Advanced Methods for Scientific Computing (AMSC)
Lecture title: Introduction to OpenMP

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.Y. 2024/2025

What is OpenMP?

OpenMP is an application interface for shared memory MIMD programming. It has been designed to facilitate incremental parallelization of a serial code (even if this way does not necessarily lead to the best performance.)

It uses special compiler directives, so it requires compiler support.

A program written using OpenMP may also run in a scalar fashion without OpenMP no effort.

A note: The latest versions of OpenMP support also GPU programming. Here an article on the subject.

Where can I find information on OpenMP?

The specification of the MPI standard (for all various versions) can be found in openmp.org, where you also find a lot of documentation, in particular reference guides.

The book by P.S.Pacheco and M. Malensek (suggested for this course) is also a great source of information (in chapter 5).

The courses of CINECA (the Italian supercomputing center), like this one, can also be a source of useful information (and they have been exploited for these slides).

How are OpenMP directives made

OpenMP parallelism is obtained by the use of directives and, additionally, some functions.

The openMP directives are the fundamental components of the OpenMP API. They are in fact directives to the compiler (pragma directives) of the form

#pragma omp <directive name> <clauses>

The # must be in the first column, directive name is the name of the directive, which can be further specified through clauses. A pragma directive must be one line long, If you want to split a long directive you need the backslash (\) character. OpenMP directives are ignored if `-fopenmp` is not used at compilation.

The OpenMP API is complemented by a set of functions, whose name starts always with omp_. You need to include <omp.h>.

Compilation

To activate OpenMP you need to use a special compiler option (of course the compiler should support OpenMP). On gnu and LLVM based C and C++ compilers the option to use (at compilation and linking stage) is `-fopenmp`. For instance

```
g++ -fopenmp a.cpp main.cpp -o main %compiling and linking
```

or

```
g++ -fopenmp -c a.cpp main.cpp %compilation
g++ -fopenmp a.o main.o -o main % linking
```

Of course, you can add all other compiler options you need.

Running an OpenMP program

Differently than MPI, there is no need of any special command to run a program compiler with OpenMP. You launch the program as usual: for instance just

```
./main
```

A first example

In Parallel/OpenMP/HelloWorld the code `main_hello_world_simple.cpp` presents a first example

```
#include <iostream>
intmain()
{
std::cout << "Hello World!" << std::endl;

#pragma omp parallel num_threads(8)
{
std::cout << "Hello World (this time in parallel!)" << std::endl;
}
return 0;
}
```

The directive **#pragma** omp parallel opens a parallel region where a team of threads is created by forking the master thread (the only one operating before and after the directive). The parallel section is the structured block (see later) following the omp parallel directive. The num_threads(8) clause indicates that we want 8 threads.

A first example

Inside the parallel region, the threads operate independently and execute the only statement here present. Indeed, you may note that the output is garbled, and possibly different at each call of the program.

At the exit of the parallel region, the threads rejoin into a single one. So just one thread is present when executing the **return** statement.

Structured block

In the OpenMP terminology a structured block is an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom, or an OpenMP construct. In practice, is either a single statement, or a set of statements enclosed into braces: {...}, or a while/for loop.

```
#pragma omp parallel
structured block (single statement)
```

```
#pragma omp parallel
{//structured block (multiple statements)
  ...
  ...
}
```

It is not allowed to exit a structured block from within. Possible exception, the use of std::exit() to close the program in case of a nonrecoverable error.

