Master MIASHS

Mathématiques, **I**nformatique **Appliquées** aux **S**ciences **H**umaines et **S**ociales

Baptiste Chapuisat

M2 MIASHS
OpenMP

UFR 6 : département de Mathématiques et d'Informatique Appliquées (MIAp)

Introduction

- OpenMP est un modèle de programmation parallèle pour architecture à mémoire partagée.
- On parle de programmation multithread.
- Il existe d'autre jeux d'instruction pour programmation multithread. Ex : pthreads

Historique

- Avant les années 2000 chaque constructeur avait son propre jeux de directives pour la programmation multithread.
- La démocratisation des machines multiprocesseurs à mémoire partagée à conduit à la définition d’un standard.
- En 1997 une majorité de constructeur adopte OpenMP comme standard dit « industriel »

Historique

- OpenMP à d'abord été formalisé pour Fortran puis pour C/C++.
- Supporté par de nombreux compilateurs (Open Source, IBM, Intel, Texas Instrument, Oracle, …).
- Aujourd'hui version 5.0.
- Également porté sur les accélérateur depuis la version 4.
- L'Architecture Review Board (ARB) est l'organisme qui est en charge de l'évolution d'OpenMP.
- http://www.openmp.org

- Avantages
  - Simple à implémenter.
  - Les communications sont à la charge du compilateur.
- Inconvénients
  - Limité en nombre de processeur.
  - Limité dans le choix des langages.

https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

Exemple : Produit scalaire

```
#include <stdio.h>
#define SIZE 256
int main () {
   double sum , a[SIZE], b[SIZE ];
   // Initialization
   sum = 0.;
   for (int i = 0; i < SIZE; i++) {
      a[i] = i * 0.5;
      b[i] = i * 2.0;
   }
   // Computation
   for (int i = 0; i < SIZE; i++)
      sum = sum + a[i]*b[i];

   printf("sum = %g\n", sum);
   return 0;
}
```

```c
#include <stdio.h>
#include <pthread.h>
#define SIZE 256
#define NUM_THREADS 4
#define CHUNK SIZE/ NUM_THREADS
int id[ NUM_THREADS ];
double sum , a[SIZE], b[SIZE ];
pthread_t tid[ NUM_THREADS ];
pthread_mutex_t mutex_sum;
void* dot(void* id) {
  size_t i;
  int my_first = *(int*)id * CHUNK;
  int my_last = (*(int*)id + 1) * CHUNK;
  double sum_local = 0.;
  // Computation
  for (i = my_first; i < my_last; i++)
    sum_local = sum_local + a[i]*b[i];
  pthread_mutex_lock (& mutex_sum );
  sum = sum + sum_local;
  pthread_mutex_unlock (& mutex_sum );
  return NULL;
  }
int main () {
  size_t i;
  // Initialization
  sum = 0.;
  for (i = 0; i < SIZE; i++) {
    a[i] = i * 0.5;
    b[i] = i * 2.0;
  }
  pthread_mutex_init (& mutex_sum , NULL );
  for (i = 0; i < NUM_THREADS; i++) {
    id[i] = i;
    pthread_create (& tid[i], NULL , dot ,
    (void*)& id[i]);
  }
  for (i = 0; i < NUM_THREADS; i++)
    pthread_join (tid[i], NULL );
  pthread_mutex_destroy (& mutex_sum );
  printf("sum = %g\n", sum);
  return 0;
}
```

