

OpenMP

Mini-introduction

Les bases

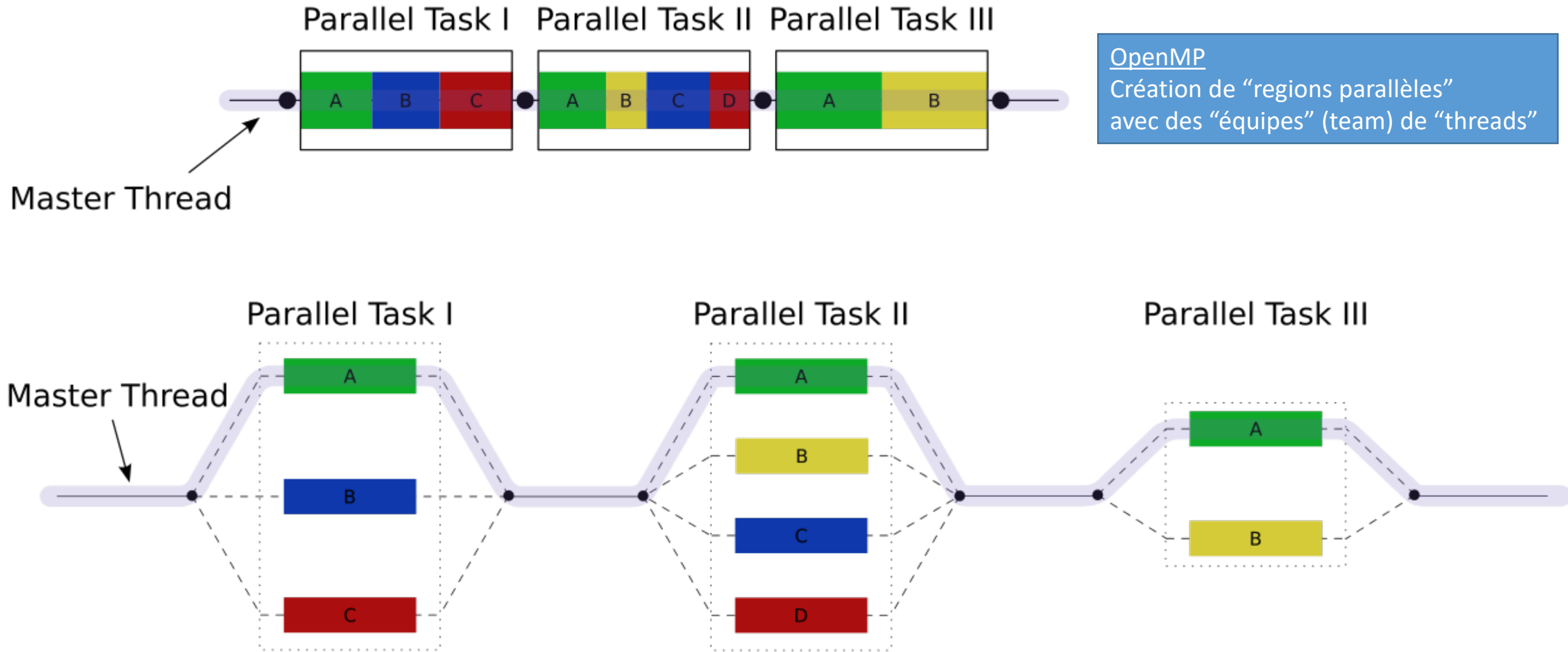
OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran,^[3] on many platforms, instruction-set architectures and operating systems, including Solaris, AIX, FreeBSD, HP-UX, Linux, macOS, and Windows.

It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.^{[2][4][5]}

OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.

An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI), such that OpenMP is used for parallelism *within* a (multi-core) node while MPI is used for parallelism *between* nodes. There have also been efforts to run OpenMP on software distributed shared memory systems,^[6] to translate OpenMP into MPI^{[7][8]} and to extend OpenMP for non-shared memory systems.^[9]

An illustration of [multithreading](#) where the primary thread forks off a number of threads which execute blocks of code in parallel



Thread creation

The pragma `omp parallel` is used to fork additional threads to carry out the work enclosed in the construct in parallel. The original thread will be denoted as *master thread* with thread ID 0.

Example (C program): Display "Hello, world." using multiple threads.

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

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

```
int main(void)
```

```
{
```

```
    #pragma omp parallel
```

```
    printf("Hello, world.\n");
```

```
    return 0;
```

```
}
```

OpenMP compiler "directives"

Use flag `-fopenmp` to compile using GCC:

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

Output on a computer with two cores, and thus two threads:

```
Hello, world.
```

```
Hello, world.
```

However, the output may also be garbled because of the [race condition](#) caused from the two threads sharing the [standard output](#).

```
Hello, wHello, woorld.
```

```
rld.
```

