Introduction to OpenMP

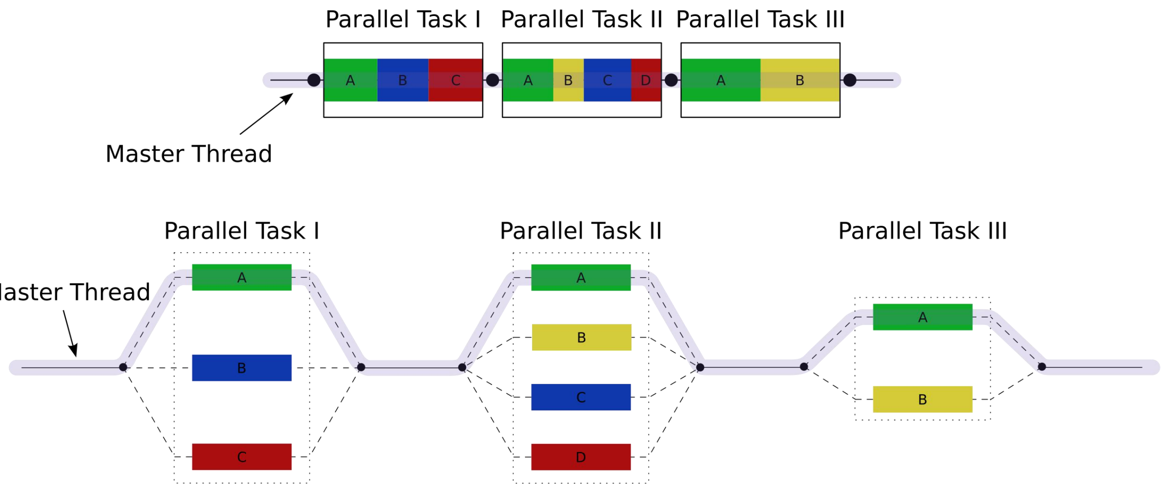
Dr. Xiaodong LIU

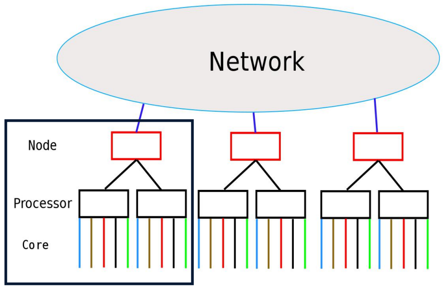
What is OpenMP?

Open Multi-Processing :
An API parallel programming in C, C++, and Fortran

Key features :

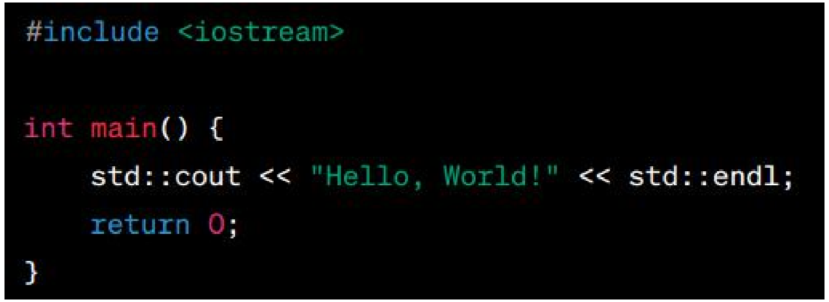
- Compiler Directives
- Shared Memory Model
- Data Scope
- Worksharing Constructs
- Synchronization
- Portability





