POLYTECHNIQUE MONTRÉAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2022)
3 crédits (3-1.5-4.5)

CORRIGÉ DU CONTRÔLE PÉRIODIQUE

DATE: Mercredi le 26 octobre 2022

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 calcul parallèle doit être effectué sur 25 noeuds. Cependant, certains noeuds peuvent ne pas être disponibles en raison d'une panne sur leur disque, sur leur mémoire vive ou sur leur carte mère. La probabilité de panne qui rend le noeud non disponible pour le calcul est 0.1 pour le disque, 0.05 pour la mémoire vive et 0.08 pour la carte mère. Il faut donc prévoir une grappe avec quelques noeuds supplémentaires afin de pouvoir trouver au moins 25 noeuds opérationnels. Quelle est la probabilité d'avoir au moins 25 noeuds opérationnels parmi 26 noeuds? **(2 points)**

La probabilité d'avoir le disque, la mémoire et la carte mère opérationnels simultanément, donc qu'un noeud soit disponible, est de (1 - 0.1) x (1 - 0.05) x (1 - 0.08) = 0.7866. On peut alors calculer la probabilité que les 26 noeuds soient opérationnels ou que 25 noeuds le soient. Tous les 26 noeuds opérationnels: $0.7866^{26} = 0.001948061$. *Exactement 25 noeuds opérationnels:* $26!/(25! \times 1!) \times 0.7866^{25} \times (1 - 0.7866)^1 = 0.013740936$. *La probabilité d'avoir au moins 25 noeuds opérationnels est donc de 0.001948061 + 0.013740936 = 0.015688997.*

b) Une boucle de calcul comporte 8 instructions et fait la somme dans un registre des éléments d'une matrice. Chaque tour de boucle prend 8 cycles lorsqu'il n'y a pas de faute de cache. Une faute de cache ajoute 24 cycles. Les instructions sont en mémoire cache. Le seul accès qui peut causer une faute de cache est d'aller chercher l'élément de matrice, `c[i][j]`, à sommer dans ce tour de boucle. Chaque bloc en cache a une longueur de 128 octets et la cache de données a une taille de 8KiO. La matrice est carrée de taille 2048 et chaque élément de la matrice est un mot de 64 bits (`double c[2048][2048]`). Votre co-équipier suggère de lire la matrice par colonnes, en faisant varier l'indice `i` le plus vite. Quelle sera la longueur moyenne d'un tour de boucle en cycles pour ce cas? Vous proposez plutôt de lire la matrice par rangée, en faisant varier l'indice `j` le plus vite. Que deviendra alors la longueur moyenne d'un tour de boucle? **(2 points)**

En lisant par colonnes, chaque accès créera une faute de cache. En effet, les éléments consécutifs d'une même colonne sont distants de 2048 x 8 octets = 16KiO. Ainsi, les accès consécutifs ne sont pas dans le même bloc en cache. Plus encore, lorsqu'on arrive à la deuxième colonne, le premier élément de la deuxième colonne est dans le même bloc de cache que le premier élément de la première colonne lu au début, mais ce bloc a été enlevé du cache depuis longtemps car celui-ci ne peut même pas contenir une colonne complète. Chaque tour de boucle prend donc 8 + 24 = 32 cycles. En lisant par rangées, une fois un bloc lu en cache, les 16 mots de 8 octets qu'il contient seront utilisés pour autant de tours de boucles, puisque les accès sont consécutifs en mémoire. Il y aura donc en moyenne 1/16 faute de cache par tour de boucle, soit une durée moyenne de 8 + 24/16 = 9.5 cycles.

