OpenMP

Introduction to OpenMP

Adrien Roussel
adrien.roussel@cea.fr

Lecture Outline

- OpenMP introduction
- Doing a parallele region
- Shared data or private (local) data
- Synchronisations
- Sharing work between threads
- Excusive execution

Introduction to OpenMP

Definition

- Definition : OpenMP (Open Multi-Processing)
  - API for parallel computing on shared-memory architecture
  - Supported on all OS
    - Unix, Windows
  - Offer API for several programming languages
    - C/C++ and Fortran.
  - API: directives + software library + environnement variable.
  - Need compiler to understand and translate directives
- OpenMP is portable
  - Quick parallel application development

- 28th october 1997, a majority of HPC industrial companies have adopted OpenMP as an « industrial » standard

Some history

- OpenMP spec now belongs to OpenMP ARB (Architecture Review Board) in charge of its evolution
  - http://www.openmp.org
  - http://www.compunity.org

- OpenMP Version 2.0 was finalized in november 2000
  - Extensions relatives to the parallelization of specific Fortran 95 constructs.
  - Version 2.5 May 2005
- Version 3.0 in May 2008
  - Task parallelism
- Version 4.5 in November 2015
  - Accelerator support
- Version 5.0 in November 2018
  - Heterogeneous memory support

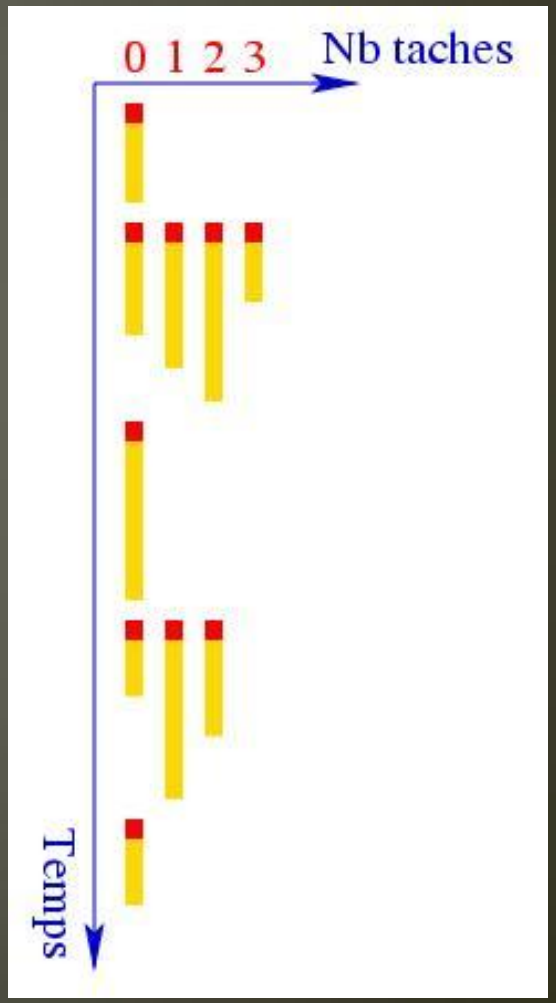
General concepts

- An OpenMP program is executed by a unique master process
  - This process activates lightweight processes, or threads, when entering a **parallel region**
- Each thread executes an implicit task composed of an set of instructions
  - Rank notion
- During a task execution, a variable can be read and/or modified in memory
  - It can be defined in the stack (local memory space) of the thread ; we say it is a *private* variable
  - It can be defined in a shared memory space ; we say it as a *shared* variable

