POLYTECHNIQUE MONTRÉAL

Département de génie informatique et génie logiciel

Cours INF8601: Systèmes informatiques parallèles (Automne 2023)
3 crédits (3-1.5-4.5)

EXAMEN FINAL

DATE: Vendredi le 22 décembre 2023

HEURE: 9h30 à 12h00

DUREE: 2H30

NOTE: Aucune documentation permise sauf un aide-memoire, 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) Le programme OpenCL suivant reçoit en argument une image en tons de gris (valeur entre 0 et 255 pour chaque point) et calcule en sortie le contenu du vecteur `histogram`. Le programme est appelé avec une taille globale de problème de 4000, et une image qui contient des lignes horizontales où chaque pixel a une valeur donnée par l'équation `image[i][j] = i % 256`. Tous les points de chaque rangée sont ainsi à la même valeur. Quel sera le contenu du vecteur `histogram` en sortie? **(2 points)**

```
__kernel void HistogramA(__global unsigned char image[4000][4000],
    __global unsigned int histogram[256])
{
  int i;
  int col = get_global_id(0);
  for(i = 0; i < 4000 ; i++) atomic_inc(histogram[image[i][col]]);
}
```

b) Deux amis vous proposent chacun une version différente du programme OpenCL `HistogramA` utilisé en a): `HistogramB` et `HistogramC`. i) Est-ce que les trois versions devraient donner le même résultat en sortie? Quelle devrait être la performance relative de chaque version (i.e. la plus rapide, la plus lente, entre les deux) et pourquoi? **(2 points)**

```
__kernel void HistogramB(__global unsigned char image[4000][4000],
    __global unsigned int histogram[256])
{
  int i;
  int row = get_global_id(0);
  for(i = 0; i < 4000 ; i++) atomic_inc(histogram[image[row][i]]);
}

__kernel void HistogramC(__global unsigned char image[4000][4000],
    __global unsigned int histogram[256])
{
  int i;
  int col = get_global_id(0), group_size = get_local_size(0);
  int chunk_size = 256 / group_size + 1;
  int start_chunk = get_local_id(0) * chunk_size;
  int end_chunk = min(256, start_chunk + chunk_size);
  unsigned int tmp_histogram[256];
  __local unsigned int local_histogram[256];

  for(i = 0; i < 256; i++) tmp_histogram[i] = 0;
  for(i = 0; i < 4000 ; i++) tmp_histogram[image[i][col]]++;
  for(i = start_chunk; i < end_chunk; i++) local_histogram[i] = 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  for(i = 0; i < 256; i++)
    atomic_add(local_histogram[i], tmp_histogram[i]);
  barrier(CLK_LOCAL_MEM_FENCE);
```

```
  for(i = start_chunk; i < end_chunk; i++)
    atomic_add(histogram[i], local_histogram[i]);
}
```

c) Lors des travaux pratiques, vous avez utilisé le langage de programmation OpenCL pour effectuer des calculs sur un GPU conçu par la compagnie AMD. Est-ce que le langage OpenCL peut aussi être utilisé pour effecturer des calculs sur les coeurs de CPU de la compagnie AMD? La plupart des utilisateurs de GPU conçus par la compagnie NVIDIA utilisent le langage de programmation CUDA. Pourraient-ils aussi utiliser le langage OpenCL sur ces GPU? **(1 point)**

Question 2 (5 points)

a) Le programme MPI suivant s'exécute sur 4 noeuds (MPI_Comm_size retourne 4). Donnez une sortie possible produite par ce programme? **(2 points)**

```
int main (int argc, char *argv[])
{
  int i, j, nb_row = 4, nb_col = 4, rank, size, m;
  int a[4][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {8, 7, 6, 5}, {4, 3, 2, 1}};
  int b[4][4] = {{0, 1, 0, 2}, {0, 3, 0, 4}, {0, 5, 0, 6}, {0, 7, 0, 8}};
  int c[4][4];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  for(i = 0; i < nb_row; i++) {
    for(j = 0; j < nb_col; j++) {
      m = a[i][rank] * b[rank][j];
      MPI_Reduce(&m, &(c[i][j]), 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    }
  }

  if(rank == 0) {
    printf("C = {");
    for(i = 0; i < nb_row; i++) {
      printf("\n      ");
      for(j = 0; j < nb_col; j++) printf("%d ", c[i][j]);
    }
    printf("\n    }\n");
  }
  MPI_Finalize();
}
```

b) Le programme MPI suivant s'exécute sur 4 noeuds (MPI_Comm_size retourne 4). Donnez une sortie possible produite par ce programme? **(2 points)**

```
int main (int argc, char *argv[])
{
  int size, rank, i, in[4], o1[4], o2[4];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  for(i = 0; i < 4; i++) { in[i] = i * i + rank * i; o1[i] = o2[i] = 0; }
  MPI_Alltoall(in, 1, MPI_INT, o1, 1, MPI_INT, MPI_COMM_WORLD);
  MPI_Allgather(o1, 1, MPI_INT, o2, 1, MPI_INT, MPI_COMM_WORLD);
  printf("%d: in={%d, %d, %d, %d};\n", rank, in[0], in[1], in[2], in[3]);
  printf("%d: o1={%d, %d, %d, %d};\n", rank, o1[0], o1[1], o1[2], o1[3]);
  printf("%d: o2={%d, %d, %d, %d};\n", rank, o2[0], o2[1], o2[2], o2[3]);
  MPI_Finalize();
}
```

c) En plus des fonctions MPI_Send et MPI_Receive, on retrouve dans la librairie MPI les fonctions MPI_Send_init et MPI_Receive_init. En quoi ces fonctions diffèrent des premières, et dans quel cas est-ce plus avantageux de les utiliser? **(1 point)**

