Cours SIM203

Initiation au calcul haute performance

Introduction au calcul parallèle à mémoire partagée — OpenMP

Axel Modave — 3 avril 2024

Introduction au calcul parallèle et à OpenMP

OpenMP – Threads et régions parallèles

OpenMP – Gestion des variables

OpenMP – Partage du travail

Architectures de calcul parallèle (RAPPEL)

Processeurs multi-cœurs
sur les machines de l'ENSTA

Machines avec **1 processeur** Intel Core i5-6500
Chaque processeur composé de **4 cœurs**
Fréquence d'horloge : **3.20 GHz**

Quelques cœurs
avec une haute fréquence …

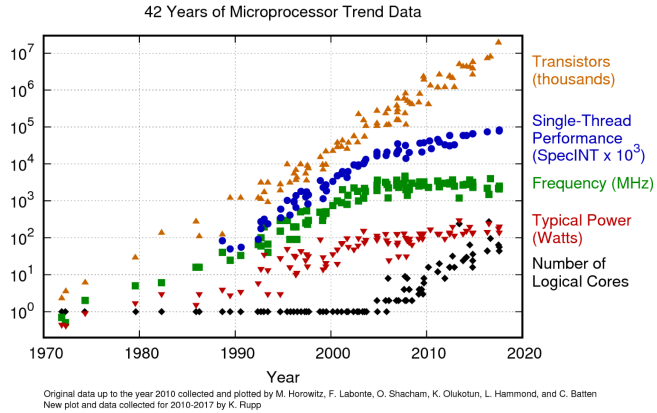
Supercalculateur LUMI-G (World #3)

https://docs.lumi-supercomputer.eu/hardware/compute/lumig/
https://www.amd.com/fr/products/server-accelerators/instinct-mi250x

Cluster avec **2560 CPU** / **10240 GPU**
Chaque proc. composé de **64** / **220 cœurs**
Fréquence d'horloge : **2** / **1.6 GHz**

Un très grand nombre de cœurs
avec une basse fréquence …

Évolution des processeurs



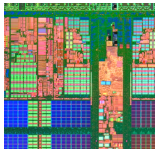
Jusqu'aux années ~2000, les performances ↗ grâce une ↗ de la fréquence.
À cause de limitations physiques *(miniaturisation vs. dissipation)*, il est difficile d' ↗ encore la fréquence.

Pour augmenter les performances, le nombre de cœurs est ↗ .
Pour des raisons économiques, la fréquence est parfois ↘ . } **Tendance**

https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

Rappel : Vision globale de calcul haute performance ...

Cœur



Calcul séquentiel
Calcul vectoriel

Séances 1 et 2

Processeur ***(plusieurs cœurs)***

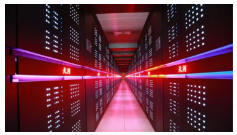
Calcul parallèle
à ***mémoire partagée***

Les cœurs peuvent travailler en parallèle sur des tâches différentes.

Ils partagent des mémoires rapides (RAM + L2 ou L3)

Séances 3 et 4

Cluster ***(plusieurs processeurs)***



Calcul parallèle
à ***mémoire distribuée***

Les processeurs peuvent travailler en parallèle sur des tâches différentes

Les données sont distribuées entre les RAM des processeurs, qui doivent communiquer (avec par ex. **MPI**).

Séance 5 *(intro)*

Cours AMS301 et AMS-I03
(3A ModSim et M2 AMS)

Questions importantes:
Comment **gérer les opérations** ?
(Quel cœur fait quoi ? Quand ?)
Comment **gérer les données** nécessaires pour les opérations ?

OpenMP *« Open Multi-Processing »*

OpenMP
Enabling HPC since 1997

- API *(Application Programming Interface)* standard pour la programmation d'applications parallèles sur **architectures à mémoire partagée**
- Le 28 octobre 1997, une majorité importante d'industriels et de constructeurs ont adopté OpenMP comme un standard industriel.
- Les spécifications OpenMP appartiennent aujourd'hui au **consortium industriel** ARB *(Architecture Review Board)*, seul organisme chargé de son évolution.