Parallel region

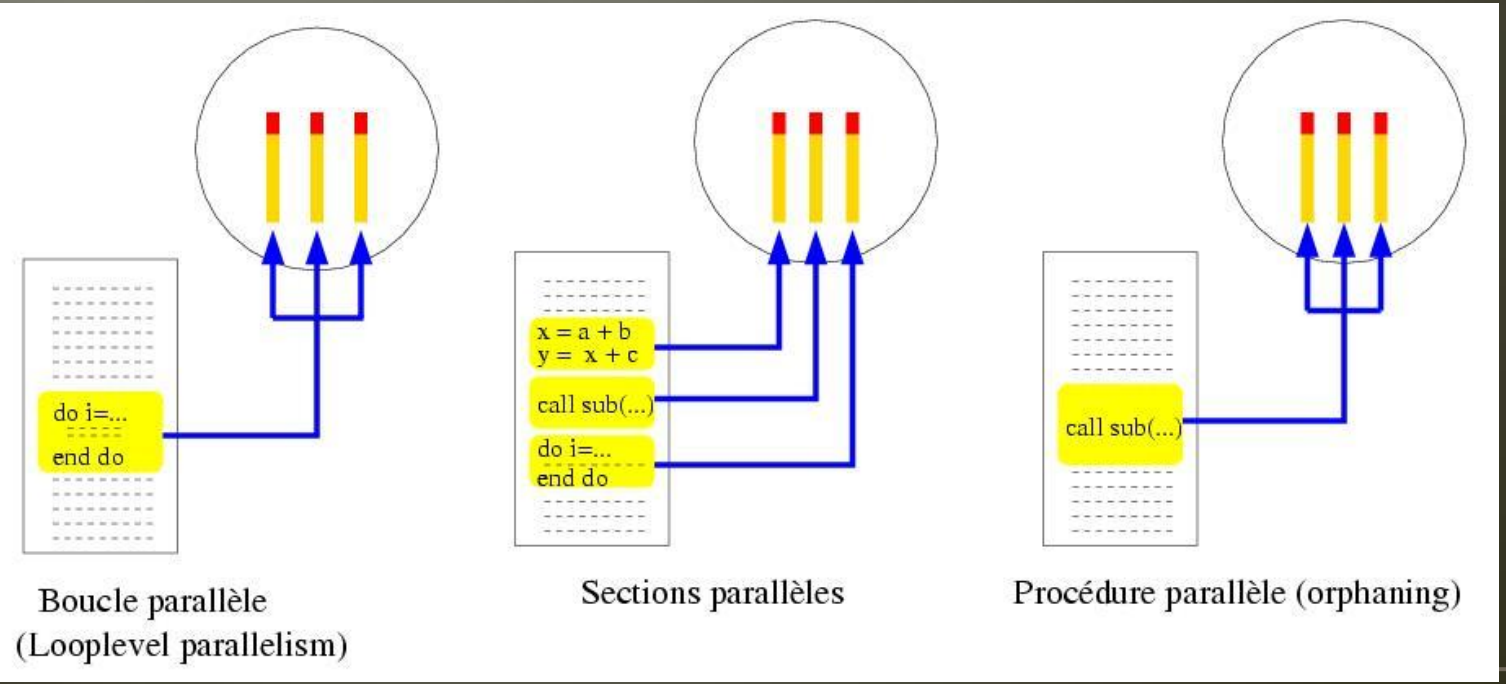
Parallel region

- An OpenMP program alternates between sequential and parallel regions
- Outside of parallel regions, a sequential region is always executed by the master thread, the one with rank 0
- A parallel region is executed by multiple threads at the same time
- Tasks can share work in the parallel region



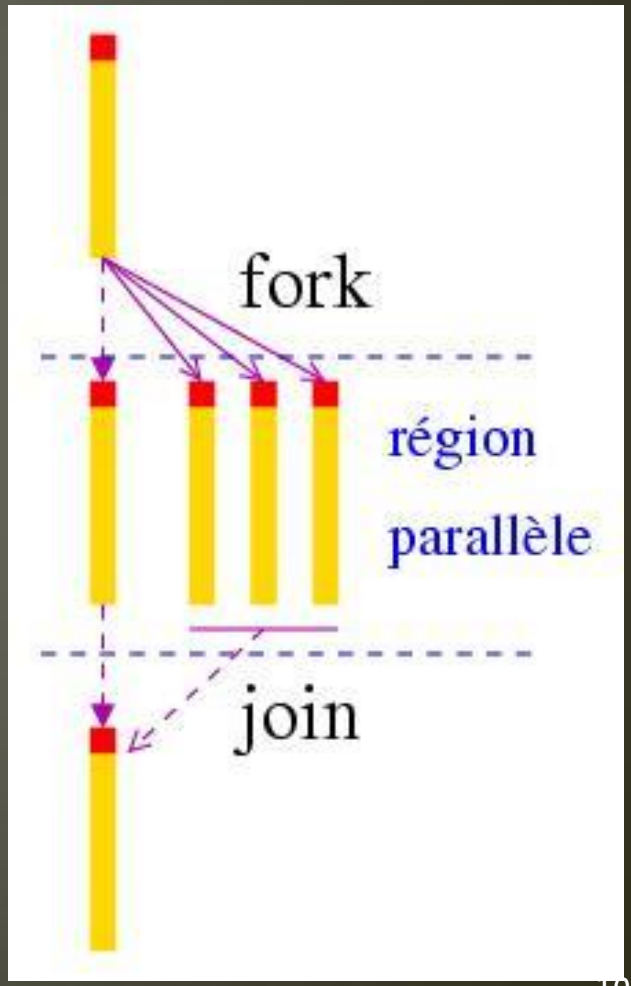
Worksharing

- Workshare consists in :
  - Spreading loop iterations between the multiple threads
  - Execute multiple code sections but only one per thread
  - Execute different, implicit or explicit, tasks



OpenMP basics

- Add openmp directives
  - Programmer responsability
- During execution, the OpenMP runtime will build a parallel region on the « fork/join » model
- At the beginning of a parallel region, the master thread creates/awakes other « children » threads
- At the end of a parallel region, the « children » threads are destroyed (or go to sleep), while the master thread continues its execution until the next parallel region



OpenMP programming

- Compilation directives and clauses :
  - Define parallel regions, workshare, synchronization, data flow, …
  - **sentinel directive [clause[clause]…]**
- Functions and subroutines: part of a library loaded during link
  - Ex: getting current thread rank
- Environment variables: once set before the execution, their values are taken into account during execution
  - Ex: number of OpenMP threads

Description of a parallel region

- The scope of a parallel region represents its influence area
  - The scope of a parallel region is inside the brackets following the parallel directive
    - Including the scope of functions called inside the parallel region
  - There is an implicit synchronizing barrier at the end of the parallel region
  - It is forbidden to branch directly inside or outside of the parallel region (ex. GOTO, CYCLE, etc.)