c) Un programme exécute 1 000 000 instructions en 3 600 000 cycles. Chaque accès en mémoire pour une instruction demande de convertir l'adresse virtuelle en adresse physique en passant par le système de traduction d'adresse. Lorsque l'entrée pour cette page virtuelle est présente dans le cache de pré-traduction d'adresse (TLB), l'accès est fait en parallèle dans le pipeline et cela ne prend aucun cycle supplémentaire. Autrement, il faut passer par la table de pages et cela génère un délai de 260 cycles. Le taux de succès dans le TLB pour les

instructions est présentement de 99%. Vous proposez d'utiliser des pages de 4MiO pour les instructions, plutôt que les pages par défaut de 4KiO. Cela diminuera beaucoup le nombre de pages différentes accédées, et le taux de succès dans le TLB deviendra de 99.9%. Que deviendra alors la durée de ce programme? **(1 point)**

Avec un taux de succès dans le TLB de 99%, on avait un total de (1-0.99) x 1 000 000 instructions x 260 cycles ajoutés = 2 600 000 cycles. Avec le nouveau taux de succès, nous aurons (1-0.999) x 1 000 000 instructions x 260 cycles ajoutés = 260 000 cycles. La durée du programme deviendra donc de 3 600 000 - 2 600 000 + 260 000 = 1 260 000 cycles.

Question 2 (5 points)

a) Considérez le code suivant utilisant les *pthreads*. Le type *queue_t* et les fonctions associées représentent une file bloquante d'entiers, similaire à celle utilisée lors du premier travail pratique. Expliquer brièvement le fonctionnement du programme. Quelle en est la sortie? Voyez-vous un quelconque intérêt à utiliser la variable `local_sum`? **(2 points)**

```
#include "queue.h"
#include <pthread.h>

void* producer(void* param) {
  queue_t* out = (queue_t*)param;
  for (int i = 0u; i < 128u; i++) queue_push(out, i % 8);
  queue_send_end(out);
}
static int sum = 0;

void* consumer(void* param) {
  queue_t* in = (queue_t*)param;
  int value, local_sum = 0;
  while (queue_pop(in, &value)) local_sum += value;
  sum = local_sum;
}

int main() {
  pthread_t workers[2];
  queue_t queue = queue_create();
  pthread_create(&workers[0], NULL, consumer, (void*)&queue);
  pthread_create(&workers[1], NULL, producer, (void*)&queue);
  pthread_join(workers[0], NULL);
  printf("Total : %d", sum);
  // --- nettoyage .. --- //
}
```

*Le programme instancie deux fils d'exécution, l'un producteur, l'autre consommateur. Le premier envoie dans la file les valeurs 0, 1, .., 7 au total 16 fois (128 / 8), tandis que le second additionne chaque valeur. Le programme principal attend la fin du consommateur avant d'afficher la valeur totale, qui est donc 16 * (0 + 1 + 2 + 3 + 4 + 5 + 6 + 7) = 448.*

La variable `local_sum` permet d'éviter de stocker chaque résultat intermédiaire dans l'espace d'adressage commun, en mémoire centrale, et peut réduire les fautes de cache si une autre variable s'y trouve. Elle permet aussi et surtout au compilateur de stocker la variable dans un registre.

b) Pour un exercice qui demandait de calculer la factorielle d'un nombre, votre partenaire de laboratoire propose le programme suivant (sans la ligne `sleep(1);`). La sortie affichée par la première exécution du programme commence alors par les deux lignes `De 0 à 4` et `De 5 à 9` et termine par la ligne `v[9] = 362880`. Vous avez des doutes sur son programme et vous décidez d'ajouter un délai dans la boucle, (la ligne `sleep(1);`), ce qui ne devrait rien changer au résultat si le programme est bien conçu. La dernière ligne affichée pour cette deuxième exécution devient alors `v[9] = 60480`. Expliquez pourquoi la sortie peut changer ainsi. Donnez la sortie complète pour la première et la deuxième exécution. **(2 points)**

```
#include <iostream>
#include <tbb/parallel_for.h>
#include <vector>

int main() {
  constexpr auto size = 10;
  std::vector<long> v; v.reserve(size);
  for (int i = 0; i < size; ++i) v.push_back(i);

  tbb::parallel_for(tbb::blocked_range<size_t>(0u, v.size(), 8),
      [&](const tbb::blocked_range<size_t>& r) {
        std::cout << "De " << r.begin() << " à " << r.end()-1 << std::e
        for (size_t i = r.begin(); i != r.end(); i++) {
          if (i == 0) v[i] = 1;
          else v[i] = v[i - 1] * i;
          sleep(1); // ligne ajoutée au second essai
        }
      }, tbb::simple_partitioner());

    for (int i = 0; i < size; i++) {
        std::cout << "v[" << i << "] = " << v[i] << std::endl;
    }
}
```