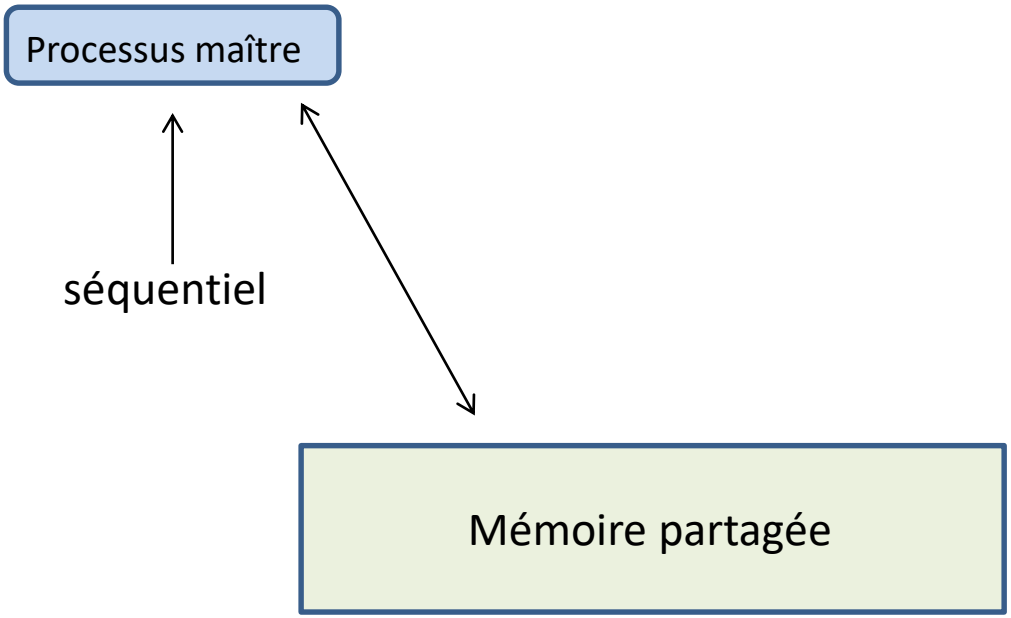
OpenMP

```
#include <stdio.h>
#include <omp.h>
#define SIZE 256
int main () {
   omp_set_num_threads(4);
   double sum , a[SIZE], b[SIZE ];
   // Initialization
   sum = 0.;
   # pragma omp parallel for
   for (int i = 0; i < SIZE; i++) {
      a[i] = i * 0.5;
      b[i] = i * 2.0;
   }
   // Computation
   #pragma omp parallel for reduction(+: sum)
   for (int i = 0; i < SIZE; i++) {
      sum = sum + a[i]*b[i];
   }
   printf("sum = %g\n", sum);
   return 0;
}
```

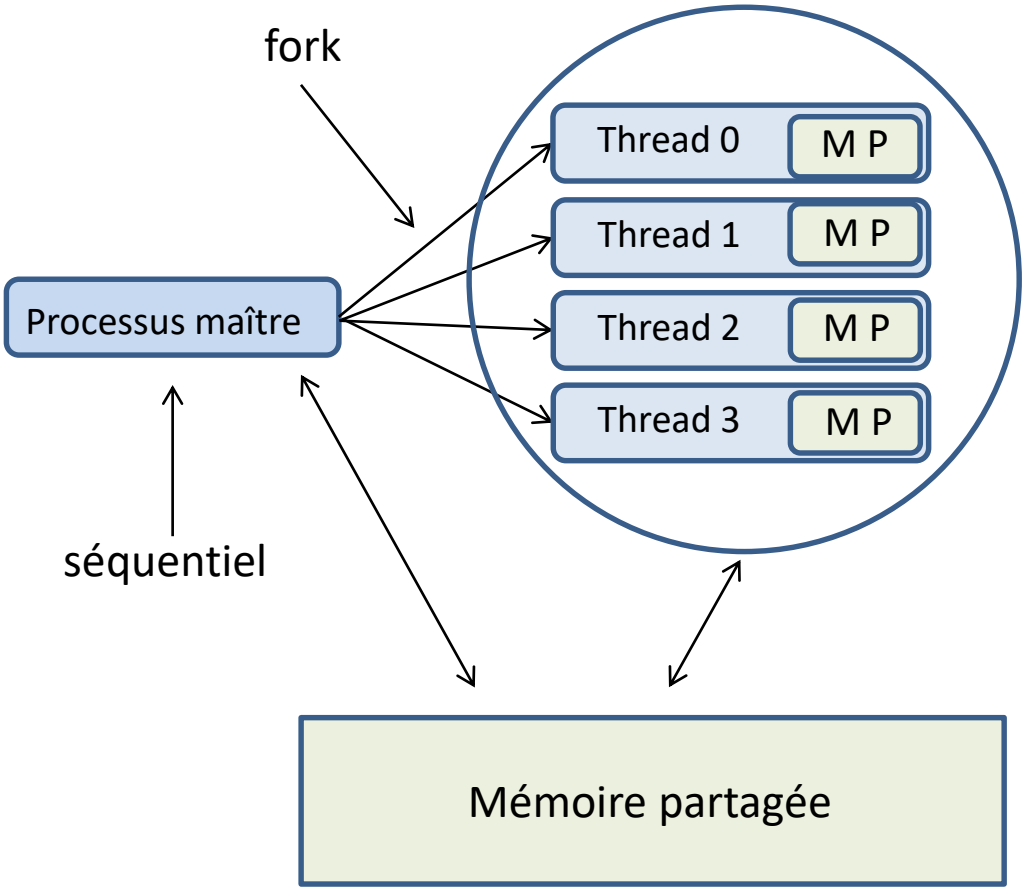
Séquentiel

```
#include <stdio.h>

#define SIZE 256
int main () {

   double sum , a[SIZE], b[SIZE ];
   // Initialization
   sum = 0.;

 for (int i = 0; i < SIZE; i++) {
      a[i] = i * 0.5;
      b[i] = i * 2.0;
   }
   // Computation

 for (int i = 0; i < SIZE; i++)
      sum = sum + a[i]*b[i];

   printf("sum = %g\n", sum);
   return 0;
}
```

