ARCHITECTURE EVOLUEE DES ORDINATEURS

Architectures multicœurs

AMINA SELMA HAICHOUR

2021-2022

Architectures multicœurs

- Partie 1 : Architecture
- **Partie 2 : Programmation parallèle**

Architectures multicœurs

- **Partie II : Programmation parallèle**
  - Comment partager le code sur plusieurs cœurs
  - **OpenMP**

Objectifs du chapitre

- Lister brièvement le contenu d'OpenMP
- Analyser un code qui contient des éléments d'OpenMP afin de décrire le parallélisme obtenu
- Paralléliser un algorithme séquentiel en identifiant ses parties parallèles
- Utiliser effectivement les **directives**, les **fonctions** et les **variables d'environnement** d'OpenMP afin d'exécuter un code sur plusieurs cœurs

Contenu du chapitre

- OpenMP
  - Directives de compilation
  - Fonctions
  - Variables d'environnement
- Synthèse et conclusion

OpenMP

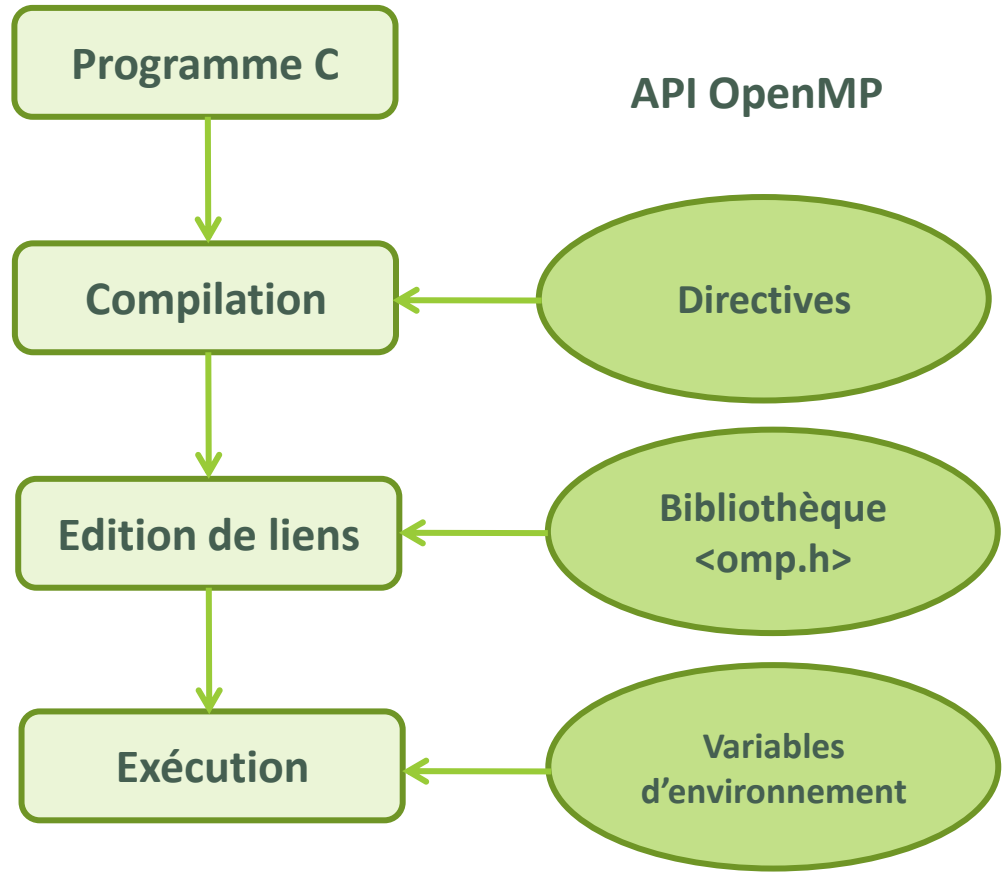
- OpenMP : **Open** specification for **M**ulti-**P**rocessing
- OpenMP est une API (Interface de programmation d'application)
- Standard pour la programmation d'applications parallèles sur des architectures à mémoire partagée (les processeurs multicœurs)
- La programmation parallèle repose sur l'utilisation de threads (programmation multithread)
- Langages C, C++ et FORTRAN
- Spécification complète : http:/ /www.openmp.org