A slightly more complete example

In the same folder, you find

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

int main() {
int num_threads;
std::cout << "Type me the number of threads you want to use\n";
cin>>num_threads;
#pragma omp parallel private(thread_id) num_threads(num_threads) \
shared(n_threads, num_threads)
{
  thread_id = omp_get_thread_num();
#pragma omp critical
  std::cout << "Hello World from thread " << thread_id << "!" << std::endl;
#pragma omp barrier
#pragma omp single
  {
  n_threads = omp_get_num_threads();
  std::cout << " - message printed by thread n."<<omp_get_thread_num()<<" -\n"
  }
}
...
```

Let's look at it in detail.

A slightly more complete example

Now the number of threads is not fixed, but given by the user. The clauses `private()` and `shared()` introduce the variables that are shared among threads and those that are private, thus replicated for each thread. Specifying shared and private variables explicitly helps program readability and safety. The default is that variables defined before the directive are shared, those defined inside the structured block are private. The default may be changed with the clause `default(none)`.

The directive `omp critical`, indicates that the following structured block may be executed by a thread at a time: inside the block only one thread is active, `omp barrier` is a synchronization point: all threads in the parallel region wait until all have reached it. Finally, `omp single` introduces a block that is executed by only one thread, without specifying which. If I had used directive, `omp master` the master thread'd have executed the statement.

`omp_get_num_threads()` is an OpenMP function that returns the number of threads, `omp_get_thread_num()` returns the rank of the thread executing the critical region.

The _OPENMP macro variable

If you use only omp directives in your code you do not need to include the header omp.h, and your code will compile also if you do not use the `-fopenmp` option or you use a compiler that does not support OpenMP: the pragmas are ignored and you have just a scalar code.
It would be nice to maintain the same feature if you use OpenMP functions as well. The standard requires that if you compile with OpenMP the preprocessor variable _OPENMP be defined. You can the use it to perform conditional compilation:

```
#ifdef _OPENMP
#include <omp.h>
#endif
 ....
#ifdef _OPENMP
   num_threads = omp_get_num_threads();
#else
   num_threads = 1;
#endif
```

Parallel loops

One of the most common OpenMP directive is omp parallel for, used to perform a single for-loop in parallel

```
#pragma omp parallel for num_thread(8) shared(a,b,c)
for (int i=0; i<n; ++i)
{
   a[i]=b[i]+c[i];
}
```

A team of 8 threads execute the loop, by default splitting it into chunks of $n/8$ or $n/8+1$ elements, and assigning the first chunk to the first team and so on. The loop must have a number of iterations determined at run time. Constructs like

```
#pragma omp parallel for num_thread(8) shared(a,b,c)
for (int i=0;; ++i)
{
 if (something) break;
}
```

are not allowed: remember the requirements for a structured block.

pragma omp for and pragma parallel for

The omp parallel (and omp parallel **for**) directive creates a team of threads, that are joined back at the end of the parallel region. Forking and joining has a cost, we should avoid to do it unnecessarily. Look at this situation

```
#pragma omp parallel for
for (i=0,i<n;++i)....
#pragma omp parallel for
for (j=0,j<m;++i)..
```

compared with

```
#pragma omp parallel
{
#pragma omp for
for (i=0,i<n;++i)....
#pragma omp for
for (j=0,j<m;++i)..
}
```

The omp **for** directive tells that the current team of threads should execute the loop in parallel. So, in the latter case we avoid joining and forking thread twice. There is an implicit barrier at the end of the omp **for** block.

Race condition and data race

We have race condition when the timing or order of events affects the correctness of a piece of code and the parallel execution may make the order indetermined. Typically it is due to a data race: one thread accesses a shared resource while another thread is writing on it[1]. A classic example:

```
#pragma omp parallel for
for (int i=1; i<n;++i)
{
    a[i+1]=a[i]+b[i];
}
```

This code suffers a race condition: thread computing a[i] with $i = 4$ may run concurrently (or even before) that handling iteration $i = 3$. Indeed, the result is non-deterministic: different runs of the code may produce different results. An example is found in folder Parallel/OpenMP/DataRace/.

[1] The two terms are often confused

reduction

Consider the following scalar code

```
sum=0
for (int i=0;i<n;++i) sum +=a[i]+b[i];
```

We can solve the potential race in the parallel version as

```
 sum=0
#pragma omp parallel for shared(sum, a, b)
 for (int i=0;i<n;++i)
 {
#pragma omp critical
 sum +=a[i]+b[i];
 }
```

But, this way parallelization is lost, the code is even less efficient than the scalar version. The correct solution is to use the `reduction` clause:

```
#pragma omp parallel for shared(a,b) reduction(+:sum)
for (int i=0;i<n;++i) sum +=a[i]+b[i];
```

sum is set to zero entering the loop automatically.

The reduction clause

It's general form is[2]

```
reduction(operator : variable list)
```

where `operator` is a binary arithmetic or logical operator (+,-,*,&&,..), and *variable list* one or more reduction variables. In the presence of a reduction clause, the reduction variable(s) is replicated in each thread, and eventually all the threads' private copies are combined according to the given operator.

Specifying the operator is necessary to initialize the reduction variable(s) and perform the final reduction correctly (e.g., for a subtraction, the final operation of the partial results is a sum).

IN Parallel/OpenMP/PI you have an example that makes use of the reduction clause.

[2] In fact we may have another term, the reduction-modifier.

Variable scope

We have seen the private and shared clauses that define the scope of a variable in a OpenMP construct. We have other possibilities, let's list them

private. The default for a variable defined in a parallel region. A different copy is made for all thread. Initialization must be made inside the parallel region.

shared. The default for variable defined outside the parallel region. A shared variable is shared by all threads.

firstprivate. The variable is replicated in each thread, but we have a shared instance used for initialization:

```
int k=100;// shared instance
#pragma omp parallel for firstprivate(k) shared(a)
for (int i=0; i<n; ++i)
{    k=k+1;
     a[i]=k
}
//a = [101, 102, ...]
```

Variable scope

lastprivate. It can be used only in a `parallel for` directive. Also in this case, like in firstprivate, we link a private variable with a shared instance, but the role here is to store in the shared instance the last value of the private instances in a parallel for.

```
int k;// shared instance
#pragma omp parallel for firstprivate(k) shared(a)
for (int i=0; i<n; ++i)
{   k=a[i];
}
//k=a[n]
```

The default clause is used to specify the default scoping rule. If you use default(none) you are obliged to specify the scoping rule of all variables used in a parallel region. Often, it is done to avoid nasty errors.

Scheduling

In a parallel loop the way iterations are distributed among threads is a scheduling strategy, which can be modified by using the strategy clause, of the form

```
strategy(shedule[, chunk_size])
```

static is the default strategy: iterations are distributed approx. equally among threads in a round-robin fashion: the first n/p (or $n/p + 1$) iterations are assigned to thread 0, the second to thread 1 etc. in a cyclic fashion. If you specify the chunk size, the number of iterations distributed is as uniform as possible, with a maximum represented by the chunk size.
The static strategy is optimal if the computational cost of each iteration is the same. Since often it is the case, it is the default.

Scheduling

dynamic. Unless specified, chunk size is here equal to 1. A number of iterations equal to chunk size is assigned to an available thread. As soon as a thread completes its job is put back in the set of available tasks. Better suited if the cost is not uniform. The choice of the chunk size may be critical for best performance.

guided. The guided strategy is similar to dynamic. The chunk size is determined dynamically by the system and decreases with the iterations. If you specify chunk size, you are in fact specifying the maximal size of the chunk.
The guided strategy may be useful when the cost of each iteration may vary largely.

runtime. It is not a real strategy. It just indicates that the strategy is specified via the environmental variable `OMP_SCHEDULE`.

Scheduling

auto. Leave the system choose for you the scheduling strategy.

If the workload is not uniform the choice of the best scheduling strategy may be not obvious. Sometimes some experimentation is needed. In this case the runtime strategy is useful:

```
#pragma omp parallel for strategy(runtime)

...

