**POLYTECHNIQUE MONTRÉAL**

**Département de génie informatique et génie logiciel**

**Cours INF8601:** *Systèmes informatiques parallèles (Automne 2020)*
*3 crédits (3-1.5-4.5)*

---

# CORRIGÉ DE L'EXAMEN FINAL

**DATE: Mardi le 15 décembre 2020**

**HEURE: 9h30 à 12h00**

**DUREE: 2H30**

**NOTE: Aucune documentation permise, aide-mémoire fourni à la fin du questionnaire, calculatrice non programmable permise**

**Ce questionnaire comprend 10 questions pour 20 points**

---

# Question 1 (2 points)

Le programme OpenCL suivant est exécuté sur une image de taille `image_width` (dimension 0) par `image_height` (dimension 1).

```
__kernel void Filter(const __global float* input,
    uint image_width, uint image_height, __global float* output)
{ int x = get_global_id(0); // variante A
  int y = get_global_id(1); // variante A
//  int x = get_global_id(1); // variante B
//  int y = get_global_id(0); // variante B

  int pos = y * image_width + x;
  float pixel = input[pos];
  if((x + y) % 2) {
    if(pixel < 100) pixel = pixel / 2;
  } else {
    if(pixel < 100) pixel = 0;
  }
  output[pos] = pixel;
}
```

a) Ce programme est exécuté sur un vecteur d'entrée, `input = {50, 150, 140, 40, 30, 130, 120, 20, 100}`, avec `image_width = 3` et `image_height = 3`. Quel sera le contenu du vecteur de sortie `output` à la fin? **(1 point)**

*La position x, y des éléments dans le vecteur est comme suit:* `input = {x0y0, x1y0, x2y0, x0y1, x1y1, x2y1, x0y2, x1y2, x2y2}`*. Le premier élément a une coordonnée de x=0 et y=0 et sera mis à 0 s'il est inférieur à 100, il en sera de même pour tous les éléments dont la somme x+y donne un nombre pair, soit x=2 y=0, x=1 y=1, x=0 y=2 et x=2 y=2. Ceci correspond aux éléments de position paire dans le vecteur linéaire. Pour les autres éléments dans le vecteur, de position impaire, leur valeur sera divisée par 2 s'ils sont inférieurs à 100. Le résultat est donc:* `output = {0, 150, 140, 20, 0, 130, 120, 10, 100}`

b) Votre co-équipier propose de modifier le programme en échangeant les dimensions 0 et 1, et donc en commentant les deux énoncés `variante A`, et en décommentant les deux énoncés `variante B`. Laquelle variante sera la plus efficace et pourquoi? **(1 point)**

*La variante A sera plus efficace car la dimension 0 (*`get_global_id(0)`*) varie le plus vite et fera accéder des cases consécutives en mémoire pour les différents fils d'un même processeur SIMD. Ceci permet de regrouper en un accès de bloc les accès mémoire à des mots consécutifs, effectués simultanément par les différents ALU d'un même processeur SIMD. En effet, dans le premier cas, x est associé à la dimension 0 et permet de passer d'une case consécutive à l'autre, pour les items consécutifs selon la dimension 0.*

# Question 2 (2 points)

Le programme OpenCL suivant est exécuté avec la fonction `clEnqueueNDRangeKernel` sur deux dimensions, `work_dim = 2`, avec les dimensions 0 et 1 de longueur 3, `global_work_size = {3, 3}`. Le contenu de toutes les variables d'entrée est montré en commentaire.

```
// float A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
// float B[] = {8, 7, 6, 5, 4, 3, 2, 1, 0};
// int widthA = 3; int widthB = 3;

__kernel void MM(__global float* C,
          __global float* A, __global float* B,
          int widthA, int widthB)
{  int i = get_global_id(1);
   int j = get_global_id(0);

   float value = 0;
   for (int k = 0; k < widthA; ++k)
   { value += A[i * widthA + k] * B[k * widthB + j];
   }
   C[i * widthB + j] = value;
}
```

a) Combien d'appels de la fonction kernel MM seront effectués pour ce calcul? **(0.5 point)**