OpenMP

- Installation : **sudo apt-get install build-essential libgomp1**
- Compilation : **gcc -fopenmp test.c -o test**
- Utilisation : **#include <omp.h>**
- Composée de :
  - Directives de compilation
  - Fonctions
  - Variables d'environnement

OpenMP

- Directives de compilation
  - Partage de travail
  - Synchronisation
  - Définition du statut privé ou partagé des données
- Fonctions
  - Font partie de la bibliothèque <omp.h> chargée à l'édition des liens
- Variables d'environnement
  - Une fois positionnées, leurs valeurs sont prise en compte à l'exécution



OpenMP

- Programme OpenMP-C (exemple)

```
#include <omp.h>

void main ()
{

   #pragma omp parallel
   {

    int ID = omp_get_thread_num ();
    printf("Hello(%d)", ID);
    printf("world(%d)", ID);

   }

}
```

Directive OpenMP

Fonction OpenMP

OpenMP

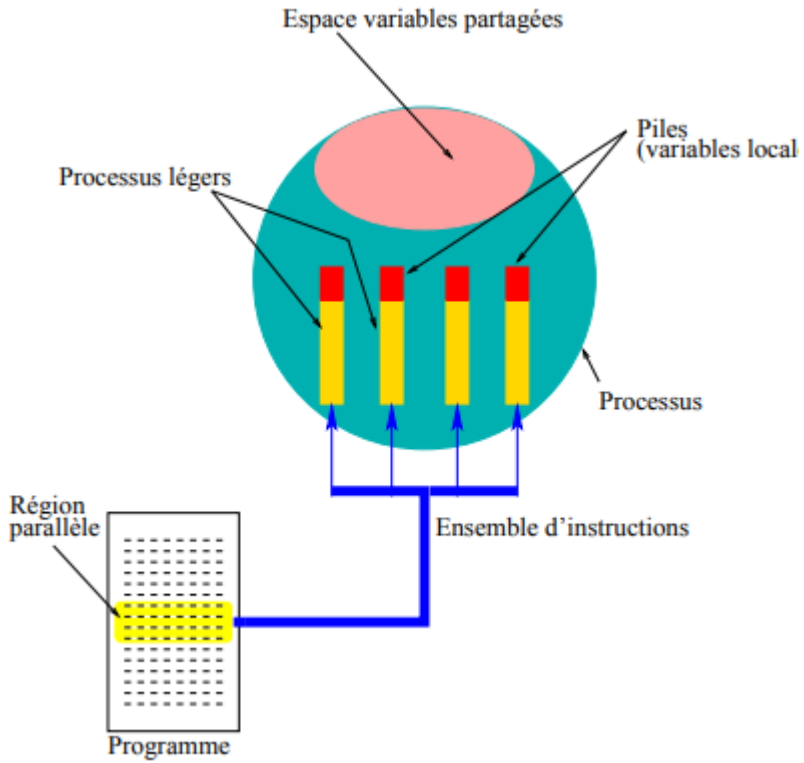
- Programme OpenMP-C (exemple)

```
#include <omp.h>

void main ()
{

   #pragma omp parallel
   {

    int ID = omp_get_thread_num ();
    printf("Hello(%d)", ID);
    printf("world(%d)", ID);

   }

}
```

Région parallèle
Code exécuté par chaque thread

→ Création de threads (processus légers)

→ Destruction de threads



OpenMP

- Programme OpenMP-C (exemple)

```
#include <omp.h>

void main ()
{

   #pragma omp parallel
   {

    int ID = omp_get_thread_num ();
    printf("Hello(%d)", ID);
    printf("world(%d)", ID);

   }

}
```