	Version	Spécifications	Commentaire
1997	OpenMP 1	45 pages	Pour processeurs multi-cœurs
2000	OpenMP 2	50 pages	
2008	OpenMP 3	152 pages	Ajout concept de tâche
2013	OpenMP 4	248 pages	Ajout support des accélérateurs (ex. GPU)
2018	OpenMP 5	669 pages	Amélioration accélérateurs, gestion des mémoires, déboggage, analyse de performance, portabilité, …

http://www.openmp.org/specifications/

OpenMP, en pratique ...

1. **Activer OpenMP à la compilation**

```
>> icpc -qopenmp code.cpp
```

2. **Directives de compilation #pragma**
   - Elles donnent des indications au compilateur pour organiser le travail et définir le statut des données.
   - Elles sont ignorées par le compilateur si OpenMP n'est pas activé.

```
#pragma omp parallel for
  for (int i=0; i<n; i++)
    a[i] = b[i] + c[i];
```

```
#pragma omp   parallel   for
```

Sentinelle *Nom* *Clause*

Sentinelle spécifique pour OpenMP

Optionnel — Il peut y en avoir aucun ou plusieurs.

OpenMP, en pratique ...

1. **Activer OpenMP à la compilation**

```
>> icpc -qopenmp code.cpp
```

2. **Directives de compilation #pragma**
   - Elles donnent des indications au compilateur pour organiser le travail et définir le statut des données.
   - Elles sont ignorées par le compilateur si OpenMP n'est pas activé.

```
#pragma omp parallel for
  for (int i=0; i<n; i++)
    a[i] = b[i] + c[i];
```

3. **Fonctions spécifiques**
   - Il faut charger les entêtes de la librairie
   - Elles permettent des actions pendant l'exécution ou donnent des informations.

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

```
void omp_set_num_threads(int n);
int omp_get_num_threads();
```

4. **Variables d'environnement** *(du système d'exploitation)*
   - Une fois positionnées, leurs valeurs sont prises en compte à l'exécution.

```
>> export OMP_NUM_THREADS=2
>> ./a.out
```

Introduction au calcul parallèle et à OpenMP

OpenMP – Threads et régions parallèles

OpenMP – Gestion des variables

OpenMP – Partage du travail

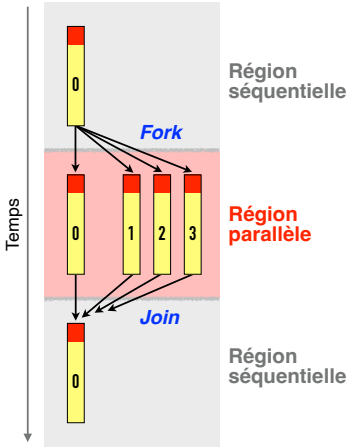
Threads et régions parallèles : Modèle de programmation

Un programme est une alternance de régions séquentielles et de régions parallèles.

- À son lancement, un programme possède un thread unique ***(région séquentielle)***.
- OpenMP permet de définir des parties de code à exécuter en parallèle ***(régions parallèles)***.

La parallélisation repose sur l'utilisation de processus légers, les "threads".

- Chaque thread exécute une tâche composée d'un ensemble d'instructions.
- Au sein d'une région parallèle, les threads sont exécutés de façon concurrente, c'est-à-dire indépendamment les uns des autres.
- Au début d'une région parallèle, le thread maître crée de nouveaux threads esclaves ***(Fork)***, qui disparaissent en fin de région parallèle ***(Join)***.



0 – Thread **maître**
1,2,3 – Threads **esclaves**

Threads et régions parallèles : Lors de l'exécution ...

