043B, cela donne 0b0000 0000 0000 0000 0000 0011 0011 1011 ou 0b0000 0000 0000 0000 000|0 0100 00|11 1011, soit étiquette 0, ensemble 0x10, décalage 0x3B. Initialement, tous les blocs en cache ont le statut invalide (I). La première opération fait charger un bloc dans le cache de P0 (premier bloc de l'ensemble 0x0A, étiquette 0, statut M). La seconde opération invalide le bloc du cache de P0 (qui sera réécrit en mémoire puisque modifié) et le charge dans le cache de P3 (premier bloc de l'ensemble 0x0A, étiquette 0, statut M). La troisième opération concerne toujours le même bloc et invalide le bloc du cache de P3 (qui sera réécrit em mémoire puisque modifié) et le charge dans le cache de P7 (premier bloc de l'ensemble 0x0A, étiquette 0, statut M). Ce même bloc est ensuite lu par P1, il est réécrit en mémoire par P7, puisque modifié, et son statut devient S avant d'être chargé par P1 (premier bloc de l'ensemble 0x0A, étiquette 0, statut S). Finalement, un bloc différent est chargé dans le cache de P2 (premier bloc de l'ensemble 0x10, étiquette 0, statut E).*

b) Trois serveurs de données redondants, chacun contenant une unité de disque RAID de 5 disques, sont utilisés pour alimenter une grappe de calcul. Chaque serveur a une probabilité de panne de 0.1 pour son électronique, de 0.15 pour l'électronique de l'unité de disque RAID et de 0.2 pour chaque disque dans les unités de disque RAID. Un serveur fonctionne en autant que son électronique, l'électronique de l'unité de disque RAID et au moins 3 disques sur les 5 sont opérationnels. i) Quelle est la probabilité globale de panne pour 1 serveur? ii) Pour les 3 serveurs simultanément? **(2 points)**

Une unité de disque RAID est fonctionnelle si 3, 4 ou 5 disques sont fonctionnels. Pour 5 disques fonctionnels: $\frac{5!}{(5!\times 0!)} \times (1-0.2)^5 \times 0.2^0 = 0.32768$. *Pour exactement 4 disques fonctionnels:* $\frac{5!}{(4!\times 1!)} \times (1-0.2)^4 \times 0.2^1 = 0.4096$. *Pour exactement 3 disques fonctionnels:* $\frac{5!}{(3!\times 2!)} \times (1-0.2)^3 \times 0.2^2 = 0.2048$. *Le total est donc pour 3, 4 ou 5 disques de:* $0.32768+0.4096+0.2048 = 0.94208$. *Un serveur est fonctionnel si ses 3 composantes sont fonctionnelles (électronique, électronique de l'unité RAID, disques) simultanément:* $(1-0.1)\times(1-0.15)\times 0.94208 = 0.7206912$. *La probabilité de panne pour 1 serveur est donc de:* $(1-0.7206912) = 0.2793088$. *La probabilité que les 3 serveurs soient en panne simultanément, et donc le service soit indisponible malgré sa redondance, est de:* $0.2793088^3 = 0.021789831$

c) Un programme qui s'exécute sur un seul fil d'exécution (thread) a une fraction parallélisable de 0.8. Quel est le facteur d'accélération qu'il sera possible d'obtenir si on l'exécute sur un processeur avec de nombreux coeurs (e.g. 64 coeurs) avec 8 fils d'exécution? Avec 16 fils d'exécution? **(1 point)**

Selon la loi d'Amhdal, $acc = 1/((1-f)+f/n)$. *Pour* $f = 0.8$ *et* $n = 8$ *ou* $n = 16$, *cela donne 3.3333 ou 4, respectivement.*

Question 2 (5 points)

a) Le programme suivant crée un certain nombre de fils d'exécution POSIX. A chaque fil créé, un message est imprimé avec un identificateur qui indique la position de ce fil dans l'arborescence de création de fils d'exécution.

