**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)

---

# CORRIGÉ DE L'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
```

*Les opérations vectorielles peuvent être chaînées lorsque qu'elles sont consécutives et utilisent toutes des unités différentes. C'est le cas des lignes 4, 5, et 6, 7, 8, 9. Les trois premières instructions prennent 1 cycle chacune. Les deux instructions suivantes peuvent se chaîner en 10 + 16 + 64 = 90 cycles. Les 4 instructions suivantes peuvent aussi se chaîner en 10 + 16 + 8 + 20 + 64 = 118 cycles. La dernière instruction prend 10 + 64 = 74 cycles. Le total est donc 3 + 90 + 118 + 74 = 285 cycles.*

```
1  L.D      F0, R1      ; +1
2  L.D      F1, R2      ; +1
3  L.D      F2, R3      ; +1
4  LV       V1, R4;     ; +10
5  MULVS.D  V1, V1, F0  ; +16+64
6  LV       V2, R5      ; +10
7  MULVS.D  V2, V2, F1  ; +16
8  ADDVV    V3, V1, V2  ; +8
9  DIVSV.D  V3, V3, F2  ; +20+64
10 SV       V3, R6      ; +10+64
```

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;
}
```

*Le pointeur `chunk` reçu en argument est accédé à répétition dans la fonction. Cela requiert des accès en mémoire à répétition car le compilateur n'ose pas mettre ces valeurs, qui pourraient être accédées de manière concurrente par plus d'un thread, dans des registres. Ceci est particulièrement coûteux pour les accès dans la boucle, puisqu'ils sont effectués un grand nombre de fois. Une seconde optimisation, moins importante, est de combiner les deux boucles en une seule.*

```
int encode_slow_a(struct chunk *chunk)
{ int i;
  uint64_t checksum = 0;
  char *data = chunk->data;
  int area = chunk->area;
  int key = chunk->key;

  #pragma omp parallel for private(i) reduction(+:checksum)
  for (i = 0; i < area; i++) {
    data[i] = data[i] + key;
    checksum += data[i];
  }
  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)**

*Puisque les deux threads s'exécutent sur le même coeur physique, le temps pour chacun pourrait doubler. Toutefois, pendant les fautes de cache d'un thread, l'autre thread peut s'exécuter. Il*

*y a donc un gain et chaque thread devrait avoir un temps qui est un peu moins que doublé. Il pourrait y avoir des cas pathologiques où chaque thread pourrait utiliser pleinement la cache mais s'en tirer avec très peu de fautes de cache s'il est seul sur un coeur physique. Lorsque deux threads semblables partageraient un coeur physique en hyperthreading, les fautes de cache exploseraient, car la capacité de la cache serait insuffisante pour soutenir deux threads de ce type. A ce moment, le temps pourrait être plus du double.*

# 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];
}
```

*Le matériel du GPU combine les accès simultanés à des cases consécutives en mémoire, par les coeurs adjacents d'un processeur SIMD, en un seul accès à un bloc de mémoire globale, ce qui est beaucoup plus efficace. Dans ce programme, un incrément de 1 sur x cause un incrément de 1 pour l'indice de in et out ((x+1)+y*w = (x+y*w)+1). A l'inverse, un incrément de 1 sur y cause un incrément de w sur l'indice de in et out (x+(y+1)*w = (x+y*w)+w). La différence tient donc à quelle dimension, 0 ou 1, avec* `get_global_id()` *est assignée à x et y. En effet, la dimension 0 est celle qui varie le plus vite, celle qui change d'un coeur à l'autre sur le même processeur SIMD. Il faut donc lui assigner l'indice x, puisque cela mènera à des accès consécutifs en mémoire, comme dans la version b. Ainsi, la version b fait en sorte que les différents work item d'un même groupe, qui sont exécutés en même temps sur les coeurs adjacents du même processeur SIMD, ayant des valeurs consécutives de* `get_global_id(0)`*, accèdent des cases consécutives en mémoire, puisque c'est l'indice x qui est associé à la dimension 0. Dans la version a, les work item sur les coeurs adjacents, avec leur* `get_global_id(0)` *consécutifs, font des accès à* `w` *mots de distance, qui ne peuvent être combinés en un seul accès de bloc à la mémoire centrale.*

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];
}
```

*Le programme effectue une transposition de matrice. L'identificateur global, `get_global_id(0)`, est utilisé pour retrouver la valeur correspondante de rangée et colonne dans la matrice `entree`. Ensuite, cet élément est stocké dans la matrice `sortie` en transposant les indices.*

```
0.000000 0.000000 0.000000
1.000000 1.000000 1.000000
2.000000 2.000000 2.000000
0.000000 1.000000 2.000000
0.000000 1.000000 2.000000
0.000000 1.000000 2.000000
```

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;
}
```

*La variable temporaire `struct rgb c` est globale et peut donc être modifiée simultanément par des appels concurrents, ce qui peut corrompre le calcul de chaque thread. Il suffit de déplacer cette déclaration à l'intérieur de la fonction afin que la variable devienne locale, propre à chaque appel de la fonction.*