Région parallèle
Code exécuté par chaque thread

Ex. Création de deux threads

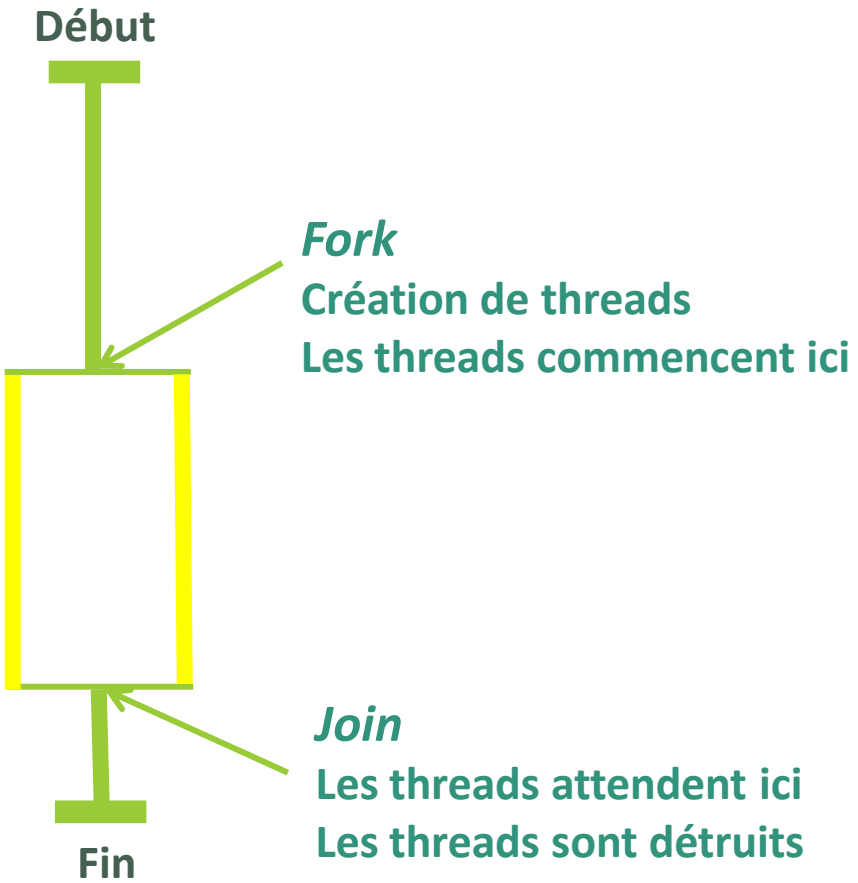
Hello(0)
Hello(1)
world(0)
world(1)

OpenMP

- Programme OpenMP-C (exemple)

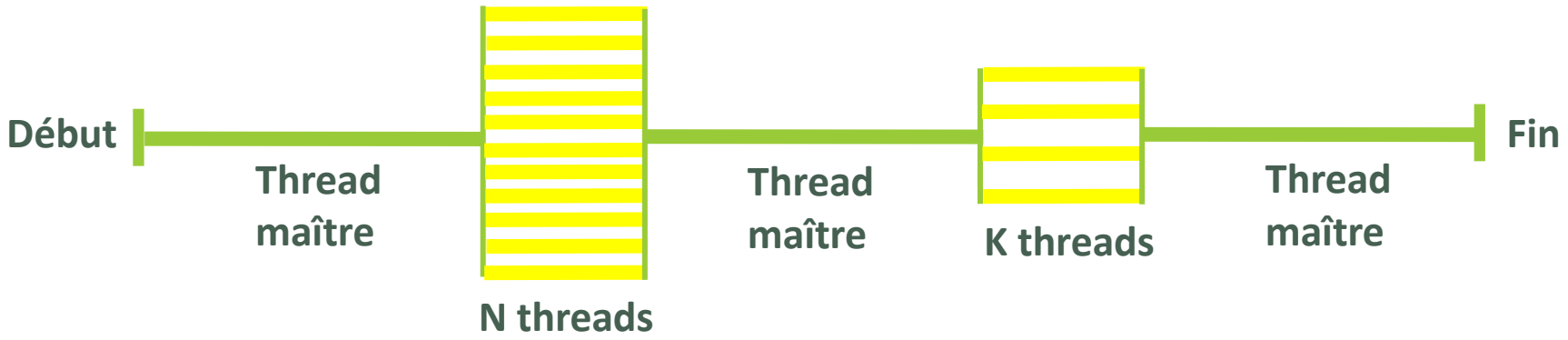
```
#include <omp.h>

void main ()
{

   #pragma omp parallel
   {

    int ID = omp_get_thread_num ();
    printf("Hello(%d)", ID);
    printf("world(%d)", ID);

   }

}
```

Région parallèle
Code exécuté par chaque thread



OpenMP

- Modèle d'exécution Fork/Join



OpenMP

- Modèle d'exécution Fork/Join
  - Un thread maître au démarrage
  - Apparition/disparition de threads esclaves au fil des régions parallèles
  - Chaque thread exécute sa propre séquence d'instructions
  - Chaque thread a sa propre mémoire locale avec ses variables dites privées
  - Chaque thread accède à l'ensemble de la mémoire partagée et donc des variables partagées (variables globales)
  - Il faut s'assurer que l'accès aux variables partagées est correct : besoin éventuel de **synchronisations**