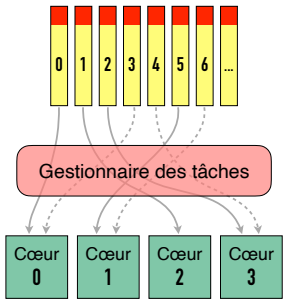
Les threads sont affectées aux cœurs par le gestionnaire des tâches du système d'exploitation.

Différents cas peuvent se présenter :

- Au mieux, à chaque instant, il existe un thread par cœur avec autant de threads que de cœurs pendant toute la durée du travail.
- Au pire, tous les threads sont traités de façon séquentielle par un et un seul cœur.
- En réalité, la situation est en général intermédiaire.

En pratique :

- Les threads d'une région parallèle ne sont donc pas forcément exécutés en même temps, mais ils sont **exécutés de façon concurrente** dans une période de temps limitée.
- La région parallèle se termine lorsque tous les threads ont été traités (**synchronisation implicite**).



Exemple : Hello World! (1)

```
#include <iostream>
int main(){
  int N;
  cout << "How many threads? ";
  cin >> N;
#pragma omp parallel num_threads(N)
  cout << "Hello world!\n";
  return 0;
}
```

```
>> icpc -qopenmp helloworld1.cpp
```

```
>> ./a.out
How many threads? 3
Hello world!
Hello world!
Hello world!
```

Directive qui indique que l'instruction qui suit sera exécuté en parallèle
*(Nombre de threads défini avec une **clause**)*

Option de compilation qui indique que OpenMP doit être activé.

Erreur courante :
Si vous oubliez `-qopenmp`, le programme va compiler, mais il sera séquentiel !

Exemple : Hello World! (2)

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

int main(){

  int N;
  cout << "How many threads? ";
  cin >> N;
  omp_set_num_threads(N);

#pragma omp parallel
  cout << "Hello world!\n";

  return 0;
}
```

```
>> icpc -qopenmp helloworld2.cpp
```

```
>> ./a.out
How many threads? 3
Hello world!
Hello world!
Hello world!
```

Entêtes de la librairie OpenMP

Fonction pour définir le nombre de threads

Directive qui indique que l'instruction qui suit sera exécutée en parallèle par le nombre de threads par défaut

Option de compilation qui indique que OpenMP doit être activé.

Erreur courante :
Si vous oubliez `#include <omp.h>`, il peut y avoir un problème à la compilation.

Exemple : Hello World! (3)

```
#include <iostream>

int main(){

#pragma omp parallel
  cout << "Hello world!\n";

  return 0;
}
```

```
>> icpc -qopenmp helloworld3.cpp
```

```
>> export OMP_NUM_THREADS=2
>> ./a.out
Hello world!
Hello world!
>> export OMP_NUM_THREADS=3
>> ./a.out
Hello world!
Hello world!
Hello world!
```

Directive qui indique que l'instruction qui suit sera exécuté en parallèle par le nombre de threads par défaut

Option de compilation qui indique que OpenMP doit être activé

Variable d'environnement qui défini le nombre de threads par défaut

Lignes de commande à tester :
```
>> export BLABLA=2
>> echo $BLABLA
```


Exemple : Synchronisation (1)

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

int main(){

  cout << "Let's start!\n";

#pragma omp parallel
  {
    int myRank     = omp_get_thread_num();
    int numThreads = omp_get_num_threads();
    cout << "Hello world! (" << myRank << "/" << numThreads << "\n";
  }

  int myRank     = omp_get_thread_num();
  int numThreads = omp_get_num_threads();
  cout << "I'm done! (" << myRank << "/" << numThreads << ")\n";

  return 0;
}
```

```
>> icpc -qopenmp helloworld4.cpp
```

```
>> export OMP_NUM_THREADS=4
>> ./a.out
Let's start!
Hello world! (2/4)
Hello world! (1/4)
Hello world! (0/4)
Hello world! (3/4)
I'm done! (0/1)
```

Région séquentielle

Région parallèle

Synchronisation implicite

Région séquentielle

Les threads travaillent de façon concurrente *(pas exactement en même temps)* dans la région parallèle.
Il n'y a pas de garantie sur l'ordre global d'exécution des instructions dans la région parallèle.

