PAR Laboratory Assignment
Lab 2: Brief tutorial on OpenMP programming model

Ll. Àlvarez, E. Ayguadé, J. R. Herrero, J. Morillo, J. Tubella, G. Utrera and Chenle Yu

Fall 2019-20

Contents

Note:

- All files necessary to do this laboratory assignment are available in `/scratch/nas/1/par0/sessions/lab2.tar.gz`. Copy the compressed file from that location to your home directory in `boada.ac.upc.edu` and uncompress it with this command line: `"tar -zxvf lab2.tar.gz"`.

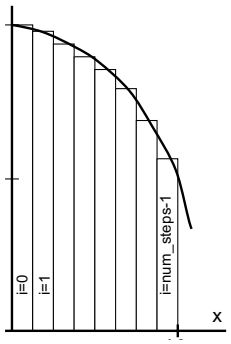
1

A very practical introduction to OpenMP (Part I)

This laboratory assignment has been prepared with the purpose of introducing the main constructs in the OpenMP extensions to the C programming language. In each of the two sessions you will first go through a set of different code versions (some of them not correct) for the computation of number Pi in parallel; and then you will be presented with a set of very simple examples that will be helpful to practice the main components of the OpenMP programming model being introduced. We ask you to fill in the questionnaire in the deliverable part for this second laboratory assignment. The session will finish observing the overheads introduced by the use of different synchronisation constructs in `OpenMP`.