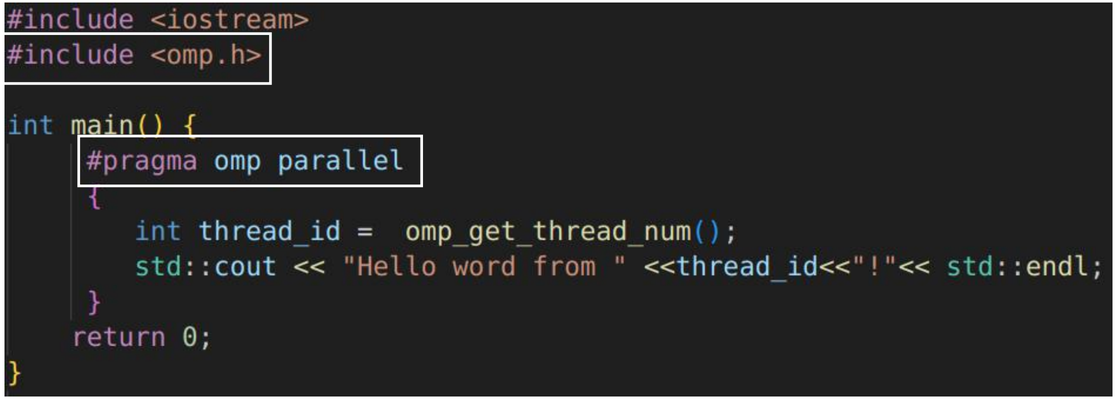
https://en.wikipedia.org/wiki/Fork%E2%80%93join_model

Directives



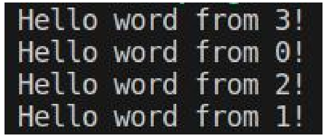
```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

#pragma omp **Directives:**
parallel, for, master, single,
sections, critical, barrier,
atomic, task,
parallel for



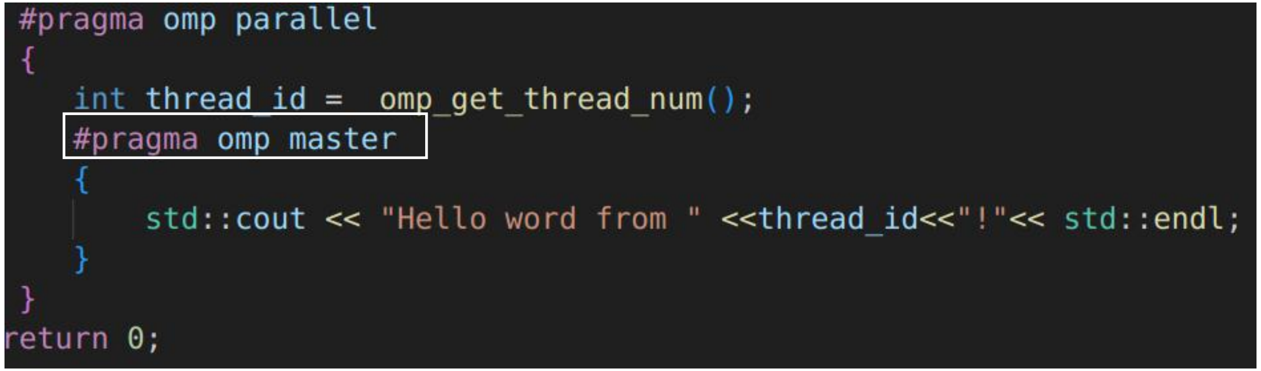
```
#include <iostream>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        int thread_id =  omp_get_thread_num();
        std::cout << "Hello word from " <<thread_id<<"!"<< std::endl;
    }
    return 0;
}
```

compile:
g++ code.cpp -fopenmp -o a.out
run:
export OMP_NUM_THREADS=4
./a.out

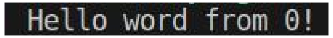


