OpenMP Examples - Part 1

Marco Ferretti, Mirto Musci

Dipartimento di Ingegneria Industriale e dell'Informazione
University of Pavia

November 26, 2018

Outline

Outline

OpenMP Syntax

- Most of the constructs of OpenMP are pragmas

```
#pragma omp construct [clause [clause] ...]
```

  - An OpenMP construct applies to a structural block
  - Usually enclosed by { }
- In addition:
  - Several `omp_<something>` function calls
  - Remember to include `omp.h`
  - Several `OMP_<something>` environment variables

Controlling OpenMP Behavior

- Function calls and matching environment variables
  - check the documentation for details

omp_set_num_threads(int)/omp_get_num_threads()

- Control the number of threads used for parallelization
- Must be called from sequential code
- Also can be set by OMP_NUM_THREADS variable

omp_get_thread_num()

- Get the current thread ID (0 for the master thread)

Measure time

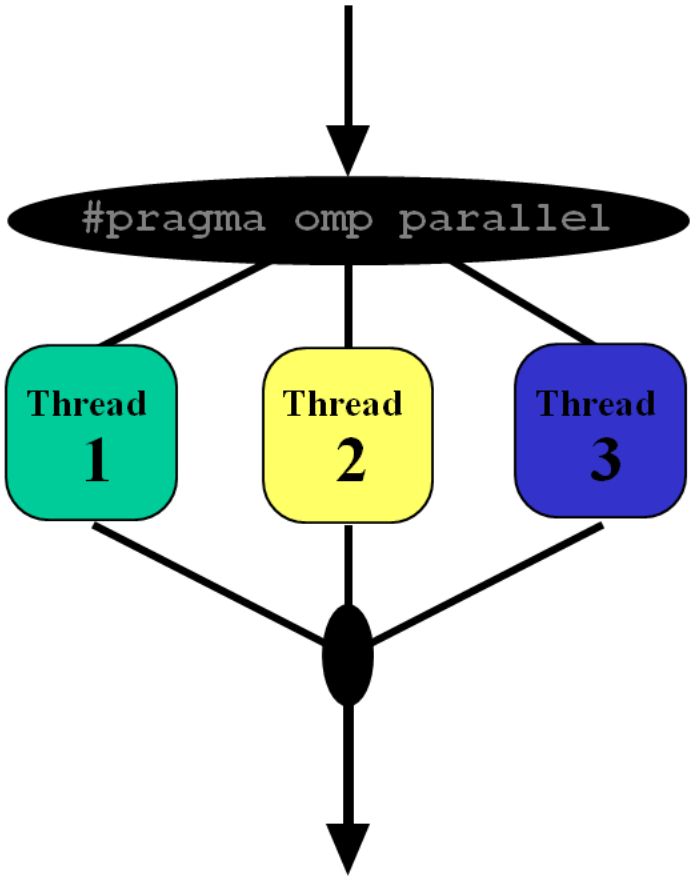
omp_get_wtime()

- A portable way to compute wall clock time
- It returns absolute values in seconds with no meaning by themselves (a timestamp)
- You must call it (at least) two times, store the values and compute a difference!
- Be careful of what you measure and remember Amdahl's law

