Programmation parallèle avec OpenMP

Présenté par Eric Giguère le 22 mai 2024

Nos membres et partenaires

Université Laval
McGill
Université de Sherbrooke
Université de Montréal
HEC Montréal
Université Concordia
Polytechnique Montréal
UQAC Université du Québec à Chicoutimi
UQTR Savoir. Surprendre.
UQÀM
ÉTS
INRS Institut national de la recherche scientifique
UQAT Université du Québec en Abitibi-Témiscamingue
Université TÉLUQ
Ouranos

Partenaire de l'
Alliance de recherche numérique du Canada

Partenaires connectivité

RISQ Le réseau au service du savoir
canarie

Partenaires financiers

INNOVATION.CA Canada Foundation for Innovation | Fondation canadienne pour l'innovation
Québec

Présentation du formateur

- Analyste en calcul scientifique à l’Université de Sherbrooke pour Calcul Québec depuis 2016.
- Études en physique des particules.

Cet après-midi

- Exercices;
- Questions;

De 13h30 à 16h

Objectifs

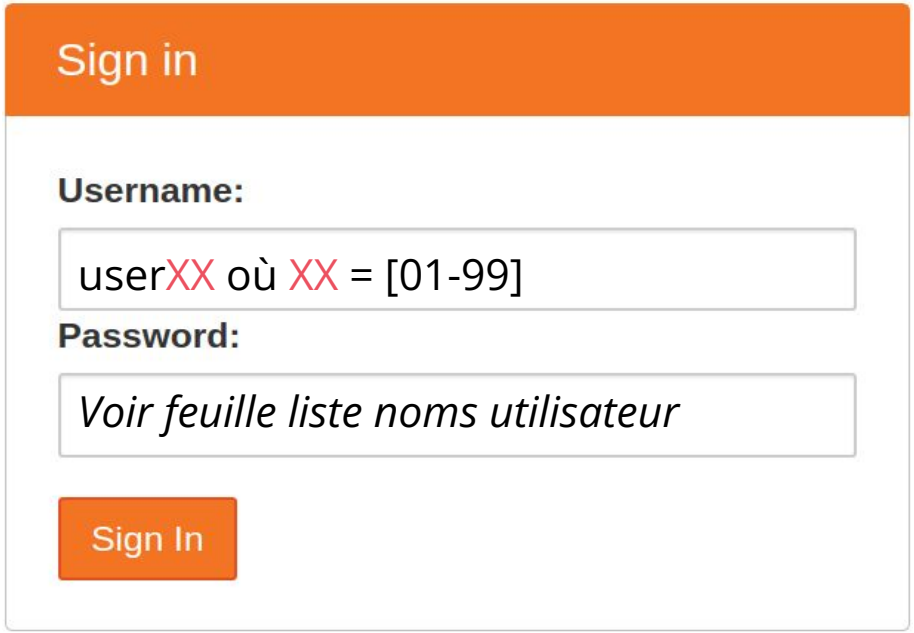
- **Se familiariser avec les concepts du calcul parallèle avec mémoire partagée.**
- **Savoir paralléliser un code avec OpenMP**
- **Connaître les particularités qui apparaissent avec OpenMP.**

Contenu de la formation

- Introduction à OpenMP
- Création de section parallèle
- Boucle parallel
- Gestion de la mémoire
- Synchronization
- Multiple boucles

Accéder à la plateforme

Navigateur web:
p-ecole.calculquebec.cloud



Préparation

Utiliser une tâche interative:
salloc --time=4:0:0 --cpus-per-task=4 --mem=8G

Exemples et exercices:
https://github.com/calculquebec/cq-formation-intro-openmp/tree/ecole_printemps

$ git clone https://github.com/calculquebec/cq-formation-intro-openmp.git
$ cd cq-formation-intro-openmp
$ git checkout ecole_printemps

Table de référence:
https://docs.computecanada.ca/wiki/OpenMP/fr

Introduction à OpenMP

De plus en plus de besoin de calcul

- Cpu plus vite/fort
- Ressource spécialisé (Gpu)
- Plus de cpus - **Parallélisme**

Plusieurs exécutions (vectorization)
Plusieurs ordinateurs (MPI)
Plusieurs coeurs de calculs (openMP)

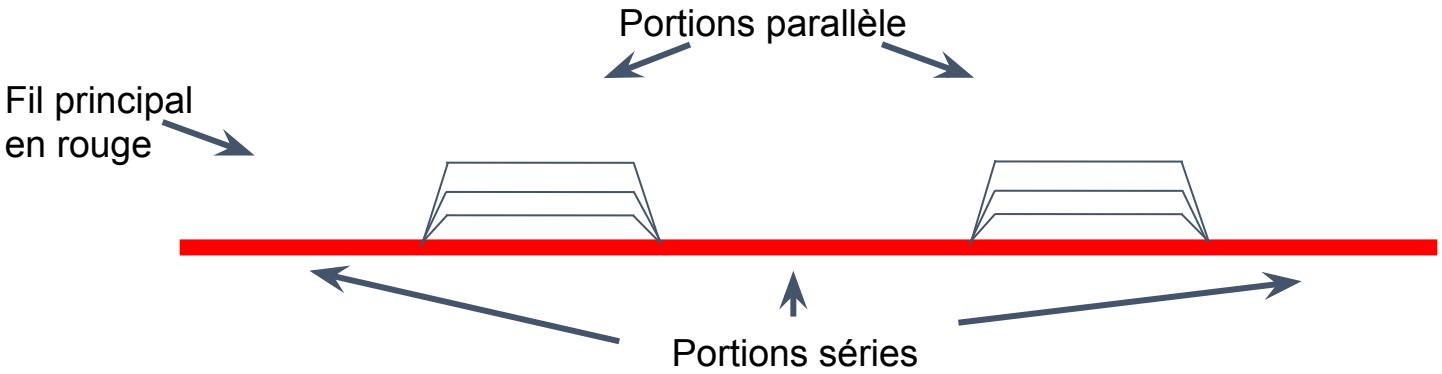
OpenMP (Open Multi-Processing) est une interface de programmation (C,C++, Fortran) pour le calcul parallèle avec mémoire partagée.