*La fonction kernel MM sera effectuée 9 fois (dimension 0 x dimension 1 = 3 x 3 = 9);*

b) Combien de multiplications à virgule flottante (float) seront effectuées pour chaque appel de la fonction kernel MM? **(0.5 point)**

*A chaque tour de boucle, une multiplication à virgule flottante est effectuée (*`value += A[...] * B[...]`*). Il y aura donc 3 multiplications à virgule flottante.*

c) Quelle sera la valeur de `C[0]` après l'exécution de cette fonction kernel sur tous les éléments? **(1 point)**

*Cette fonction effectue une multiplication de matrices. Il faut donc multiplier les éléments de la première rangée de A avec ceux de la première colonne de B puis faire la somme. Cela donne* $0 \times 8 + 1 \times 5 + 2 \times 2 = 9$.

# Question 3 (2 points)

Le programme MPI suivant s'exécute sur 4 noeuds (MPI_Comm_size retourne 4). Quelle est la sortie produite par ce programme à l'écran?

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

int main (int argc, char *argv[])
{ int i, rank, size, chunk, rest, start, end, sum, res;
  int v[] = {1, 2, 3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3, 2, 1};
  MPI_Status status;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  chunk = sizeof(v) / sizeof(int) / size;
  rest = sizeof(v) / sizeof(int) % size;

  if(rank < rest) { chunk++; start = rank * chunk; }
  else { start = rank * chunk + rest; }
  end = start + chunk;
  sum = 0;
  for(i = start; i < end; i++) sum += v[i];
  printf("Node %d, chunk %d, sum %d\n", rank, chunk, sum);
  MPI_Reduce(&sum,&res,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
  if(rank == 0) printf("Result: %d\n", res);
  MPI_Finalize();
}
```

*Le nombre d'éléments est divisé par le nombre de processus, 15/4 = 3 reste 3. Ainsi, les premiers 3 processus (rank 0, 1, 2) auront 4 éléments (chunk = 4) alors que le dernier processus (rank = 3) en aura 3. Ensuite, chaque noeud effectue la somme des éléments qui lui sont réservés. Ainsi, le processus 0 fait 1+2+3+4=10, 1 fait 5+6+7+8=26, 2 fait 7+6+5+4=22 et 3 fait 3+2+1=6. Le total est de 10+26+22+6=64.*

```
Node 0, chunk 4, sum 10
Node 1, chunk 4, sum 26
Node 2, chunk 4, sum 22
Node 3, chunk 3, sum 6
Result: 64
```

# Question 4 (2 points)

Le programme MPI suivant s'exécute sur 4 noeuds (MPI_Comm_size retourne 4). Quelle est la sortie produite par ce programme à l'écran?

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

int main (int argc, char *argv[])
{
  int size, rank, i, in[4], o1[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 - rank * i + i * i; o1[i] = 0; }

  MPI_Alltoall(in, 1, MPI_INT, o1, 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]);
  MPI_Finalize();
}
```

*Il suffit d'exécuter le programme pour obtenir le résultat suivant:*

```
0: in={0, 2, 6, 12};
1: in={0, 1, 4, 9};
2: in={0, 0, 2, 6};
3: in={0, -1, 0, 3};
0: o1={0, 0, 0, 0};
1: o1={2, 1, 0, -1};
2: o1={6, 4, 2, 0};
3: o1={12, 9, 6, 3};
```

# Question 5 (2 points)

Un outil de mesure de performance, comme CPU Profile, échantillonne, à un intervalle de temps régulier de 1ms, le contenu de la pile d'appels (la fonction courante et celles dans le chemin d'appel). Lorsqu'un échantillon se répète, ce qui arrive souvent, un facteur multiplicatif est ajouté plutôt que de sauver le même chemin d'appel deux 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).

```
main, f1: 10
main, f1, f4: 22
main, f1, f3: 8
main, f1, f3, f8: 6
main, f2, f5: 5
main, f2, f7: 9
main, f3, f7: 11
main, f3, f8: 16
```