# 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();
}
```

*Chaque noeud effectue la somme d'une partie des éléments, noeud 0: sum(1, 5, 9) = 15, noeud 1: sum(2, 6, 10) = 18, noeud 2: sum(3, 7, 11) = 21, noeud 3: sum(4, 8, 12) = 24 et imprime sa somme. Ensuite une réduction de sommation est faite et le noeud racine, 0, obtient la somme globale, sum(15, 18, 21, 24) = 78, qu'il imprime.*

```
Sum 0: 15
Sum 1: 18
Sum 3: 24
Sum 2: 21
Result: 78
```

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();
}
```

*Chaque noeud remplit le vecteur `in` de valeurs un peu différentes car la formule tient compte de `i` et `rank`. Ensuite, la fonction `MPI_Allgather` prend un élément de chaque noeud (`in[0]` du noeud 0, textttin[1] du noeud 1...) et propage cela au vecteur `out` de chaque noeud. Le contenu du vecteur `out` imprimé par les différents noeuds est donc identique.*

```
0: in = { 0 1 4 9 }
1: in = { 0 -1 0 3 }
1: out = { 0 -1 -4 -9 }
2: in = { 0 -3 -4 -3 }
2: out = { 0 -1 -4 -9 }
3: in = { 0 -5 -8 -9 }
3: out = { 0 -1 -4 -9 }
0: out = { 0 -1 -4 -9 }
```

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

*Un gestionnaire de ressource gère les ressources informatiques d'un calculateur parallèle et les alloue aux différentes tâches soumises, en fonctions de leurs différentes contraintes et de leur niveau de privilège et de priorité. C'est l'outil qui permet aux différents utilisateurs d'un calculateur parallèle de soumettre leurs tâches dans la queue d'exécution, afin qu'elles soient traitées à leur tour.*

# 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)**

*Le temps passé dans chaque fonction elle-même (self) est donné par le nombre des échantillons, chacun comptant pour 1ms. Pour le temps passé dans chaque fonction incluant les fonctions appelées (self+childs), il faut imputer le temps des fonctions appelées aux fonctions appelantes au prorata des appels. On commence par les fonctions feuilles (f3, f4, f5 et f6) où le temps self+childs est le temps de chaque fonction elle-même (self). Les fonctions f3, f6, f1 et f2 n'ont qu'un seul appelant et leur temps self+childs pourra donc être imputé directement à leur seul appelant. Cependant, le temps de f4, 12ms, doit être imputé à f1 et f2, en proportion des appels: 5/12 x 12ms = 5ms à f1 et 7/12 x 12ms = 7ms à f2. Le temps de f5, 7ms, doit aussi être imputé à f1 et f2, en proportion des appels: 6/14 x 7ms = 3ms à f1 et 8/14 x 7ms = 4ms à f2. Le temps de f1 est donc son temps self plus le temps de self+childs de f3 et les contributions de f4 et f5, soit 6ms + 2ms + 5ms + 3ms = 16ms. Il suffit de continuer ainsi en remontant jusqu'au programme principal.*

```
Fonction  self  self+childs
main       2ms  2+16(f1)+23(f2) = 41ms
f1         6ms  6+2(f3)+5(f4)+3(f5) = 16ms
f2         4ms  4+7(f4)+4(f5)+8(f6) = 23ms
f3         2ms  2ms
f4        12ms  12ms
f5         7ms  7ms
f6         8ms  8ms
```

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;
```

*Pour chaque entrée, on ajoute le nombre d'octets au* self *de la première fonction du chemin d'appel (celle qui a appelé malloc/new directement). On ajoute aussi le nombre d'octets au* self+childs *de chaque fonction sur le chemin d'appel. Ceci donne le résultat suivant.*

```
Fonction  self           self+childs
main                     64+32+18+24+12+8 = 158
f1        12             64+32+18+12 = 126
f2                       64+32 = 96
f3        64+8 = 72      64+8 = 72
f4        32             32+24 = 56
f5        24             18+24 = 42
f6        18             18
```

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)**

*OpenMP est typiquement utilisé pour programmer un ordinateur multi-coeur à mémoire partagée, par exemple pour partager l'exécution d'une boucle d'un programme entre plusieurs threads parallèles qui roulent sur autant de coeurs du même noeud. OpenCL est typiquement utilisé pour programmer les GPU. La décomposition du travail en très nombreux work item convient bien aux très nombreux coeurs des GPU, groupés en processeurs SIMD. Finalement, MPI permet de faire communiquer de très nombreux noeuds parallèles afin de résoudre un gros calul. Ces trois langages et librairies ont été prévus pour fonctionner ensemble. Plusieurs programmes exploitent donc ainsi à l'aide de ces trois technologies un très grand nombre de noeuds, chacun avec plusieurs threads sur autant de coeurs, et avec en plus une partie des calculs délégués au GPU. La plupart des ordinateurs au sommet du Top 500 contiennent ainsi un grand nombre de noeuds avec chacun plusieurs coeurs parallèles et quelques GPU.*

Le professeur: Michel Dagenais