i) Quel est le nombre de fils d'exécution créés à l'aide de pthread_create par ce programme? ii) Donnez une sortie possible de ce programme? **(2 points)**

```
void *create_child_threads(void *arg) {
  int id = (uintptr_t)arg, level = 0;
  pthread_t t;
  fprintf(stderr, "T%d ", id);
  for(int tmp = id; tmp > 0; level++) tmp = tmp / 10;
  if(level <= 2) {
    int nb_child = 3 - level;
    for(int i = 0; i < nb_child; i++)
      pthread_create(&t,NULL,create_child_threads,(void *)((uintptr_t)(id*10+i+1)));
  }
  pthread_exit(NULL);
}

int main(int argc, char **argv) {
  create_child_threads((void *)((uintptr_t)0));
  return(0);
}
```

Il y a 3 + 3 x 2 + 3 x 2 x 1 = 15 fils d'exécution créés. Une sortie possible est la suivante. L'ordre entre les valeurs imprimées peut changer.

T0 T1 T3 T2 T21 T11 T12 T111 T22 T211 T31 T221 T32 T121 T311 T321

b) Un ordinateur 64 bits, semblable au x86-64, dont la mémoire est adressable à l'octet, utilise des tables de pages à 4 niveaux (en fait un arbre de noeuds, chacun occupant une page, à 4 niveaux) et des pages de 4KiO. Les 16 bits les plus significatifs de l'adresse sont inutilisés. Un programme ajoute à son espace adressable une zone de 20MiO calquée sur un fichier. La table de page doit donc être augmentée pour couvrir ces nouveaux 20MiO. i) Comment se décompose l'adresse en ses différents champs (décalage dans la page, index à chaque niveau dans la table de pages...). ii) Combien de noeuds devront être ajoutés à l'arbre pour couvrir ces 20MiO au minimum? Au maximum? **(2 points)**

Dans l'adresse, les 16 bits les plus significatifs sont inutilisés, le décalage dans la page occupe les 12 bits les moins significatifs ($2^12 = 4096$ ou 4KiO). Les 36 bits entre les deux représentent les 4 index de 9 bits, un pour chaque niveau de la table de pages. En effet, chaque noeud de 4KiO dans la table de pages contient des entrées de 8 octets (64 bits), soit 4096/8 = 512 entrées, ce qui requiert un index de 9 bits ($2^9 = 512$).

L'espace de 20MiO représente 20MiO / 4KiO = 5x1024 = 5120 pages de mémoire physique. Ceci requiert autant d'entrées qui occupent 5120/512= 10 noeuds au niveau L1 dans la table des pages dans le meilleur des cas. Dans le meilleur cas, le noeud parent au niveau L2 était pré-existant et les 10 entrées étaient donc libres. Ceci donne donc un minimum de 10 noeuds à ajouter.

Dans le pire cas, seulement la racine de l'arbre existe pour cette région de mémoire virtuelle, au niveau L4, et les entrées se répartissent au niveau L1 sur 11 noeuds. Les 11 entrées correspondantes pourraient se répartir sur 2 noeuds au niveau L2 et encore 2 noeuds au niveau L3. Le nombre total de noeuds ajoutés serait donc de 11 + 2 + 2 = 15.

c) Pour un ordinateur qui utilise des adresses de 64 bits, comme l'architecture x86-64, chaque entrée dans la table de page a une longueur de 64 bits. Cependant, pour chaque entrée, on veut mémoriser l'adresse physique de la page (adresse de 64 bits) ainsi que plusieurs bits de statut comme les permissions (valide, lecture, écriture, exécution) pour cette page. Comment peut-on placer toute cette information dans une entrée de 64 bits? **(1 point)**

Puisque les pages de mémoire physique sont toujours à une adresse qui est un multiple de 4KiO, les 12 bits les moins significatifs sont nécessairement à 0 et il n'est pas nécessaire de les mémoriser dans les entrées de la table de page. De plus, les 16 bits les plus significatifs sont inutilisés et pourraient eux aussi être mis à profit.

Néanmoins, ces bits inutilisés sont généralement conservés pour des besoins d'expansion future de la capacité mémoire des prochaines générations de processeurs.