*L'exécution complète a pris 87ms. Pour chaque fonction, on compte le nombre de fois qu'elle arrive en début de pile pour le temps self. Pour le temps self + child, on compte le nombre de fois pour chaque fonction qu'elle est présente dans la pile d'appel. Dans chaque cas, on tient bien sûr compte du facteur multiplicatif.*

| fonction | self | self+child |
|---|---|---|
| main | 0 | 10+22+8+6+5+9+11+16 = 87 |
| f1 | 10 | 10+22+8+6 = 46 |
| f2 | 0 | 5+9 = 14 |
| f3 | 8 | 8+6+11+16 = 41 |
| f4 | 22 | 22 |
| f5 | 5 | 5 |
| f6 ? | | |
| f7 | 9+11 = 20 | 9+11 = 20 |
| f8 | 6+16 = 22 | 6+16 = 22 |

# Question 6 (2 points)

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 peut-on en déduire? Dans chaque cas, expliquez le problème et donnez le numéro de la ligne correspondante dans la trace.

```
01: malloc, 0x10000020, 0x10
02: write, 0x10000024, 0x4
03: read, 0x10000024, 0x4
04: read, 0x10000028, 0x4
05: malloc, 0x10000040, 0x20
06: malloc, 0x10000080, 0x10
07: write, 0x10000040, 0x8
08: read, 0x10000040, 0x8
09: free, 0x10000020
10: read, 0x10000084, 0x4
11: write, 0x10000084, 0x4
12: free, 0x10000080
13: read, 0x10000024, 0x4
14: write, 0x10000050, 0x10
15: read, 0x10000050, 0x10
16: free, 0x10000020
```

*A la ligne 4, puis à la ligne 10, on lit une case jamais initialisée (écrite). A la ligne 13, on lit une case dans un objet qui a été libéré à la ligne 9. A la ligne 16, on libère un objet déjà libéré à la ligne 9. A la fin du programme, l'objet alloué à la ligne 5 n'est jamais libéré.*

# Question 7 (2 points)

Un programme parallèle calcule la multiplication de deux matrices, $C = AB$, où A, B et C sont des matrices de nombres à virgule flottante (float), chacune d'une taille de 64 x 64. Ce calcul est effectué sur un ordinateur à mémoire partagée qui compte 16 coeurs, avec 1 fil d'exécution par coeur.

a) Si chaque fil d'exécution s'occupe du calcul de 4 rangées de la matrice résultat C, combien de nombres à virgule flottante différents de A et B seront accédés par ce fil? Décrivez desquels il s'agit, par exemple des rangées ou des colonnes complètes de A ou B. **(1 point)**

*Si le fil s'occupe des 4 rangées i à i+3 de C, il faudra que le fil accède les 4 rangées correspondantes de A (i à i+3) et toutes les colonnes de B, un total de 4 x 64 + 64 x 64 = 4352*

b) Si chaque fil d'exécution s'occupe du calcul d'une sous-matrice de C d'une taille de 8 x 8, combien de nombres à virgule flottante différents de A et B seront accédés par ce fil? Décrivez desquels il s'agit, par exemple des rangées ou des colonnes complètes de A ou B. **(1 point)**

*Si le fil s'occupe d'une sous-matrice de C de taille 8 x 8 qui va de (i,j) à (i+7, j+7), il faudra que le fil accède les 8 rangées correspondantes de A (i à i+7) et les 8 colonnes correspondantes de B (j à j+7), un total de 8 x 64 + 8 x 64 = 1024.*

# Question 8 (2 points)

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 ($x_1$).

```
A = np.array([
    [5, 2, 1],
    [2, 6, 2],
    [1, 2, 7]
])
b = np.array([29, 31, 26])
x = np.array([1, 1, 1])
```

*Il suffit de créer les matrices D et O et d'appliquer l'équation.*

```
[0, 2, 1] [1]   [3]  | [29]   [3]   [26]
[2, 0, 2] [1] = [4]  | [31] - [4] = [27]
[1, 2, 0] [1]   [3]  | [26]   [3]   [23]

[1/5   0   0]   [26]   [5.2 ]
[  0 1/6   0] x [27] = [4.5 ]
[  0   0 1/7]   [23]   [3.28]
```

