POLYTECHNIQUE MONTRÉAL

Département de génie informatique et génie logiciel

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

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)**

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)**

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)**

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 .. --- //
}
```

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::endl;
        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;
    }
}
```

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)**

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)**

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;
                               }
```

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)**

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]);
}
```

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);
}
```

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)**

Le professeur: Michel Dagenais