1.1 Computing number Pi

In the documentation for the previous laboratory assignment we already showed how to compute the number Pi using numerical integration. To distribute the work for the parallel version each processor will be responsible for computing some rectangles in Figure 1.1 (in other words, to execute some iterations of the loop that traverses those rectangles). The parallelisation should also guarantee that there are no data races when accessing to variable `sum` in order to accumulate the contribution of each rectangle.




```
static long num_steps = 100000;
void main () {
    double x, pi, step, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (long int i=0; i<num_steps; ++i) {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

Figure 1.1: Pi computation

In order to parallelise the sequential code we will proceed through a set of versions `"pi-vx.c"` inside the `lab2/pi` directory, being `x` the version number. We provide two entries in the `Makefile` to compile them: `"make pi-vx-debug"` and `"make pi-vx-omp"`. The binary generated with the `"make pi-vx-debug"` option prints which iterations are executed by each thread and the value of Pi that is computed, which will be useful to understand what the program is doing. You can use the script `run-debug.sh` to interactively execute the binary with a very small input value (set to 32 in the script), just by typing `"./run-debug.sh pi-vx"`. In order to time the parallel execution you need to choose the second option `"make pi-vx-omp"` and queue the execution of the binary with a much larger input value (set to 100000000 in the script), just by typing `"qsub -l execution ./submit-omp.sh pi-vx"`).

You can always interactively run this version using the `run-omp.sh` script, but as you know this is not recommend for timing purposes. Finally, to instrument the execution with `Extrae` and visualize the parallel execution with `Paraver` you will need to submit for execution the `submit-extrae.sh` script, which sets the input to a smaller value (set to 100000 in the script), just by typing `"qsub -l execution ./submit-extrae.sh pi-vx"`.

1.2 Parallelisation with OpenMP

1.2.1 Defining the parallel region

1. Compile and run the initial sequential code `pi-v0.c`. This initial version introduces the use of `omp_get_wtime` runtime call to measure wall clock execution time. The result computed for Pi as well as the execution time for this version will be taken as reference for the other versions.

2. In a first attempt to parallelise the sequential code, `pi-v1.c` introduces the `parallel` construct, which creates/activates the team of threads. With `parallel` all threads in the team created/activated replicate the execution of the body of the parallel regions delimited by the open/close curly brackets. However, just adding `parallel` makes our parallel code incorrect; in order to understand why we need to know that in OpenMP all variables used inside the parallel region are shared by default unless declared private (either by defining the variable inside the scope of the parallel region or declaring it `private` in the `parallel` construct). In particular, in this code the (shared) access to the loop control variable `i` and the temporary variable `x` causes the data races that make the execution not correct.

3. In order to partially correct it, `pi-v2.c` adds the `private` clause for variables i and x. Now observe that when i is private each thread executes all iterations of the loop, so we are not taking benefit of parallelism.

4. In order to avoid the total replication of work `pi-v3.c` uses the runtime call `omp_get_num_threads()` and changes the for loop in order to distribute the iterations of the loop among the participating threads. Which iterations is each thread executing? Observe that the result is not correct due to a race condition (perhaps you don't observe the error when using the small input value, but for sure you will observe when running with the large one). The access to which variable is causing the data race, and therefore the incorrect result?

1.2.2 Using synchronization to avoid data races

1. Next version `pi-v4.c` uses the `critical` construct to provide a region of mutual exclusion, that is a region of code in which only one thread executes it at any given time. This version should be correct although, as you will observe when submitting the execution of `submit-omp.sh`, it introduces large synchronization overheads.

2. Version `pi-v5.c` uses the `atomic` construct to guarantee the indivisible execution of read-operate-write operations, which is more efficient than the `critical` construct used in the previous version. Compare the execution time of both `pi-v4` and `pi-v5` when submitted for execution to the queue.

3. In order to reduce the amount of synchronisation needed to protect the access to `sum`, version `pi-v6.c` defines a private copy for each thread `sumlocal` that is used to accumulate its partial contribution to the global variable `sum`. This new variable is initialised to 0 and before the thread finishes it accumulates its value into `sum` making use of either `atomic` or `critical`.

4. Since the previous code transformation is very common, `OpenMP` offers the `reduction` clause. Version `pi-v7.c` makes use of this `reduction` clause. The compiler creates a private copy of the reduction variable which is used to store the partial result computed by each thread; this variable is initialised to the neutral value of the operator specified (0 in the case of `+`). At the end of the region, the compiler ensures that the shared variable is properly updated combining the partial results computed by each thread using the operator specified (`+` in this case). Run it and compare the execution time of `pi-v7` with the previous three versions.

1.2.3 The `for` work–sharing construct