*La réponse converge après plusieurs itérations vers x = [4.13414634, 3.03658537, 2.25609756]*

# Question 9 (2 points)

Soit le code suivant qui implémente une convolution 5x5 en OpenCL. Le programme et le noyau compilent sans erreur. L'exécution du code en entier ne donne aucune erreur, mais le résultat de la convolution n'est pas correct. Des questions se posent sur ces résultats incorrects.

```
// convolution.c
//
// Le noyau OpenCL est déjà compilé
// La file de commandes est déjà créée.
// La vérification d'erreurs est omise.

typedef struct pixel {
  cl_uchar bytes[3];
} pixel_t;

typedef struct image {
  size_t width;
  size_t height;
  pixel_t* pixels;
} image_t;

typedef struct __attribute__((packed)) params {
  cl_uint image_width;
  cl_uint image_height;
  cl_int convolution_coefficients[25];
  cl_float convolution_factor;
} params_t;

void execute(
    cl_context ctx, cl_command_queue queue,
    image_t* in, image_t* out) {
  unsigned int image_size = in->width *
      in->height * sizeof(pixel_t);

  cl_mem buf_in = clCreateBuffer(ctx,
     CL_MEM_READ_WRITE,image_size,NULL,&status);
  cl_mem buf_out = clCreateBuffer(ctx,
     CL_MEM_READ_WRITE,image_size,NULL,&status);
  clEnqueueWriteBuffer(queue, buf_in, CL_TRUE,
      0, image_size, in->pixels, 0, NULL, NULL);

  params_t params = { /* empty for now */ };
  clSetKernelArg(kernel,0,sizeof(buf_in),&buf_in);
  clSetKernelArg(kernel,1,sizeof(buf_out),&buf_out);
  clSetKernelArg(kernel,2,sizeof(params_t*),&params);

  size_t work_size[2] = {in->width, in->height};
  clEnqueueNDRangeKernel(queue, kernel, 2, NULL,
      work_size, NULL, 0, NULL, NULL);
  clFinish(queue);

  clEnqueueReadBuffer(queue,buf_out,CL_TRUE,0,
      image_size, out->pixels, 0, NULL, NULL)
  clFinish(queue);
}

// convolution.cl
typedef struct params {
  uint image_height;
  uint image_width;
  int convolution_coefficients[25];
  float convolution_factor;
} params_t;

typedef struct pixel {
  uchar bytes[3];
} pixel_t;

__kernel void convolution_kernel(
    __global pixel_t* in, __global pixel_t* out,
    __global params_t* params) {
  int x = get_global_id(0);
  int y = get_global_id(1);
  int pixel_index = x + y * (params.image_width);

  /* le calcul est omis, mais fonctionnel */
  do_convolution(in, out, params, pixel_index);
}
```

a) Un pointeur non initialisé est envoyé comme troisième paramètre du noyau à son exécution. Est-ce un problème? Quel en sera l'impact? **(0.5 point)**

*En effet, la variable* `params` *n'est pas initiallisée. Par conséquent, le noyau effectue des lectures vers de la mémoire non-initialisée. Ceci est un problème et peut facilement expliquer les résultats invalides.*

b) La définition d'une des structures de données dans le code C ne concorde pas avec la définition correspondante de la structure de données dans le code OpenCL. Est-ce un problème? Quel en sera l'impact? **(0.5 point)**

*En effet, les champs* `image_width` *et* `image_height` *sont dans l'ordre inverse. Les mauvaises valeurs seront donc lues dans le code OpenCL, pour la largeur et la hauteur de l'image, et le calcul sera fait avec les mauvais pixels de l'image. Cela aussi pourrait causer un mauvais résultat.*

c) Le tampon `buf_in` devrait être configuré avec `CL_MEM_READ_ONLY` et le tampon `buf_out` avec `CL_MEM_WRITE_ONLY`, afin que le noyau effectue seulement des lectures dans le premier et des écritures dans le deuxième. Est-ce un problème? Quel en sera l'impact? **(0.5 point)**