- Parallélisation “facile” de code et boucle dans un nœud.
- Peut-être utilisé avec MPI pour de la parallélisation hybride.

Parallélisation à mémoire partagée/fils d'exécutions (openmp)

Plusieurs fils d'exécutions

- Le fil principal lance un groupe de fil au besoin pour exécuter les portions parallèles du code.
- Chaque fil d'une même portion parallèle exécute une copie du même code/instructions
- Chaque fil a accès à l'espace mémoire du fil principal

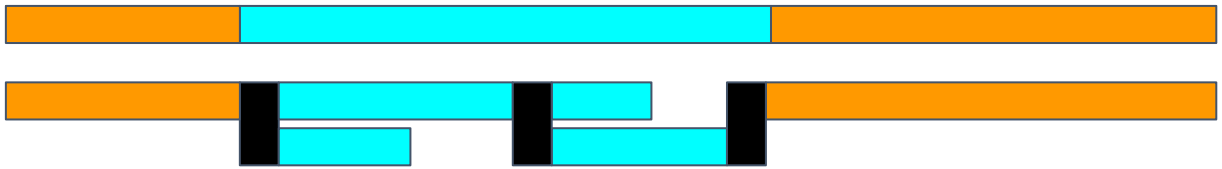


Défis

“Race condition”
- L’accès impromptu à des données partagées peut entraîner des situations/états inattendus

Synchronisation
- Permet de partager de l’information entre les tâches et de contrôler l’accès aux données partagées
- Trop de synchronisation impacts les performances et peut causer des ‘deadlocks’

Accélération/efficacité



Création de section parallèle

Structures openMP:

- `#include <omp.h>`
- `-fopenmp / -openmp`
- `#pragma omp parallel [...]`
- `$export OMP_NUM_THREADS=N`
- `omp_set_num_threads(N)`
- `omp_get_num_threads()`
- `omp_get_thread_num()`

OpenMP - les bases

‘#pragma omp parallel’ indique la section a faire en parallèle.

Généralement: #pragma omp [directive]
S’applique à un bloc({…}) de code
Pour fortran: !$OMP PARALLEL
!$OMP END PARALLEL

```
#pragma omp parallel
{
    printf(“Hello ”);
    printf(“world\n”);
}
```

Output:

```
Hello world
Hello world
```

```
#pragma omp parallel
    printf(“Hello ”);
    printf(“world”);
```

Output:

```
Hello Hello world
```

OpenMP - hello world

HelloWorld.c

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

int main(void) {
   printf("Début de la section parallèle\n");

#pragma omp parallel
  {
     printf("Hello world - fil #%d de %d\n",
     omp_get_thread_num(),omp_get_num_threads());
  }

  printf("Fin de la section parallèle\n");
}
```

Calcul Québec

```
$ gcc -fopenmp -o hello-world.out hello-world.c

$ OMP_NUM_THREADS=2 ./omp-hello-world
Début de la section parallèle
Hello world - fil #0 de 2
Hello world - fil #1 de 2
Fin de la section parallèle

$ OMP_NUM_THREADS=6 ./omp-hello-world
Début de la section parallèle
Hello world - fil #2 de 6
Hello world - fil #1 de 6
Hello world - fil #3 de 6
Hello world - fil #0 de 6
Hello world - fil #5 de 6
Hello world - fil #4 de 6
Fin de la section parallèle
```

Exemple 1 : Hello World

Le compilateur doit supporter OpenMP et être appelé avec l'option appropriée:

-fopenmp pour gcc

-openmp pour icc

OMP_NUM_THREADS détermine le nombre de fils. Dans le code **omp_set_num_threads**() contrôle le nombre de fils.

Parallélisation de boucle*

boucle.c

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

int main(void) {
   float A[100];
   float sum = 0.;
#pragma omp parallel
  {
    int fil_n = omp_get_thread_num();
    int num_fils = omp_get_num_threads();
    for(int i=fil_n; i<100; i+=num_fils){
        A[i] = 1./i/i;
    }
  }

  for(int i=0; i<100; i++){
     sum += A[i];
  }
  printf("La somme est %d", sum);
}
```

Exercice

Qu'est ce qui a priorité: **OMP_NUM_THREADS** ou **omp_set_num_threads**?
Partir de **HelloWorld2.c** pour faire le test.