Outline

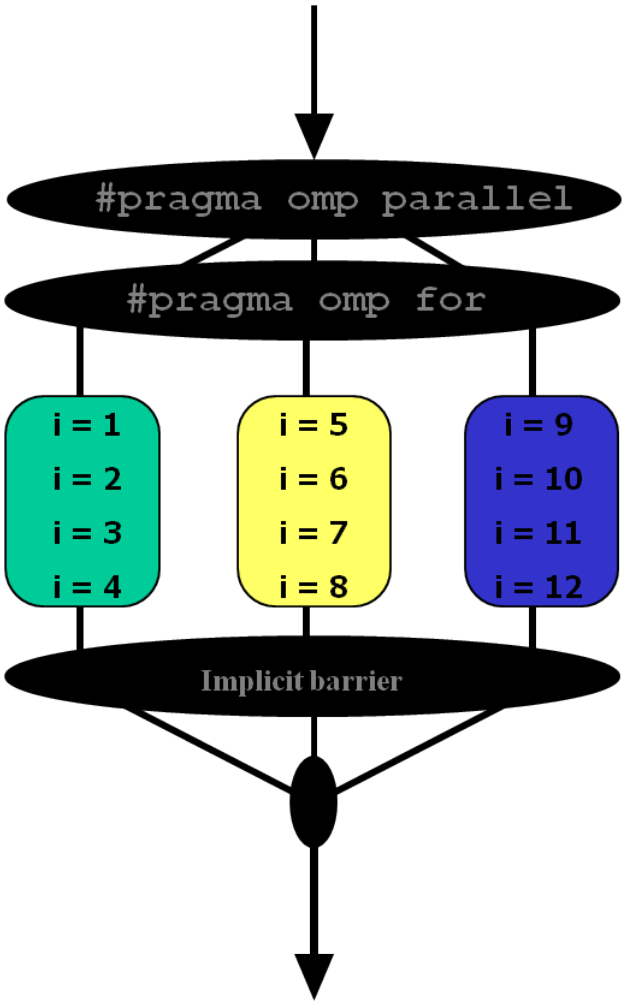
Parallel Regions

- #pragma omp parallel
- Defines a parallel region over structured block of code
- Threads are created as "parallel" pragma is crossed
- Every instruction inside is executed by each thread
- Threads wait at end of region (implicit barrier)



Work Sharing: For

- Used to assign each thread an independent set of iterations (chunks)
- Implicit barrier at the end
- Can combine the directives:
  - #pragma omp parallel for
- Only simple kinds of for loops
- Can control the distribution of chunks using theschedule"option"



Schedule Clause: Controlling Work Distribution

- schedule(static [, chunksize])
  - Default: chunks of circa the same size, one to each thread
  - If more chunks than threads: assigned using round-robin
- schedule(dynamic [, chunksize])
  - Threads receive chunk assignments dynamically
  - I.e. the first available thread gets a chunk
  - Default chunksize is one iteration!
    Best load balancing, worse granularity!

Work Sharing: Sections

- Easy way to parallelize independent computations!
- The first thread to encounter a section execute it.
- If more threads than section(s), they wait at the end

```
#pragma omp sections
{
#pragma omp section
  answer1 = long_computation_1();
#pragma omp section
  answer2 = long_computation_2();
}
if (answer1 != answer2) { ... }
```

Outline

Data Visibility

- Most variables are shared by default
  - Such as global variables
  - They can be modified concurrently by different threads!

```
{
  int sum = 0;
  #pragma omp parallel for
  for (int i=0; i<N; i++) sum += i;
}
```

- Some variables can be private, that is a local copy is created for each thread
  - Such as variables declared inside parallel regions
  - Variables can be explicitly declared as private

Declaring a variable as private

private clause:

- Override default behavior
- A copy of the variable is created for each thread.
- No connection between the original variable and the private copies

```
int i;

#pragma omp parallel for \
  private(i)

for (i=0; i<n; i++) { ... }
```

Outline

Single

#pragma omp single

- Only one thread will execute the following block of code
  - The rest will wait for it to complete (implicit barrier!)
  - Good for non-thread-safe regions of code (such as I/O)
  - Must be used in a parallel region

Master

#pragma omp master

- The block will be executed by the master thread only
- No implicit barrier
- Must be used in a parallel region

```
#pragma omp parallel
{
  do_preprocessing();

  #pragma omp single
  read_input();
  #pragma omp master
  notify_input_consumed();

  do_processing();
}
```

Critical Sections

- #pragma omp critical [name]
  - Only one thread at the time can execute the protected code
- Critical sections are global in the program
  - Can be used to protect a single resource in different functions
- Critical sections are identified by the name
  - All the unnamed critical sections are mutually exclusive
  - All the critical sections having the same name are mutually exclusive between themselves

```
int x = 0;
#pragma omp parallel shared(x)
{
  #pragma omp critical
  x++;
}
```

Atomic Execution

- Efficient critical sections
  - Protects a single variable update
  - Usually just a dedicated assembly instruction
- #pragma omp atomic
  update_statement
- Update statement is one of:
  - var= var op expr
  - var op= expr
  - var++ / var–.