```
Hello word from 3!
Hello word from 0!
Hello word from 2!
Hello word from 1!
```

Master & Single

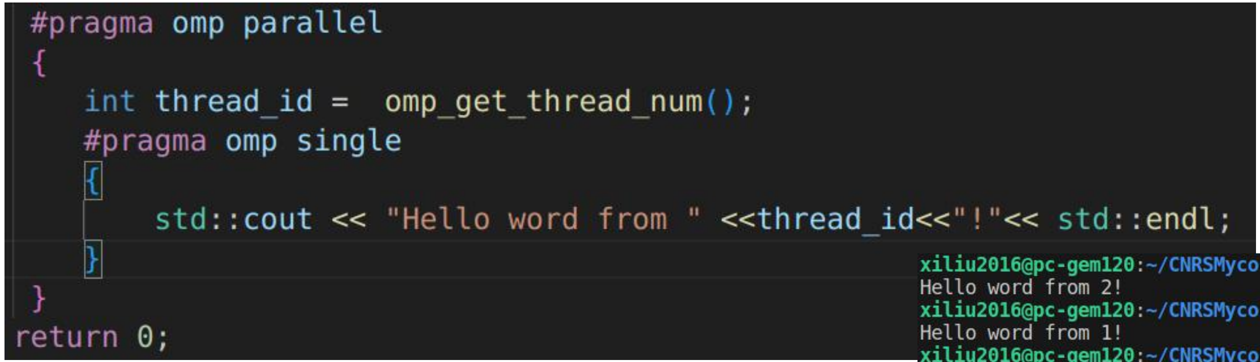


```
#pragma omp parallel
{
    int thread_id =  omp_get_thread_num();
   #pragma omp master
    {
        std::cout << "Hello word from " <<thread_id<<"!"<< std::endl;
    }
}
return 0;
```



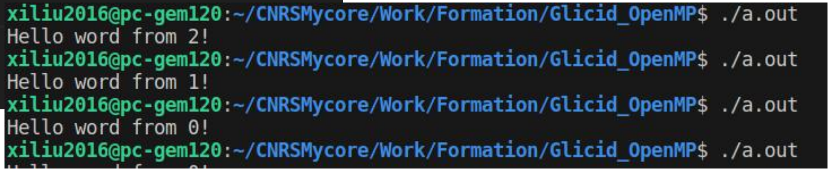
```
Hello word from 0!
```

Master
The block is executed only by the master thread



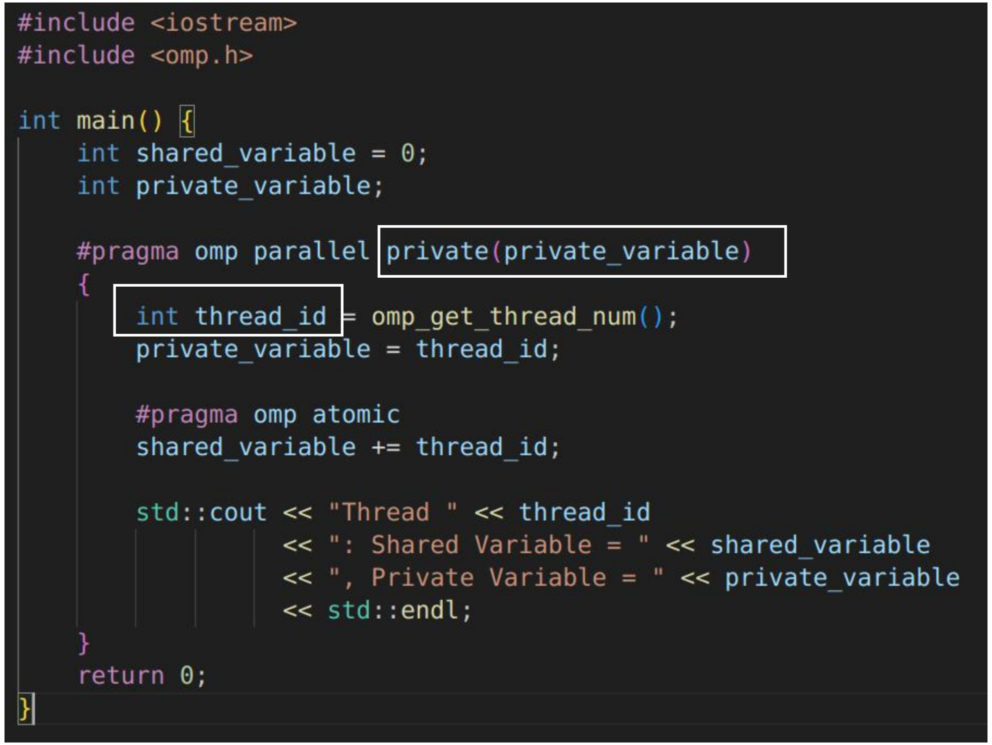
```
#pragma omp parallel
{
    int thread_id =  omp_get_thread_num();
    #pragma omp single
    {
        std::cout << "Hello word from " <<thread_id<<"!"<< std::endl;
    }
}
return 0;
```

Single
The block is executed on a single thread, not necessarily the main thread.

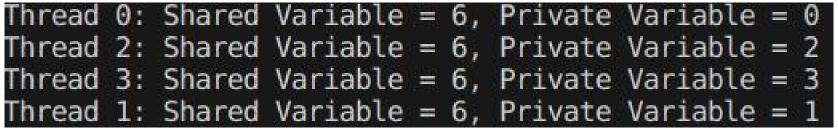