Exemple : Synchronisation (2)

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

void myPrint(string message){
  int myRank     = omp_get_thread_num();
  int numThreads = omp_get_num_threads();
  cout << message << " (" << myRank << "/" << numThreads << ")\n";
}

int main(){

  cout << "Let's start!\n";

#pragma omp parallel
  myPrint("Hello world!");

  myPrint("I'm done!");

  return 0;
}
```

Région séquentielle

Région parallèle

Région séquentielle

```
>> icpc -qopenmp helloworld5.cpp
```

```
>> export OMP_NUM_THREADS=4
>> ./a.out
Let's start!
Hello world! (1/4)
Hello world! (3/4)
Hello world! (2/4)
Hello world! (0/4)
I'm done! (0/1)
```

La fonction `myPrint()` est appelée plusieurs fois en parallèle dans la région parallèle, et une fois en séquentiel dans la deuxième région séquentielle.

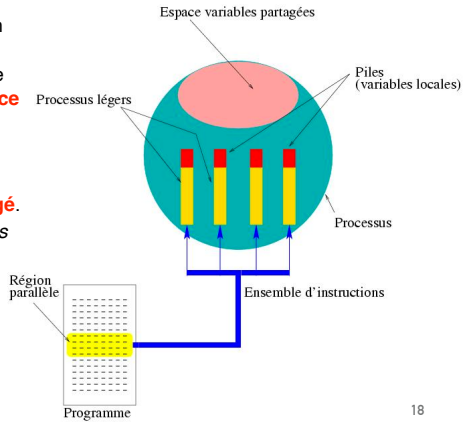
Introduction au calcul parallèle et à OpenMP
OpenMP - Threads et régions parallèles
OpenMP - Gestion des variables
OpenMP - Partage du travail

Gestion des variables

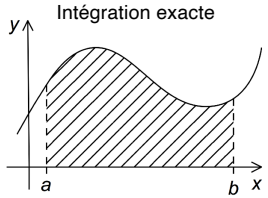
Pendant une région parallèle, les threads peuvent allouer/désallouer de la mémoire, et ils peuvent lire/écrire des variables et des tableaux de variables.

Deux statuts de variable sont possibles :

- **Variable privée**
  Chaque thread possède sa propre version de la variable.
  Un seul nom de variable est utilisé dans le code, mais chaque thread alloue un **espace mémoire privé** dans sa pile *(stack)*.
- **Variable partagée**
  Les threads accèdent à la même variable, stockée dans un **espace mémoire partagé**.
  Attention : Les threads ne sont pas censés la modifier en même temps!



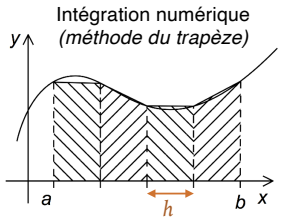
Exemple : Méthode du trapèze (1)



$$I = \int_a^b f(x)\,dx$$

$$I = h\left(\frac{f(a)}{2} + \sum_{i=1}^{N-1} f(a+ih) + \frac{f(b)}{2}\right)$$

avec $h = (b-a)/N$



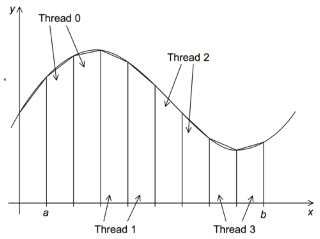
N trapèzes

```
double trapeze(double a, double b, int N){
  double h = (b-a)/N;
  double approx = (f(a) + f(b))*0.5;
  for(int i=1; i<=N-1; i++){
    double x_i = a + i*h;
    approx += f(x_i);
  }
  approx *= h;
  return approx;
}
```