Concepts généraux

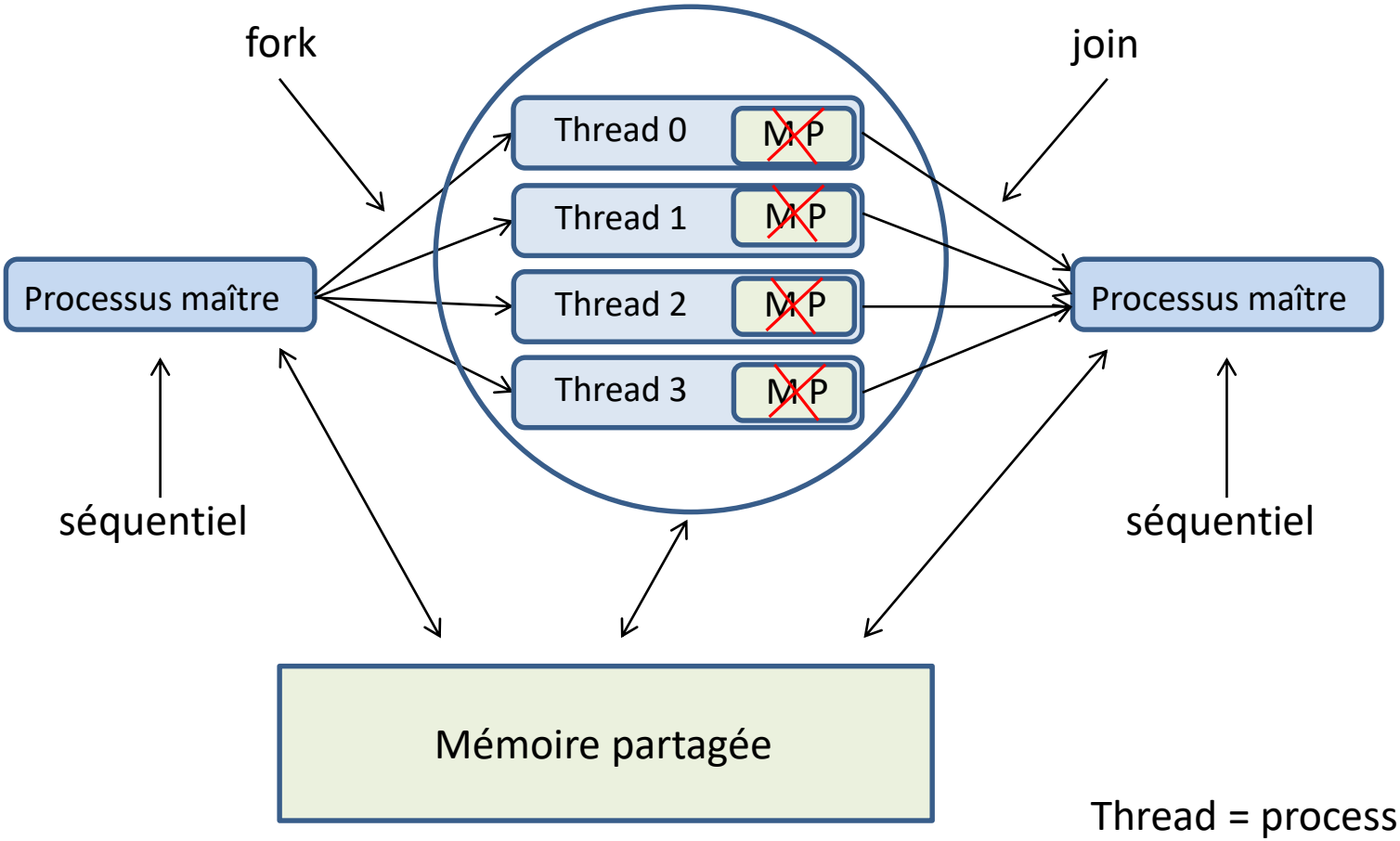


Concepts généraux



Thread = processus léger
M P = Mémoire Privée

Concepts généraux



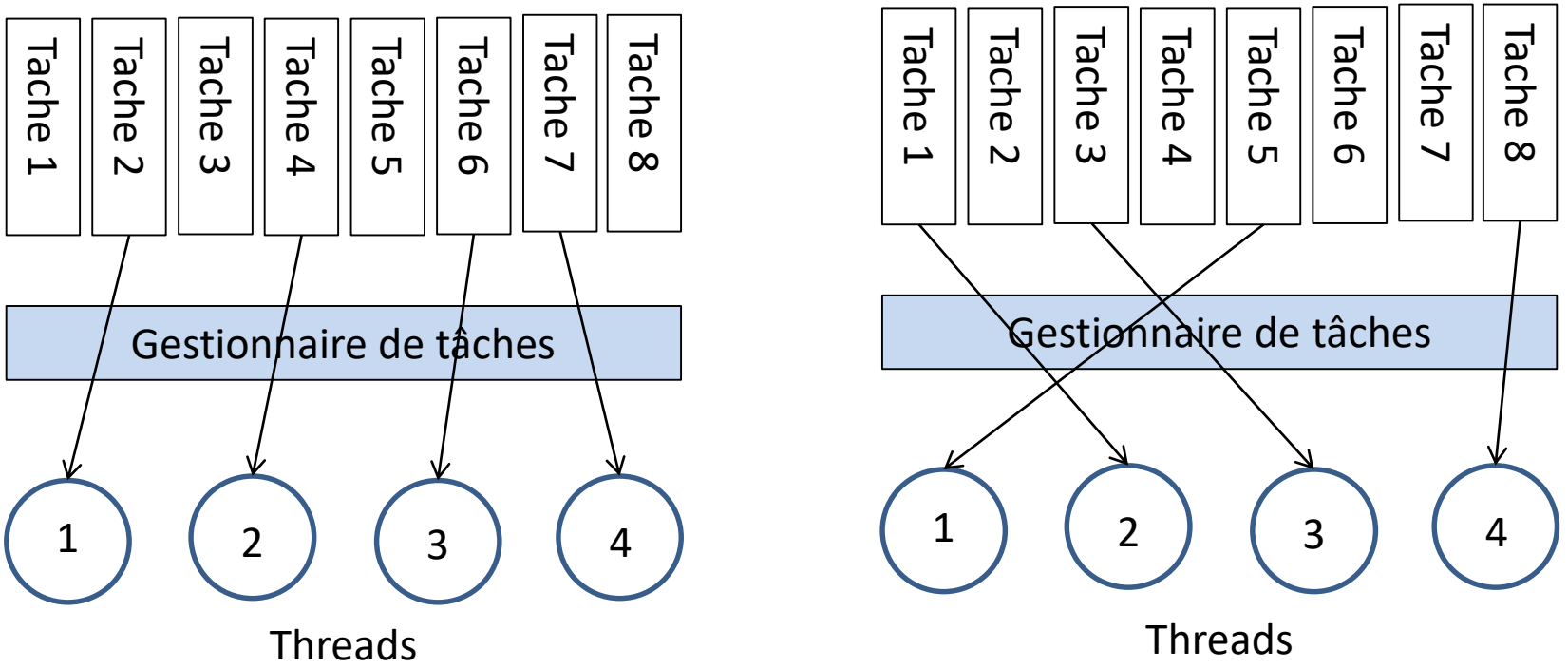
Thread = processus léger

Concepts généraux

- Chaque thread exécute sa propre séquence d'instructions.
- C'est le système d'exploitation qui choisit l'ordre d'exécution des threads. Il les affecte aux unités disponibles.
- On ne connait pas l'ordre global d'exécution des instructions.

Concepts généraux

omp_set_num_threads(8);



La compilation

Compilation avec GNU :

```
$ module load cv-standard gcc/5.3.0
$ gcc -fopenmp -o prog.exe prog.c
```

Compilation avec Intel :

```
$ module load cv-standard intel/compiler/64/2016.3.210
$ icc -openmp -o prog.exe prog.c
```

Principes OpenMP

- Les directives et clause de compilation : Elles définissent le partage du travail, la synchronisation et le statut des données (privées, partagées...).
- La bibliothèque OpenMP et les fonctions.
- Les variables d’environnement : Elles sont prise en compte à l’exécution.

Directives OpenMP

- Les directives sont déclarées comme des commentaires et ne sont prises en compte que si l'option de compilation OpenMP à été spécifier

- Format C/C++

```
#pragma omp <directive> [ clause [ clause ] . . . ]
{
…
}
```

- Format Fortran90

```
! $OMP PARALLEL <directive> [ clause [ clause ] . . . ]
…
…
! $OMP END PARALLEL
```

Les fonctions (Runtime Library Routine)

- En C/C++ (fichier header)

  # include " omp.h"

- En Fortran90 (module)

  !$ use OMP_LIB

Quelques fonctions OpenMP

- omp_get_num_threads() : retourne le nombre total de threads utilisés
- omp_set_num_threads(int) : spécifie un nombre de thread dans une région parallèle
- omp_get_thread_num() : retourne le numéro du thread courant
- omp_in_parallel() : retourne F en séquentiel et T en parallèle
- omp_get_num_procs() : retourne le nombre de threads disponibles sur la machine
- omp_get_wtime() : retourne le temps écoulé d'un certain point

Les variables d'environnements

- Elles permettent de définir les paramètre d'exécution du programme.
- Le nom de la variable est toujours écrit en majuscule
- Elles doivent être définies avant le début du programme.