...

export OMP_SCHEDULE=static,4 #try with static, chunk size 4
time ./myprog;
export OMP_SCHEDULE=dynamic,4 #try with dynamic, chunk size 4
time ./myprog;
```

In the folder Parallel/OpenMP/Scheduling you have an example that show the effect of different strategies.

critical sections, atomic and locks

OpenMP offers three ways to ensure mutual exclusive access to a portion of code. We have already seen omp critical. It is used to define a structured block that can be accessed by a thread at a time. However, look at this code,

```
#pragma omp parallel num_threads(8)
{
...
#pragma omp critical
  x += fun();
...
#pragma omp critical
  y *= b;
}
```

It is very inefficient, since openMP treats a critical region in a parallel block as a single region. A trhet entering the first critical region blocks a thread entering the second region, even if there is no need.
Solution: use named critical sections (see next slide)

named critical sections

```
#pragma omp parallel num_threads(8)
{
...
#pragma omp critical(one)
   x += fun();
...
#pragma omp critical(two)
   y *= b;
}
```

When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name.

Atomic directive

The atomic directive is a special critical section where you must have just one statement of the form

`var <op>= <expression>`

It uses some characteristics of the hardware to ensure mutual exclusive access to the variable `var`.

```
#pragma omp parallel num_threads(8)
{
 #pragma omp atomic
    x+=fun();
}
```

more than one atomic

However,

```
#pragma omp parallel num_threads(8)
{
...
#pragma omp atomic
x += fun();
...
#pragma omp atomic
 y*= b;
}
```

is inefficient, for the same reason for which the named critical sections were introduced.
Unfortunately, there are not named atomic directive, so in the previous example we cannot use atomic, we have to use named critical sections, omp critical(name).

Locks

locks are another mechanism of mutual exclusion, and they are normally used to allow only one thread at a time to access an object.
A lock is a variable of type omp_lock_t[3], and may be operated upon with 4 omp functions:
1. omp_init_lock(omp_lock_t* lock_p). Initializes a lock as unset. It should be called before a lock is used.
2. omp_set_lock(omp_lock_t* lock_p). Sets the lock. If the lock is already set the thread has to wait until the lock is unset.
3. omp_unset_lock(omp_lock_t* lock_p) Unsets the lock. Another thread is now able to set it.
4. omp_destroy_lock(omp_lock_t* lock_p). Uninitializes the lock.

[3] This is a *simple* lock, we omit the discussion of *nested* locks

The use of locks

Suppose we have a class that contain a member that can be updated and we want to make sure that only a thread at a time can update it:

```
class Foo{
public:
  Foo(){omp_int_lock(&mylock);};
  void update_c(){
   omp_set_lock(&mylock);
   ++counter;
   omp_unset_lock(&mylock);
 }

private:
  omp_lock_t mylock;
  int counter;
};
```

If I call update_c on a Foo object inside a parallel region I am sure that only one thread at a time updates the counter. In apscParallel/OpenMP/Locks you find a simple examples.

Critical, atomic, locks:when?

From the advices by P Pacheco and M Malensek.

1. Do not mix in the same parallel region `atomic` and `critical` directive that involve the same variable. Use only `atomic` if possible (it is more efficient) otherwise only `critical`.
2. Nesting mutual exclusion construct may be dangerous! This code causes a deadlock:

```
double f(double x double  & z){
#pragma omp critical
 z+=g(x);
  ...}

#pragma omp parallel
{
 #pragma omp critical
 y+= f(x,z)
 ...
```

The problem can be solved using named critical sections: nested critical sections should be named.

Critical, atomic, locks:when?

The solution

```
double f(double x double  & z){
#pragma omp critical (f)
 z+=g(x);
  ...}

#pragma omp parallel
{
 #pragma omp critical (outer)
 y+= f(x,z)
  ...
```

Why the deadlock in the unnamed case? Because of the way critical sections work: before entering a CS a task ask: is anybody in there? And indeed the nested section is occupied.... by itself! Giving names, the nested section is treated as a separate critical section.

Critical, atomic, locks:when?

3. Preferably use `locks` when the mutual exclusive access is associated to objects more than portions of code.
4. Use omp barrier directive whenever a synchronization point for all thread is needed. However, use it only when necessary, otherwise you are causing a useless overhead.
5. Use omp single directive to select portion of code that should be run by just one thread. Prefer it to omp master, the latter should be used only when the executing thread must be the master thread.

Tasks

The tools we have seen so far are effective to parallelize codes that have a fixed or predetermined number of parallel blocks or loop iterations to schedule across threads. When this is not the case, OpenMP since version 3.0 offers the possibility of creating tasks. Tasks are structured block of code introduced by the directive `omp task` inside a parallel region, but generally launched by only one thread of the team:

```
#pragma omp parallel
#pragma omp single
{
#pragma omp task
{
...
}
#pragma omp task
{
...
}
}
```

Tasks

When the parallel directive creates a team of threads one of the thread is in charge of creating task instances. The OpenMP runtime schedule the task code for execution. Different tasks will then be executed concurrently by the team of threads. We report in Parallel/OpenMP/Fibonacci a first example inspired by the code in P. Pacheco book, but made more "C++ style". It is a the parallel code that computes Fibonacci numbers using tasks and recursion.

A warning, the code is just an example of the use of tasks. It is highly inefficient, unless you have a machine with a high number of threads and you have to compute a large Fibonacci number. We comment here the main part of the code. In the README file more comments (as usual).

Excerpt from the Fibonacci code

```
 auto const & compute(long unsigned int n) {
  fibs.resize(n+1ul,0ul);
#pragma omp parallel num_threads(num_threads)
#pragma omp single
  fib(n);
  return fibs;
}

  auto fib(long unsigned int n){
 long unsigned int i{0ul};
 long unsigned int j{0ul};
 if(n<=1ul)
 {
   fibs[n]=n;
   return n;
 }
#pragma omp task shared(i) if (n>par_limit)
i = fib(n-1ul);
#pragma omp task shared(j) if (n>par_limit)
j = fib(n-2ul);
#pragma omp taskwait
fibs[n]= i+j;
return fibs[n];
}
```

Explanation

The compute() method is the driver of the parallel computation. It creates a team of threads with the omp parallel directive and delegates to a single thread to launch fib(n). The latter is a parallel and recursive implementation of the computation of the Fibonacci sequence, where the computation of F_{n-1} and F_{n-2} is assigned to 2 different tasks, which may be run in parallel by the given team of threads. When $n \leq 1$ we break the recursion.

The clause `if (n>par_limit)` expresses a conditional condition for the creation of the task, if not satisfied the code is simply executed by the current thread.

Before computing $F_n = i + j$, we need to be sure that the tasks have completed their job. The omp taskwait is used for that purpose: it is a synchronization barrier for the tasks.

Got it?

The single task entering fib(n) creates two parallel tasks, that launch two different instances of fib() which launch two tasks each, and so until $n \leq 1$.

Unfortunately, each task here does very little work, the overhead linked to thread creation and handling by the operative system is such that normally the parallel version is very inefficient.

A final note: The recursive implementation of the Fibonacci sequence computation is rather elegant and also close to the mathematical definition of the Fibonacci sequence! However, remember that recursive calls have an overhead, non-recursive implementations con be more efficient.

Another example of use of tasks

In Parallel/OpenMP/Sort you have a parallel implementation of the QuickSort algorithm for sorting vectors. Here, we use again recursion and tasks. However, differently from the previous example, the tasks have more work to do, and indeed for sufficiently large vectors the OpenMP implementation beats the code implemented in `std::sort()` (which is also quicksort), in both the serial and multithreaded version.

Moreover, you may also note in the code the different position of the taskwait directive: the QuickSort algorithm has better parallel characteristics than the computation of Fibonacci sequence.

Cache coherence and false sharing

In the book of P. Pacheco M Malensek there is a whole section dedicated to cache coherence and the so called false sharing. We do not go into details here, being rather technical, but we point out cache coherence can greatly reduce the efficiency of a memory shared parallel algorithm! And it may happen when active parallel tasks access nearby memory location.

The term false sharing is used to indicate when cache coherence forces threads to access the main memory for data that could instead be accessed in the cache memory! Since accessing main memory can be order of magnitude slower, false sharing can be a major drawback. Unfortunately, the analysis of the cache usage of an algorithm is complex and beyond the scope of these lectures.