Le deuxième fil d'exécution utilise la valeur v[5 - 1] (donc v[4]) pour calculer v[5]. Initialement, v[4] = 4. Une fois que le premier fil d'exécution termine son travail, v[4] est remplacé

par 24. Les résultats obtenus par le deuxième fil d'exécution dépendent donc du moment où il lit v[4], lorsqu'il est à sa valeur initiale de 4 ou après que le premier fil d'exécution l'ait changé pour 24. Lors de la première exécution, il y a un délai avant que le second fil d'exécution ne soit créé et il lit alors v[4] = 24. Le calcul de factoriel se fait correctement et v[9] = 362880. Dans le second cas, avec le délai, les deux fils d'exécution progressent à peu près au même rythme et le second fil d'exécution a largement le temps de lire v[4] à 4, avant que le premier fil d'exécution ne le modifie. La valeur obtenue pour v[9] est alors incorrecte, diminuée par un facteur de 4/24. En effet, 362880 x 4 / 24 = 60480.

```
De 0 à 4
De 5 à 9
v[0] = 1
v[1] = 1
v[2] = 2
v[3] = 6
v[4] = 24
v[5] = 120
v[6] = 720
v[7] = 5040
v[8] = 40320
v[9] = 362880

De 0 à 4
De 5 à 9
v[0] = 1
v[1] = 1
v[2] = 2
v[3] = 6
v[4] = 24
v[5] = 20
v[6] = 120
v[7] = 840
v[8] = 6720
v[9] = 60480
```

c) Vous avez réalisé lors du premier travail pratique un pipeline de traitement d'image où un travailleur n'exécute qu'une seule tâche avant de transmettre le résultat à un autre. Décrivez brièvement l'impact de cette organisation sur la latence du temps de traitement de chaque image et sur le débit du système, respectivement. **(1 point)**

Une organisation en pipeline permet d'améliorer le débit total (objets traités par seconde), en raison du nombre de fils d'exécution qui travaillent en parallèle. Le débit est déterminé par l'étape la plus longue du pipeline. En contrepartie, la latence est égale ou plus grande (temps total par objet), déterminée par l'étape la plus longue fois le nombre d'étapes. Dès que le découpage en étapes n'est pas parfaitement égal, ce qui est généralement le cas, la latence

augmente dans le pipeline. En effet, les autres étages dans le pipeline doivent attendre après l'étage le plus lent, ce qui crée une latence qui n'existait pas sans le pipeline.

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 4MiO, utilisé pour son code exécutable et le monceau au tout début de son espace adressable, et 4MiO, utilisé à la toute fin de son espace adressable pour la pile. Ce processus demande alors de créer une nouvelle zone mémoire (par exemple avec l'appel mmap) d'une taille de 9000KiO, commençant à l'adresse 0x0000080000000000. Combien de noeuds devront être ajoutés à l'arbre qui constitue la table de pages de ce processus, en raison de l'ajout de cette zone mémoire? **(2 points)**