```
double f(double x){
  return sin(x);
}
```


Exemple : Méthode du trapèze (2)

```
int main(){
  double a = 0.;
  double b = M_PI;
  int    N = 1000;
#pragma omp parallel
  {
    int myRank     = omp_get_thread_num();
    int numThreads = omp_get_num_threads();
    double my_a = a + (b-a)*myRank/numThreads;
    double my_b = a + (b-a)*(myRank+1)/numThreads;
    int    my_N = N/numThreads;
    double approx = trapeze(my_a,my_b,my_N);
    cout << "Result: " << approx << "\n";
  }
  return 0;
}
```




```
>> icpc -qopenmp trapeze2.cpp
```

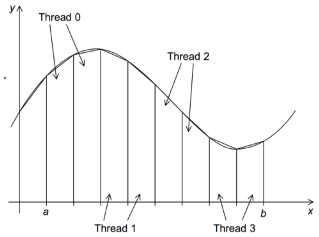
```
>> export OMP_NUM_THREADS=1
>> ./a.out
Result: 2
>> export OMP_NUM_THREADS=2
>> ./a.out
Result: 1
Result: 0.999999
>> export OMP_NUM_THREADS=4
>> ./a.out
Result: 0.292893
Result: 0.292893
Result: 0.707106
Result: 0.707106
```

Chaque thread s'occupe d'une partie de l'intégrale numérique.

Par **défaut**, les **variables déclarées avant la région parallèle** seront **partagées** par les threads. *(en rouge)*
Par **défaut**, les **variables déclarées dans la région parallèle** seront **privées**. *(en bleu)*

Exemple : Méthode du trapèze (3)

```
int main(){
  double a = 0.;
  double b = M_PI;
  int    N = 1000;

  double my_a = 0;
  double my_b = 0;
  int    my_N = 0;
  double approx = 0;

#pragma omp parallel shared(a,b,N,cout) \
    private(my_a,my_b,my_N,approx)
  {
    int myRank     = omp_get_thread_num();
    int numThreads = omp_get_num_threads();

    my_a = a + (b-a)*myRank/numThreads;
    my_b = a + (b-a)*(myRank+1)/numThreads;
    my_N = N/numThreads;
    approx = trapeze(my_a,my_b,my_N);

    cout << "Result: " << approx << "\n";
  }

  return 0;
}
```



Le comportement par défaut est modifié en utilisant des **clauses supplémentaires**.

Les variables dans **shared**(…) sont partagées dans la région parallèle.

Les variables dans **private**(…) sont privées dans la région parallèle.

- Même si les variables "originales" sont initialisées avant la région, la version privée des variables ne sont pas initialisées.
- Les valeurs privées sont perdues à la fin de la région parallèle.
- Les variables *"originales"* reprennent leur valeur antérieure après la r. p.

Clauses pour variables déclarées avant la région parallèle

`#pragma omp parallel …`

shared(*liste de variables***)**
Chaque variable de la liste est **partagée**.

private(*liste de variables***)**
Chaque variable de la liste est **privée**. À l'entrée d'une région parallèle, un espace mémoire privé est alloué, mais la variable n'est **pas initialisée**.

firstprivate(*liste de variables***)**
Chaque variable de la liste est **privée**. À l'entrée d'une région parallèle, un espace mémoire privé est alloué et **initialisé** avec la valeur de la variable pré-existante.

default(none)
Annule tous les comportement par défaut. Toutes les variables utilisées dans les threads doivent être listées dans `shared(…)`, `private(…)` ou `firstprivate(…)`.
(Cette clause force le programmeur à expliciter tous les usages. C'est simplement une sécurité!)

Exemple d'utilisation dans `variables1.cpp`.

Exemple : Méthode du trapèze (4)

Comment **additionner le résultat** des **différents threads** *(les intégrales partielles)* pour obtenir le **résultat total** *(l'intégrale totale)* ?