```
xiliu2016@pc-gem120:~/CNRSMycore/Work/Formation/Glicid_OpenMP$ ./a.out
Hello word from 2!
xiliu2016@pc-gem120:~/CNRSMycore/Work/Formation/Glicid_OpenMP$ ./a.out
Hello word from 1!
xiliu2016@pc-gem120:~/CNRSMycore/Work/Formation/Glicid_OpenMP$ ./a.out
Hello word from 0!
xiliu2016@pc-gem120:~/CNRSMycore/Work/Formation/Glicid_OpenMP$ ./a.out
```

Clauses



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

int main() {
    int shared_variable = 0;
    int private_variable;

    #pragma omp parallel private(private_variable)
    {
        int thread_id = omp_get_thread_num();
        private_variable = thread_id;

        #pragma omp atomic
        shared_variable += thread_id;

        std::cout << "Thread " << thread_id
                  << ": Shared Variable = " << shared_variable
                  << ", Private Variable = " << private_variable
                  << std::endl;
    }
    return 0;
}
```



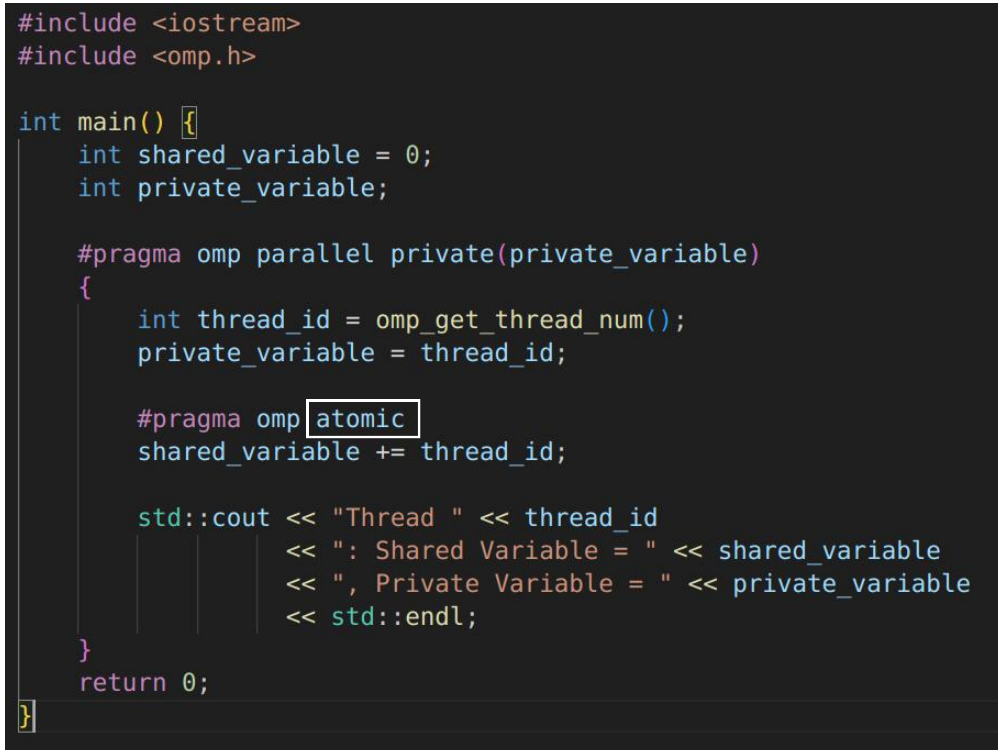
```
Thread 0: Shared Variable = 6, Private Variable = 0
Thread 2: Shared Variable = 6, Private Variable = 2
Thread 3: Shared Variable = 6, Private Variable = 3
Thread 1: Shared Variable = 6, Private Variable = 1
```