Une zone de 9000KiO représente 9000KiO / 4KiO = 2250 pages. Ceci implique autant d'entrées au dernier niveau de la table de pages. L'adresse 0x0000080000000000 donne 0b0000000000000000 pour les 16 bits inutilisés, et 0b000010000 pour les 9 bits d'index au niveau de la racine puis 0 pour l'index au niveau 2, 3 et 4. Puisque chaque noeud contient 512 entrées (4KiO / 8 octets par entrée = 512 entrées), et que les entrées commencent au début d'un noeud de dernier niveau, il faudra 2250 / 512 = 4.39453125 donc 5 noeuds au dernier niveau. Le noeud racine au premier niveau existe déjà, il faudra simplement ajouter un noeud au second niveau et un noeud au troisième niveau pour connecter ces 5 nouveaux noeuds, qui sont au début du noeud de troisième niveau, au reste de l'arbre. Le total est donc de 7 noeuds.

b) Un fil d'exécution sur le coeur 0 place des valeurs mises à jour dans les variables `d1a` et `d1b`, et change ensuite `v1` à 1 pour indiquer que ces variables sont prêtes. Il fait la même chose avec les variables `d2a` et `d2b`, et change ensuite `v2` à 1. Avec la librairie standard du langage C, des barrières mémoire sont ajoutées pour s'assurer que les valeurs mises à jour sont bien lues par un fil d'exécution sur le coeur 1, tel que montré dans l'extrait de programme qui suit. Vous devez enlever les barrières mémoire de la librairie standard du langage C (`cmm_smp_...`) et insérer des directives OpenMP (`#pragma omp ...`) qui auront un effet équivalent pour assurer que les valeurs mises à jour sont lues sur le coeur 1. **(2 points)**

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

d1a = 16;                      if(v1) {
d1b = 14;                        cmm_smp_rmb();
d2a = 5;                         new1 = d1a + d1b;
d2b = 7;                       }
```

```
cmm_smp_wmb();                  if(v2) {
v1 = 1;                           cmm_smp_rmb();
v2 = 1;                           new2 = d2a - d2b;
                                }
```

Le modèle des barrières mémoire en OpenMP diffère de celui du langage C. En C, `cmm_smp_mb` indique que tous les accès avant la barrière seront réalisés avant tous les accès qui viennent après la barrière. De plus, on considère que toute modification apportée à une variable sera propagée éventuellement vers la mémoire centrale et les autres copies en cache. En OpenMP, l'énoncé `#pragma omp flush` indique que toute modification de variable est propagée vers la mémoire centrale et que toute mise à jour en mémoire est prise en compte dans les variables. Ainsi, si une copie locale avait été prise pour une variable, la valeur mise à jour de cette variable sera propagée à la copie originale. A l'inverse, si la copie originale a été modifée, la copie locale est abandonnée et la copie originale est relue. Le terme flush *fait référence à évincer la copie locale, par exemple d'un registre, d'une copie locale ou du cache, après l'avoir réécrit si elle avait été modifiée. Ce faisant, on s'assure de reprendre la copie originale, plutôt qu'une vieille copie locale, au prochain accès. Ce modèle demande donc deux énoncés, là où on n'en avait qu'un seul avec les barrières du modèle du langage C. En effet, il faut d'abord envoyer les valeurs de `d1a, d1b, d2a, d2b` et ensuite, après avoir modifié `v1` et `v2`, il faut les envoyer à leur tour, dans un second temps. Avec le modèle du langage C, on suppose qu'une fois `v1` et `v2` modifiés, il vont se propager automatiquement vers la mémoire centrale relativement rapidement. Le modèle OpenMP ne donne pas cette assurance. Ainsi, de la même manière, il faut aller chercher la valeur mise à jour de `v1` avant de la tester, et ensuite on va chercher les valeurs mises à jour de `d1a` et `d1b`. En effet, là encore, le modèle OpenMP suppose qu'autrement les copies locales pourraient ne pas voir les modifications apportées sur les copies originales, tant qu'une barrière de mémoire implicite (single, parallel...) ou explicite (flush) n'est pas effectuée.*