*C'est une optimisation manquée, puisque le contenu des tampons pourrait être copié dans les deux directions au lieu d'une seule. Cependant, ce n'est pas une erreur qui explique de mauvais résultats.*

d) Seulement le `global_work_size` est envoyé à la fonction `clEnqueueNDRangeKernel`, alors que le `local_work_size` est `NULL`. Est-ce un problème? Quel en sera l'impact? **(0.5 point)**

*C'est une valeur acceptable. Dans ce cas, l'environnement d'exécution choisira la taille du groupe automatiquement.*

# Question 10 (2 points)

Soit une base de données distribuée implémentée en MPI. Une requête pour effectuer une recherche dans cette base de données est décrite par la structure `search` ci-dessous en C. Le noeud principal MPI envoie cette requête à tous les autres noeuds de la base de données. Cette structure est envoyée en 2 parties. La partie 1 contient: `keywords_length`, `date_after`, `date_before`, `authors_length` et `result_count`. La partie 2 contient: `keywords` et `authors`. Le code initialisant le type MPI pour la première partie est fourni après la définition de la structure `search`. Il manque toutefois le code pour `MPI_Aint displacements[5] = {...}`. Fournissez le code manquant pour initialiser correctement le vecteur `displacements`.

```
typedef struct __attribute__((packed)) search {
  unsigned int keywords_length;
  char* keywords;
  unsigned long date_after;
  unsigned long date_before;
  unsigned int authors_length;
  char* authors;
  unsigned long result_count;
} search_t;

int count = 5;
int blocklengths[5] = {1, 1, 1, 1, 1};
MPI_Aint displacements[5] = { /* à compléter */ };
MPI_Datatype types[5] = {
    MPI_UNSIGNED,
    MPI_UNSIGNED_LONG,
    MPI_UNSIGNED_LONG,
    MPI_UNSIGNED,
    MPI_UNSIGNED_LONG,
};

MPI_Datatype type;
MPI_Type_create_struct(count, blocklengths,
    displacements, types, &type);
MPI_Type_commit(&type);
```

*Il y a plusieurs bonnes réponses et aussi des erreurs à éviter. La réponse a) utilise l'aide du compilateur, pour calculer le décalage de chaque champ dans la structure, et est probablement la meilleure réponse. Le code de b) est correct mais plus compliqué et plus sujet à erreur. Le code de b) ne tient pas compte de contraintes d'alignement qui pourraient faire que le compilateur insère des coussins (padding) entre les champs. Ceci est correct car l'attribut* `packed` *a été précisé pour la structure* `search`*. Le code c) est bon pour l'architecture x86_64 utilisée dans les laboratoires GIGL mais ce code n'est pas portable, par exemple sur une architecture 32 bits avec des pointeurs de 4 octets.*

*Le code d) essaie de soustraire quelque chose pour les deux champs sautés, ce qui est une erreur. La fonction offset nous donne directement la bonne valeur. Le code e) donne les adresses de fin des champs plutôt que de début. Le premier élément doit être 0 pour l'adresse du premier champ.*

```
// a) bonne réponse
MPI_Aint displacements[5] = {
    offsetof(search_t, keywords_length),
    offsetof(search_t, date_after),
    offsetof(search_t, date_before),
    offsetof(search_t, authors_length),
    offsetof(search_t, result_count),

// b) bonne réponse
MPI_Aint displacements[5] = {
    0,
    sizeof(unsigned int)
        + sizeof(char*),
    sizeof(unsigned int)
        + sizeof(char*)
        + sizeof(unsigned long),
    sizeof(unsigned int)
        + sizeof(char*)
        + 2 * sizeof(unsigned long),
    2 * sizeof(unsigned int)
        + 2 * sizeof(char*)
        + 2 * sizeof(unsigned long),
};

// c) bonne réponse
MPI_Aint displacements[5] = {0, 12, 20, 28, 40};
```

```
// d) mauvaise réponse
MPI_Aint displacements[5] = {
    offsetof(search_t, keywords_length),
    offsetof(search_t, date_after)
        - offsetof(search_t, keywords),
    offsetof(search_t, date_before),
    offsetof(search_t, authors_length),
    offsetof(search_t, result_count)
        - offsetof(search_t, authors),
};

// e) mauvaise réponse
MPI_Aint displacements[5] = {
    sizeof(unsigned int),
    sizeof(unsigned int)
        + sizeof(char*)
        + sizeof(unsigned long),
    sizeof(unsigned int)
        + sizeof(char*)
        + 2 * sizeof(unsigned long),
    2 * sizeof(unsigned int)
        + sizeof(char*)
        + 2 * sizeof(unsigned long),
    2 * sizeof(unsigned int)
        + 2 * sizeof(char*)
        + 3 * sizeof(unsigned long),
};
```