Ordered

- #pragma omp ordered
  statement
- Executes the statement in the sequential order of iterations
- Example:

```
#pragma omp parallel for
for (j=0; j<N; j++) {
  int result = heavy_computation(j);
  #pragma omp ordered
  printf("computation(%d) = %d\n", j, result);
}
```

Barrier synchronization

- #pragma omp barrier
- Performs a barrier synchronization between all the threads in a team at the given point.
- Example:

```
#pragma omp parallel
{
  int result = heavy_computation_part1();
  #pragma omp atomic
  sum += result;
  #pragma omp barrier
  heavy_computation_part2(sum);
}
```

Reduction

```
for (j=0; j<N; j++) {
  sum = sum+func(j);
}
```

- To parallelize use the reduction clause!
  #pragma omp parallel for reduction(+: sum)
- In general can be any associative operation on the shared variable
  - sum, subtraction, moltiplication, logical or, etc.

Outline

Getting started

- Login on your machine
- Browse to the course website
  `http://www-5.unipv.it/mferretti/cdol/aca/`
- Click on lecture notes link
- Download the exercises source code
  - look for source code for examples
  - today we are going to use the lab1 file.

Work at home: Linux

Everything should work out of the box!

If not

- install the GNU compiler with your packet manager
  - OpenMP is supported since veryo old gcc versions
- or install almost any other compiler
- open the source files with your favorite editor (e.g. gedit)
- Start coding and remember to compile with `-fopenmp`!

Work at home: Windows

There are a lot of possibilites, for instance CodeBlocks

- Download the binary file from `http://www.codeblocks.org/downloads/26`
- Also download separately MinGW from `https://sourceforge.net/projects/mingw-w64/`
  - Paste mingw folder in CodeBlocks directory
- Launch CodeBlocks; clickCompiler undersettings menu
  - put `-fopenmp` in Other compiler options
  - put `-lgomp -pthread` in Other linker options
  - if doesn't work,manually link the `libgomp` library from MinGW folder
- Start coding! (but it was a little bit longer, wasn't it?)

Work at home: Mac

This only works for recent versions of Mac. I'm not an expert on this, please check online.

- Download Homebrew `https://brew.sh/index_it.html`
- Install it on terminal `/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"`
- Insert your admin password and follow instructions
- Install the latest `gcc` (it will take a few minutes)
  `brew install gcc --without-multilib`
- To compile, please use the proper command (i.e. `gcc-7`, if latest version is 7.X).

Outline

Exercise 1: Hello World

- Take a moment to examine the source code and note how OpenMP directives and library routines are being used.
- Use the following command to compile the code:

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

- To run the code, simply type the command `hello` and the program should run.
- How many threads were created? Why?

Exercise 1: Hello World

- Vary the number of threads and re-run Hello World
- Set the number of threads to use by means of the OMP_NUM_THREADS environment variable.

```
OMP_NUM_THREADS=4
```

- Do you know other ways to set the number of threads?
- Your output should look similar to below. The actual order of output strings may vary.

```
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 3
Hello World from thread = 1
Hello World from thread = 2
```

Exercise 2: Environment Information

- Starting from scratch, write a simple program that obtains information about your openMP environment.
  - Alternately, you can modify the "hello" program to do this.
- Using the appropriate openMP functions, have the master thread query and print the following:
  - The number of processors available
  - The number of threads being used
  - The maximum number of threads available
  - If you are in a parallel region
  - If dynamic threads are enabled
  - If nested parallelism is supported
- If you need help, consult the omp_getEnvInfo example file.

Exercise 3: Parallel For

- This example demonstrates use of the OpenMP for work-sharing construct.
- It specifies dynamic scheduling of threads and assigns a specific number of iterations to be done by each thread.
- After reviewing the source code, compile and run the executable. (Assuming OMP_NUM_THREADS still set to 4).

```
gcc -fopenmp omp_workshare1.c -o workshare1
workshare1 | sort
```

- Review the output. Note that it is piped through the sort utility. This will make it easier to view how loop iterations were actually scheduled.

Exercise 3: Parallel For

- Run the program a couple more times and review the output. What do you see?

- Typically, dynamic scheduling is not deterministic.
- Everytime you run the program, different threads can run different chunks of work.
- It is even possible that a thread might not do any work because another thread is quicker and takes more work.
- It might be possible for one thread to do all of the work.