Question 3 (5 points)

a) La fonction suivante `CheckVector` est appelée avec comme argument `v` un vecteur de 30 `float` {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 0.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0, 0.0, 21.0, 22.0, 33.0, 24.0, 25.0, 26.0, 27.0, 28.0, 29.0, 0.0}, et `size` un entier qui vaut 30. Expliquez brièvement ce que fait le programme. Quelle est la valeur retournée par la fonction à la fin? **(2 points)**

```
struct Count {
  int value; float *v;
  Count() : value(0) {}
  Count( Count& c, split ) { v = c.v; value = 0; }

  void operator()( const blocked_range<int>& r ) {
    int temp_count = value;
    for(int i = r.begin(); i != r.end(); i++ )
      if(v[i] < v[i - 1]) temp_count++;
    value = temp_count;
  }
  void join( Count& rhs ) {value += rhs.value;}
};

int CheckVector(float v[], int size) {
  Count c; c.v = v;
  parallel_reduce( blocked_range<int>(1, size), c);
  return c.value;
}
```

Le programme vérifie si le vecteur est en ordre croissant. Il incrémente un décompte à chaque fois qu'un nombre décroit par rapport au nombre précédent. Ceci arrive 4 fois dans ce vecteur et la fonction retourne donc la valeur 4.

b) La section de code suivante s'exécute sur un ordinateur avec ordonnancement faible. Les variables f1, f2, f3, valid1 et valid2 sont initialement à 0. Quelles barrières devrait-on insérer, et où, pour avoir comme seules sorties possibles: `0 0 0` , `22 34` ou `22 34 42` ? **(2 points)**

```
Processeur 0                    Processeur 1

f1 = 22;                          if(valid1) {
f2 = 34;                            if(valid2) {
valid1 = 1;                           printf("%d %d %d\n", f1, f2, f3);
f3 = 42;                            } else {
valid2 = 1;                           printf("%d %d\n", f1, f2);
                                    } else {
                                      printf("0 0 0\n");
                                    }
```

La barrière d'écriture, cmm_smp_wb(), assure que ce qui a été écrit avant sera propagé avant en mémoire. Ainsi, les nouvelles valeurs de f1 et f2 (f3) seront propagées de P0 vers la mémoire centrale avant valid1=1 (valid2=1). De l'autre côté, pour la lecture sur P1, il faut s'assurer, avec une barrière de lecture, cmm_smp_rmb(), que si valid1 (valid2) n'est plus 0, les autres variables, f1 et f2 (f3), auront aussi eu leurs anciennes valeurs invalidées en cache, afin que tout accès obtienne les nouvelles valeurs.

```
Processeur 0                    Processeur 1

f1 = 22;                          if(valid1) {
f2 = 34;                            if(valid2) {
cmm_smp_wmb()                         cmm_smp_rmb()
valid1 = 1;                           printf("%d %d %d\n", f1, f2, f3);
f3 = 42;                            } else {
cmm_smp_wmb()                         cmm_smp_rmb()
valid2 = 2;                           printf("%d %d\n", f1, f2);
                                    } else {
                                      printf("0 0 0\n");
                                    }
```

c) Pour implémenter un verrou, il est possible de faire une boucle sur un échange atomique. On échange le contenu du verrou avec la valeur 1. Si l'échange donne 1, le verrou était pris et on boucle jusqu'à ce que la valeur 0 soit obtenue, ce qui indique que le verrou était libre. Une implémentation plus efficace est d'abord de boucler sur une lecture du verrou pour attendre que le verrou soit libre, et ensuite essayer un échange atomique. Le code suivant illustre cette différence. Pourquoi est-ce plus efficace? **(1 point)**

```
        DADDUI R2, R0, #1
lockit: EXCH R2, 0(R1)
        BNEZ R2, lockit
```

```
lockit: LD R2, 0(R1)
        BNEZ R2, lockit
        DADDUI R2, R0, #1
        EXCH R2, 0(R1)
        BNEZ R2, lockit
```

L'échange atomique implique une opération coûteuse sur le coeur de processeur concerné, mais aussi sur le circuit de gestion de la cohérence entre les caches et la mémoire centrale. De plus, une écriture (causée par l'échange) sur le verrou forcera l'invalidation de ce bloc dans les caches des autres processeurs. Cela peut causer beaucoup d'activité au niveaux des caches si ce verrou est en forte contention. L'implémentation plus sophistiquée, qui attend que le verrou soit libre en bouclant sur sa lecture, peut éviter un grand nombre d'opérations atomiques et d'invalidations de blocs en cache, ce qui sauve des ressources au niveau des caches.

