**ECOLE POLYTECHNIQUE DE MONTREAL**

**Département de génie informatique et génie logiciel**

**Cours INF8601:** Systèmes informatiques parallèles (Automne 2018)
3 crédits (3-1.5-4.5)

---

**EXAMEN FINAL**

**DATE: Dimanche le 9 décembre 2018**

**HEURE: 13h30 à 16h00**

**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) Une unité centrale de traitement vectorielle est telle que décrite dans le livre et contient une de chacune des unités suivantes: i) addition, soustraction et comparaison point flottant, ii) multiplication point flottant, iii) division point flottant, iv) opérations entières, v) opérations logiques, et vi) rangement et chargement. Ces unités en pipeline permettent de produire une nouvelle valeur à chaque cycle. Les instructions scalaires (e.g. LD, DRSL) prennent un cycle d'exécution chacune. Dans ces unités en pipeline, les additions, soustractions et comparaisons point flottant ont une profondeur de pipeline de 8, les multiplications point flottant de 16, les divisions point flottant de 20 et les chargements et rangements de 10. Les registres vectoriels contiennent 64 mots de 64 bits. Calculez le temps requis pour l'exécution de la séquence d'instructions suivante. **(2 points)**

```
1  L.D      F0, R1
2  L.D      F1, R2
3  L.D      F2, R3
4  LV       V1, R4
5  MULVS.D  V1, V1, F0
6  LV       V2, R5
7  MULVS.D  V2, V2, F1
8  ADDVV    V3, V1, V2
9  DIVSV.D  V3, V3, F2
10 SV       V3, R6
```

b) La fonction suivante était utilisée dans le travail pratique 2 et présente des problèmes de performance. Identifiez ces problèmes de performance et suggérez une version améliorée. **(2 points)**

```
int encode_slow_a(struct chunk *chunk)
{ int i, j;
  uint64_t checksum = 0;

  #pragma omp parallel for private(i,j) reduction(+:checksum)
  for (i = 0; i < chunk->height; i++) {
    for (j = 0; j < chunk->width; j++) {
      int index = i * chunk->width + j;
      chunk->data[index] = chunk->data[index] + chunk->key;
      checksum += chunk->data[index];
    }
  }
  chunk->checksum = checksum;
  return 0;
}
```

c) Deux threads, un chacun de deux processus différents, qui ne partagent donc aucune mémoire, s'exécutent sur les deux coeurs logiques du même coeur physique, en hyperthreading. Est-ce que chaque thread va s'exécuter à la même vitesse, plus vite (combien) ou plus lentement (combien) que s'il était seul sur un coeur physique? Expliquez. **(1 point)**

# Question 2 (5 points)

a) On vous propose deux versions, a et b, de la fonction `copy` qui copie les éléments d'une matrice. Les deux versions produisent exactement le même résultat, car l'appel à la fonction avec `clEnqueueNDRangeKernel` est ajusté pour mettre les bonnes valeurs de taille (nombre de rangées ou colonnes) dans les dimensions 0 et 1. Cependant, une version offre une beaucoup meilleure performance. Laquelle? Expliquez pourquoi. **(2 points)**

```
kernel void copy_a(global const float * in,
    global float * out, int w,int h)
{ int x = get_global_id(1), y = get_global_id(0);
  out[x+y*w] = in[x+y*w];
}

kernel void copy_b(global const float * in,
    global float * out, int w,int h)
{ int x = get_global_id(0), y = get_global_id(1);
  out[x+y*w] = in[x+y*w];
}
```

b) Un programme utilisant OpenCL définit une matrice `entree` et une matrice `sortie`. La fonction `UneFonction` lit `entree` et écrit `sortie` comme s'ils étaient des vecteurs. Voici le code en C qui définit les matrices, appelle le `kernel` OpenCL `UneFonction` et imprime le résultat. La partie du programme qui initialise et appelle le code OpenCL, et crée, envoie et reçoit des tampons pour les matrices `entree` et `sortie`, est sans intérêt et n'est pas fournie. Quelle sera la sortie imprimée par ce programme? **(2 points)**

```
// Extraits importants du programme C
#define NRANGEES 256
#define NCOLONNES 128
float entree[NRANGEES][NCOLONNES];
float sortie[NCOLONNES][NRANGEES];
int nrangees = NRANGEES, ncolonnes = NCOLONNES;
size_t size = nrangees * ncolonnes;
cl_int i, j, err;

for(i = 0; i < nrangees; i++)
  for(j = 0; j < ncolonnes; j++) entree[i][j] = i;

for(i = 0; i < 3; i++) {
  for(j = 0; j < 3; j++) printf("%f ", entree[i][j]);
  printf("\n");
}
// Initialiser OpenCL et tout préparer pour fournir
// entree, sortie, nrangees et ncolonnes en argument
...
err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &size,
         NULL, 0, NULL, NULL);
// Récupérer sortie et libérer les ressources OpenCL
...
for(i = 0; i < 3; i++) {
  for(j = 0; j < 3; j++) printf("%f ", sortie[i][j]);
  printf("\n");
}

// Programme OpenCL
__kernel void UneFonction(__global float *entree,
    __global float *sortie, int nrangees, int ncolonnes)
{ unsigned int xIndex = get_global_id(0) / ncolonnes;
  unsigned int yIndex = get_global_id(0) % ncolonnes;
  unsigned int index_entree = xIndex * ncolonnes + yIndex;
  unsigned int index_sortie = yIndex * nrangees + xIndex;
  sortie[index_sortie] = entree[index_entree];
```

```
}
```