  - OMP_NUM_THREADS
  - OMP_THREAD_LIMIT
  - OMP_DEFAULT_DEVICE

Un premier exemple

```
#include <stdio.h>


int main() {
  int val = 100;
  int rang;




        val = val + rang;
        printf("dans le thread %d la valeur de val est %d\n", rang, val);


  printf("En dehors la valeur de val est %d\n", val);
}
```

Un premier exemple

- Compiler le programme
- Écrire un fichier batch SLURM qui exécute trois fois le programme.
- Lancer le job.
- Que peut on dire de la variable « val » ?

Un premier exemple

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

int main() {
  int val = 100;
  int rang;

  omp_set_num_threads(4);




        val = val + rang;
        printf("dans le thread %d la valeur de val est %d\n", rang, val);


  printf("En dehors la valeur de val est %d\n", val);
}
```

Un premier exemple

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

int main() {
  int val = 100;
  int rang;

  omp_set_num_threads(4);

  # pragma omp parallel
  {

        val = val + rang;
        printf("dans le thread %d la valeur de val est %d\n", rang, val);
  }

  printf("En dehors la valeur de val est %d\n", val);
}
```

Un premier exemple

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

int main() {
  int val = 100;
  int rang;

  omp_set_num_threads(4);

  # pragma omp parallel
  {
        rang = omp_get_thread_num();
        val = val + rang;
        printf("dans le thread %d la valeur de val est %d\n", rang, val);
  }

  printf("En dehors la valeur de val est %d\n", val);
}
```

Statut des variables

- Par défaut les variables sont publiques.
- Pour qu'une variable soit privée il faut rajouter la clause *private(nom_var)* dans la directive de paralllélisation.

  # pragma omp parallel private(val)

Statut des variables

- Par défaut les variables sont publiques.
- Pour qu'une variable soit privée il faut rajouter la clause *private(nom_var)* dans la directive de paralllélisation.

  # pragma omp parallel private(val)
- On peut récupérer une variable publique et l'utiliser comme une variable privée avec la *clause firstprivate(nom_var).*

  # pragma omp parallel firstprivate(val)

Statut des variables

- Par défaut les variables sont publiques.
- Pour qu'une variable soit privée il faut rajouter la clause *private(nom_var)* dans la directive de paralllélisation.

  # pragma omp parallel private(val)
- On peut récupérer une variable publique et l'utiliser comme une variable privée avec la *clause firstprivate(nom_var).*

  # pragma omp parallel firstprivate(val)
- La clause *lastprivate(nom_var)* permet de rendre public une variable privée.

Statut des variables

- Par défaut, les variables sont partagées mais pour éviter les erreurs, on peut définir le statut de chaque variable explicitement.
- La clause DEFAULT(NONE) permet d’obliger le programmeur à expliciter le statut de chaque variable private(nom_var) ou share(nom_var)

Statut des variables

- La directive **threadprivate**

  # pragma omp threadprivate(var1, var2, …)

- Spécifie que les variables listées seront privées et persistantes à chaque thread au travers de l'exécution de multiples régions parallèles.
- La directive doit suivre la déclaration des variables globales ou statiques concernées.
- Le nombre de threads doit être fixe

Définition du nombre de thread

- Avec une fonction dans le programme
  omp_set_num_threads(4);

- Dans une directive parallèle
  #pragma omp parallel num_threads(4)

- Avec une variable d'environnement
  - *format bash*
  export OMP_NUM_THREAD=4
  - *format csh*
  setenv OMP_NUM_THREAD 4

La clause if

- Quand la clause if est présente, une équipe de threads n'est créée que si l'expression scalaire est différente de zéro, sinon la région est exécutée séquentiellement par le thread maître.

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

int main() {
   int par = 1;

   # pragma omp parallel if(par)
   {
      printf("Region parallele\n");
   }

   par = 0;
   # pragma omp parallel if(par)
   {
      printf("Region sequentiel\n");
   }
}
```

Exécution exclusive

- Les directives **single** et **master** permettent d'exécuter en séquentiel une portion de code se trouvant à l'intérieur d'une région parallèle.
- Avec la directive **single** la portion de code est exécutée par le premier thread disponible. **single** peut être utilisé avec les clauses **private()**, **firstprivate()**, **copyprivate()** et **nowait**.

```
# pragma omp parallel
        ….
   # pragma omp single [clause()]
```

- Avec la directive **master** la portion de code est exécutée par le thread maître (thread 0).
- La clause **copyprivate()** permet de diffuser une variable privée aux autres threads.
- Exercice : écrire un programme qui affiche le numéro de chaque thread et ou le thread maître affiche le nombre total de thread (omp_get_num_threads()).