1. In our current parallel version of the code we manually assign iterations to threads. Next we will use the `for` construct to automatically distribute the iterations of the loop among the threads of the team in different ways. This is the `pi-v8.c` version. Run it and notice which iterations are assigned to each thread. The execution time should be similar to the previous version, although the iterations are distributed among threads in a different way.

2. **Investigate yourself:** The `for` construct accepts an `schedule` clause to determine which iterations are executed by each thread; iterations are assigned to threads in chunks. There are three different options for schedule: (i) `static` (ii) `dynamic` and (iii) `guided`, all of them with an optional `chunk` value which indicates the number of consecutive iterations per chunk. Compile and run the different versions of the code provided (`pi-v8.c` and `pi-v9.c` for static; `pi-v10.c` for dynamic; and `pi-v11.c` for guided). Use the `debug` version to see which iterations are executed by each thread with the different schedules. Submit the execution of `submit-omp.sh` to observe the impact of the different schedules in the execution time. By instrumenting the execution with `Extrae` and visualising the parallel execution with `Paraver` you can observe the overheads that are introduced by `dynamic` and `guided` due to the *scheduling* points where chunks of iterations are dynamically assigned to threads. Compare versions v8, v10 and v11 by submitting to the execution queue the `submit-extrae.sh` script with the proper parameters and analysing the results.

1.2.4 Summary of code versions

The following table summarises all the codes used during the process. Changes accumulate from one version to the following.

Version	Description of changes	Correct?
v0	Sequential code. Makes use of `omp_get_wtime` to measure execution time	yes
v1	Added `parallel` construct and `omp_get_thread_num()`	no
v2	Added `private` for variables `x` and `i`	no
v3	Manual distribution of iterations using `omp_get_num_threads()`	no
v4	Added `critical` construct to protect `sum`	yes
v5	Added `atomic` construct to protect `sum`	yes
v6	Private variable `sumlocal` and final accumulation on `sum`	yes
v7	Use of `reduction` clause on `sum`	yes
v8	Added `for` construct to distribute iterations of loop (default schedule: static)	yes
v9	Example of `schedule(static,1)`	yes
v10	Example of `schedule(dynamic,1000)`	yes
v11	Example of `schedule(guided,10)`	yes

1.3 Test your understanding

Next you will go through a set of very simple examples that will be helpful to practice the components of the OpenMP programming model that have been used to parallelise the Pi computation. For each example answer the question(s) in the questionnaire that you will deliver as part of the deliverable for this second laboratory assignment. In order to follow them, you will need:

- The set of files inside the indicated directories in `lab2/openmp`.
- In case you want to know details about the `OpenMP` constructs, the set of slides in *Part I* and *Part II* in *Short tutorial on OpenMP* (available through "Atenea").

Use the appropriate entry in the `Makefile` to individually compile each program (e.g. `"make 1.hello"`). In order to do the executions simply run each program interactively (e.g. `"./1.hello"` or `"OMP_NUM_THREADS=4 ./1.hello"` if you want to externally set the number of threads to be used).

1.3.1 Parallel regions

1. Get into directory `lab2/openmp/basics`. Open each file in the directory (the examples are ordered), compile and execute; then answer the question(s) in the questionnaire associated to it.

1.3.2 Loop parallelism

1. Get into directory `lab2/openmp/worksharing`. Open each file in the directory (the examples are ordered), compile and execute; then answer the question(s) in the questionnaire associated to it.

1.3.3 Synchronisation

1. Get into directory `lab2/openmp/synchronization`. Open each file in the directory (the examples are ordered), compile and execute; then answer the question(s) in the questionnaire associated to it.

1.4 Observing overheads

1.4.1 Synchronisation overheads

To finish this session we ask you to compile and execute four different versions of the Pi computation parallel program, each one making use of a different synchronisation mechanism to perform the update of the global variable `sum`. Codes are available inside directory `lab2/overheads`.

- `pi_omp_critical.c`: a `critical` region is used to protect every access to `sum`, ensuring exclusive access to it. This version is equivalent to `pi-v4`.
- `pi_omp_atomic.c`: it makes use of `atomic` to guarantee atomic (indivisible) access to the memory location where variable `sum` is stored. This version is equivalent to `pi-v5`.
- `pi_omp_reduction.c`: it makes use of the `reduction` clause applied to the global variable `sum`. This version is equivalent to `pi-v7`.
- `pi_omp_sumlocal.c`: a "per–thread" private copy `sumlocal` is used followed by a global update at the end using only one `critical` region. This version is equivalent to `pi-v6`.

Take a look at the four different versions and make sure you understand them. For example, how many synchronisation operations (`critical` or `atomic`) are executed in each version?

Compile all four versions (use the appropriate entries in the `Makefile`) and queue the execution of the binaries generated using the `submit-omp.sh` script (which requires the name of the binary, the number of iterations for Pi computation and the number of threads). Answer the following questions:

1. If executed with only 1 thread and 100.000.000 iterations, do you observe any major overhead in the execution time caused by the use of the different synchronisation mechanisms? You can compare with the baseline execution time of the sequential version in `pi_sequential.c`.
2. If executed with 4 and 8 threads and the same number of iterations, do the 4 programs benefit from the use of several processors in the same way? Can you guess the reason for this behaviour?