Le professeur: Michel Dagenais

# Annexe: aide-mémoire, ces pages ne seront pas corrigées

**Équations**

Loi d'Amdahl

$$A(p) = \frac{1}{(1-f) + \frac{f}{p}}$$

Probabilité k sur n fonctionnels

$$p = \binom{n}{k} \times p_d^k \times (1-p_d)^{n-k}$$

Probabilité p de deux événements **indépendants** p1 et p2: p = p1 x p2

**Barrière Papillon**

chaque thread communique avec un autre à chaque étape (i+1 % n, i+2 % n, i+4 % n...). Tous reçoivent l'information que tous sont prêts en même temps. Pas de phase d'annonce de fin!

**Verrous RCU**

Structures qui utilisent un pointeur en entrée (liste). Surtout des lectures. Lors d'un changement, atomiquement enlever le pointeur à l'ancienne donnée. Lorsque tous les lecteurs sortent, libère la mémoire.

**Opération sur les matrices**

$$A + B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a+e & b+f \\ c+g & d+h \end{pmatrix} = C$$

$$A - B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} - \begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} a-e & b-f \\ c-g & d-h \end{pmatrix} = C$$

$$k \times \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} ka & kb \\ kc & kd \end{pmatrix}$$

$$a_{11} \times b_{11} + a_{12} \times b_{21} + a_{13} \times b_{31} = c_{11}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

$$D = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_n \end{bmatrix} \Rightarrow D^{-1} = \begin{bmatrix} 1/d_1 & 0 & \cdots & 0 \\ 0 & 1/d_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1/d_n \end{bmatrix}$$

**Algorithme de Jacobi**

*itératif*

$$x^{(k+1)} = D^{-1}(b - Ox^{(k)})$$

Où D diagonale de A, O=A-D

**Mémoire**

Endroits où l'ordre peut changer : compilateur, pipeline, multiprocesseurs

Modèle séquentiel: Attendre propagation du write, ou attendre que les write de partout soient effectifs avant de faire read ou write suivant. Trop contraignant.

Total Order: Read avant écriture

Partial Order: Accès à var différentes réordonnancés par rapport aux write

Weak ordering: accès à var différentes réordonnancés

Barrières mémoires

Pleine : smp_mb() accès effectués avant tous ceux qui suivent

Pour lecture : smp_rmb() read effectués avant read qui suivent

Pour écriture : smp_wmb() write effectués avant write qui suivent

Lectures dépendantes : smp_read_barrier_depends() Read effectués avant read dépendants qui suivent

**Lorsque l'on utilise des barrières mémoires, il faut des barrières de chaque côté!!!**

**Valgrind**

Memcheck: Enregistrement de l'état de chaque bit. 10 à 30x plus lent.

Helgrind : 100x plus lent.

Cachegrind : Deux niveaux de cache L1 et LL (last-level). 50x plus lent.

Callgrind: Permet de voir l'arbre d'appel ie A->B->...->Z

Massif : Head profiler

**Google**

Thread Sanitizer: Division en segment de synchronisation. 2 à 20x plus lent, 5 à 10x plus de mémoire. Garde les quelques derniers accès.