Exercise 3: Parallel For

- Edit the workshare1 source file and switch to static scheduling.
- Recompile and run the modified program. Notice the difference in output compared to dynamic scheduling.
  - Specifically, notice that thread 0 gets the first chunk, thread 1 the second chunk, and so on.
- Rerun the program. Does the output change?
- With static scheduling, the allocation of work is deterministic and should not change between runs.
  - Every thread gets work to do.
- Reflect on possible performance differences between dynamic and static scheduling.

Exercise 4: Sections

- This example demonstrates use of the OpenMP sections work-sharing construct.
- Note how the parallel region is divided into separate sections, each of which will be executed by one thread.
- As before, compile and execute the program after reviewing it.

```
gcc -openmp omp_workshare2.c -o workshare2
workshare2
```

- Run the program several times and note differences in output.

Exercise 4: Sections

- Because there are only two sections, you should notice that some threads do not do any work.
- You may/may not notice that the threads doing work can vary.
  - For example, the first time thread 0 and thread 1 may do the work, and the next time it may be thread 0 and thread 3.
- Which thread does work is non-deterministic in this case.
  - It is even possible for one thread to do all of the work!

Exercise 5: Orphan Directive

- This example computes a dot product in parallel.
- It differs from previous examples because the parallel loop construct is orphaned
  - It's contained in a subroutine outside the main program's parallel region.
- After reviewing the source code, compile and run the program

```
gcc -fopenmp omp_orphan.c -o orphan
orphan | sort
```

- Note the result...and the fact that this example will come back as omp_bug6 later!

Exercise 6: Matrix Multiply

- This example performs a matrix multiply by distributing the iterations of the operation between available threads.
- After reviewing the source code, compile and run the program

```
gcc -fopenmp omp_mm.c -o matmult
```

- Review the output. It shows which thread did each iteration and the final result matrix.
- Run the program again, however this time sort the output to clearly see which threads execute which iterations:

```
matmult | sort | grep Thread
```

- Do the loop iterations match the schedule(static, chunk) clause for the matrix multiple loop in the code?

Outline

When things go wrong...

- There are many things that can go wrong when developing OpenMP programs.
- The omp_bugX.X series of programs demonstrate just a few.
- See if you can figure out what the problem is with each case and then fix it.
- The buggy behavior will differ for each example. Some hints are provided in the next slide.
- More in details explanations are provided in the following.
- Don't cheat!

Hints

Code	Hint
omp_bug1	Fails compilation. Solution provided.
omp_bug2	Thread identifiers are wrong. Wrong answers.
omp_bug3	Run-time error, hang.
omp_bug4	Causes a segmentation fault. Script provided.
omp_bug5	Program hangs. Solution provided.
omp_bug6	Failed compilation.

Explanations: omp_bug1

- The exercise shows the combined parallel for directive.
  - Fails compilation because the loop doesn't come immediately after the directive.
- Corrections include removing all statements between the parallel for directive and the actual loop.
- Logic is added to preserve the ability to query the thread id and print it from inside the loop.
  - But it is not necessary: simpler solutions work too
  - Notice the use of the firstprivate clause to intialize the flag.

Explanations: omp_bug2

- The bugs in this case are caused by neglecting to scope the tid and total variables as private.
  - By default, most OpenMP variables are scoped as shared.
  - These variables need to be unique for each thread.
- Alternatively, you could solve the exercise considering total to be shared.
  - In this case, you should protect access to total
  - Move the initialization to zero outside the loop or use a barrier
  - Use a reduction in the for loop

Explanations: omp_bug3

- The run time error is caused by by the omp barrier directive in the print_results subroutine.
- By definition, an omp barrier can not be nested outside the static extent of a sections directive.
- In this case it is orphaned outside the calling sections block.
  - You need to remove something...

Explanations: omp_bug4

- OpenMP thread stack size is implementation dependent
  - It represent how much "space" each thread has to store private variables
  - In this case, the array is too large to fit into the thread stack space and causes the segmentation fault.
- Solution provided - note that it is a script and will need to be "sourced".
  - For example: `source omp_bug4fig`.
  - It only works on Linux...
  - Be sure to examine the solution file to see what's going on.
  - In the last line you may need to change the name of the executable to match yours.
- Alternatively: avoid making an array a private variable!