Sections parallèles

- Une section est une portion de code exécutée par un seul thread.
- A la différence des directives **single** et **master** la directive **section** permet d'exécuter en parallèle plusieurs sections de code différentes. **section** peut être utilisé avec les clauses **private()**, **firstprivate**, **lastprivate()**, **reduction()** et **nowait**.

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
       < code bloc 1>
    #pragma omp section
      < code bloc 2>
  }
}
```

Résumer

- On déclare une zone parallèle avec

  # pragma omp parallel <directive> [clause …] {

  … }

- Directive sur le statut des variables :

  Par défaut les variables sont globales.

  private(), firstprivate(), threadprivate, lastprivate(), copyprivate()

- Directives sur les zones parallèles

  single, master, sections, if()

Parallélisassions des boucles for

- Pour paralléliser une boucle for on utilise la directive # pragma omp parallel for

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

int main() {

   int num_proc;

   # pragma omp parallel for
   for(int i=0; i<12; i++) {
      num_proc = omp_get_thread_num()
      printf("Le traitement %d est executer par le thread %d\n", i, num_proc);
   }
}
```

Parallélisassions des boucles for

- Par défaut une synchronisation globale est effectuée en fin de construction *for* mais on peut la supprimer avec la clause *nowait*.
- Attention !!! Les boucles do while ne sont pas parallélisable avec OpenMP. Pourquoi ?

La clause lastprivate

- Utilisable avec la directive section ou dans une boucle for.
- Permet de récupérer la dernière valeur d'une variable.

```
# pragma omp parallel for lastprivate(num_proc)
 for(int i=0; i<4; i++) {
    num_proc = omp_get_thread_num();
 }
 printf("num_proc est egale a %d\n", num_proc);
```

La clause collapse

- La clause collapse(n) indique le nombre de boucles dans un ensemble de boucles imbriquées qui doivent être regroupées en une seule itération.

  #pragma omp for collapse(2)

- La clause collapse permet de gagner en performance mais elle n'est utilisable que si les boucles sont imbriquées et sans dépendances.
- Attention !!! Les boucle for doivent se suivre sans instruction intermédiaire.

Exercice

- Écrire la version parallèle OpenMP du produit matrice vecteur dont la version séquentielle est

```
for (i=0 ;i<SIZE ;i++) {
   C[i]=0 ;
   for (j=0 ;j<SIZE ;j++)
      C[i]+=mat[i][j]*vect[j] ;
}
```

1. La directive collapse peut-elle fonctionner ? Expliquer pourquoi.

La réduction

- Une réduction est une opération associative appliquée à une variable partagée..

  #pragma omp parallel for reduction(op: var1, var2, …)

- En langage C « op » peut prendre les valeurs suivantes : +, -, *, &, |, ˆ, &&.

- Exercice : Produit scalaire

  Initialisez deux vecteurs a et b avec les indices d'une boucle for tel que *a[i] = i * 0.5* et *b[i] = i * 2.0*

$$\text{Sum} = \sum_{i=0}^{10} a[i] * b[i]$$

La clause schedule()

- Les itérations sont réparties automatiquement par le compilateur sur chaque thread. Cette répartition peut être contrôlée avec la clause schedule().

  #pragma omp for schedule(type)

  - schedule(type,N)
    - type : mode de répartition des paquets d'itérations (static, dynamic, guided ou runtime)
    - N : nombre d'itérations dans un paquet (facultatif, prend la valeur 1 s'il n'est pas spécifié)

La clause schedule()

- **(static, *chunk_size*)** : les itérations sont réparties sur chaque thread par paquets en proportion égale, sans accorder de priorité aux threads.

- **(dynamic, *chunk_size*)** : Les paquets d'itérations sont distribués sur les threads les plus rapides. A chaque fois qu'un thread termine un paquet d'itérations il reçoit directement un autre paquet.

- **(guided , *chunk_size*)** : la taille des paquets d'itérations attribués à chaque thread décroit exponentiellement, chaque paquet est attribué au thread le plus rapide. Cas particulier d'allocation dynamic.