Take note of all the results that you obtain and reach your conclusions about the overheads associated with these `OpenMP` constructs. You will have to deliver them in the appropriate section of the deliverable for this second laboratory assignment.

2

A very practical introduction to OpenMP (Part II)

As in the previous session, in this one you will first go through a set of different code versions (some of them not correct) for the computation of number Pi in parallel but now using the tasking model. After that you will be presented with a set of very simple examples that will be helpful to practice the main components of the OpenMP programming model being presented. We ask you to fill in the rest of the questionnaire in the deliverable for this second laboratory assignment. The session will finish observing the overheads related with the creation of parallel regions and tasks in `OpenMP`.

2.1 Parallelisation with OpenMP

As you did in the previous session, use the two entries provided in the `Makefile` in order to compile the different versions that will be explored in this session: `"make pi-vx-debug"` and `"make pi-vx-omp"`. Execute the binaries generated using the `run-debug.sh` (interactive execution), `run-omp.sh` (interactive execution) and `submit-omp.sh` (execution queue) scripts. You can also instrument the execution with `Extrae` and visualize the parallel execution with `Paraver` by submitting the `submit-extrae.sh` script.

2.1.1 The `single` work–sharing construct

1. Version `pi-v12.c` manually divides the execution of the loop in two loops, each computing half of the total number of iterations. With this code we will exemplify the use of the **single** construct. With `single` only one of the threads executes the associated code; the others just wait at the end of the `single`. Are we exploiting any parallelism in this version? Execute it several times and verify your answer.

2. Unfortunately only one thread is doing useful work at any time. In order to avoid this, we introduce the `nowait` clause in version `pi-v13.c`. Now answer the same question as before: are we exploiting any parallelism in this version? Execute it several times and verify your answer.

2.1.2 Tasking execution model

1. `pi-v14.c` simply replaces the use of `single` by `task`. When a thread encounters a `task` construct in generates a task and places it in a pool of tasks; any thread in the team can execute it. In other words, the `task` construct provides a way of defining a deferred unit of computation that can be executed by any thread in the team. If you execute it you will see two things: 1) the value of pi is not correct (in fact is zero); and 2) each iteration is executed 4 times. For the first we give you the reason: reductions are not implemented in `task`. For the second, can you guess why?

2. `pi-v15.c` makes use of the `single` construct explained before to make the parallelization correct: only one thread generates the tasks, the others are ready to execute them. Execute this version and check the result that is obtained. Can you explain how the sharing of variable `sum` is done?

Observe that this version also introduces the use of the `taskwait` construct, which forces the thread that entered into the `single` region and created the two tasks to wait for their termination before accumulating the partial results that they have computed.

3. `pi-v16.c` introduces the use of dependencies between tasks. The `depend` clause is used to specify variables that are `in`, `out` or `inout` to the task (in other words, read, written or both). With this a task order can be established so that a task is not executed until all its dependences are satisfied. This is the case for the third task defined in `pi-v16.c`, which waits for the termination of the other two. This is replacing the `taskwait` construct in `pi-v15.c`.

4. Finally, `pi-v17.c` makes use of the `taskloop` construct to generate a task for a certain number of consecutive iterations, controlled either with the `num_tasks` or the `grainsize` clauses; the first one specifies the number of tasks to generate while the second one controls the number of consecutive iterations per task. Take a look at the code and try both options commenting one option or the other.

2.1.3 Summary of code versions

The following table summarises all the codes used during the process. Changes accumulate from one version to the following.

Version	Description of changes	Correct?
v12	Use of `single` construct	yes
v13	Use of `single` construct and nowait clause	yes
v14	Use of `task` construct	no
v15	Use of `single` to have just one `task` generator and `taskwait`	yes
v16	Use of `task` with dependences (`depend` clause)	yes
v17	Use of `taskloop` to generate tasks from loop iterations	yes

2.2 Test your understanding

Now you are ready to go through the set of very simple examples that will be helpful to practice the main components of the OpenMP programming model introduced in this session, filling-in the rest of the questionnaire in the deliverable for this second laboratory assignment. In order to follow them, you will need:

- The set of files inside the `lab2/openmp/tasks` directory.
- In case you want to know details about the `OpenMP` constructs, the set of slides in *Part III* of *Short tutorial on OpenMP* available through "Atenea".

2.2.1 Task parallelism

1. Get into the directory `lab2/openmp/tasks`. Open each of the codes in the directory (the examples are ordered), compile and execute; then answer the questions in the questionnaire associated to it.

2.3 Observing overheads

To finish with this section we propose you to observe the overheads related with the creation of `parallel` regions, `task` creation and synchronization. Codes are available inside directory `lab2/overheads`. Take note of all the results that you obtain and reach your conclusions about the overheads associated with these `OpenMP` constructs. You will have to deliver them in the appropriate section of the deliverable for this second laboratory assignment.