- In a parallel region, all the threads execute the same code.
  - Each thread has its own unique ID: rank
  - Use this unique ID to give different part of the global work to each thread
    - Get thread local id: omp_get_thread_num()
    - Get number of threads in parallel region: omp_get_num_threads()

First program example

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


void main() {
  #pragma omp parallel
    {
      printf( "Hello from thread %d\n",
        omp_get_thread_num() ) ;
    }
}
```

Header

Directive

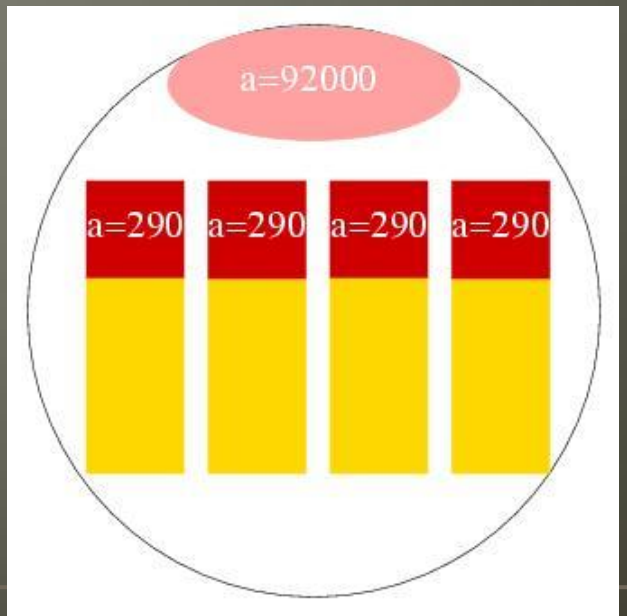
Parallel region

```
$ gcc –o test –fopenmp test.c

$ export OMP_NUM_THREADS=4
$ ./test
Hello from thread 1
Hello from thread 2
Hello from thread 3
Hello from thread 0
```

Data flow

Dataflow

- As OpenMP is a shared-memory model, variables declared outside a parallel region are shared inside the parallel region
  - Default behavior
- A clause allows to change this default behavior
  - Shared: data is shared between threads
  - Private: data is duplicated in each thread



Private data

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

int main() {
  float a;
  a = 92000.;
  printf( "Out region: %p\n", &a);
#pragma omp parallel
  {
    printf("In region: %p thread %d\n",
      &a, omp_get_thread_num() );
  }
  return 0;
}
```

```
$ gcc –fopenmp –o test test.c

$ OMP_NUM_THREADS=4
$ ./test
Out region: 0xbf8d9a9c
In region: 0xbf8d9a9c thread 1
In region: 0xbf8d9a9c thread 2
In region: 0xbf8d9a9c thread 3
In region: 0xbf8d9a9c thread 0
```

Private data

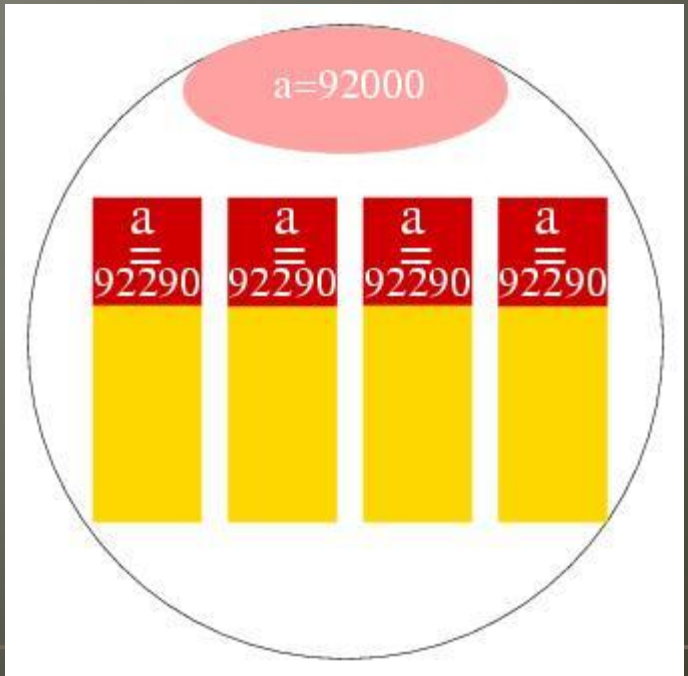
```
#include <stdio.h>
#include <omp.h>

int main() {
  float a;
  a = 92000.;
  printf( "Out region: %p\n", &a);
#pragma omp parallel private(a)
  {
    printf("In region: %p thread %d\n",
      &a, omp_get_thread_num() );
  }
  return 0;
}
```