Question 4 (5 points)

a) Le programme OpenMP suivant est exécuté. Donnez les lignes qui seront générées en sortie. **(2 points)**

```
int main(int argc, char **argv)
{ int nb_thread, thread, p, s;
  omp_set_num_threads(3);
  #pragma omp parallel private(thread, nb_thread)
  { printf("a)\n");
    for(int i = 0; i < 3; i++) {
      nb_thread = omp_get_num_threads();
      thread = omp_get_thread_num();
      printf("b) thread %d / %d, i = %d\n", thread, nb_thread, i);
    }
    printf("c) thread %d / %d\n", thread, nb_thread);
    #pragma omp for schedule(static,1)
    for(int i = 0; i < 4; i++) {
      nb_thread = omp_get_num_threads();
      thread = omp_get_thread_num();
```

```
      printf("d) thread %d / %d, i = %d\n", thread, nb_thread, i);
    }
  }
}
```

Ce programme génère la sortie suivante. L'ordre entre les sorties produites par les différents fils d'exécution, entre deux barrières de rendez-vous implicites, peut changer d'une exécution à l'autre.

```
a)
b) thread 1 / 3, i = 0
b) thread 1 / 3, i = 1
b) thread 1 / 3, i = 2
c) thread 1 / 3
d) thread 1 / 3, i = 1
a)
b) thread 0 / 3, i = 0
b) thread 0 / 3, i = 1
b) thread 0 / 3, i = 2
c) thread 0 / 3
d) thread 0 / 3, i = 0
d) thread 0 / 3, i = 3
a)
b) thread 2 / 3, i = 0
b) thread 2 / 3, i = 1
b) thread 2 / 3, i = 2
c) thread 2 / 3
d) thread 2 / 3, i = 2
```

b) On vous fournit la section de code suivante en OpenMP. Les matrices `double` a[NRA][NCA] et b[NCA][NCB] ont été initialisées aux valeurs du problème et la matrice c[NRA][NCB] et le vecteur d[ND] `double` ont été initialisés à 0. i) Vérifiez que le code est correct (déterministe en présence de fils d'exécution parallèles), et suggérez au besoin un correctif. ii) Proposez au moins deux améliorations, par rapport au programme avec votre correctif inclus, qui permettront d'obtenir le même résultat mais plus efficacement. Expliquez brièvement chaque amélioration apportée. **(2 points)**

```
int main(int argc, char **argv) {
  #pragma omp parallel for shared(a,b,c,d) private(i,j,k)
  for (int i=0; i<NRA; i++)
    for(int j=0; j<NCB; j++)
      for(int k=0; k<NCA; k++) {
        c[i][j] += a[i][k] * b[k][j];
        d[(int)(c[i][j]) % ND]++;
      }
}
```

L'accès dans d n'est pas synchronisé et n'est donc pas déterministe. Plusieurs fils d'exécution qui s'occupent de différentes valeurs de i pourraient incrémenter en même temps une même case de d. Ainsi, certains incréments pourraient être perdus. On pourrait en faire une région critique (pragma omp critical) mais il est plus efficace d'utiliser une opération d'incrément atomique. Il est encore mieux d'utiliser une réduction sur d. Avec les boucles telles qu'elles sont, on pourrait utiliser une variable temporaire pour sommer calculer la nouvelle valeur de c[i][j] dans la boucle k la plus interne. Par ailleurs, le choix de l'ordre des indices est très important, pour que l'indice le plus à droite dans une matrice varie le plus vite (boucle la plus imbriquée). On veut ainsi i avant j pour c, i avant k pour a, et k avant j pour b. Le meilleur choix est donc d'avoir i, k, j.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <stdint.h>
#include <omp.h>
#include <math.h>

double fn(int n1, int n2) {
  return fabs(sin((double)(n1 * n2)));
}

#define NRA 1000
#define NCA 2000
#define NCB 1500
#define ND 128

int i, j, k;
double a[NRA][NCA], b[NCA][NCB], c[NRA][NCB], d[ND];

int main(int argc, char **argv)
{
  int test = atoi(argv[1]);