```
double approx = 0;
#pragma omp parallel
{
  …
  approx += trapeze(my_a,my_b,my_N);
}
cout << "Result: " << approx << "\n";
```

Idée : Les threads peuvent accumuler **(+=)** les résultats partial dans une variable partagée.

Exemple : Méthode du trapèze (4)

Comment **additionner le résultat** des **différents threads** *(les intégrales partielles)* pour obtenir le **résultat total** *(l'intégrale totale)* ?

```
double approx = 0;
#pragma omp parallel
{
  …
  approx += trapeze(my_a,my_b,my_N);
}
cout << "Result: " << approx << "\n";
```

Idée : Les threads peuvent accumuler **(+=)** les résultats partial dans une variable partagée.

```
>> icpc -qopenmp trapeze4.cpp
```

```
>> export OMP_NUM_THREADS=100
>> ./a.out
Result: 1.999505
>> ./a.out
Result: 2
>> ./a.out
Result: 2
>> ./a.out
Result: 1.964347
```

Mauvaise solution !

Les résultats sont inconsistants.
Les threads modifient la même variable en même temps … et il y a des accidents.

(***data race*** — *dans un prochain cours!)*

Directive CRITICAL

- La directive `#pragma omp critical` assure que la section de code qui suit est exécutée par tous les threads de façon séquentielle *(mais dans n'importe quel ordre)*.
- Elle permet de travailler sur des variables partagées en évitant le *data race*.
- Attention : Le code devient en partie séquentiel, avec possible perte de performance.

```
double approx = 0;
#pragma omp parallel
{
  …
  double approxLoc = trapeze(my_a,my_b,my_N);
#pragma omp critical
  {
    approx += approxLoc;
  }
}
cout << "Result: " << approx << "\n";
```

```
>> icpc -qopenmp trapeze5.cpp
```

```
>> export OMP_NUM_THREADS=100
>> ./a.out
Result: 2
>> ./a.out
Result: 2
>> ./a.out
Result: 2
>> ./a.out
Result: 2
```


Clause reduction

Comment **additionner le résultat** des **différents threads** *(les intégrales partielles)* pour obtenir le **résultat total** *(l'intégrale totale)* ?

```
double approx = 0;
#pragma omp parallel reduction(+:approx)
{
  …
  approx += trapeze(my_a,my_b,my_N);
}
cout << "Result: " << approx << "\n";
```

```
>> icpc -qopenmp trapeze6.cpp
```

```
>> export OMP_NUM_THREADS=100
>> ./a.out
Result: 2
>> ./a.out
Result: 2
>> ./a.out
Result: 2
>> ./a.out
Result: 2
```

Avec la **clause reduction** :

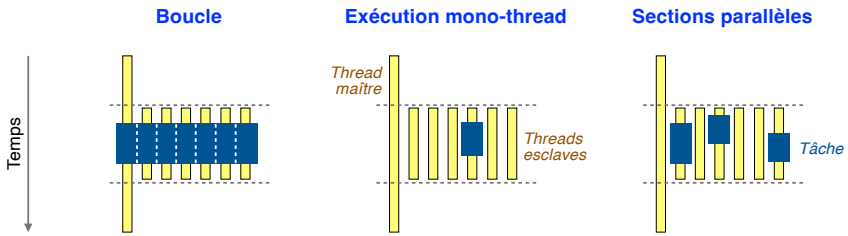
- La variable **approx** est privée dans la région parallèle.
- Les valeurs finales **approx** de tous les threads sont additionnés à la fin de la région.
- Après la fin de la région, le résultat final est contenu dans **approx**.

`reduction(`*opération* `:` *variable*`)`

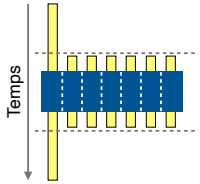
Différentes opérations possibles :
+ – * / min max

Quelques directives pour partager le travail

- Directives qui répartissent le travail entre les différents threads d'une région parallèle
- Elles sont **incluses dans une région parallèle**
- Elles sont rencontrées par tous les threads *(ou aucun)*



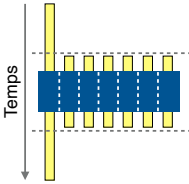
Partage du travail : Boucles