Clause

```
$ gcc –fopenmp –o test test.c

$ OMP_NUM_THREADS=4
$ ./test
Out region: 0xbf887e2c
In region: 0xbf887dfc thread 0
In region: 0xb67cf2ec thread 3
In region: 0xb6fd02ec thread 2
In region: 0xb77d12ec thread 1
```

Initialized private data

- By default, private data are not initialized
- Clause « firstprivate » allows to initialize private variables
  - With the value they have when entering the parallel region



Initialized private data

```
#include <stdio.h>

int main() {
  float a;
  a = 92000.;
#pragma omp parallel default(none) \
    firstprivate(a)
  {
    a = a + 290.;
    printf("a vaut : %f\n",a);
  }
  printf("Hors region, a vaut : %f\n",
    a);
  return 0;
}
```

```
$ gcc –o prog -fopenmp prog.c

$ export OMP_NUM_THREADS=4
$ ./prog
a vaut : 92290.
a vaut : 92290.
a vaut : 92290.
a vaut : 92290.
Hors region, a vaut : 92000.
```

Static variables

- A variable is static if its location in memory is defined at its declaration at compile time
  - Fortran: COMMON variables, or in a MODULE, or declared SAVE, or initialized atdéclaration (ex. PARAMETER, DATA, etc…)
  - C: extern variables, or declared static
- THREADPRIVATE directive allows privatizing a static variable
  - And keep its location through multiple parallel regions
  - COPYIN clause gives the static value to all private instances
    - Like firstprivate clause for non-static variables

Static variables

```
#include <stdio.h>
#include <omp.h>
int a;


int main() {
  a = 92000;
  #pragma omp parallel
  {
    a = a + omp_get_thread_num();
    sub();
  }
  printf(
   "Hors region, A vaut: %d\n",a);
  return 0;
}
```

```
#include <stdio.h>

int a;


void sub(void) {
  int b;
  b = a + 290;
  printf("b vaut : %d\n",b);
}
```

```
└─ $ ./main_no_threadprivate.pgr
b vaut : 92291
b vaut : 92291
b vaut : 92296
b vaut : 92293
Hors region, A vaut: 92006
```

Static variables

```
#include <stdio.h>
#include <omp.h>
int a;
#pragma omp threadprivate(a)

int main() {
  a = 92000;
  #pragma omp parallel copyin(a)
  {
    a = a + omp_get_thread_num();
    sub();
  }
  printf(
   "Hors region, A vaut: %d\n",a);
  return 0;
}
```

```
#include <stdio.h>

int a;
#pragma omp threadprivate(a)

void sub(void) {
  int b;
  b = a + 290;
  printf("b vaut : %d\n",b);
}
```

```
└─ $ ./main_no_threadprivate.pgr
b vaut : 92290
b vaut : 92291
b vaut : 92292
b vaut : 92293
Hors region, A vaut: 92000
```

Other clauses

- The parallel directive also accpets other clauses
  - REDUCTION : allows to apply reduction (same operation on the same shared variable betwenn mutiple threads) ;
  - NUM_THREADS : allows to specifiy number of threads for the current parallel region

- It is possible to nest several parallel regions if this mode has been set with OMP_SET_NESTED

Synchronizations

Synchronizations

- Synchronization may be necessary in the following situations:
  1. To ensure each thead has reached a common location in the program (global barrier)
  2. To serialize the execution of all threads when writing shared variables

Synchronizations

- Synchronization may be necessary in the following situations:
  1. To ensure each thead has reached a common location in the program (global barrier)
  2. To serialize the execution of all threads when writing shared variables
     1. It is necessary to have exclusive writing to be sure to have the correct result

```
a = 920;
#pragma omp parallel shared(a)
{
   a++;
}
printf("a = %d\n", a); // may be 921 instead of 922
```

Synchronizations

- Synchronization may be necessary in the following situations:
  1. To ensure each thead has reached a common location in the program (global barrier)
  2. To serialize the execution of all threads when writing shared variables
     1. It is necessary to have exclusive writing to be sure to have the correct result

```
a = 920
#omp parallel shared(a)
{
   a++;
}
```

Synchronizations

- Synchronization may be necessary in the following situations:
  1. To ensure each thead has reached a common location in the program (global barrier)
  2. To serialize the execution of all threads when writing shared variables
     1. It is necessary to have exclusive writing to be sure to have the correct result

```
a = 920
#omp parallel shared(a)
        THREAD 0

Load a  // a = 920

a = a+1 // a = 921

Store a
```

Synchronizations

- Synchronization may be necessary in the following situations:
  1. To ensure each thead has reached a common location in the program (global barrier)
  2. To serialize the execution of all threads when writing shared variables
     1. It is necessary to have exclusive writing to be sure to have the correct result

```
a = 920
#omp parallel shared(a)
        THREAD 0                    THREAD 1

Load a  // a = 920
                            Load a  // a = 920
a = a+1 // a = 921
                            a = a+1 // a = 921
Store a
                            Store a
Printf("a = %d\n", a); // 921 instead of 922
```