  #pragma omp parallel shared(a,b,c,d) private(i,j,k)
  {
    // initialisation ~ 0.1s
    #pragma omp for
    for(i=0; i<NRA; i++)
      for(j=0; j<NCA; j++) a[i][j] = fn(i,j);
    #pragma omp for
    for(i=0; i<NCA; i++)
      for(j=0; j<NCB; j++) b[i][j] = fn(i,j);
    #pragma omp for
    for(i=0; i<NRA; i++)
      for(j=0; j<NCB; j++) c[i][j] = 0;
    #pragma omp for
    for(i=0; i<ND; i++) d[i] = 0;

    // boucles j, i, k, sans atomic (course), 8.26s / 32.86s
    // sans optimisation, 20.45s / 78.91s
    if(test == 0) {
      #pragma omp for
      for(j=0; j<NCB; j++) {
        for(i=0; i<NRA; i++) {
          for(k=0; k<NCA; k++) {
            c[i][j] += a[i][k] * b[k][j];
            d[(int)(c[i][j]) % ND]++;
    } } } }
    // boucles j, i, k, 8.15s / 32.43s
    if(test == 1) {
      #pragma omp for
      for(j=0; j<NCB; j++) {
        for(i=0; i<NRA; i++) {
          for(k=0; k<NCA; k++) {
```

```
        c[i][j] += a[i][k] * b[k][j];
        #pragma atomic
        d[(int)(c[i][j]) % ND]++;
} } } }
// boucles i, j, k, 9.88s / 39.45s
if(test == 2) {
  #pragma omp for
  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];
        #pragma atomic
        d[(int)(c[i][j]) % ND]++;
} } } }
// boucles i, k, j, 6.86s / 27.06s
if(test == 3) {
  #pragma omp for
  for(i=0; i<NRA; i++) {
    for(k=0; k<NCA; k++) {
      for(j=0; j<NCB; j++) {
        c[i][j] += a[i][k] * b[k][j];
        #pragma atomic
        d[(int)(c[i][j]) % ND]++;
} } } }
// ajout de collapse(2), 6.84s / 27.08s
// peut introduire une course
if(test == 4) {
  #pragma omp for collapse(2)
  for(i=0; i<NRA; i++) {
    for(k=0; k<NCA; k++) {
      for(j=0; j<NCB; j++) {
        c[i][j] += a[i][k] * b[k][j];
        #pragma atomic
        d[(int)(c[i][j]) % ND]++;
} } } }
// boucles i, k, j, sans atomic (course), 6.93s / 27.28s
if(test == 5) {
  #pragma omp for
  for(i=0; i<NRA; i++) {
    for(k=0; k<NCA; k++) {
      for(j=0; j<NCB; j++) {
        c[i][j] += a[i][k] * b[k][j];
        d[(int)(c[i][j]) % ND]++;
} } } }
// boucles i, k, j, reduction de d, 4.96s / 19.77s
// sans optimisation, 16.55s / 65.97s
if(test == 6) {
  #pragma omp for reduction(+:d)
  for(i=0; i<NRA; i++) {
    for(k=0; k<NCA; k++) {
      for(j=0; j<NCB; j++) {
        c[i][j] += a[i][k] * b[k][j];
        d[(int)(c[i][j]) % ND]++;
} } } }
```

```
} }
```

c) La commande `#pragma omp barrier` effectue un rendez-vous entre tous les fils d'exécution de la région parallèle. Comment peut-on implémenter efficacement un rendez-vous en mémoire partagée? **(1 point)**

En mémoire partagée, un compteur peut être incrémenté de manière atomique à chaque fil d'exécution qui rejoint le rendez-vous. Si le décompte n'a pas atteint le nombre de participants, le fil d'exécution attend sur une condition. Si le nombre de participants est atteint, le fil d'exécution peut signaler la condition aux autres et poursuivre.

Le professeur: Michel Dagenais