Question 3 (5 points)

a) Un outil de mesure de performance, comme CPU Profile, échantillonne le contenu de la pile d'appels (la fonction courante et celles dans le chemin d'appel) à un intervalle de temps régulier de 1ms. Lorsqu'un échantillon se répète, ce qui arrive souvent, un facteur multiplicatif est ajouté, plutôt que de sauver le même chemin d'appel plusieurs fois dans le fichier de sortie. Voici les échantillons récoltés, chacun suivi du nombre de fois qu'il a été rencontré. Calculez le temps passé dans chaque fonction elle-même (self) et dans chaque fonction et ses appels (self+childs). **(2 points)**

```
(main, f1), 9
(main, f2), 7
(main), 6
(main, f1, f2, f4, f5, f6), 5
(main, f1, f3), 11
(main, f2, f4, f6), 8
(main, f1, f2, f3, f4), 3
(main, f1, f3, f5), 10
```

b) Un outil de vérification des accès et de l'allocation de mémoire, semblable à Memcheck, instrumente toutes les lectures (read, adresse, nombre d'octets) et écritures (write, adresse, nombre d'octets) en mémoire, de même que les allocations (malloc, adresse, nombre d'octets) et libérations (free, adresse). Les nombres d'octets et adresses sont donnés en hexadécimal. Voici la trace de l'information récoltée, en ordre chronologique, pendant l'exécution d'un petit programme. Quelles erreurs d'utilisation de la mémoire ou fuites de mémoire à la fin peut-on en déduire? Dans chaque cas, expliquez le problème et donnez le numéro de la ligne correspondante dans la trace. **(2 points)**

```
01: malloc, 0x40000000, 0x10
```

```
02: malloc, 0x40000040, 0x20
03: write, 0x40000050, 0x4
04: read, 0x40000050, 0x8
05: write, 0x40000004, 0x4
06: malloc, 0x40000060, 0x20
07: malloc, 0x40000080, 0x20
08: write, 0x40000000, 0x8
09: read, 0x40000000, 0x8
10: free, 0x40000040
11: free, 0x40000060
12: read, 0x40000084, 0x8
13: write, 0x40000084, 0x8
14: free, 0x40000060
15: read, 0x40000050, 0x4
16: free, 0x40000080
```

c) Pour chacun des deux cas suivants d'ennuis avec un logiciel, quel serait un bon outil à suggérer pour trouver le problème: i) un problème de corruption est présent et on suspecte un accès incorrect à la mémoire comme un débordement de vecteur en C/C++, et ii) le temps écoulé est très long, mais le temps en exécution en mode usager est normal, si bien que le problème semble se situer dans l'interaction avec le système d'exploitation ou avec les autres tâches. **(1 point)**

Question 4 (5 points)

a) Quelle est la sortie produite par ce programme? Comment se nomme l'opération bien connue effectuée sur le vecteur v dans ce programme? **(2 points)**

```
void seqprfsum(int *u, int m)
{ int s = 0; for(int i = 0; i < m; i++) { u[i] += s; s = u[ i ]; }}

void parprfsum (int *x, int n, int *z)
{
  #pragma omp parallel
  { int i, j, me = omp_get_thread_num ( ),
        nth = omp_get_num_threads( ), chunksize = n / nth,
        start = me * chunksize;
    seqprfsum(&x[start], chunksize);

    #pragma omp barrier
    #pragma omp single
    { for(i = 0; i < nth - 1; i++) z[i] = x[(i+1) * chunksize - 1];
      seqprfsum(z ,nth - 1);
    }
    if(me > 0 ) for(j = start; j < start + chunksize; j ++) x[j] += z[me - 1];
  }
}
```

```
int main(int argc, char **argv)
{ int v[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
  int sp[4];

  omp_set_num_threads(4);
  parprfsum(v, 16, sp);
  for(int i = 0; i < 16; i++) printf("%d ", v[i]);
  printf("\n");
}
```

b) Vous proposez de réaliser un programme parallèle pour solutionner un système d'équations linéaires, $b = Ax$, par la méthode itérative de Jacobi. Soit A la matrice des coefficients, b le vecteur des valeurs, et x le vecteur des variables, on peut définir D la matrice diagonale de A, et O la matrice en enlevant la diagonale de A ($O = A - D$). La solution à l'itération $k+1$ est donnée en fonction de la solution à l'itération k par $x_{k+1} = D^{-1}(b - Ox_k)$. On vous demande de démontrer comment fonctionne la méthode itérative de Jacobi en l'appliquant manuellement sur les données suivantes, avec le vecteur x de solution initiale (i.e. pour x_0) tel que spécifié. Donnez la valeur obtenue pour le vecteur x après la première itération (x_1). **(2 points)**

```
A = np.array([
    [4, 2, 1, 1],
    [3, 4, 1, 0],
    [3, 0, 5, 1],
    [1, 2, 1, 4]
])
b = np.array([5, 0, 10, 8])
x_init = np.array([1, 1, 1, 1])
```

c) On veut paralléliser la multiplication de grandes matrices, C = AB, de dimension $n \times n$, sur m noeuds à l'aide de la librairie MPI. Est-il préférable de diviser le travail en donnant à calculer à chaque noeud pour la matrice résultat C: i) n/m rangées, ou ii) une sous-matrice de $(n/\sqrt{m}) \times (n/\sqrt{m})$? Pourquoi? Combien d'additions et de multiplications chaque noeud devrait-il faire dans chaque cas? **(1 point)**

Le professeur: Michel Dagenais