  3. To sync two specific threads (lock mechanism).

Barrier

- The BARRIER directive synchronizes all concurrent threads in the same team of a parallel region

```
#pragma omp barrier
```

- Each thread waits for the others to reach the barrier to continue, together, the execution of their insctructions

- Implicit barrier
  - Most of the OpenMP construct (parallel region, work sharing constructs, …) have an implicit barrier at the end of their scope

Atomic update

- The ATOMIC directive ensures that a shared variable will be read and modified atomically
  - As if all was done in one clock cycle
  - Only one thread at a time can do it

```
#pragma omp atomic
```

- Its effect is local to the following instruction
  - Must be a supported operation (see p.235 & 236 of OpenMP 5.0 spec)

Atomic update

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



int main()
{
        int a = 920;
#pragma omp parallel shared(a)
        {
                #pragma omp atomic
                a++;
        }
        printf("a = %d\n", a); // always 920 + number of threads
        return 0;
}
```

Critical region

- A CRITICAL region can be seen as an ATOMIC directive generalized to a group of instructions represented by its scope
  - Or instruction not supported by the atomic directive
- All threads executes the region, in a non-deterministic way, but only one thread at a time
- A critical region is defined with the CRITICAL directive
  - Apply to the scope defined by its brackets

```
#pragma omp critical
{
…
}
```

- If the atomic directive support the desired operation, it is usually less performant to use a critical section instead

Critical regions

```
#include <stdio.h>

int main()
{
    int s, p;

    s = 0, p = 1;
    #pragma omp parallel
    {
        #pragma omp critical
        {
            s++;
            p*=2;
        }
    }
        printf("Somme et produit finaux : %d, %d\n",s,p);
    return 0;
}
```

Workshare

Workshare

- As we have seen so far, using a parallel construct and the use of some OpenMP functions can be enough to parallelize a code
  - And share work between the threads
- To be easier to use, OpenMP also provides work sharing constructs
  - For
  - Sections
  - Tasks
- These constructs allow the user to finely direct how the work will be spread among the threads
  - No need to get the ID of the thread and give some work accordingly
- Warning : work spread on multiple threads should be independent!
  - A thread should not read some data written by another thread (race condition)
  - Compiler and runtime will not check if the work is independent! It is the responsibility of the user!

Parallel for loop

- For loop iteration domain is cut into pieces and given to the threads
  - Pieces are called “chunks”
  - A chunk is a set of continuous iterations
  - Applied to a set of perfect nested loops
  - Restrictions on the loop structure (no *while loops, no irregular loops*)
  - Loop indexes are private
- Use of FOR directive
  - Apply on the following loop
  - “For” loops, no while loops, no irregular loops
  - Can apply on nested loops with the clause COLLAPSE
- The SCHEDULE clause allows driving how the iterations are cut into chunks
  - May help with load balancing
- By default, there is an implicit barrier ate the end of the FOR directive
  - Can be avoided with NOWAIT clause

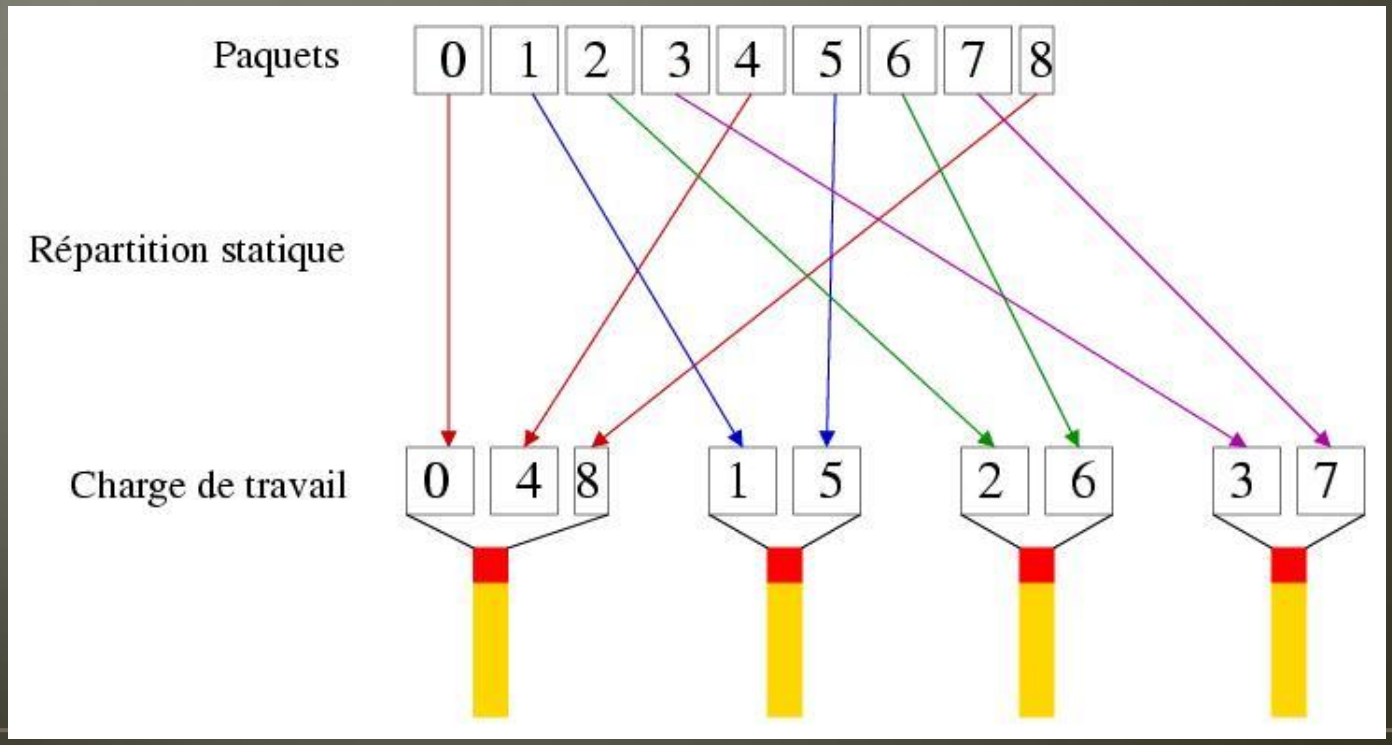
Parallel for loop