```
double a[N], b[N], c[N];
#pragma omp parallel
  {
    int myRank     = omp_get_thread_num();
    int numThreads = omp_get_num_threads();
    int my_start = N*myRank/numThreads;
    int my_end   = N*(myRank+1)/numThreads;
    for(int i=my_start; i<my_end; i++)
      c[i] = a[i] + b[i];
  }
```

Exemple : Addition de deux vecteurs

Partage du travail : Boucles



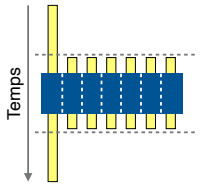

```
double a[N], b[N], c[N];
#pragma omp parallel
  {
    int myRank     = omp_get_thread_num();
    int numThreads = omp_get_num_threads();
    int my_start = N*myRank/numThreads;
    int my_end   = N*(myRank+1)/numThreads;
    for(int i=my_start; i<my_end; i++)
      c[i] = a[i] + b[i];
  }
```

Exemple : Addition de deux vecteurs

```
double a[N], b[N], c[N];
#pragma omp parallel
  {
#pragma omp for
    for(int i=0; i<N; i++)
      c[i] = a[i] + b[i];
  }
```

- La directive **`for`** partage les itérations d'une boucle parallèle.
- Les itérations sont attribuées automatiquement aux différents threads.
  Il n'est pas nécessaire de répartir "à la main" les itérations.
- Des clauses supplémentaires permettent de guider la répartition. *(dans un prochain cours)*

Partage du travail : Boucles




```
double a[N], b[N], c[N];
#pragma omp parallel
  {
    int myRank     = omp_get_thread_num();
    int numThreads = omp_get_num_threads();
    int my_start = N*myRank/numThreads;
    int my_end   = N*(myRank+1)/numThreads;
    for(int i=my_start; i<my_end; i++)
      c[i] = a[i] + b[i];
  }
```

Exemple : Addition de deux vecteurs

```
double a[N], b[N], c[N];
#pragma omp parallel
  {
#pragma omp for
    for(int i=0; i<N; i++)
      c[i] = a[i] + b[i];
  }
```

```
double a[N], b[N], c[N];
#pragma omp parallel for
    for(int i=0; i<N; i++)
      c[i] = a[i] + b[i];
```

- Lorsque la région parallèle ne contient que la boucle, le code peut être simplifié !
- Pour paralléliser une boucle, une seule commande suffit !

Exemple : Méthode du trapèze (version finale)

```
int main(){
  double a = 0.;
  double b = M_PI;
  int    N = 1000;
  double h = (b-a)/N;
  double approx = (f(a) + f(b))*0.5;
#pragma omp parallel for reduction(+:approx)
  for(int i=1; i<=N-1; i++){
    double x_i = a + i*h;
    approx += f(x_i);
  }
  approx *= h;
  cout << "Result: " << approx << "\n";
  return 0;
}
```

```
>> icpc -qopenmp trapeze8.cpp
```

```
>> export OMP_NUM_THREADS=4
>> ./a.out
Final: 2
```

Par rapport au code séquentiel, une seule ligne de code permet de paralléliser le travail !

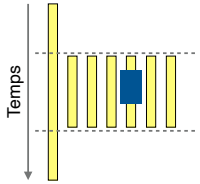
… mais elle signifie beaucoup de choses !

`parallel`
→ Région parallèle

`for`
→ Parallélisation de la boucle

`reduction(+:approx)`
→ Addition des valeurs locales de **`approx`** en fin de région parallèle

Partage du travail : Exécution mono-thread



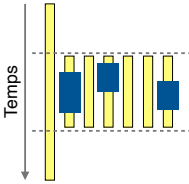