#pragma omp **clause (val):**
private(variable_list),
shared(variable_list),
firstprivate(variable_list),
reduction(operator:variable_list)

shared variable is by **defaut** when defined **outside** the parallel region

Thread ID starts by 0

Atomic



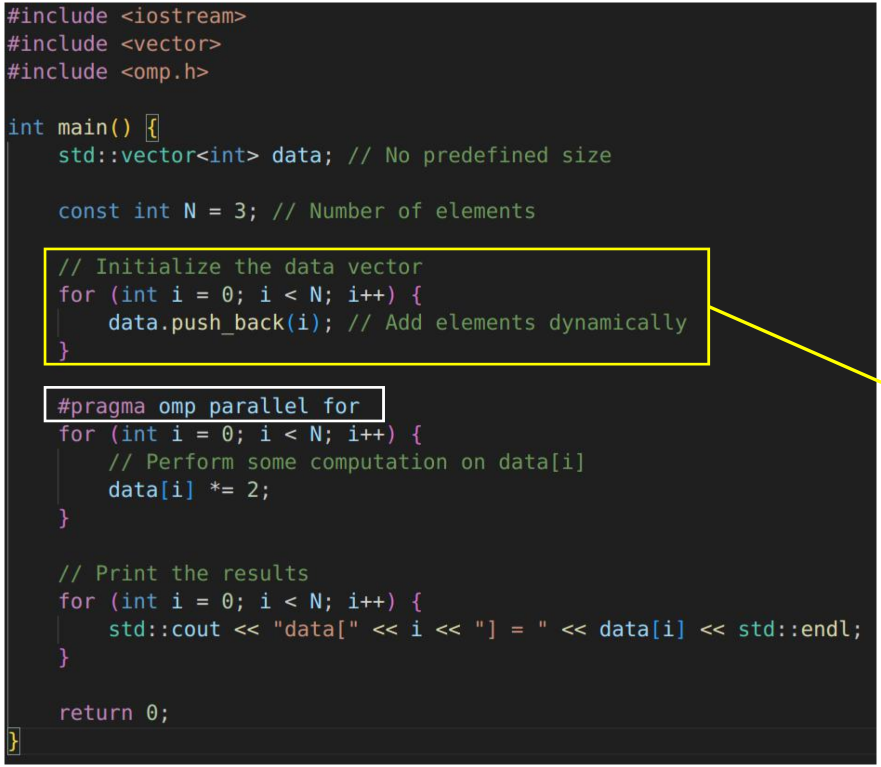
```
#include <iostream>
#include <omp.h>

int main() {
    int shared_variable = 0;
    int private_variable;

    #pragma omp parallel private(private_variable)
    {
        int thread_id = omp_get_thread_num();
        private_variable = thread_id;

        #pragma omp atomic
        shared_variable += thread_id;

        std::cout << "Thread " << thread_id
                  << ": Shared Variable = " << shared_variable
                  << ", Private Variable = " << private_variable
                  << std::endl;
    }
    return 0;
}
```

Perform simple atomic updates on shared variables.
modifications atomically

Critical

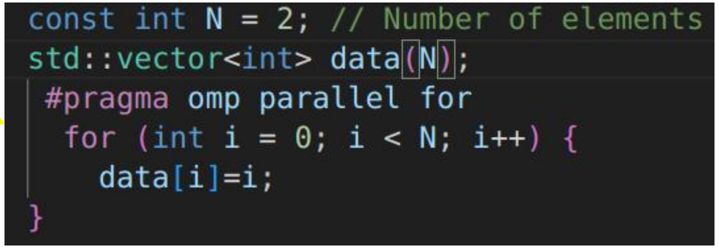
Specifies a code block that is restricted to access by only one thread at a time.