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

int main(void) {
   printf("Début de la section parallèle\n");
   omp_set_num_threads(4);
#pragma omp parallel
   {
      printf("Hello world - fil #%d de %d\n",
             omp_get_thread_num(),
             omp_get_num_threads());
   }
   printf("Fin de la section parallèle\n");
}
```

Boucle parallel

Structures openMP:

- `#pragma omp for [...]`
- `schedule [static, dynamic]`

Quelles boucles peuvent être faites en parallèle avec openmp?

1:
```
for(i=0; i<N; i++){
    A[i] = A[i] + B[N-1-i]
}
```

2:
```
for(i=0; i<N; i++){
    A[i] = A[i] + A[N-1]
}
```

3:
```
for(i=0; i<N; i++){
    for(j=i; j<N; j++){
        C[j] = A[i] * B[j-i]
    }
}
```

4:
```
for(i=0; i<N; i++){
    A[i] = A[i] * A[i+1]
}
```

5:
```
for(i=0; i<N; i+=2){
    A[i] = A[i] + A[i+3]
}
```

6:
```
for(i=0; i<N; i++){
    for(j=i; j<N; j++){
        C[i] += A[j] * B[j-i]
    }
}
```

Parallélisation de boucle*

boucle.c

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

int main(void) {
   float A[100];
   int i;
   float sum = 0.;
#pragma omp parallel
  {
    int fil_n = omp_get_thread_num();
    int num_fils = omp_get_num_threads();
    for(i=fil_n; i<100; i+=num_fils){
        A[i] = 1./i/i;          // <- Itération indépendante (inverse)
    }
  }

  for(int i=0; i<100; i++){
     sum += A[i];
  }
  printf("La somme est %d", sum);
}
```

pragma omp for

boucle2.c

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

int main(void) {
   float A[100];
   int i;
   float sum = 0.;
#pragma omp parallel
  {
#pragma omp for
    for(i=0; i<100; i++){
        A[i] = 1./i/i;
    }
  }

  for(i=0; i<100; i++){
     sum += A[i];
  }
  printf("La somme est %d", sum);
}
```

Ordre des itérations

Exemple 2 : schedule1.c

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h>
#define N 100

int main(void) {
int i;
#pragma omp parallel
  {
       char done_here[N+1];
       done_here[N] = 0;
       for(i=0; i<N; i++){done_here[i] = '.';}
#pragma omp for
       for(i=0; i<N; i++){
       usleep(1000*((i*i)%10));
       done_here[i] = 'o';
       }
  printf("Fil %d: %s \n", omp_get_thread_num(), done_here);
  }
}
```

Ordre des itérations

Exemple 2 : schedule1.c

```
$ OMP_NUM_THREADS=4 ./schedule1.out
Fil 0: ooooooooooooooooooooooooo.....................................................................
Fil 1: .........................ooooooooooooooooooooooooo.....................................................
Fil 2: ...................................................ooooooooooooooooooooooooo.........................
Fil 3: .....................................................................................ooooooooooooooooooooooooo
```

Le partage de boucle static: change fil reçois 1/N itérations consécutives*.

-Pas idéale si chaque itération prend un temps différent.

-L'option **schedule(method, N)** permet de contrôler le partage.

Ordre des itérations

Exemple 2 : schedule2.c

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h>
#define N 100

int main(void) {
int i;
#pragma omp parallel
  {
       char done_here[N+1];
       done_here[N] = 0;
       for(i=0; i<N; i++){done_here[i] = '.';}
#pragma omp for schedule(static, 2)
       for(i=0; i<N; i++){
       usleep(1000*((i*i)%10));
       done_here[i] = 'o';
       }
  printf("Fil %d: %s \n", omp_get_thread_num(), done_here);
  }
}
```

Ordre des itérations

Exemple 2 : schedule1.c

```
$ OMP_NUM_THREADS=4 ./schedule1.out
Fil 0: oooooooooooooooooooooooo........................................................................
Fil 1: ........................oooooooooooooooooooooooo........................................................
Fil 2: ................................................oooooooooooooooooooooooo........................
Fil 3: ........................................................................oooooooooooooooooooooooo
```

schedule(static, 2)

Exemple 2 : schedule2.c

```
$ OMP_NUM_THREADS=4 ./schedule2.out
Fil 0: oo......oo......oo......oo......oo......oo......oo......oo......oo......oo......oo......oo..
Fil 1: ..oo......oo......oo......oo......oo......oo......oo......oo......oo......oo......oo......oo
Fil 2: ....oo......oo......oo......oo......oo......oo......oo......oo......oo......oo......oo......
Fil 3: ......oo......oo......oo......oo......oo......oo......oo......oo......oo......oo......oo....
```

Exercise - schedule

Tester l'option schedule. (static, dynamic, guided)

Solution: schedule3.c

```
#include <stdio.h>
#include <omp.h>
#include <unistd.h>
#define N 100

int main(void) {
int i;
printf("schedule(..., ...)\n");
#pragma omp parallel
  {
      char done_here[N+1];
      done_here[N] = 0;
      for(i=0; i<N; i++){done_here[i] = '.';}
#pragma omp for schedule(..., ...)
      for(i=0; i<N; i++){
      usleep(1000*((i*i)%10));
      done_here[i] = 'o';
      }
  printf("Fil %d: %s \n", omp_get_thread_num(), done_here);
  }
}
```

Exercise - défit

$$\pi = 4\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1} = 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + - \cdots\right)$$

pi1.c

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

int main(void) {
int i, N=1000000;
double sum = 0.;

  for(i=0; i<N; i+=2){
    sum += 4./(2.*i+1.);
    sum -= 4./(2.*i+3.);
  }
  printf("pi = %f\n",sum);
}
```

Paralléliser le code.