```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(static)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d running iteration %d\n",
         omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
}
```

```
$ gcc –fopenmp –o prog prog.c

$ OMP_NUM_THREADS=4 ./a.out
Thread 0 running iteration 0
Thread 0 running iteration 1
Thread 0 running iteration 2
Thread 3 running iteration 9
Thread 2 running iteration 6
Thread 2 running iteration 7
Thread 2 running iteration 8
Thread 1 running iteration 3
Thread 1 running iteration 4
Thread 1 running iteration 5
```

Static schedule

- The Schedule(STATIC) clause divides evenly the iterations in chunks
  - By default, a chunk size is maximal
  - Each chunk is then given to a thread in a round-robin (cyclic) way
    - i.e., thread 0 gets chunk 0, thread 1 gets chunk 1, thread n gets chunk n, thread 0 gets chunk n+1



Static schedule: size 1

```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(static,1)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d running iteration %d\n",
         omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
}
```

```
$ gcc –fopenmp –o prog prog.c

$ OMP_NUM_THREADS=4 ./prog
Thread 0 running iteration 0
Thread 0 running iteration 4
Thread 0 running iteration 8
Thread 3 running iteration 3
Thread 3 running iteration 7
Thread 2 running iteration 2
Thread 2 running iteration 6
Thread 1 running iteration 1
Thread 1 running iteration 5
Thread 1 running iteration 9
```

Static schedule: size 2

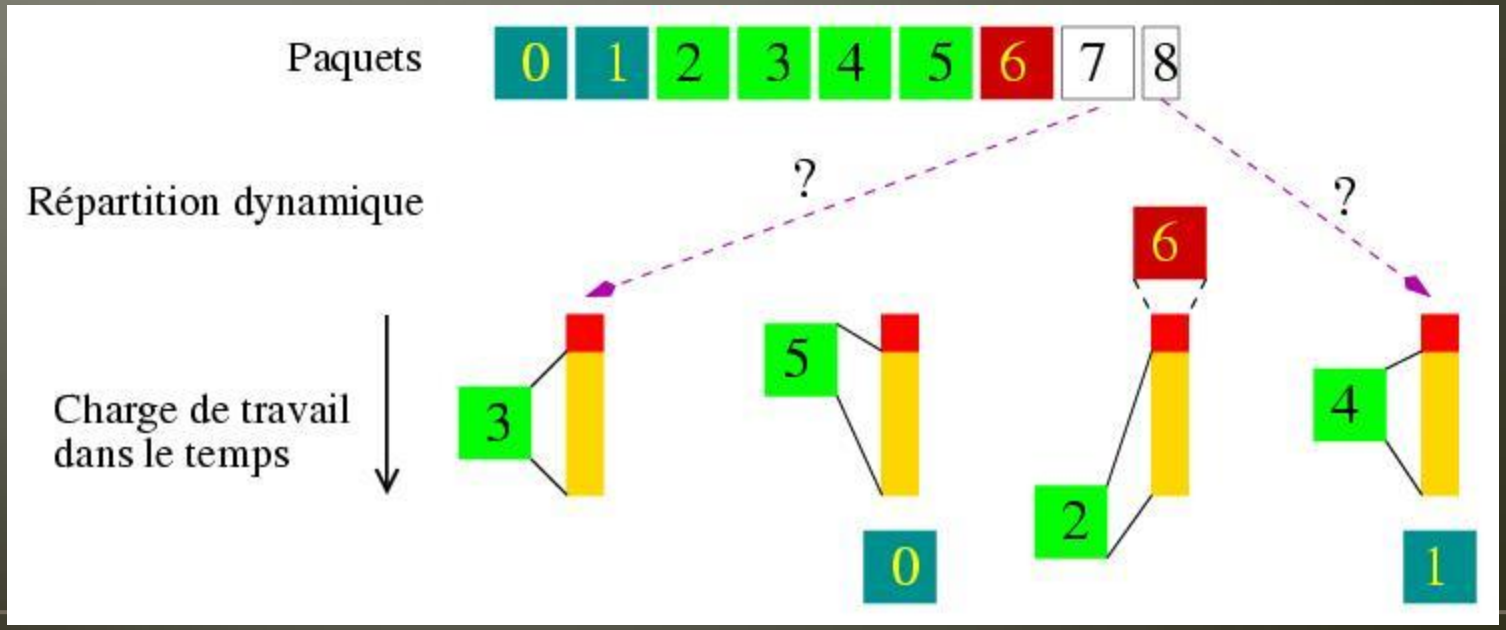
```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(static,2)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d running iteration %d\n",
         omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
}
```