OpenMP

- Directives de compilation
  - Création de régions parallèles
  - Partage du travail
  - Synchronisation

OpenMP

- Directives de compilation
  - Création de régions parallèles
  - Partage du travail
  - Synchronisation

OpenMP

- Directives de compilation
  - Création de régions parallèles
    - **#pragma omp parallel [clause, …] {}**
    - [clause] = {**if**, **num_threads**, **private**, **shared**, **firstprivate**}
    - Crée une région parallèle
    - **Fork** au début de la région et **Join** à sa fin
    - Les variables sont partagées par défaut dans la région parallèle

OpenMP

- Directives de compilation
  - Création de régions parallèles
    - **#pragma omp parallel [clause, …] {}**
    - Clause **if (expression logique)**
    - Clause **num_threads (expression arithmétique)**
    - Clause **private (liste de variables)**
    - Clause **shared (liste de variables)**
    - Clause **firstprivate (liste de variables)**

OpenMP

- Directives de compilation
  - Création de régions parallèles
    - **#pragma omp parallel if (n > 50000) {}**
    - Si la condition est fausse, la région parallèle n'est pas créée

OpenMP

- Directives de compilation
  - Création de régions parallèles
    - **#pragma omp parallel num_threads (10) {}**
    - Définit le nombre de threads pour la région parallèle (en l'occurrence ici **10**)

OpenMP

- Directives de compilation
  - Création de régions parallèles
    - **#pragma omp parallel private (v1, v2) {}**
    - Chaque thread possède sa propre liste des variables **v1** et **v2**
    - Les valeurs de **v1** et **v2** sont différentes pour chaque thread
    - En dehors de la région parallèle, les valeurs de **v1** et **v2** ne sont pas affectées par les threads

OpenMP

- Directives de compilation
  - Création de régions parallèles

```
int main() {
  float X [100000];
  int iam, nt, ipoints, istart, npoints=100000;
  #pragma omp parallel private (iam,nt,ipoints,istart)
   {
     iam=omp_get_thread_num ();
     nt=omp_get_num_threads ();
     ipoints=npoints/nt;
     istart=iam*ipoints;
     if (iam==nt-1)
        ipoints = npoints - istart;
     subdomain (x, istart, ipoints);
   }
}
```

#pragma omp parallel private () (exemple)

Chaque thread utilise des valeurs différentes pour **iam**, **nt**, **ipoint** et **istart**

x et **npoint** sont paratgées

OpenMP

- Directives de compilation
  - Création de régions parallèles
    - **#pragma omp parallel shared (N, M) {}**
    - Les variables **N** et **M** sont partagées entre tous les threads
    - C'est le mode par défaut dans **OpenMP**
    - Les variables peuvent être changées par les threads et leurs valeurs sont visibles à tous les threads

OpenMP

- Directives de compilation
  - Création de régions parallèles
    - **#pragma omp parallel firstprivate (V) {}**
    - La variable **V** est unique pour chaque thread
    - La valeur initiale de **V** est considérée à l'entrée dans la région parallèle
    - A l'intérieur des threads, la valeur de **V** peut changer

OpenMP

- Directives de compilation
  - Création de régions parallèles

#pragma omp parallel firstprivate () (exemple)

```
int main() {
   int count = 2;
   omp_set_num_threads (NT);
   #pragma omp parallel firstprivate (count)
   {
       count++;
       printf("count= %d\n", count);
   }
}
```

cout = 3

...

cout = 3

#pragma omp parallel private () (exemple)

```
int main() {
   int count = 2;
   omp_set_num_threads (NT);
   #pragma omp parallel private (count)
   {
       count++;
       printf("count= %d\n", count);
   }
}
```

cout = 1

...

cout = 1

OpenMP

- Directives de compilation
  - Création de régions parallèles
  - Partage du travail
  - Synchronisation

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp for**
    - **#pragma omp sections {}**
    - **#pragma omp master {}**
    - **#pragma omp single {}**

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp for [clause, …]**
    - Distribue les itérations d'une boucle sur les threads disponibles

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp for [clause, …]** (exemple)

Début de la région parallèle (création de threads)
Partage du travail entre les threads crées

Fin de la région parallèle (destruction de threads)