Whether `printf` is atomic depends on the underlying implementation^[15] unlike C++'s `std::cout`.

```

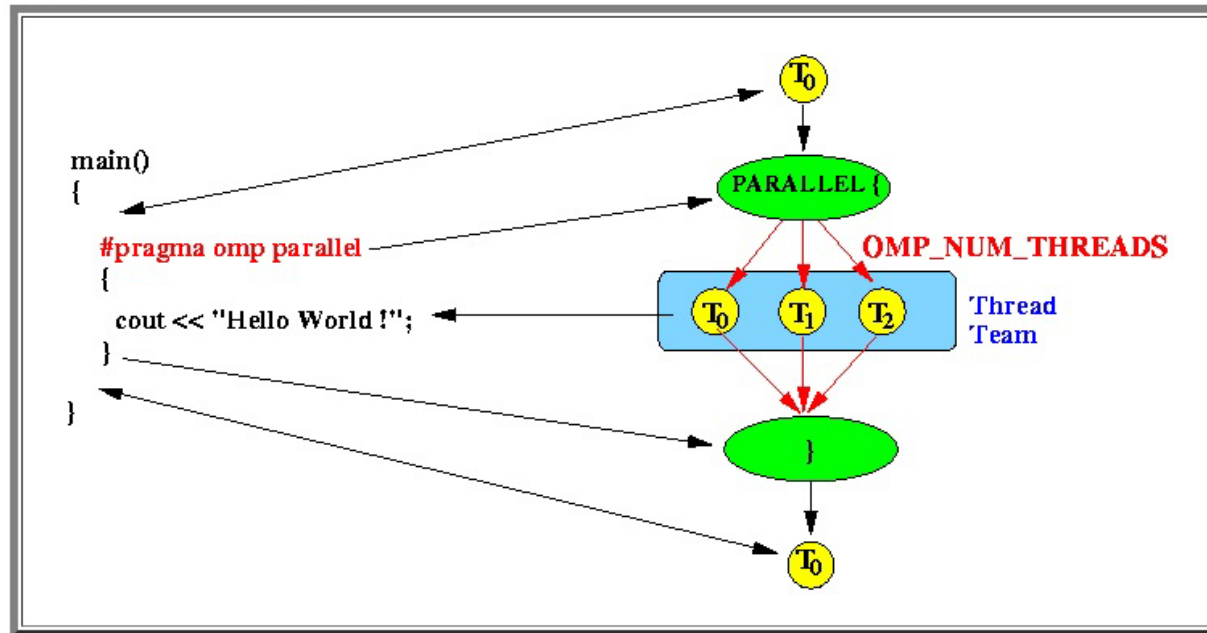
#include <iostream>    // NO: .h !!!
#include <omp.h>

using namespace std;    // You need to use this namespace

int main(int argc, char *argv[])
{
    #pragma omp parallel
    {
        cout << "Hello World !!!" << endl;
    }
}

```

- What is happening in an OpenMP program ?



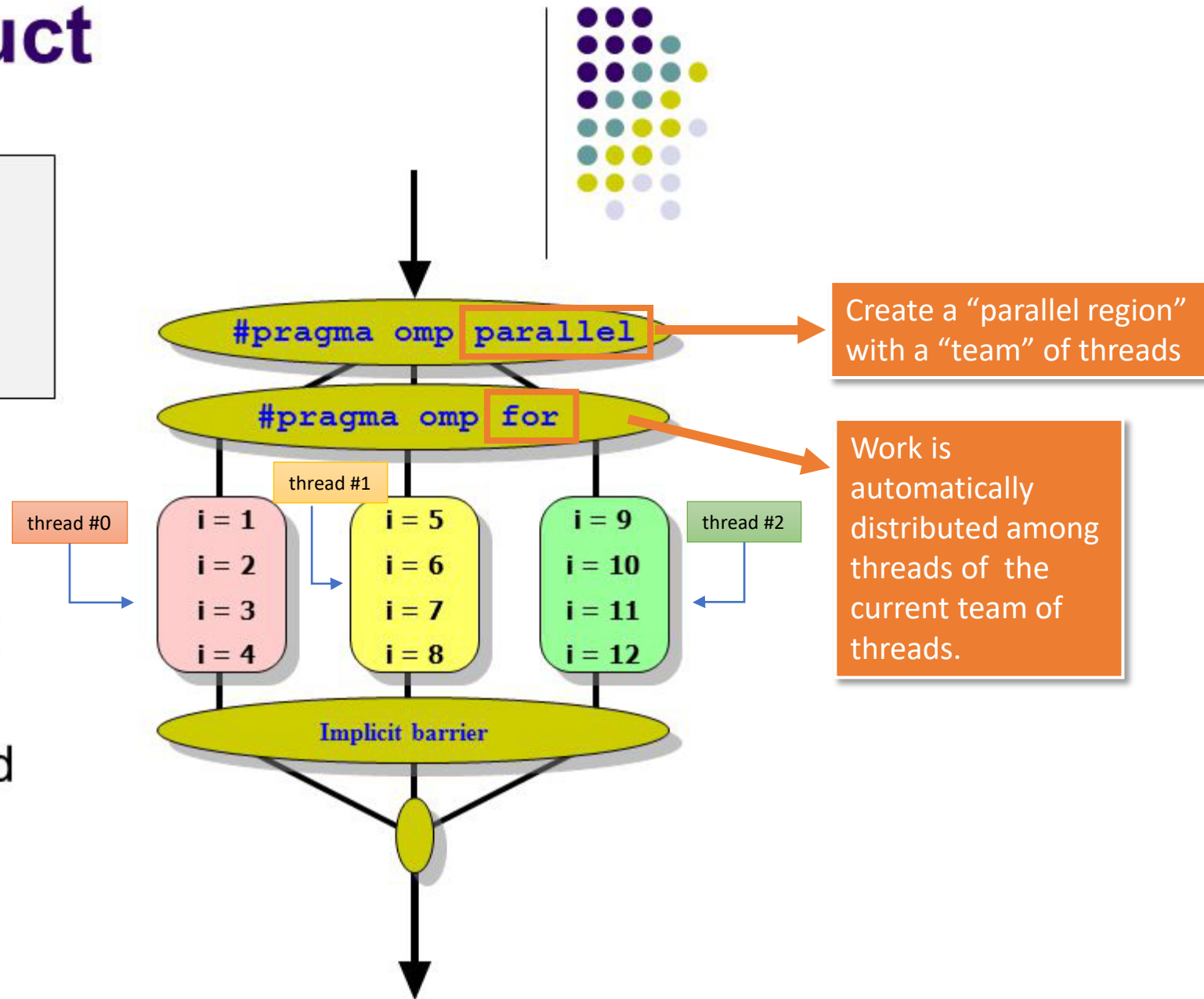
- Before the directive `#pragma omp parallel` the program is *single-threaded* - this thread is also known as the **MASTER THREAD**
- The directive `#pragma omp parallel` tells the compiler to create a thread team. The number of threads in the team is determined by the *environment variable* **OMP_NUM_THREADS**
- **At the end of the parallel section**, ALL threads are *implicitly joined* (i.e., the master thread calls "pthread_join" on all worker threads)