Est-ce plus vite?
$ time ./pi1.out

Est-ce que pi est bon?

Schedule

- **schedule(static, N)**
  Les itérations sont distribuées en bloc de N en ordre.
  Le fil 0 aura les itérations (0, N-1)
  Le fil 1 aura les itérations (N, 2N-1), etc.

- **schedule(dynamic, N)**
  Les itérations sont distribuées en bloc de N sans ordre particulier.
  Dès qu'un fil est libre, il prend le prochain bloc disponible.

- **schedule(guided, N)**
  Les fils prennent des blocs de (N_itération_restante / N_fils) avec un minimum de N, dès qu'ils sont libres.
  100 itérations avec 5 fils résultent en blocs de (20, 16, 13, ...)

Gestion de la mémoire

Structures openMP:

- `Private, public, firstprivate`
- `default(None)`

Calcul de Pi

$$\pi = 4\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1} = 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + - \cdots\right)$$

pi1.c

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

int main(void) {
int i, N=1000000;
double sum = 0.;
for(i=0; i<N; i+=2){
     sum += 4./(2.*i+1.);
     sum -= 4./(2.*i+3.);
   }
   printf("pi = %f\n",sum);
}
```

Calcul de Pi

$$\pi = 4\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1} = 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + - \cdots\right)$$

pi2.c

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

int main(void) {
int i, N=1000000;
double sum = 0.;
#pragma omp parallel for
  for(i=0; i<N; i+=2){
     sum += 4./(2.*i+1.);
     sum -= 4./(2.*i+3.);
  }
  printf("pi = %f\n",sum);
}
```

Calcul de Pi

$$\pi = 4\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1} = 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + - \cdots\right)$$

pi2.c

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

int main(void) {
int i, N=1000000;
double sum = 0.;
#pragma omp parallel for
  for(i=0; i<N; i+=2){
     sum += 4./(2.*i+1.);
     sum -= 4./(2.*i+3.);
  }
  printf("pi = %f\n",sum);
}
```

```
$ OMP_NUM_THREADS=1 ./pi2.exe
pi = 3.141583
$ OMP_NUM_THREADS=2 ./pi2.exe
pi = 3.059115
$ OMP_NUM_THREADS=2 ./pi2.exe
pi = 3.016857
$ OMP_NUM_THREADS=4 ./pi2.exe
pi = 2.720920
$ OMP_NUM_THREADS=4 ./pi2.exe
pi = -0.005669
$ OMP_NUM_THREADS=8 ./pi2.exe
pi = -0.062608
$ OMP_NUM_THREADS=8 ./pi2.exe
pi = 3.149340
```

Calcul de Pi

$$\pi = 4\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1} = 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + - \cdots\right)$$

pi2.c

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

int main(void) {
int i, N=1000000;
double sum = 0.;
#pragma omp parallel for
  for(i=0; i<N; i+=2){
     sum += 4./(2.*i+1.);
     sum -= 4./(2.*i+3.);
  }
  printf("pi = %f\n",sum);
}
```

“Race condition” pour `sum`. La synchronisation n’est pas instantanée.

Calcul de Pi

pi2.1.c

```
#include <stdio.h>
#include <omp.h>
#define MAX_NUM_THREADS 40

int main(void) {
int i, N=1000000, fil_n, num_threads;
double Sum =0., sum[MAX_NUM_THREADS];

#pragma omp parallel
  {
      fil_n = omp_get_thread_num();
      num_threads = omp_get_num_threads();
      sum[fil_n] = 0.;
#pragma omp for
      for(i=0; i<N; i+=2){
        sum[fil_n] += 4./(2.*i+1.);
        sum[fil_n] -= 4./(2.*i+3.);
      }
  }
  for(i=0; i<num_threads; i++){
    Sum += sum[i];
  }
  printf("pi = %f\n",Sum);
}
```

Calcul de Pi

pi2.1.c

```
#include <stdio.h>
#include <omp.h>
#define MAX_NUM_THREADS 40

int main(void) {
int i, N=1000000, fil_n, num_threads;
double Sum =0., sum[MAX_NUM_THREADS];

#pragma omp parallel
  {
      fil_n = omp_get_thread_num();
      num_threads = omp_get_num_threads();
      sum[fil_n] = 0.;
#pragma omp for
      for(i=0; i<N; i+=2){
        sum[fil_n] += 4./(2.*i+1.);
        sum[fil_n] -= 4./(2.*i+3.);
      }
  }
  for(i=0; i<num_threads; i++){
    Sum += sum[i];
  }
  printf("pi = %f\n",Sum);
}
```

```
$ OMP_NUM_THREADS=1 ./pi2.1.exe
pi = 3.141583
$ OMP_NUM_THREADS=2 ./pi2.1.exe
pi = 3.200359
$ OMP_NUM_THREADS=2 ./pi2.1.exe
pi = -0.000386
$ OMP_NUM_THREADS=4 ./pi2.1.exe
pi = 0.000008
$ OMP_NUM_THREADS=4 ./pi2.1.exe
pi = 2.994991
$ OMP_NUM_THREADS=8 ./pi2.1.exe
pi = -0.000019
$ OMP_NUM_THREADS=8 ./pi2.1.exe
pi = 3.141348
```

Clause ‘shared’, ‘private’, ‘default’

Automatiquement, toutes les variables sont partagées, ce qui peut causer des erreurs.