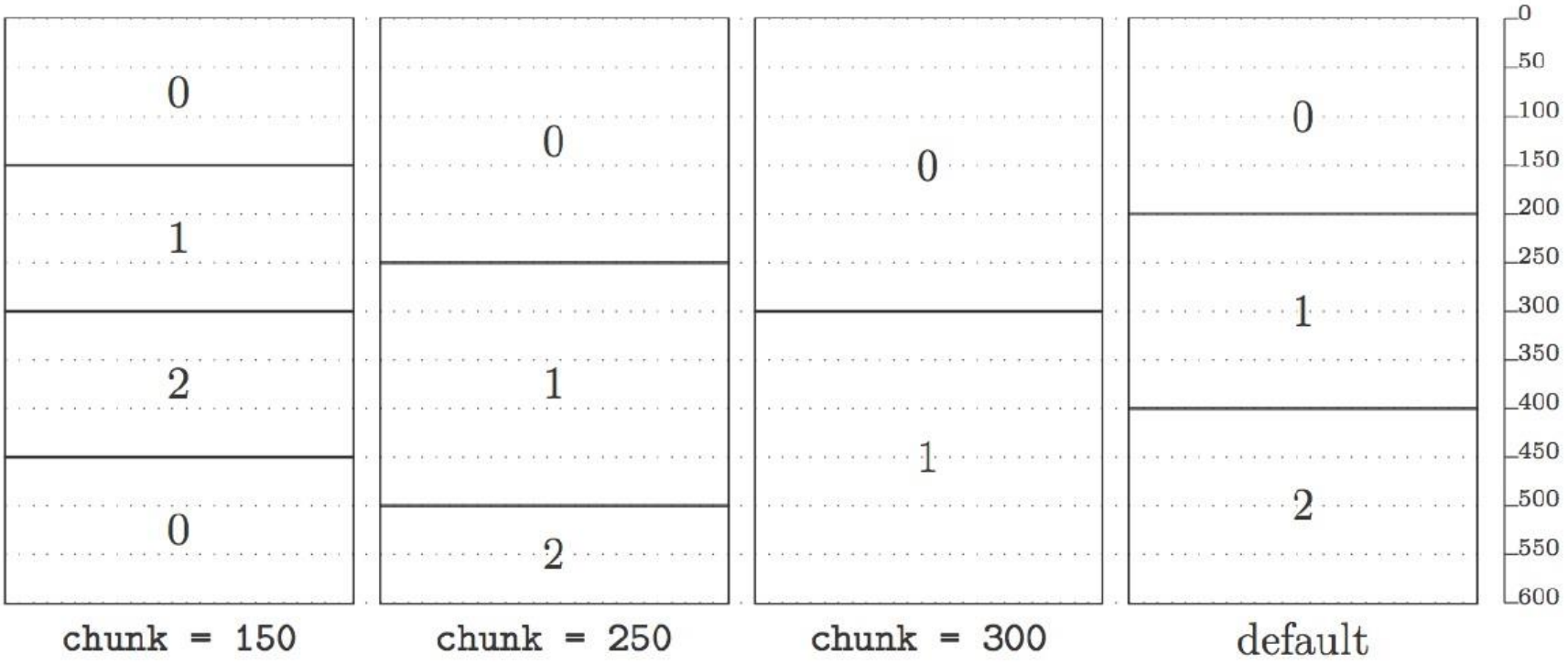
- **(runtime , *chunk_size*)** : permet de de choisir le mode de répartition des itérations avec la variable d'environnement OMP_SCHEDULE

  $ export OMP_SCHEDULE =" static ,10"

La clause schedule()