Then the **master thread** T_0 continue with the *single-threaded* execution **following the PARALLEL section**.

“omp for” construct

```
// assume N=12
#pragma omp parallel
#pragma omp for
    for(i = 1, i < N+1, i++)
        c[i] = a[i] + b[i];
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct



Parallel region: max number of threads

```
#pragma omp parallel
{
    // Code inside this region runs in parallel.
    printf( "Hello!\n" );
}
```

Parallel region: explicitly set the number of threads in the team of “this” parallel region

```
#pragma omp parallel num_threads(2)
{
    // Code inside this region runs in parallel.
    printf( "Hello!\n" );
}
```

Parallel region: explicitly set the number of threads in teams in future parallel regions

```
// Set the number of threads
omp_set_num_threads( nbThreads );

#pragma omp parallel
{
    // Code inside this region runs in parallel.
    printf( "Hello!\n" );
}
```

Note : il existe aussi une variable d'environnement OpenMP (configurable) qui fixe le nombre maximum de threads.
OMP_NUM_THREADS=...

OpenMP : “Shared-Memory” Model

(mémoire partagée)

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



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

La variable “n” est “partagée” par tous les threads
⇒ variable globale partagée si en dehors de la region parallèle
⇒ la variable “n” de la boucle sera incrémentée par tous les threads en parallèle, ainsi chaque chaque thread n’effectuera qu’une partie de la boucle, voire même rien du tout si “n” a déjà atteint le maximum (n=10)
⇒ ATTENTION => “race condition” potentielle sur “n” !

La variable “n” est “privée” (*private*) pour chaque thread
⇒ locale
⇒ chaque thread a une copie indépendante de “n”, et effectue la boucle “for” complète, soit 10 fois
⇒ le mot clef “private” permet de créer une variable locale privée par thread.

NOTE : si la variable est déclarée dans la region parallèle, elle est locale et automatiquement privée pour chacun des threads. Exemple :

```
#pragma omp parallel  
{  
    int n;    // private variable  
}
```


OpenMP library “routines”

Comme en Python, en C++, etc..., on a accès au nombre de threads, aux identifiants de threads, etc...

```
// OpenMP
//
// g++ -std=c++0x TD1_main.cpp -o TD1_main_program -lpthread -fopenmp
//
// export OMP_NUM_THREADS=5
```

```
#pragma omp parallel num_threads(4)
{
    // Parallel region code
    printf( "Hello World... from thread = %d\n", omp_get_thread_num() );
}
```

OpenMP compiler “directives”

OpenMP library “routines”

```
int nthreads, tid;

// Begin of parallel region
#pragma omp parallel private(nthreads, tid)
{
    // Getting thread number
    tid = omp_get_thread_num();
    printf( "Welcome to GFG from thread = %d\n", tid );

    if ( tid == 0 ) {
        // Only master thread does this
        nthreads = omp_get_num_threads();
        printf( "Number of threads = %d\n", nthreads );
    }
}
```

OpenMP compiler “directives”

OpenMP library “routines”

OpenMP

On peut faire bien plus avec OpenMP (ex: synchronisations, régions parallèles d'équipes de threads imbriquées, réductions, ...)

On a juste vu les notions de base.