#pragma omp parallel **default(none)**

- Protège contre le partage non voulu.

#pragma … **shared(A,B,C) private(D,E,F)**

- Listes de variables partagées et unique à chaque fils.

Les variables définies dans la partie parallèle sont privées.

Calcul de Pi

pi2.2.c

```
#include <stdio.h>
#include <omp.h>
#define MAX_NUM_THREADS 40

int main(void) {
int i, N=1000000, fil_n, num_threads;
double Sum =0., sum[MAX_NUM_THREADS];

#pragma omp parallel default(none) \
private(fil_n, i) shared(num_threads, sum, N)
  {
       fil_n = omp_get_thread_num();
       num_threads = omp_get_num_threads();
       sum[fil_n] = 0.;
#pragma omp for
       for(i=0; i<N; i+=2){
          sum[fil_n] += 4./(2.*i+1.);
          sum[fil_n] -= 4./(2.*i+3.);
       }
  }
  for(i=0; i<num_threads; i++){
    Sum += sum[i];
  }
  printf("pi = %f\n",Sum);
}
```

Partage des variables:
Automatiquement, les variables sont partagées.

Un seul “fil_n” pour tous les fils...

Calcul de Pi

pi2.2.c

```
#include <stdio.h>
#include <omp.h>
#define MAX_NUM_THREADS 40

int main(void) {
int i, N=1000000, fil_n, num_threads;
double Sum =0., sum[MAX_NUM_THREADS];

#pragma omp parallel default(none) \
private(fil_n, i) shared(num_threads, sum, N)
  {
       fil_n = omp_get_thread_num();
       num_threads = omp_get_num_threads();
       sum[fil_n] = 0.;
#pragma omp for
       for(i=0; i<N; i+=2){
          sum[fil_n] += 4./(2.*i+1.);
          sum[fil_n] -= 4./(2.*i+3.);
       }
  }
  for(i=0; i<num_threads; i++){
    Sum += sum[i];
  }
  printf("pi = %f\n",Sum);
}
```

```
$ OMP_NUM_THREADS=1 ./pi2.2.exe
pi = 3.141583
$ OMP_NUM_THREADS=8 ./pi2.2.exe
pi = 3.141583
```

Calcul de Pi

pi2.2.c

```
#include <stdio.h>
#include <omp.h>
#define MAX_NUM_THREADS 40

int main(void) {
int i, N=1000000, fil_n, num_threads;
double Sum =0., sum[MAX_NUM_THREADS];

#pragma omp parallel default(none) \
private(fil_n, i) shared(num_threads, sum, N)
  {
       fil_n = omp_get_thread_num();
       num_threads = omp_get_num_threads();
       sum[fil_n] = 0.;
#pragma omp for
       for(i=0; i<N; i+=2){
          sum[fil_n] += 4./(2.*i+1.);
          sum[fil_n] -= 4./(2.*i+3.);
       }
  }
  for(i=0; i<num_threads; i++){
    Sum += sum[i];
  }
  printf("pi = %f\n",Sum);
}
```

```
$ OMP_NUM_THREADS=1 ./pi2.2.exe
pi = 3.141583
$ OMP_NUM_THREADS=8 ./pi2.2.exe
pi = 3.141583
$ time OMP_NUM_THREADS=1 ./pi2.2.exe
real    0m8.633s
user    0m8.636s
sys     0m0.000s
$ time OMP_NUM_THREADS=2 ./pi2.2.exe
real    0m7.660s
user    0m15.104s
sys     0m0.000s
$ time OMP_NUM_THREADS=8 ./pi2.2.exe
real    0m8.204s
user    0m56.532s
sys     0m0.024s
```

Calcul de Pi

pi2.2.c

```
#include <stdio.h>
#include <omp.h>
#define MAX_NUM_THREADS 40

int main(void) {
int i, N=1000000, fil_n, num_threads;
double Sum =0., sum[MAX_NUM_THREADS];

#pragma omp parallel default(none) \
private(fil_n, i) shared(num_threads, sum, N)
  {
       fil_n = omp_get_thread_num();
       num_threads = omp_get_num_threads();
       sum[fil_n] = 0.;
#pragma omp for
       for(i=0; i<N; i+=2){
          sum[fil_n] += 4./(2.*i+1.);
          sum[fil_n] -= 4./(2.*i+3.);
       }
  }
  for(i=0; i<num_threads; i++){
    Sum += sum[i];
  }
  printf("pi = %f\n",Sum);
}
```

Faux partage de sum.