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

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

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp for [clause, …]**
    - Clause **schedule (type, chunk)**
    - Clause **private (liste de variables)**
    - Clause **firstprivate (liste de variables)**
    - Clause **lastprivate (liste de variables)**
    - Clause **reduction (opérateur:liste de variables)**
    - Clause **nowait**

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp for schedule (type, chunk)**
    - Définit la distribution des itérations sur les threads
    - Type = **static** – ordonnancement statique déterminé à la compilation
    - Les itérations sont divisées en blocs de **chunk** itérations consécutives
    - Les blocs sont assignés aux threads en **round-robin**
    - Si **chunk** n'est pas précisé, des blocs de tailles similaires sont créés, un par thread

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp for schedule (type, chunk)**
    - Définit la distribution des itérations sur les threads
    - Type = **static** – ordonnancement statique déterminé à la compilation

```
int main(){
  int A[MAX];
  #pragma omp parallel for schedule(static, 8)
    for (int i=0;i< MAX; i++)
    {
      A[i] = big(i);
    }
}
```

#pragma omp for schedule () (exemple)

Ex. 3 threads sont disponibles
MAX = 64
Le chunk = 8 (bloc de 8 itérations)
Le thread 0 reçoit 8*3 itérations
Le thread 1 reçoit 8*3 itérations
Le thread 2 reçoit 8*2 itérations

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp for schedule (type, chunk)**
    - Définit la distribution des itérations sur les threads
    - Type = **dynamic** – ordonnancement dynamique déterminé à l'exécution
    - Les itérations sont divisées en blocs de **chunk** itérations consécutives
    - Chaque thread demande un bloc de **chunk** itérations consécutives dès qu'il n'a pas de travail
    - Si **chunk** n'est pas précisé, il vaudra 1

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp for schedule (type, chunk)**
    - Définit la distribution des itérations sur les threads
    - Type = **dynamic** – ordonnancement dynamique déterminé à l'exécution

```
int main(){
  int A[MAX];
  #pragma omp parallel for schedule(dynamic, 8)
    for (int i=0;i< MAX; i++)
    {
      A[i] = big(i);
    }
}
```

#pragma omp for schedule () (exemple)

Ex. 2 threads sont disponibles

MAX = 24

Le chunk = 8 (bloc de 8 itérations)

Le thread 0 reçoit 8

Le thread 1 reçoit 8

Le thread 1 termine avant le thread 0 et redemande 8

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp for lastprivate (V)**
    - La variable **V** est unique pour chaque thread
    - A l'intérieur des threads, la valeur de **V** peut changer
    - La valeur de **V** de la dernière itération est considérée à la sortie de la région parallèle

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp for lastprivate (V)**

```
int main(){
  int i;
 #pragma omp parallel
 {
  #pragma omp for lastprivate (i)
      for (i=0; i<n-2; i++)
          a[i] = b[i] + b[i+1];
 }
 a[i+1]=b[i+1];
 for (i=0; i<n; i++)
      printf("%d\n", a[i]);
}
```

#pragma omp for lastprivate () (exemple)

Ex. n = 100
« i » en dehors de la région parallèle reçoit la valeur de la copie « i » du thread responsable de la dernière itération , donc i = 98
a[99] = b[99]

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp for reduction (opérateur:liste de variables)**
    - Définit l'opérateur à appliquer aux valeurs des variables à la sortie des threads
    - L'opérateur peut être : +, -, *, etc.

#pragma omp for reduction () (exemple)

```
int main()
{
  double ave=0.0, A[MAX];
  #pragma omp parallel for reduction(+:ave)
    for (int i=0;i< MAX; i++)
       ave + = A[i];
}
```

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp for nowait**
    - Le thread qui termine n'attend pas les autres

#pragma omp for (exemple)

```
int main()
{…
 #pragma omp parallel private (i)
 {
   #pragma omp for
      for (i=1; i<n; i++)
        b[i] = (a[i] + a[i-1]) / 2.0;
   #pragma omp for
      for (i=0; i<m; i++)
        y[i] = sqrt(z[i]);
 }
}
```

Synchronisation implicite des threads i.e
le thread qui termine attend tous les autres

