ECOLE POLYTECHNIQUE DE MONTREAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2016)
3 crédits (3-1.5-4.5)

CORRIGÉ DU CONTRÔLE PÉRIODIQUE

DATE: Mercredi le 26 octobre 2016

HEURE: 9h30 à 11h20

DUREE: 1H50

NOTE: Toute documentation permise, calculatrice non programmable permise

Ce questionnaire comprend 4 questions pour 20 points

Question 1 (5 points)

a) Dans une grappe de calcul, la probabilité qu'un ordinateur donné fonctionne sans panne pendant une tâche de durée t (en heures) est donnée par la fonction $p(t) = 1/(.01t + 1)$. Par exemple, la probabilité qu'une tâche puisse se terminer correctement, sans panne du noeud qui l'exécute, est de 1 pour $t = 0$, 0 pour $t = \infty$ et 0.80645 pour $t = 24h$. Si une grappe contient 100 noeuds, quelle est la probabilité que tous les noeuds soient opérationnels pendant une tâche parallèle de 1h? De 2 h? **(2 points)**

Pour 1h, la probabilité d'exécution correcte est de 1/1.01. Pour 2h elle est de 1/1.02. La probabilité que tous les noeuds soient opérationnels est donc de $(1/1.01)^{100} = 0.3697$ *pour 1h et de* $(1/1.02)^{100} = 0.1380$ *pour 2h.*

b) Lors du premier TP, vous avez utilisé l'outil Callgrind de l'environnement Valgrind. Quelle information est-ce que cet outil fournit? Comment avez-vous utilisé cette information, quelle était son utilité? **(1 point)**

L'outil Callgrind permet de calculer le nombre de fois que chaque fonction est exécutée et d'estimer le temps passé en exécution dans chaque fonction elle-même (sans les appels imbriqués) et chaque fonction ainsi que celles appelées de manière imbriquée. Cette information aide à décider quelles fonctions paralléliser et permet d'estimer le temps qui peut être sauvé en divisant le temps des fonctions ciblées par le facteur de parallélisme attendu.

c) Dans le cadre du premier TP, vous avez effectué le même calcul à l'aide des POSIX Threads et à l'aide de TBB. Quelle version était la plus efficace? Est-ce que la différence était grande? Comment expliquez-vous cette différence? **(1 point)**

La librairie TBB est en général un peu plus efficace que les POSIX Threads. La différence n'est cependant pas très grande. Un des facteur de cette performance supérieure est la répartition dynamique du travail entre les threads et l'organisation récursive, en arbre, pour la création des threads, comparé à la plupart des programmes POSIX Threads qui font une boucle dans le thread principal qui crée sériellement tous les thread parallèles.

d) Un programme est composé de sections parallélisables et non parallélisables. La fraction parallélisable est de 88%. Ce programme est parallélisé et exécuté sur un ordinateur de 4 coeurs, quel sera le facteur d'accélération? Sur un ordinateur à 64 coeurs? **(1 point)**

Soit t0 le temps initial, t1 le nouveau temps deviendra $t1 = (1-fp)t0 + fp \times t0/n$, *l'accélération est donc de* $a = t0/t1 = 1/((1-fp)+fp/n)$, *soit* $1/(.12+.88/4) = 2.94$ *pour 4 coeurs, et* $1/(.12+.88/64) = 7.47$ *pour 64 coeurs.*

Question 2 (5 points)

a) Un ordinateur 64 bits, dont la mémoire est adressable à l'octet, utilise des tables de pages à 4 niveaux et des pages de 8Kio. Chaque noeud de la table de pages entre dans une page. Montrez comment l'adresse se décompose en décalage dans la page, index pour chacun des 4 niveaux dans la table, et s'il y a lieu les bits restants. A quoi servent les bits restants s'il y en a? Décomposez l'adresse virtuelle 0x0000ABCDEF012345 en ces différentes composantes (décalage, index3, index2...). **(2 points)**