Address Sanitizer: Enregistrement de l'état de chaque bloc de 8 octets. 2x plus lent.

CPU profile : échantillonnage dans le temps de la chaine d'appel

**Promela/Spin**: simuler des programmes, de quelques dizaines de lignes, pour lesquels l'ordre d'exécution n'est pas prévisible.

**Lockdep** : graphe d'ordre de prise des verrous par variable et par structure (inode, page...). Vérification du contexte (normal ou interruption).

**Gcov**: nb de fois que chaque ligne est exécutée. Surcout 10 à 15%

**GProf** : nb d'appels à chaque fonction avec provenance. Échantillonnage dans le temps. Hypothèse: les différents appels prennent le même temps, donc le temps self+childs n'est pas très fiable

**Oprofile**: Échantillonnage des instructions avec les compteurs matériels. Surcoût < 0.5%

**Algo Multiplication de matrices** (envoi des données seulement lorsque nécessaire)

Si p nœuds, décomposer matrice nxn en matrices de taille $m = n/\sqrt{p}$

```
1  iup  = i+1 mod m;
2  idown = i-1 mod m;
3  for (k = 0; k < m; k++) {
4       km = (i+k) mod m;
5       broadcast(A[i,km]) to all nodes handling row i of C;
6       C[i,j] = C[i,j] + A[i,km]*B[km,j]
7       send B[km,j] to the node handling C[idown,j]
8       receive new B[km+1 mod m,j] from the node handling C[iup,j]
9  }
```

**TBB**

parallel_for(blocked_range<size_t>(0,n), MyClosure(a));
parallel_reduce(blocked_range<size_t>(0, n), MySum(a));
parallel_reduce Utilise une Structure en arbre
parallel_scan(blocked_range<int>(0,n), MyScan(a, b));
parallel_do( _rst, last, MyClosure );
parallel_sort(begin, end);

Exemple de code (parallel_reduce)

```
struct Sum {
   float value;
   Sum() : value(0) {}
   Sum(Sum& s, split) { value = 0; }
   void operator()(const blocked_range<float*>& r) {
      float temp = value;
      for(float* a=r.begin(); a!=r.end(); ++a) temp +=
*a;
         value = temp;}
   void join(Sum& rhs) {value += rhs.value;}
};
float ParallelSum(float array[], size_t n) {
   Sum total;
parallel_reduce(blocked_range<float*>(array,
array+n),total);
   return total.value;}
```

Exemple de code (parallel_pipeline)

```
float RootMeanSquare( float* first, float* last ) {
   float sum=0;
   parallel_pipeline(16,
      make_filter<void,float*>(filter::serial,
         [&](flow_control& fc)-> float*{
         if( first<last ) return first++;
         else { fc.stop(); return NULL; }
      }) &
      make_filter<float*,float>(filter::parallel,
      [](float* p){return (*p)*(*p);}) &
      make_filter<float,void>(filter::serial,
      [&](float x) {sum+=x;})
  );
  return sqrt(sum);}
```

**OpenMP**

Vars déclarées avant *omp parallel* sont globales
Vars déclarées après *omp parallel* sont locales
*Omp single* un seul thread execute cette partie
Barrière implicite à la fin des blocs, *nowait* enlève la barrière
*Flush* de la mémoire implicite à la fin des blocs

**OpenCL**

Synchronisation

Marqueur pour savoir lorsque les commandes précédentes d'une queue sont terminées.
Barrière pour assurer que toutes les commandes précédentes sont faites avant de commencer les suivantes.
void barrier (cl_mem_fence__ags _ags): synchroniser tous les workItem du workGroup.
void mem_fence (cl_mem_fence__ags _ags) barrières mémoires pour mémoire du workItem.

**MPI**

MPI_Wait est bloquant | MPI_Test est non bloquant

```
int MPI_Type_create_struct(int count, int blocklengths[],
   MPI_Aint displacements[], MPI_Datatype types[],
   MPI_Datatype *newtype);
Int MPI_Type_commit(MPI_Datatype *newtype);
```