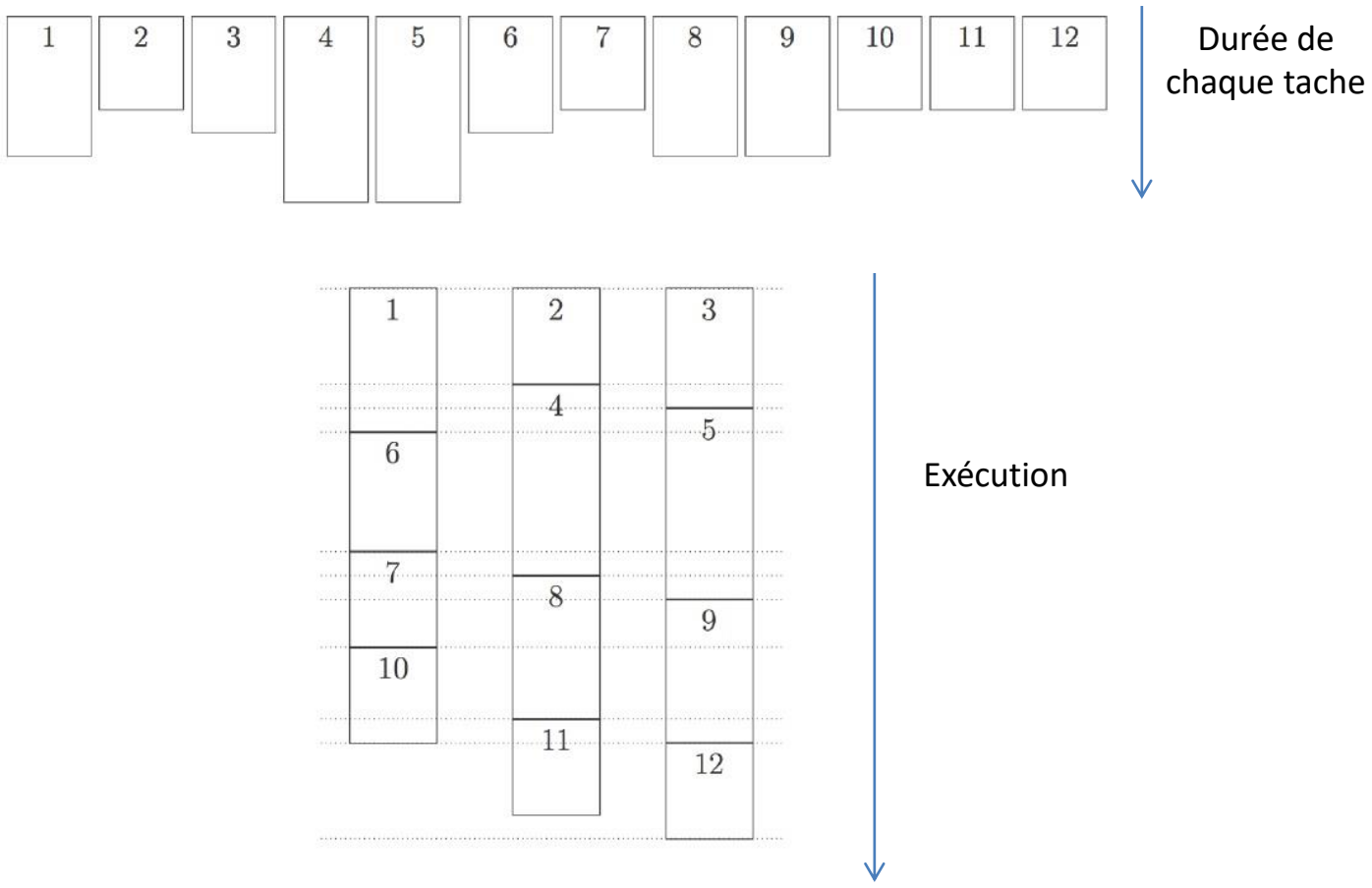
- L'ordonnancement du travail de type STATIC

Exemple pour 600 itérations sur trois threads



La clause schedule()

- L'ordonnancement du travail de type DYNAMIC



Les directives de synchronisation

- En l’absence de clause **nowait** une synchronisation est automatiquement appliquée en fin de région parallèle.
- Une synchronisation explicite peut être nécessaire dans certains cas.

Les directives de synchronisation

- La directive **barrier** synchronise l'ensemble des threads dans une région parallèle.

  #pragma omp barrier

- La directive **critical** spécifie que le bloc d'instruction suivant la directive ne doit être exécuté que par un seul thread à la fois.

  #pragma omp critical (nom)

- 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 ait terminé avant de commencer l'exécution du bloc

- La directive **ordered** permet l'exécution d'une boucle dans l'ordre des indices. L'exécution est équivalente à une exécution séquentielle.

  #pragma omp ordered

- Cette directive est principalement utilisée pour le debogage

Les directives de synchronisation

- La directive **atomic** est une directive de mise à jour. Elle assure qu'une variable partagée est lue et modifié en mémoire par un seul thread à la fois.

  #pragma omp atomic [clause]
- La clause peut être read, write, update, capture

- Les valeurs des variables partagées peuvent rester temporairement dans des registres pour des raisons de performances. La directive **flush** garantit que chaque thread a accès aux valeurs des variables partagées modifiées par les autres threads.

  #pragma omp flush (var1, var2, …)

Les procédures orphelines

- On qualifie de « orpheline » les procédures contenant des régions parallèles qui sont appelées dans un région parallèle.
- Le résultat de l’exécution (nombre de threads) dépend du mode de compilation.

Conclusion

- Un programme séquentiel et facilement parallélisable avec OpenMP.
- Mais l'optimisation dépend de la complexité du programme.
- La difficulté porte principalement sur le partage des variables et sur la synchronisation.