Une page de 8Kio (2^{13}*) requiert 13 bits pour le décalage et peut contenir 1024 entrées de 8 octets (64 bits) lorsqu'utilisée comme noeud dans la table de pages. L'index dans un noeud de la table de pages requiert donc 10 bits pour indexer ces 1024 entrées. Le reste des bits* $64-13-4\times10 = 11$ *est inutilisé. Ces bits inutilisés limitent l'espace virtuel adressable par un processus à* 2^{53}*. L'adresse virtuelle se décompose donc ainsi: 11 bits inutilisés, 10 bits index0, 10 bits index1, 10 bits index2, 10 bits index3, 13 bits décalage. Ceci donne pour l'adresse 0x0000ABCDEF012345: 0b0000 0000 0000 0000 1010 1011 1100 1101 1110 1111 0000 0001 0010 0011 0100 0101 en binaire, ou 0b000 0000 0000|00 0001 0101|01 1110 0110|11 1101 1110|00 0000 1001|0 0011 0100 0101 en séparant les champs. On obtient ainsi champ par champ en hexadécimal: inutilisés 0x0000, index0 0x015, index1 0x1E6, index2 0x3DE, index3 0x009, décalage 0x0345.*

b) Un ordinateur possède une cache L1 MESI de 64Kio, avec des ensembles de 8 blocs, pour chacun de ses 4 coeurs. Chaque bloc en cache contient 128 octets. Cette cache est initialement vide. Les adresses suivantes sont accédées en séquence, en lecture (R) ou en écriture (W), sur les processeurs spécifiés (P0 à P3). Donnez l'état des cases non vides (M, E, S ou I) après chaque accès. **(2 points)**

```
P0 R 0x12A; P0 R 0x02A; P1 W 0x12A 2;
P2 W 0x12A 4; P2 R 0x12A; P1 W 0x02A 8
```

Pour des blocs de 128 octets, les 7 derniers bits d'adresse représentent le décalage dans le bloc. La cache contient $2^{16}/2^7 = 2^9 = 512$ blocs de 128 octets soit $512/8 = 64$ ensembles de 8 blocs. Il faut donc 6 bits pour représenter l'ensemble en cache et le reste constitue l'étiquette du bloc (qui distingue un bloc des autres qui peuvent se retrouver dans le même ensemble). Pour l'adresse 0x12A, cela donne 0b0001 0010 1010 ou 0b\|00 0010\|010 1010, soit étiquette 0, ensemble 0x02, décalage 0x2A. Pour l'adresse 0x02A, cela donne 0b0000 0010 1010 ou 0b\|00 0000\|010 1010, soit étiquette 0, ensemble 0x0, décalage 0x2A. Initialement, tous les blocs en cache ont le statut invalide (I). Les deux premières opérations font charger deux blocs dans la cache de P0 (premier bloc de l'ensemble 2, étiquette 0, statut exclusif (E), premier bloc de l'ensemble 0, étiquette 0, status E). L'opération suivante invalide le premier bloc de l'ensemble 2 de P0, charge le premier bloc de l'ensemble 2 de P1 et le modifie, ce qui donne le statut modifié (M). La même chose se produit ensuite sur P2 avec la quatrième opération qui invalide le premier bloc de l'ensemble 2 de P1. La cinquième opération ne modifie pas le statut du premier bloc de l'ensemble 2 de P2 qui reste à M. Finalement, la dernière opération invalide le premier bloc de l'ensemble 0 de P0, charge le premier bloc de l'ensemble 0, étiquette 0, de P1 et le modifie, ce qui donne le statut modifié.

c) La cache de pré-traduction d'adresse (TLB) permet de convertir les adresses virtuelles d'un processus en ses adresses physiques, en mémorisant un sous-ensemble (cache) du contenu de la table de pages d'un processus. Sur certaines variantes plus avancées de TLB, une étiquette est ajoutée aux adresses virtuelles dans le TLB pour identifier le processus associé. Doit-on mettre à jour le contenu du TLB lorsque l'espace d'adressage du processus est modifié (e.g., une zone de mémoire partagée est ajoutée par un appel à mmap())? Expliquez. Que doit-on faire avec le contenu du TLB, selon que ce soit un TLB régulier ou une "variante avancée", lorsque l'ordonnanceur change de thread mais tout en restant dans le même processus? Lorsque l'ordonnanceur change de processus? **(1 point)**