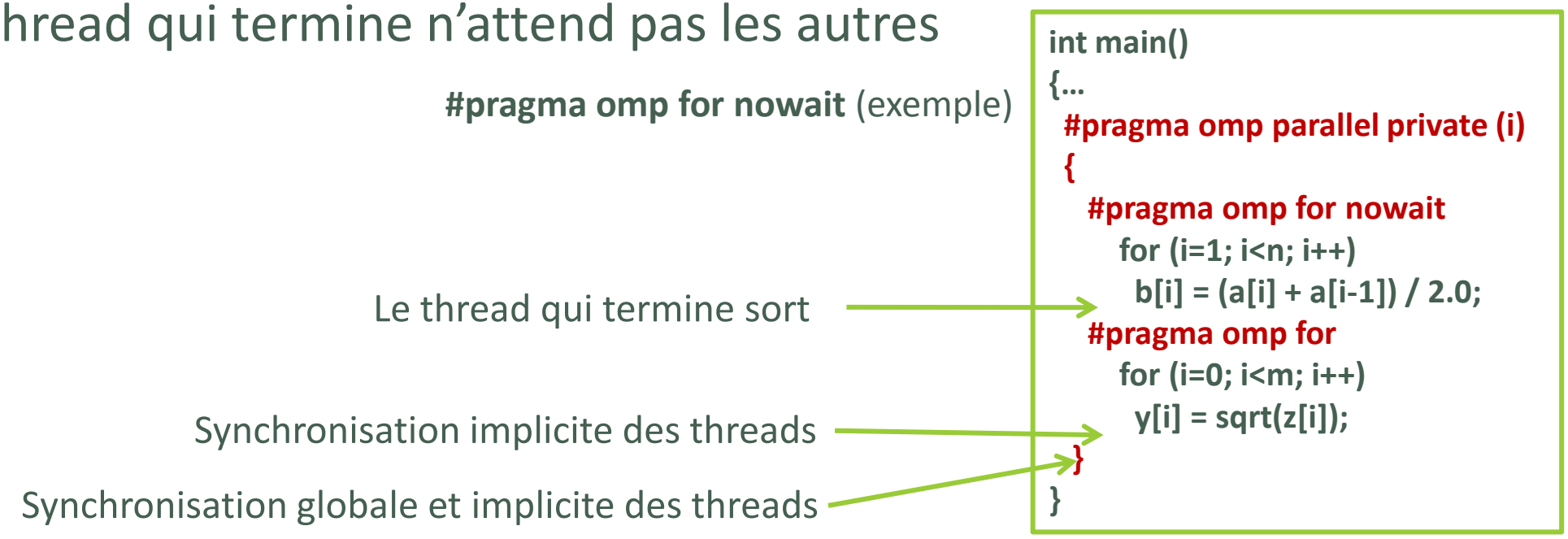
Synchronisation implicite des threads

Synchronisation globale et implicite des threads

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp for nowait**
    - Le thread qui termine n'attend pas les autres

#pragma omp for nowait (exemple)



OpenMP

- Directives de compilation
  - Partage du travail

```
#define N 1000
double Tab1[N], Tab2[N];
…
main() {
   int i;
   double alpha = 2.0;
   …
   #pragma omp parallel for private (i)
    for (i = 0; i < N; i++) {
      double coef;
      coef = pow(Tab1[i],alpha);
      Tab2[i] = coef*2;
    }
}
```

#pragma omp for (exemple)

Tab1, **Tab2**, **i** et **alpha** sont des variables partagées

Chaque thread a une copie de **i** (variable privée)

Chaque thread a une variable privée **coef**

OpenMP

- Directives de compilation
  - Partage du travail

```
#define N 1000
double Tab1[N], Tab2[N];
…
main() {
   double alpha = 2.0;
   …
   #pragma omp parallel for
    for (int i = 0; i < N; i++) {
      double coef;
      coef = pow(Tab1[i],alpha);
      Tab2[i] = coef*2;
    }
}
```

#pragma omp for (exemple)

Tab1, **Tab2** et **alpha** sont des variables partagées

Chaque thread a une variable privée **i**

Chaque thread a une variable privée **coef**

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp sections [clause, …] {}**
    - Divise le code en sections qui s'exécutent en parallèle
    - Clause **private (liste de variables)**
    - Clause **firstprivate (liste de variables)**
    - Clause **lastprivate (liste de variables)**
    - Clause **reduction (opérateur:liste de variables)**
    - Clause **nowait**

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp sections [clause, …] {}**

Partage du travail entre les threads crées

Un thread réalise X_calculation()

Un autre thread réalise Y_calculation()

```
int main(){
  #pragma omp parallel
    {
    #pragma omp sections
      {
       #pragma omp section
         X_calculation();
       #pragma omp section
         y_calculation();
      }
    }
}
```