Loop parallelization



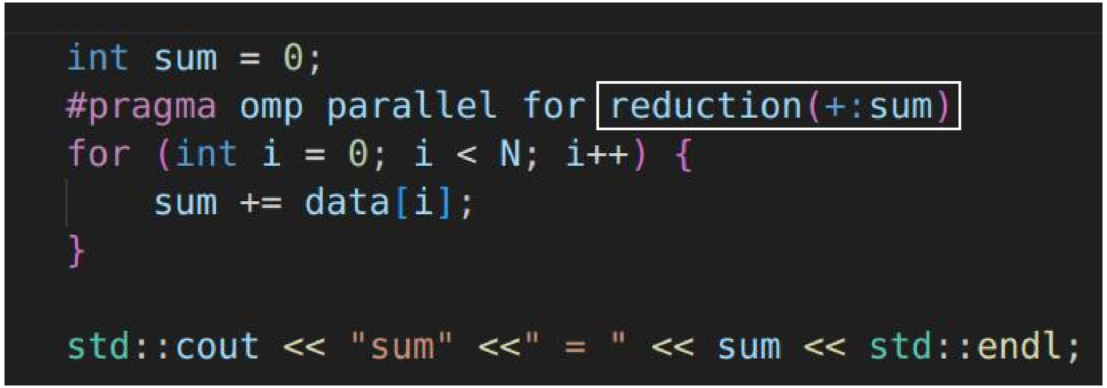
```cpp
#include <iostream>
#include <vector>
#include <omp.h>

int main() {
    std::vector<int> data; // No predefined size

    const int N = 3; // Number of elements

    // Initialize the data vector
    for (int i = 0; i < N; i++) {
        data.push_back(i); // Add elements dynamically
    }

    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        // Perform some computation on data[i]
        data[i] *= 2;
    }

    // Print the results
    for (int i = 0; i < N; i++) {
        std::cout << "data[" << i << "] = " << data[i] << std::endl;
    }

    return 0;
}
```

Each thread will execute a portion of the loop's iterations



```cpp
const int N = 2; // Number of elements
std::vector<int> data(N);
 #pragma omp parallel for
  for (int i = 0; i < N; i++) {
    data[i]=i;
}
```

Reduction



```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; i++) {
    sum += data[i];
}

std::cout << "sum" <<" = " << sum << std::endl;
```

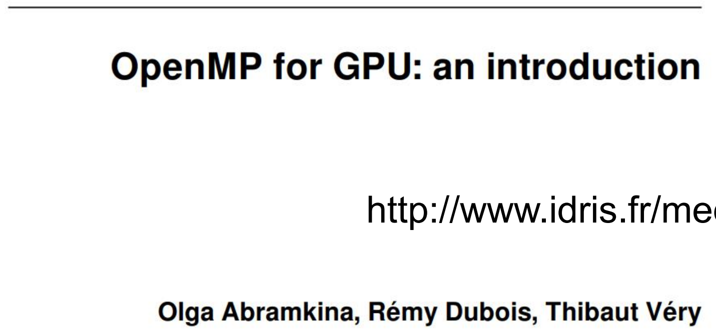
Each thread computes its part of the sum, and the final result is automatically combined at the end of the parallel region.

Others operators: max, min, *, &, |, ^

You are thirsty?

https://www.openmp.org

http://www.idris.fr/formations/openmp/



OpenMP for GPU: an introduction

http://www.idris.fr/media/formations/openacc/openmp_gpu_idris_c.pdf

Olga Abramkina, Rémy Dubois, Thibaut Véry