c) La fonction suivante (dont une partie répétitive sans intérêt est omise) calcule la couleur, `struct rgb *color`, en fonction d'une valeur, `float value`. La fonction donne le résultat attendu lorsqu'exécutée sériellement, mais fonctionne de manière erratique lorsque le programme est parallélisé et cette fonction est appelée de plusieurs threads. Quelle est l'erreur? **(1 point)**

```
struct rgb c;

void value_color(struct rgb *color, float value, int interval,
    float interval_inv)
{ int x = (((int)value % interval) * 255) * interval_inv;
  int i = value * interval_inv;
  switch(i) {
    case 0: c.r = 0; c.g = x; c.b = 255; break;
    case 1: c.r = 0; c.g = 255; c.b = 255 - x; break;
// tous les autres cas: 2, 3, 4...
...
    default: c = white; break;
  }
  *color = c;
}
```

# Question 3 (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 à l'écran? **(2 points)**

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{ int v[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
  int i, rank, size, chunk, res, sum = 0;
  int v_size = sizeof(v) / sizeof(v[0]);
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  for(i = rank; i < v_size; i += size) sum +=v[i];

  printf("Sum %d: %d\n", rank, sum);
  MPI_Reduce(&sum,&res,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
  if(rank == 0) printf("Result: %d\n", res);
  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 à l'écran? **(2 points)**

```
#include "mpi.h"
```

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{ int size, rank, i;

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

c) Expliquez quelle est la fonction d'un logiciel gestionnaire de ressources comme SLURM ou Red Hat Grid (Condor). **(1 point)**

# Question 4 (5 points)

a) Vous compilez un programme avec les options de profilage de `gprof` et l'exécutez ensuite. Ce programme ne comporte qu'un seul thread, qui est interrompu par `gprof` à chaque 1ms afin de noter la fonction dans laquelle se trouve le compteur de programme. De plus, à chaque appel, `gprof` note le décompte du nombre d'appels de chaque provenance (fonction appelante). L'information suivante est ainsi produite pendant l'exécution. Sur 41 échantillons du compteur de programme, récoltés à intervalle régulier de 1ms, 2 sont dans la fonction main, 6 dans f1, 4 dans f2, 2 dans f3, 12 dans f4, 7 dans f5 et 8 dans f6. Le nombre d'appels de chaque fonction (avec la provenance des appels entre parenthèses) est `main`: 1 (pas d'appelant), f1:2 (main:2), f2:3 (main:3), f3:4 (f1:4), f4:12 (f1:5, f2:7), f5:14 (f1:6, f2:8), f6:9 (f2:9). Calculez pour chaque fonction le temps passé dans la fonction elle-même (self) et le temps passé dans la fonction elle-même plus ses enfants (self+childs), comme le ferait l'outil `gprof`. **(2 points)**

b) Un programme semblable à `heap profile` intercepte les appels à `malloc` ou `new` et note à chaque allocation le nombre d'octets alloués ainsi que le contenu de la pile d'appel, soit le chemin d'appel qui commence par la fonction ayant appelé malloc/new directement (self) et se poursuit par celles ayant appelé indirectement. Les données suivantes sont fournies: une liste d'entrées séparées par des ";", chaque entrée donnant le nombre d'octets ":" le chemin d'appel (noms des fonctions appelantes séparés par des ","). On vous demande de calculer le nombre d'octets alloués directement (self) et indirectement (self+childs) pour chaque fonction. **(2 points)**

```
64: f3, f2, f1, main;
32: f4, f2, f1, main;
18: f6, f5, f1, main;
24: f5, f4, main;
12: f1, main;
8: f3, main;
```

c) Dans le cadre du cours, trois paradigmes de programmation ont été utilisés, OpenMP, OpenCL et MPI. Pour chacun, expliquez dans quelle situation il est utile. Sont-ils mutuellement exclusifs ou peuvent-ils être utilisés de concert pour solutionner un problème? **(1 point)**

Le professeur: Michel Dagenais