#pragma omp sections {} (exemple)

```
int main(){
  #pragma omp parallel sections
    {
     #pragma omp section
     X_calculation();
     #pragma omp section
      y_calculation();
    }
}
```

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp sections [clause,…] {}**

#pragma omp sections {} (exemple)

```
#include <omp.h>
#define N 1000
main()
{
  int i; float a[N], b[N], c[N], d[N];
  for (i=0; i < N; i++){
      a[i] = i * 1.5;
      b[i] = i + 22.35;
  }
#pragma omp parallel shared(a,b,c,d) private(i)
{
  #pragma omp sections nowait
  {
    #pragma omp section
    for (i=0; i < N; i++) c[i] = a[i] + b[i];
    #pragma omp section
    for (i=0; i < N; i++) d[i] = a[i] * b[i];
  }
}
}
```

Partage du travail entre les threads créés

Le thread qui termine sort

Synchronisation globale et implicite des threads

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp master {}**
    - La région qui suit la directive est exécutée par le thread maître seulement

#pragma omp master {} (exemple)

Tous les threads créés y compris le thread maitre exécutent x_calculation et y_calculation

```
int main(){
 #pragma omp parallel
  {
    X_calculation();
    y_calculation();
    #pragma omp master
    {
      z_calculation();
    }
  }
}
```

Seul le thread maitre exécute z_calculation

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp single [clause, …] {}**
    - La région qui suit la directive est exécutée par un seul thread (n'importe lequel)
    - Clause **private (liste de variables)**
    - Clause **firstprivate (liste de variables)**
    - Clause **nowait**

OpenMP

- Directives de compilation
  - Partage du travail
    - **#pragma omp single [clause, …] {}**

#pragma omp single {} (exemple)

```
int main(){
 #pragma omp parallel
  {
    X_calculation();
    y_calculation();
    #pragma omp single
    {
      z_calculation();
    }
  }
}
```

OpenMP

- Directives de compilation
  - Création de régions parallèles
  - Partage du travail
  - Synchronisation

OpenMP

- Directives de compilation
  - Synchronisation
  - **#pragma omp barrier**
  - **#pragma omp critical {}**
  - **#pragma omp atomic {}**

OpenMP

- Directives de compilation
  - Synchronisation
  - **#pragma omp barrier**
  - Synchronisation entre tous les threads d'une équipe
  - Si un thread arrive à la directive ''**barrier**'', il attend que tous les autres threads y soient arrivés. Quand cela arrive, les threads poursuivent leur exécution en parallèle

OpenMP

- Directives de compilation
  - Synchronisation
  - **#pragma omp barrier**

#pragma omp barrier(exemple)

```
#include <omp.h>
void work1(int k) {
// large amount of work }
 void work2(int k) {
// large amount of work that must all happen after work1 is finished }
int main() {
  int n=1000000;
   #pragma omp parallel private(i) shared(n)
    {
     #pragma omp for
       for (i=0; i<n; i++)
         work1(i);
     #pragma omp barrier
     #pragma omp for
       for (i=0; i<n; i++)
       work2(i);
     }
  return 0;
}
```

Synchronisation explicite des threads
Tous les threads se rejoignent ici

Synchronisations implicites des threads
Tous les threads se rejoignent ici

OpenMP

- Directives de compilation
  - Synchronisation
  - **#pragma omp critical {}**
  - Spécifie que le bloc d'instructions suivant la directive doit être exécuté par un seul thread à la fois
  - Si un thread exécute un bloc protégé par la directive ''**critical**'' et qu'un second arrive à ce bloc, alors le second devra attendre que le premier termine avant de commencer l'exécution du bloc

OpenMP

- Directives de compilation
  - Synchronisation
  - **#pragma omp critical {}**

#pragma omp critical {} (exemple)

Un seul thread qui accède à la fois

```
...
int sum_shared = 0;
#pragma omp parallel
{
  int sum_local = 0;
  #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
      sum_local += i; }
  #pragma omp critical
  {
    sum_shared += sum_local;
  }
}
printf("%d \n", sum_shared);
...
```

OpenMP

- Directives de compilation
  - Synchronisation
  - **#pragma omp atomic {}**
  - Spécifie que l'instructions suivant la directive doit être exécutée par un seul thread à la fois
  - Si un thread exécute l'instruction protégée par la directive ''**critical**'' et qu'un second arrive à cette instruction, alors le second devra attendre que le premier termine avant de commencer l'exécution de l'instruction