```
Coeur 0                                   Coeur 1

                                        #pragma omp flush(v1)
d1a = 16;                               if(v1) {
d1b = 14;                                 #pragma omp flush(d1a,d1b)
d2a = 5;                                  new1 = d1a + d1b;
d2b = 7;                                }
#pragma omp flush(d1a,d1b,d2a,d2b)      #pragma omp flush(v2)
v1 = 1;                                 if(v2) {
v2 = 1;                                   #pragma omp flush(d2a,d2b)
#pragma omp flush(v1,v2)                  new2 = d2a - d2b;
                                        }
```

c) Un système multi-coeur possède un cache L1 pour chaque coeur et utilise le protocole de cohérence de cache MESI. Le coeur 2 rencontre une faute de cache au moment de faire une écriture. Le bloc de mémoire correspondant sera donc chargé dans le cache de ce coeur et

l'écriture y sera effectuée. Que deviendra alors l'état de ce bloc dans le cache du coeur 2? Si ce bloc était présent dans le cache d'autres coeurs, qu'adviendra-t-il de leur état dans les caches des autres coeurs? **(1 point)**

Une fois le bloc modifié dans le cache du coeur 2, il aura l'état modifié. En parallèle, le système de cohérence de cache va invalider les autres copies de ce bloc dans les caches des autres coeurs qui seront alors dans l'état invalide.

Question 4 (5 points)

a) Le programme OpenMP suivant est exécuté. Quelle sera la sortie générée par le printf à la fin? Est-ce que la sortie générée change si on intervertit l'ordre des 3 énoncés for imbriqués pour le calcul final de c? Présentement, l'ordre des for imbriqués est i, j, k (i.e. i varie le moins vite et k le plus vite). Quel serait un ordre plus efficace en temps d'exécution pour ces trois for? **(2 points)**

```
#define NB_RANGEE_A 8
#define NB_COL_A 8
#define NB_COL_B 8

int i, j, k;
double a[NB_RANGEE_A][NB_COL_A],
        b[NB_COL_A][NB_COL_B],
        c[NB_RANGEE_A][NB_COL_B];

int main(int argc, char **argv)
{
  #pragma omp parallel shared(a,b,c) private(i,j,k)
  {
    #pragma omp for
    for(i = 0; i < NB_RANGEE_A; i++)
      for(j = 0; j < NB_COL_A; j++) a[i][j] = i + j;
    #pragma omp for
    for(i = 0; i < NB_COL_A; i++)
      for(j = 0; j < NB_COL_B; j++) b[i][j] = i * j;
    #pragma omp for
    for(i = 0; i < NB_RANGEE_A; i++)
      for(j = 0; j < NB_COL_B; j++) c[i][j] = 0;

    #pragma omp for
    for(i = 0; i < NB_RANGEE_A; i++) {
      for(j = 0; j < NB_COL_B; j++) {
        for(k = 0; k < NB_COL_A; k++) {
          c[i][j] += a[i][k] * b[k][j];
```

```
    } } }
  }
  printf("Multiplication c = a * b, c[1][1] = %g\n", c[1][1]);
}
```

L'ordre dans lequel les boucles imbriquées sont placées ne change en théorie pas le résultat final, car on somme finalement les mêmes éléments provenant de a et b dans le même élément de c, mais dans un ordre différent, et la somme est commutative. Cependant, il pourrait y avoir une course qui pourrait parfois affecter la somme, si l'indice k était celui de la première boucle (la plus externe). En effet, on sait que différents fils d'exécution ont différentes valeurs de l'indice de la première boucle, puisque celui-ci est visé par l'énoncé de division du travail `omp for`*. Si cet indice est* `i` *ou* `j`*, nous sommes certains que deux fils d'exécution ne peuvent pas calculer le même élément de* `c[i][j]`*. Par contre, si le premier indice est* `k`*, deux fils d'exécution pourraient être rendus à faire une somme dans le même élément de* `c[i][j]`*, et cette course pourrait faire perdre certains des éléments à sommer et donc changer le résultat final. Afin d'avoir l'ordre le plus efficace en temps d'exécution, l'idée est d'accéder les matrices par rangées en autant que possible. Pour le calcul* `c[i][j] += a[i][k] * b[k][j]` *on veut donc faire varier j avant i, k avant i et j avant k. On peut donc faire varier j en premier, k en second et i en dernier. Heureusement, ce n'est pas k qui devrait aller comme premier indice. Ceci donne:*