Produit de vecteur

vec_product.c

```
void vec_product(
  double left[], double right[],
  double alpha, double out[], int N)
{
#pragma omp parallel for schedule(...)
    for (int i=0; i<N; i++){
       out[i] = left[i] * right[i] * alpha;
    }
}
```

```
$ ./vec_product.o # (static)
real    0m0.397s
user    0m4.681s
sys    0m0.008s

$ ./vec_product.o # (static, 1)
real    0m4.908s
user    0m56.698s
sys     0m0.136s

$ ./vec_product.o # (dynamic)
real    0m38.817s
user    5m10.488s
sys     0m0.012s
```

Exercice: Parallelizer

euler.c

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

int main(void) {
  int i;
  double fact[100], sum=0.;

  fact[0]=1
  for(i=1; i<100; i++){
    fact[i] = i*fact[i-1];
  }

  for(i=0; i<100; i++){
    sum += 1./fact[i];
  }
  printf("The sum is %f/n",sum)
}
```

Exercice: Trouver l'erreur

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

int main(void) {
int i, j;
double temp, a[100][100], b[100], c[100];

for(i=0; i<100; i++){
  b[i] = i + i*i/10 - 0.23*i*i*i;
  c[i] = i - i*i/10 + 0.64*i*i*i;
}

#pragma omp parallel for private(temp)
  for(i=0; i<100; i++){
    for(j=0; j<100; j++){
      temp = b[i] - c[j];
      a[i][j] = temp*temp - b[i];
    }
  }
}
```

Synchronization

Structures openMP:

- `#pragma omp critical`
- `#pragma omp atomic`
- `reduction`

Calcul de Pi

pi2.2.c

```
#include <stdio.h>
#include <omp.h>
#define MAX_NUM_THREADS 40

int main(void) {
int i, N=1000000, fil_n, num_threads;
double Sum =0., sum[MAX_NUM_THREADS];

#pragma omp parallel default(none) \
private(fil_n, i) shared(num_threads, sum, N)
  {
       fil_n = omp_get_thread_num();
       num_threads = omp_get_num_threads();
       sum[fil_n] = 0.;
#pragma omp for
       for(i=0; i<N; i+=2){
          sum[fil_n] += 4./(2.*i+1.);
          sum[fil_n] -= 4./(2.*i+3.);
       }
  }
  for(i=0; i<num_threads; i++){
    Sum += sum[i];
  }
  printf("pi = %f\n",Sum);
}
```

Calcul de Pi -- mieux?

pragma omp critical/atomic : instruction à être exécuté par un seul fil à la fois.

pi3.c

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

int main(void) {
int i, N=1000000000;
double pi_i, sum = 0.;
#pragma omp parallel for private(pi_i)
  for(i=0; i<N; i+=2){
     pi_i = 4./(2.*i+1.);
     pi_i -= 4./(2.*i+3.);
#pragma omp atomic
     sum += pi_i;
  }
  printf("pi = %f\n",sum);
}
```

```
#pragma omp atomic
sum += pi_i;
```

Un fil à la fois pour une opération simple, synchronisation par cpu.

```
#pragma omp critical
sum = f(sum, pi_i);
```

Un fil à la fois, synchronisation par openmp. Toutes opérations supportées.

Calcul de Pi -- mieux?

pragma omp critical/atomic : instruction à être exécuté par un seul fil à la fois.

pi3.c

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

int main(void) {
int i, N=1000000000;
double pi_i, sum = 0.;
#pragma omp parallel for private(pi_i)
  for(i=0; i<N; i+=2){
     pi_i = 4./(2.*i+1.);
     pi_i -= 4./(2.*i+3.);
#pragma omp atomic
     sum += pi_i;
  }
  printf("pi = %f\n",sum);
}
```

```
$ time OMP_NUM_THREADS=1 ./pi3.c.o
pi = 3.141593
real    0m9.430s
user    0m9.428s

$ time OMP_NUM_THREADS=2 ./pi3.c.o
pi = 3.141593
real    0m6.023s
user    0m11.864s

$ time OMP_NUM_THREADS=4 ./pi3.c.o
pi = 3.141593
real    0m14.736s
user    0m57.392s

$ time OMP_NUM_THREADS=8 ./pi3.c.o
pi = 3.141593
real    0m15.625s
user    1m52.292s
```

Calcul de Pi -- mieux?

pragma omp critical/atomic : instruction à être exécuté par un seul fil à la fois.

pi3.1.c

```
int main(void) {
int i, N=1000000;
double sum=0., pi_fil;

#pragma omp parallel private(pi_fil)
{
  pi_fil= 0.;
#pragma omp for
  for(i=0; i<N; i+=2){
    pi_fil += 4./(2.*i+1.);
    pi_fil -= 4./(2.*i+3.);
  }
#pragma omp atomic
  sum += pi_fil;
}
  printf("pi = %f\n",sum);
}
```

Moins de synchronisation.