```
$ gcc –fopenmp –o prog prog.c

$ OMP_NUM_THREADS=4 ./prog
Thread 0 running iteration 0
Thread 0 running iteration 1
Thread 0 running iteration 8
Thread 0 running iteration 9
Thread 3 running iteration 6
Thread 3 running iteration 7
Thread 1 running iteration 2
Thread 1 running iteration 3
Thread 2 running iteration 4
Thread 2 running iteration 5
```

Clause schedule

- The Schedule(DYNAMIC) clause divides evenly the iterations in chunks
  - Once a thread finishes its current chunk, it will get another one

```
$ export OMP_SCHEDULE="DYNAMIC,480"
$ export OMP_NUM_THREADS=4 ; ./prog
```



Static schedule: size 1

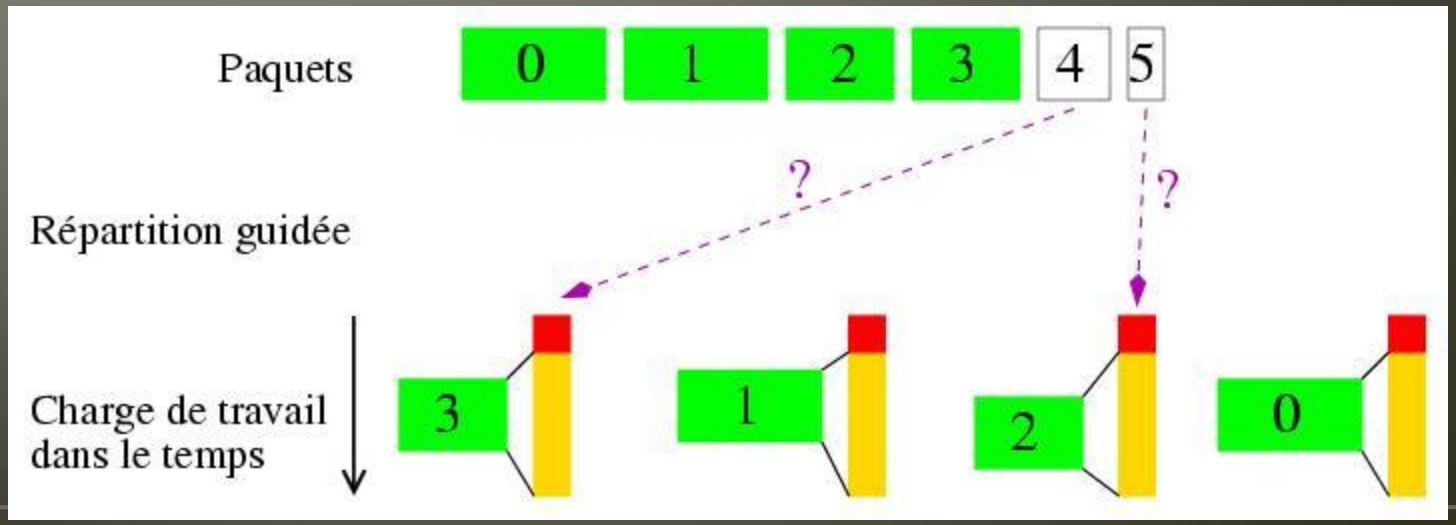
```
#include <stdio.h>

int main( int argc, char ** argv ) {
  int N ;

  N = 10 ;

#pragma omp parallel
  {
    int i ;
#pragma omp for schedule(dynamic,1)
    for ( i = 0 ; i < N ; i++ ) {
      printf( "Thread %d running iteration %d\n",
         omp_get_thread_num(), i ) ;
    }
  }
  return 0 ;
}
```

```
$ gcc –fopenmp –o prog prog.c

$ OMP_NUM_THREADS=4 ./prog
Thread 0 running iteration 1
Thread 0 running iteration 4
Thread 0 running iteration 5
Thread 0 running iteration 6
Thread 0 running iteration 7
Thread 1 running iteration 0
Thread 1 running iteration 9
Thread 2 running iteration 2
Thread 0 running iteration 8
Thread 3 running iteration 3
```

Clause schedule

- The Schedule(guided) clause divides the iterations in decreasing sizes
  - All chunks have a size above a given value (except for the last)
  - Like for dynamic, as soon as a thread finishes its current chunk, it gets another one