Lorsqu'une partie de l'espace d'adressage d'un processus est modifiée, le contenu correspondant dans la table de pages change et les entrées correspondantes dans le TLB doivent être invalidées car elles ne seront plus à jour en fonction de ces changements. Sur certaines architectures, des opérations spécifiques sont disponibles pour cela. Autrement, il faut remettre à zéro le contenu du TLB. Lorsque l'ordonnanceur change de thread mais reste dans le même processus, nous restons dans le même espace virtuel et cela n'affecte pas le TLB. Toutefois, si on change de processus, il faut normalement remettre à zéro le contenu du TLB. Cependant, si le TLB contient des étiquettes qui identifient le processus, chaque pré-traduction se fait en tenant compte du processus présent et il n'y a pas de problème. Dans ce cas, il n'est pas nécessaire de changer le contenu du TLB lors d'un tel réordonnancement, même s'il change le processus en exécution. Cela permet possiblement de conserver les entrées de l'autre processus pour quand il reviendra.

Question 3 (5 points)

a) Ecrivez un programme, utilisant la librairie TBB, qui offre la fonction *int CheckSorted(float v[], int size)* permettant d'effectuer le calcul suivant. L'argument v contient un vecteur de

nombres à virgule flottante alors que l'argument size contient la taille de v. La fonction CheckSorted doit calculer et retourner le nombre de fois que des items dans le vecteur ne sont pas en ordre croissant (i.e., l'item à la position i est plus petit que l'item en position i - 1, pour i entre 1 et size - 1 inclusivement). **(3 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 CheckSorted(float v[], int size) {
  Count c;
  c.v = v;
  parallel_reduce( blocked_range<int>(1, size), c);
  return c.value;
}
```

b) Un programme multi-thread s'exécute sur plusieurs processeurs en parallèle. Afin d'aller plus vite, il n'utilise pas de verrou. Un thread veut mettre à jour des valeurs data_a_1 et data_a_2 puis data_b_1 et data_b_2 et indiquer lorsqu'elles sont valides en mettant à 1 valid_a et valid_b 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 aient été mises à jour)? Si oui, modifiez le programme en ajoutant les barrières mémoire minimales requises? **(2 points)**

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

data_a_1 = 20;                   if(valid_a) {
data_a_2 = 18;                     new_a = data_a_2;
data_b_2 = 8;                    }
data_b_1 = 10;                   if(valid_b) {
valid_b = 1;                       new_b = data_b_1;
valid_a = 1;                     }
```

Il faut insérer une barrière d'écriture pour s'assurer que toutes les valeurs ont bien été écrites en mémoire commune avant de ne confirmer le tout par le changement à 1 de valid_a et valid_b. Une seule barrière mémoire suffit pour les deux puisque les accès aux data_a et data_b sont regroupés. Le pendant est qu'il faut une barrière de lecture après avoir détecté que valid_a (ou valid_b) est à 1 afin de s'assurer que les mises à jour des variables data sont bien arrivées au processeur à partir de la mémoire commune. Il n'y a pas de garantie d'ordre d'arrivée entre valid_a et valid_b 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. En pratique, on pourrait argumenter que chacune des deux peut agir comme validation pour les data_a et les data_b, puisque leur affectation est groupée, et réorganiser le code avec `if(valid_a || valid_b)` *et n'avoir qu'une seule barrière en lecture.*

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

data_a_1 = 20;                   if(valid_a) {
data_a_2 = 18;                     cmm_smp_rmb(); new_a = data_a_2;
data_b_2 = 8;                    }
data_b_1 = 10;                   if(valid_b) {
cmm_smp_wmb();                     cmm_smp_rmb(); new_b = data_b_1;
valid_b = 1;                     }
valid_a = 1;
```

Question 4 (5 points)

a) On vous fournit la section de code suivante en OpenMP. Vous devez proposer plusieurs améliorations qui permettront d'obtenir le même résultat mais plus efficacement. Fournissez une nouvelle version de cette section de programme et expliquez en une ou deux lignes de texte chaque amélioration apportée. **(2 points)**

```
#pragma openmp parallel for
for(int i = 0; i < nb_cpu; i++) {
  for(int j = 0; j < nb_task; j++) {
    for(int k = 0; k < nb_resource; k++) {
      by_cpu[i] += time[k][i][j];
      #pragma openmp atomic
      sum += time[k][i][j];
} } }

#pragma openmp parallel for
for(int i = 0; i < nb_cpu; i++) {
  for(int j = 0; j < nb_task; j++) {
    for(int k = 0; k < nb_resource; k++) {
```