OpenMP

- Directives de compilation
  - Synchronisation
  - **#pragma omp atomic {}**

#pragma omp atomic {} (exemple)

Un seul thread qui accède à la fois

```
…
int sum_shared = 0;
#pragma omp parallel
{
  int sum_local = 0;
  #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
      sum_local += i; }
  #pragma omp atomic
  {
    sum_shared += sum_local;
  }
}
printf("%d \n", sum_shared);
…
```

OpenMP

- Fonctions
  - **omp_get_wtime()**
    - Retourne le temps réel, en secondes
  - **omp_get_num_threads()**
    - Retourne le nombre de threads
  - **omp_get_thread_num()**
    - Retourne le numéro du thread courant
  - **omp_get_num_procs()**
    - Retourne le nombre de processeurs/cœurs

OpenMP

- Fonctions
  - **omp_get_wtime()** (exemple)

```
…
start = omp_get_wtime();
#pragma omp parallel private(cycle,i,j)
 {
  … // PARALLEL COMPUTATIONS
 }
finish = omp_get_wtime();
duration = finish - start;
…
```

Début de la mesure du temps

Fin de la mesure du temps

Retourne le nombre de secondes depuis la valeur initiale de l'horloge temps réel de l'OS

Portable sous **Linux** et **Windows**

OpenMP

- Variables d'environnement
  - **OMP_NUM_THREADS** **nombre** (ex. $ export **OMP_NUM_THREADS** 8)
    - Définit le nombre de threads dans la région parallèle
  - **OMP_SCHEDULE** (**type**, **chunk**)
    - Définit la distribution des itérations sur les threads

OpenMP

- Exercice d'application
  - Ecrire la version parallèle (OpenMP) pour 4 cœurs du programme suivant en utilisant la directive **#pragma omp parallel**
  - Proposer une deuxième solution avec une directive de partage de travail

```
For (i =0 ; i < 100 ; i++)
    C[i] = (A[i] * B[i])
```

OpenMP

- Exercice d'application

Solution 1 avec **#pragma omp parallel**

```
#include <omp.h>
#include <stdio.h>
int main () {
      …
      int nthreads = 4;
      omp_set_num_thread(nthreads);

     #pragma omp parallel
     {
         int id = omp_get_thread_num();
         int istart = id*(100/nthreads);
         int iend = (id+1)*(100/nthreads);
         for (int i = istart ; i < iend ; i++)
            C[i] = (A[i] * B[i]);
      }
}
```

OpenMP

- Exercice d'application

Solution 2 avec **#pragma omp parallel for**

```
#include <omp.h>
#include <stdio.h>
int main () {
    …
    omp_set_num_thread(4);

    #pragma omp parallel for
        for (int i = 0 ; i < 100 ; i++)
            C[i] = (A[i] * B[i]);
}
```

OpenMP

- Exercice d'application
  - Ecrire la version parallèle (OpenMP) pour 4 cœurs du programme suivant en utilisant la clause ''**reduction**'' de la directive de partage de travail
  - Proposer une deuxième solution en utilisant **#pragma omp critical**

```
For (i =0 ; i < 100 ; i++)
a + = b[i]
```

OpenMP

- Exercice d'application

Solution 1 avec **la clause reduction**

```
#include <omp.h>
#include <stdio.h>
int main () {
    …
    omp_set_num_thread(4);

    #pragma omp parallel for reduction (+:a)
        for (int i = 0 ; i < 100 ; i++)
            a + = b[i];
}
```

OpenMP

- Exercice d'application

Solution 2 avec **#pragma omp critical**

```
#include <omp.h>
#include <stdio.h>
int main () {
    …
    omp_set_num_thread(4);
    #pragma omp parallel
    {
       int a_local = 0;
       #pragma omp for nowait
         for (int i = 0 ; i < 100 ; i++)
           a_local + = b[i];
       #pragma omp critical
          a + = a_local;
    }
}
```

Contenu du chapitre

- OpenMP
  - Directives de compilation
  - Fonctions
  - Variables d'environnement
- Synthèse et conclusion

Synthèse et conclusion

- **OpenMP** : ensemble de directives de compilation, fonctions et variables d'environnement
- Programmation Parallèle des multicœurs
  - Créer des régions parallèles dans le code
  - Partager le code entre les threads
  - Synchroniser le travail entre les threads