Explanations: omp_bug5

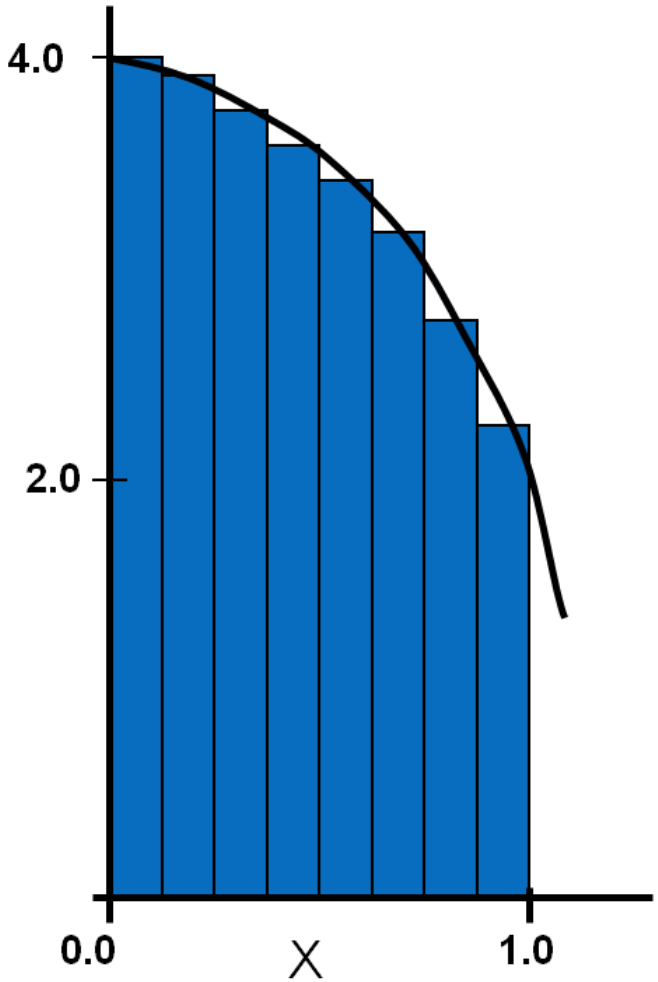
- The problem is that the first thread acquires locka and then tries to get lockb before releasing locka .
- Meanwhile, the second thread has acquired lockb and then tries to get locka before releasing lockb .
- The solution overcomes the deadlock by using locks correctly.
- Remember: avoid using explicit locking!
  - This is what is actually implemented "behind the scene" when using a critical section.
  - Try to implement the exercise using critical...

Explanations: omp_bug6

- With orphaned directives, correct scoping is critical.
  - The sum variable is scoped incorrectly.
- See the omp_orphan routine in the "Basic" section for one example of correct scoping.
- Note that there are other ways to solve the exercise
  - You can rework the code avoiding orhpan directives altogether...

Outline

Numerical Integration



- Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

- We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\,\Delta x \approx \pi$$

- Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial Code

```
static long num_steps=100000;
double step, pi;

void main()
{   int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;




    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

- Parallelize the numerical integration code using OpenMP
- What variables can be shared?
- What variables need to be private?
- What variables should be set up for reductions?

Parallel Code

```
static long num_steps=100000;
double step, pi;

void main()
{  int i;
   double x, sum = 0.0;

   step = 1.0/(double) num_steps;

#pragma omp parallel for \
      private(x) reduction(+:sum)

   for (i=0; i< num_steps; i++){
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0 + x*x);
   }
   pi = step * sum;
   printf("Pi = %f\n",pi);
}
```

- Parallelization code is a one-liner!
- sum is a reduction, hence shared, variable
- i is private since it is the loop variable

Assignment

Modify the PI calculation example, so you can:

- vary, at run time, the number of steps
  - Will the calculated pi value change?
- get the total time for the calculation using omp_get_wtime
- Implement the computational core in a separate function, and call it with different number of threads
  - Observe differences in elapsed time
  - What happens if you use more threads than available processors?
- Advanced: try to implement it without the reduction clause
  - it is slower? faster?

For Further Reading


Blaise Barney
OpenMP Exercise, 2011
https://computing.llnl.gov/tutorials/openMP/exercise.html