```
> export OMP_SCHEDULE="GUIDED,256"
> export OMP_NUM_THREADS=4 ; ./prog
```



Reduction

- A reduction is an associative operation performed on a shared variable
- Operation can be :
  - arithmetic : +, --, * ;
  - logic : .AND., .OR., .EQV., .NEQV. ;
  - Intrinsic function: MAX, MIN, IAND, IOR, IEOR.
- Each thread computes a local partial result…
  - Only considering the iterations it works on
  - Independent from the other threads local result
- … then synchronizes with the other threads at the end to compute the global results

Reduction

```
#include <stdio.h>
#define N 5
int main()
{
  int i, s=0, p=1, r=1;

  #pragma omp parallel
  {
    #pragma omp for reduction(+:s) reduction(*:p,r)
    for (i=0; i<N; i++) {
      s = s + 1;
      p = p * 2;
      r = r * 3;
    }
  }
  printf("s = %d ; p = %d ; r = %d\n",s,p,r);
  return 0;
}
```

```
$ gcc –o prog –fopenmp prog.c

$ export OMP_NUM_THREADS=4
$ ./prog
s = 5 ; p = 32 ; r = 243
```

Dataflow in for directive

- The for directive also accepts dataflow clauses
  - PRIVATE : a variable will be private (i.e., one copy per thread) inside the loop;
  - FIRSTPRIVATE : a variable will be private inside the loop, and the copy local to the thread will be initialized with the value of the original variable when entering the loop
  - LASTPRIVATE : a variable will be private inside the loop, and the variable outside the loop will get back, after the loop, the value of the local copy computed by the thread doing the last iteration

Combined directives

- The « parallel for » directive is the fusion of the two distinct directives « parallel » and « for »
  - Possible clauses are the union of the clauses of the two directives
- The end of scope is synchronize with an implicit barrier.
  - As for the parallel directive, it is not possible to skip the implicit barrier by using the NOWAIT clause.

Parallel sections

- A section is a piece of the program executed by only one thread
- Several independent pieces of the program can be defined by the user with several SECTION directives inside the scope of an englobing SECTIONS directive
- The goal is to define different pieces of code that can run concurrently on different threads
- There is an implicit barrier at the end of the SECTIONS clause
  - Implicit barrier can be ignored with the nowait clause

Parallel sections

```
int main() {
    int i, rang;
    float pas_x, pas_y;
    float coord_x[M],coord_y[N];
    float a[M][N], b[M][N];

    #pragma omp parallel private(rang) num_threads(3)
    {
        rang=omp_get_thread_num();
        #pragma omp sections nowait
        {
            #pragma omp section
            {
                lecture_champ_initial_x(a);
                printf("Tâche numéro %d : init. champ en X\n",rang);
            }
            #pragma omp section
            {
                lecture_champ_initial_y(b);
                printf("Tâche numéro %d : init. champ en Y\n",rang);
            }
        }
    }
    return 0;
}
```

Exclusive execution

Exclusive directives

- It might be necessary to have a piece of the code to be executed by only one thread
  - While the other threads are waiting for this piece to be done

- As opening and clausing parallel regions can be expensive, OpenMP provides two directives to have a sequential part of the code in a parallel directive
  - MASTER directive
  - SINGLE directive.

Master directive

- The code inside a « master » directive is executed only by the master thread (i.e., with rank 0)
  - No clause, no implicit barrier

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

int main()
{
    int rang;
    float a;

    #pragma omp parallel private(a,rang)
    {
        a = 92290.;

        #pragma omp master
        {
            a = -92290.;
        }

        rang=omp_get_thread_num();
        printf("Rang : %d ; A vaut : %f\n",rang,a);
    }
    return 0;
}
```

Single directive

- The code inside a « single » directive is executed only by one thread
  - Do not know which one
    - First to arrive ti the single directive
  - Generalization of MASTER construct
- There is an implicit barrier at the end of the single directive
  - Can be skipped with the nowait clause
- Accept a new clause: COPYPRIVATE.
  - Broadcast the value of the variable in the copyprivate clause to the other threads when exiting the single directive
- Also accept private and firstprivate clauses

An example datarace

```
#include <stdio.h>

int main()
{
    float s;

    #pragma omp parallel default(none)
shared(s)
    {
        #pragma omp single
        {
            s=1.;
        }
        printf("s = %f\n",s);
        s=2.;
    }
    return 0;
}
```

An example datarace

```
#include <stdio.h>

int main()
{
    float s;

    #pragma omp parallel default(none)
shared(s)
    {
        #pragma omp single nowait
        {
            s=1.;
        }
        printf("s = %f\n",s);
        s=2.;
    }
    return 0;
}
```

```
$ gcc –fopenmp –o main.c

$  OMP_NUM_THREADS=16 ./a.out
s = 1.000000
s = 1.000000
s = 1.000000
s = 1.000000
s = 1.000000
s = 1.000000
s = 1.000000
s = 1.000000
s = 1.000000
s = 1.000000
s = 1.000000
s = 1.000000
s = 2.000000
s = 2.000000
s = 1.000000
s = 2.000000
```