```
int main(){
#pragma omp parallel
  {
    double a = 92290.;
#pragma omp single
    a = -92290.;
    int myRank = omp_get_thread_num();
    cout << "Rank " << myRank << ": " << a << "\n";
  }
  return 0;
}
```

- La directive **`single`** permet de faire exécuter une portion de code par un et un seul thread, sans pouvoir indiquer lequel.
- Tous les threads sont synchronisés à la fin de la *"section single"*.

```
>> icpc -qopenmp single.cpp
```

```
>> export OMP_NUM_THREADS=2
>> ./a.out
Rank 0: -92290.000000
Rank 1: 92290.000000
>> ./a.out
Rank 1: -92290.000000
Rank 0: 92290.000000
```


Partage du travail : Sections parallèles




```
int main(){
#pragma omp parallel
  {
    int n = omp_get_thread_num();
    cout << n <<" (I am alive!)\n";
#pragma omp sections
    {
#pragma omp section
      cout << n << "(I am the first!)\n";
#pragma omp section
      {
        cout << n << " (I am the second!)\n";
      }
    }
  }
  return 0;
}
```

- Une **`section`** est une portion de code exécutée par un et un seul thread.
- Plusieurs sections peuvent être définies au sein d'une construction **`sections`**.
- Le but est de pouvoir exécuter plusieurs portions de code de façon concurrente, avec *a priori* plusieurs threads.

```
>> icpc -qopenmp sections.cpp
```

```
>> export OMP_NUM_THREADS=3
>> ./a.out
0 (I am alive!)
1 (I am alive!)
0 (I am the first!)
2 (I am alive!)
1 (I am the second!)
>> ./a.out
2 (I am alive!)
0 (I am alive!)
0 (I am the first!)
1 (I am alive!)
1 (I am the second!)
```


Résumé des commandes

Directives de compilation

Sentinelle	*Nom*	*Clauses*	
`#pragma omp`	`parallel`	`num_threads(…)` `default(none)` `private(…)` `firstprivate(…)` `shared(…)` `reduction(…:…)`	**Région parallèle (RP)** *+ Option pour définir le nombre de thread* *+ Options pour caractériser les variables* *+ Opération de réduction*
`#pragma omp`	`critical`		**Zone séquentielle dans une RP**
`#pragma omp`	`for`		**Boucle parallèle dans une RP**
`#pragma omp`	`parallel for`		**Région parallèle + Boucle parallèle**
`#pragma omp`	`single`		**Exécution sur un thread dans une RP**
`#pragma omp`	`sections` `section`		**Sections parallèles dans une RP** **Section dans un groupe de sections parallèles**

Librairie et fonctions

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

```
void omp_set_num_threads(int n);
int omp_get_num_threads();
int omp_get_thread_num();
```

Compilation et variables d'environ.

```
>> icpc -qopenmp myCode.cpp
```

```
>> export OMP_NUM_THREADS=2
>> ./a.out
```

Ressources

Documentation officiel

- Site Internet d'OpenMP
  http://www.openmp.org/specifications/

Cours en ligne

- Formation OpenMP de l'IDRIS
  http://www.idris.fr/formations/openmp/
- An Overview of OpenMP
  Ruud van der Pas (Sun Microsystems)
  http://www.openmp.org/wp-content/uploads/ntu-vanderpas.pdf
- Cours d'introduction à OpenMP
  Cédric Bastoul (U. de Strasbourg)
  http://icps.u-strasbg.fr/people/bastoul/public_html/teaching/openmp/bastoul_cours_openmp.pdf

Livres

- *An Introduction to Parallel Programming*
  P. Pachenco
- *Using OpenMP*
  B. Chapman, G. Jost, R. Van Der Pas