```
      by_task[j] += time[k][i][j];
} } }

#pragma openmp parallel for
for(int i = 0; i < nb_cpu; i++) {
  for(int j = 0; j < nb_task; j++) {
    for(int k = 0; k < nb_resource; k++) {
      by_resource[k] += time[k][i][j];
} } }
```

Plusieurs optimisations sont possibles. Premièrement, on peut fusionner les trois boucles, ce qui évite de recalculer les indices trois fois et améliore la localité de référence. Ensuite, on peut convertir l'opération atomique en réduction, ce qui est plus efficace, changer l'ordre des indices pour varier le plus à droite en dernier dans la matrice time, ce qui améliore la localité de référence en cache, et éviter d'accéder à répétition l'élément de la matrice, calcul inutile que le compilateur optimiseur éviterait possiblement. Par ailleurs, le programme original est problématique car les différents threads pouvaient accéder en parallèle les mêmes entrées de by_task[j] et by_resource[k]. Il est maintenant possible, avec OpenMP 4.x, de faire des réductions sur des vecteurs et ainsi éliminer cette course dans le programme.

```
#pragma openmp parallel for reduction(+:sum) \
    reduction(+:by_cpu) reduction(+:by_task) \
    reduction(+:by_resource)
for(int k = 0; k < nb_resource; k++) {
  for(int i = 0; i < nb_cpu; i++) {
    for(int j = 0; j < nb_task; j++) {
      int element = time[k][i][j];
      by_cpu[i] += element;
      by_task[j] += element;
      by_resource[k] += element;
      sum += element;
} } }
```

b) Le programme OpenMP suivant s'exécute et produit une sortie sur stdout. Donnez une sortie possible produite par l'exécution de ce programme. Est-ce que la sortie peut changer d'une exécution à l'autre? Expliquez. **(2 points)**

```
int main(int argc, char **argv)
{ int nb_thread, thread;
  printf("a)\n");
  omp_set_num_threads(4);
  #pragma omp parallel private(nb_thread, thread)
  { printf("b)\n");
    #pragma omp for schedule(static)
```

```
    for(int i = 0; i < 8; i++) {
      nb_thread = omp_get_num_threads();
      thread = omp_get_thread_num();
      printf("c) thread %d / %d, i = %d\n", thread, nb_thread, i);
    }
    #pragma omp master
    printf("d) thread %d / %d\n", thread, nb_thread);
    #pragma omp critical
    printf("e) thread %d / %d\n", thread, nb_thread);
    #pragma omp single
    printf("f) thread %d / %d\n", thread, nb_thread);
  }
  printf("g) thread %d / %d\n", thread, nb_thread);
}
```

La sortie peut en effet changer car, si l'ordre d'exécution à l'intérieur d'un thread reste le même dans ce cas-ci (pas de données partagées et répartition statique de la boucle), l'ordre entre les threads peut changer. Ainsi, a et g sont hors de la section parallèle et ne changeront pas d'ordre. Les autres printf peuvent changer d'ordre. De plus, la sortie f n'est effectuée que par un seul thread mais celui-ci peut changer d'une fois à l'autre (pragma omp single), ce qui changera le numéro de thread associé à f.

```
a)
b)
c) thread 0 / 4, i = 0
c) thread 0 / 4, i = 1
b)
c) thread 3 / 4, i = 6
c) thread 3 / 4, i = 7
b)
c) thread 2 / 4, i = 4
c) thread 2 / 4, i = 5
b)
c) thread 1 / 4, i = 2
c) thread 1 / 4, i = 3
d) thread 0 / 4
e) thread 1 / 4
e) thread 2 / 4
e) thread 3 / 4
e) thread 0 / 4
f) thread 1 / 4
g) thread 0 / 0
```

c) Peut-on changer la taille de la pile en OpenMP? Comment? Comment sait-on si on doit changer la taille de la pile? **(1 point)**

OpenMP permet de spécifier la taille à utiliser pour les piles via une variable d'environnement, OMP_STACKSIZE. Si la taille est trop petite, le programme essaiera d'accéder des cases mémoire qui dépassent la zone disponible, ce qui peut résulter en une interruption pour cause d'erreur de segmentation, ou même en la corruption de zones mémoires dédiées à d'autres fins dans notre processus.

Le professeur: Michel Dagenais