```
for(i = 0; i < NB_RANGEE_A; i++) {
  for(k = 0; k < NB_COL_A; k++) {
    for(j = 0; j < NB_COL_B; j++) {
```

La valeur affichée est:

```
Multiplication c = a * b, c[1][1] = 168
```

b) 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 nb_thread, thread;
  omp_set_num_threads(4);
  printf("a) thread %d / %d\n", thread, nb_thread);
  #pragma omp parallel private(thread) shared(nb_thread)
  { nb_thread = omp_get_num_threads();
    thread = omp_get_thread_num();
    #pragma omp for schedule(static,1) ordered
    for(int i = 0; i < 8; i++) {
      printf("b) thread %d / %d, i = %d\n", thread, nb_thread, i);
      fflush(stdout);
```

```
      #pragma omp ordered
      printf("c) thread %d / %d, i = %d\n", thread, nb_thread, i);
      fflush(stdout);
    }
    #pragma omp single
    printf("d) thread %d / %d\n", thread, nb_thread);
    #pragma omp masked
    printf("e) thread %d / %d\n", thread, nb_thread);
  }
  printf("f) thread %d / %d\n", thread, nb_thread);
}
```

En a) les deux variables affichées ne sont pas initialisées et ce peut être n'importe quoi. Ensuite, les b) à chaque tour de boucle peuvent être dans n'importe quel ordre. Par contre, le c) d'un tour de boucle ne peut être affiché avant que les b) des indices qui précèdent (ou égal) ne soient affichés. La valeur `thread` *de d) peut changer. Cependant, d) vient après les c) en raison de la barrière implicite après le for, et e vient après d) en raison de la barrière implicite après le single. Finalement, f) termine le tout mais la valeur* `thread` *peut changer, car non initialisée dans ce contexte.*

```
a) thread 100 / 0
b) thread 3 / 4, i = 3
b) thread 0 / 4, i = 0
c) thread 0 / 4, i = 0
b) thread 0 / 4, i = 4
b) thread 2 / 4, i = 2
b) thread 1 / 4, i = 1
c) thread 1 / 4, i = 1
b) thread 1 / 4, i = 5
c) thread 2 / 4, i = 2
b) thread 2 / 4, i = 6
c) thread 3 / 4, i = 3
b) thread 3 / 4, i = 7
c) thread 0 / 4, i = 4
c) thread 1 / 4, i = 5
c) thread 2 / 4, i = 6
c) thread 3 / 4, i = 7
d) thread 1 / 4
e) thread 0 / 4
f) thread 100 / 4
```

c) Plusieurs librairies de calcul parallèle offrent des fonctions pour l'allocation de blocs de mémoire alignés, par exemple `cache_aligned_allocator` dans la librairie TBB. Selon quel critère ces blocs sont-ils *alignés*? Pourquoi est-il plus intéressant d'avoir des blocs

alignés? Quel est le désavantage de forcer l'alignement des blocs alloués en mémoire? **(1 point)**

Les blocs sont alignés sur un multiple de la taille d'un bloc en cache. Ceci permet de minimiser le nombre de blocs de cache requis pour ce bloc de mémoire. De plus, deux blocs alignés sont assurés de ne pas partager un bloc de cache, ce qui pourrait causer un faux partage. Le fait de forcer un tel alignement peut causer de la fragmentation, i.e. des espaces laissés inutilisés en mémoire.

Le professeur: Michel Dagenais