Calcul de Pi -- mieux?

pragma omp critical/atomic : instruction à être exécuté par un seul fil à la fois.

pi3.1.c

```
int main(void) {
int i, N=1000000;
double sum=0., pi_fil;

#pragma omp parallel private(pi_fil)
{
  pi_fil= 0.;
#pragma omp for
  for(i=0; i<N; i+=2){
    pi_fil += 4./(2.*i+1.);
    pi_fil -= 4./(2.*i+3.);
  }
#pragma omp atomic
  sum += pi_fil;
}
  printf("pi = %f\n",sum);
}
```

```
$ time OMP_NUM_THREADS=1 ./pi3.1.c.o
pi = 3.141593
real    0m8.674s
user    0m8.676s

$ time OMP_NUM_THREADS=2 ./pi3.1.c.o
pi = 3.141593
real    0m4.388s
user    0m8.728s

$ time OMP_NUM_THREADS=4 ./pi3.1.c.o
pi = 3.141593
real    0m2.238s
user    0m8.928s

$ time OMP_NUM_THREADS=8 ./pi3.1.c.o
pi = 3.141593
real    0m1.221s
user    0m9.072s
```

Calcul de Pi -- encore mieux

reduction : joindre les valeurs de chaque fils en une.

pi4.c

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

int main(void) {
int i, N=1000000000;
double sum = 0.;
#pragma omp parallel for reduction(+:sum)
  for(i=0; i<N; i+=2){
     sum += 4./(2.*i+1.);
     sum -= 4./(2.*i+3.);
  }
  printf("pi = %f\n",sum);
}
```

```
$ time OMP_NUM_THREADS=1 ./pi4.c.o
pi = 3.141593
real    0m8.626s
user    0m8.628s

$ time OMP_NUM_THREADS=8 ./pi4.c.o
pi = 3.141593
real    0m1.251s
user    0m9.240s
```

Exercice: Parallelizer (2)

euler.c

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

int main(void) {
  int i;
  double fact[100], sum=0.;

  fact[0]=1
  for(i=1; i<100; i++){
    fact[i] = i*fact[i-1];
  }

  for(i=0; i<100; i++){
    sum += 1./fact[i];
  }
  printf("The sum is %f/n",sum)
}
```

Refaire le dernier exercice avec atomic, critical et reduction.

Quel est le plus facile à utiliser, le plus rapide?

Multiple boucles

Structures openMP:

- `collapse`
- `#pragma omp single`
- `#pragma omp master`
- `#pragma omp barrier`
- `nowait`

Boucles imbriqué

```
int i,j, N=1000;
double A[1000][1000] = 0.;
for(i=0; i<N; i++){
  for(j=0; j<N; j++){
     A[i][j]=f(i,j);
  }
}
```

Boucles imbriqué

collapse : joindre des boucles.

collapse.c

```
int i,j, N=1000;
double A[1000][1000];
#pragma omp parallel for private(i,j) shared(A) \
            collapse(2) schedule(dynamic)
  for(i=0; i<N; i++){
    for(j=0; j<N; j++){
       A[i][j]=f(i,j);
    }
  }
```

Normalization de vecteur

```
...
#define N 100
int main(void){
  double vec[N], norm=0., norm2=0.;

  for(i=0; i<N; i++){
    vec[i] = i*(1+i*(0.05+i*0.0025));
  }

  for(i=0; i<N; i++){
    norm2 += vec[i]*vec[i];
  }

  printf("pi = %f\n",norm2);
  norm = sqrt(norm2);

  for(i=0;i<N;i++){
    vec[i] /= norm;
  }
  norm2 = 0.;
  for(i=0; i<N; i++){
    norm2 += vec[i]*vec[i];
  }
  printf("pi = %f\n",norm2);
}
```

Créer des fils prend du temps. Idéalement, on veut créer les fils une fois pour toutes les boucles.

Normalization de vecteur

```
#pragma omp parallel default(none) shared(vec,norm,norm2) private(i)
{
#pragma omp for
  for(i=0;i<N;i++){
       vec[i] = i*(1+i*(0.05+i*0.0025));
  }
#pragma omp for reduction(+:norm2)
  for(i=0;i<N;i++){
       norm2 += vec[i]*vec[i];
  }
#pragma omp single
{
  printf("pi = %f\n",norm2);
  norm = sqrt(norm2);
  norm2 = 0.;
}
#pragma omp for nowait schedule(static)
  for(i=0;i<N;i++){
       vec[i] /= norm;
  }
#pragma omp for schedule(static) nowait reduction(+:norm2)
  for(i=0;i<N;i++){
       norm2 += vec[i]*vec[i];
  }
#pragma omp barrier
#pragma omp master
  printf("pi = %f\n",norm2);
}
```

Exercise - Laplace

Parallelizer avec une section parallèle.

```
while(var > tol && iter <= maxIter){
  ++iter;
  var = 0.0;
  for (i=1; i<=n; ++i)
  for (j=1; j<=n; ++j) {
    Tnew[i*n2+j] = 0.25*(T[(i-1)*n2 + j] +T[(i+1)*n2 + j]
                         + T[i*n2 + (j-1)] +T[i*n2 + (j+1)]);
    var = fmax(var, fabs(Tnew[i*n2 + j] - T[i*n2 + j]));
  }
  Tmp=T; T=Tnew; Tnew=Tmp;
  if (iter%100 == 0)
  printf("iter: %8u, variation = %12.4lE\n", iter, var);
}
```

openmp task

Structures openMP:

- #pragma omp task
- #pragma omp taskwait

Quicksort

quicksort.c

```
void quicksort(double array[], int low,
int high){
  int pi;
  if (low < high) {
    pi = partition(array, low, high);
    quicksort(array, low, pi - 1);
    quicksort(array, pi + 1, high);
  }
}
```

```
int partition(double array[], int low, int high){

  double pivot = array[high];
  int i = low;

  for(int j=low; j<=high-1; j++) {
    if (array[j] <= pivot){
      swap(&array[i], &array[j]);
      i++;
    }
  }
  swap(&array[i], &array[high]);
  return i;
}
```

Quicksort

quicksort_omp.c

```
void quicksort(double array[], int low,
int high){
  int pi;
  if (low < high) {
    pi = partition(array, low, high);
#pragma omp task
    quicksort(array, low, pi - 1);
#pragma omp task
    quicksort(array, pi + 1, high);
  }
}

void main(){
…
#pragma omp parallel shared(array, N)
#pragma omp single
  quicksort(array, 0, N-1);
…
}
```

```
$ ./quicksort.o
time: 179.846000 ms

$ OMP_NUM_THREADS=1 ./quicksort_omp.o
time: 219.646116 ms
$ OMP_NUM_THREADS=4 ./quicksort_omp.o
time: 75.847382 ms
$ OMP_NUM_THREADS=8 ./quicksort_omp.o
time: 61.349241 ms
```

Quicksort

quicksort_omp_2.c

```
void quicksort(double array[], int low,
int high){
  int pi;
  if(high-low <= 0){return;}
  if(high-low <= 1000){
    pi = partition(array, low, high);
    quicksort(array, low, pi - 1);
    quicksort(array, pi + 1, high);
  } else {
    pi = partition(array, low, high);
    #pragma omp task
    quicksort(array, low, pi - 1);
    #pragma omp task
    quicksort(array, pi + 1, high);
  }
}
```

```
$ ./quicksort.o
time: 179.846000 ms

$ OMP_NUM_THREADS=1 ./quicksort_omp.o
time: 219.646116 ms
$ OMP_NUM_THREADS=4 ./quicksort_omp.o
time: 75.847382 ms
$ OMP_NUM_THREADS=8 ./quicksort_omp.o
time: 61.349241 ms

$ OMP_NUM_THREADS=1 ./quicksort_omp_2.o
time: 190.328505 ms
$ OMP_NUM_THREADS=4 ./quicksort_omp_2.o
time: 69.644457 ms
$ OMP_NUM_THREADS=8 ./quicksort_omp_2.o
time: 49.826447 ms
```

Fibonacci

fib.c

```
int fib(int n){
  int i, j;
  if (n<2)
      return n;
  else
      {
      #pragma omp task shared(i)
firstprivate(n)
      i=fib(n-1);

      #pragma omp task shared(j)
firstprivate(n)
      j=fib(n-2);

      #pragma omp taskwait
      return i+j;
      }
}
```

Exercise - paths

Parallelizer “paths.c” avec task.

```
#include <stdio.h>

long count_path(int col, int row){
  if(col==1 || row==1) return 1;
  return count_path(col-1, row) + count_path(col, row-1);
}

void main(){
  int col=18;
  int row=18;

  printf("Paths(%i, %i) = %ld\n", col, row, count_path(col, row));
}
```

Auto Parallélisation

Structures:

- `icc -parallel`

Auto-parallélisation

Avec OpenMP, nous n'avons qu'à indiquer quelles sont les boucles à paralléliser.

pi1.c

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

int main(void) {
int i, N=1000000;
double sum = 0.;

  for(i=0; i<N; i+=2){
     sum += 4./(2.*i+1.);
     sum -= 4./(2.*i+3.);
  }
  printf("pi = %f\n",sum);
}
```

pi4.c

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

int main(void) {
int i, N=1000000000;
double sum = 0.;
#pragma omp parallel for reduction(+:sum)
  for(i=0; i<N; i+=2){
     sum += 4./(2.*i+1.);
     sum -= 4./(2.*i+3.);
  }
  printf("pi = %f\n",sum);
}
```

Auto-parallélisation

Avec OpenMP, nous n'avons qu'à indiquer quelles sont les boucles à paralléliser.
Le compilateur d'intel inclut l'option de détecter et paralléliser automatiquement les boucles.
icc -parallel -qopt-report=3 -guide-par

pi1.c

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

int main(void) {
int i, N=1000000;
double sum = 0.;

  for(i=0; i<N; i+=2){
     sum += 4./(2.*i+1.);
     sum -= 4./(2.*i+3.);
  }
  printf("pi = %f\n",sum);
}
```

```
$ icc -parallel pi1.c -o pi1.auto
$ time ./pi1.auto
pi = 3.141593

real    0m0.277s
user    0m5.702s
sys     0m0.113s

$ time OMP_NUM_THREADS=2 ./pi1.auto
pi = 3.141593

real    0m2.291s
user    0m4.533s
sys     0m0.016s
```

Extra

- *Petite fraction d'openmp: simd, XeonPhi, … etc.*
- *http://www.nersc.gov/assets/omp-common-core-iwomp2.pdf*
- *http://www.openmp.org/resources/tutorials-articles/*
- *http://www.archer.ac.uk/training/course-material/2018/03/openmp-soton/TipsTricksGotchas.pdf*

Suite

Sondage

Cette après midi:
Exercice

- *https://github.com/calculquebec/cq-formation-convolution*
- *https://github.com/calculquebec/cq-formation-nbody*