2.3.1 Thread creation and termination

1. Open the `pi_omp_parallel.c` file and look at the changes done to the parallel version of pi. This new version is based on a function named `difference` which computes the difference in time between the sequential execution and the parallel execution when using a certain number of threads, using the `OpenMP` intrinsic function `omp_set_num_threads` to change the number of threads. The execution of both the sequential and parallel versions is done `NUMITERS` times in order to average the execution time of one iteration. In the `main` program, there is a loop to iterate over the number of threads (between 2 and `max_threads`, one of the input arguments of the execution) and prints the difference in time reported by function `difference`, which is the overhead introduced by the `parallel` construct (try to understand what is printed there).

2. Compile using the appropriate target in `Makefile` and submit the execution of the binary generated `pi_omp_parallel` with just one iteration and a maximum of 24 threads (i.e. `"qsub -l execution ./submit-omp.sh pi_omp_parallel 1 24"`). Do not expect a correct result for the value of Pi!

3. How does the overhead of creating/terminating threads varies with the number of threads used? Which is the order of magnitude for the overhead of creating/terminating each individual thread in the parallel region?

2.3.2 Task creation and synchronization

1. Open the `pi_omp_tasks.c` file. This version is creating tasks inside function `difference` by a single thread in the parallel region. The function measures the difference between the sequential execution and the version that creates the tasks; each version is repeated `NUMITERS` times in order to average the execution time of one iteration. In the `main` program, there is a loop to iterate over the number of tasks that we want to generate (between `MINTASKS` and `MAXTASKS`, and prints the difference in time reported by function `difference`, which is the overhead introduced by the `task and taskwait` constructs (try to understand what is printed there).

2. Compile using the appropriate target in `Makefile` and submit the execution of the binary generated `pi_omp_tasks` with just 10 iterations and one thread (i.e. `"qsub -l execution ./submit-omp.sh pi_omp_tasks 10 1"`). Do not expect a correct result for the value of Pi!

3. How does the overhead of creating/synchronising tasks varies with the number of tasks created? Which is the order of magnitude for the overhead of creating/synchronising each individual task?

3

Deliverable

After the last session for this laboratory assignment, and before starting the next one, you will have to deliver a **report** in `PDF` format (other formats will not be accepted) containing the answers to the following questionnaire and reporting your conclusions about the overheads of the different `OpenMP` constructs. Your professor will open the assignment at the Raco website and set the appropriate dates for the delivery. Only one file has to be submitted per group through the Raco website.

Important: In the front cover of the document, please clearly state the name of all components of the group, the identifier of the group (username `parXXYY`), title of the assignment, date, academic course/semester, ... and any other information you consider necessary.

3.1 OpenMP questionnaire

When answering to the questions in this questionnaire, please DO NOT simply answer with yes, no or a number; try to minimally justify all your answers and if necessary include any code fragment you need to support your answer. Sometimes you may need to execute several times in order to see the effect of data races in the parallel execution.

A) Parallel regions

1.hello.c

1. How many times will you see the `"Hello world!"` message if the program is executed with `"./1.hello"`?
2. Without changing the program, how to make it to print 4 times the `"Hello World!"` message?

2.hello.c:

1. Is the execution of the program correct? (i.e., prints a sequence of `"(Thid) Hello (Thid) world!"` being `Thid` the thread identifier). If not, add a data sharing clause to make it correct?
2. Are the lines always printed in the same order? Why the messages sometimes appear intermixed? (Execute several times in order to see this).

3.how_many.c: Assuming the `OMP_NUM_THREADS` variable is set to 8 with `"export OMP_NUM_THREADS=8"`

1. How many `"Hello world ..."` lines are printed on the screen?
2. What does `omp_get_num_threads` return when invoked outside and inside a parallel region?

4.data_sharing.c

1. Which is the value of variable `x` after the execution of each parallel region with different data-sharing attribute (`shared`, `private`, `firstprivate` and `reduction`)? Is that the value you would expect? (Execute several times if necessary)

B) Loop parallelism

1.schedule.c

1. Which iterations of the loops are executed by each thread for each `schedule` kind?

2.nowait.c

1. Which could be a possible sequence of printf when executing the program?
2. How does the sequence of `printf` change if the `nowait` clause is removed from the first `for` directive?
3. What would happen if dynamic is changed to static in the schedule in both loops? (keeping the nowait clause)

3.collapse.c

1. Which iterations of the loop are executed by each thread when the `collapse` clause is used?
2. Is the execution correct if the `collapse` clause is removed? Which clause (different than `collapse`) should be added to make it correct?

C) Synchronization

1.datarace.c (execute several times before answering the questions)

1. Is the program always executing correctly?
2. Add two alternative directives to make it correct. Explain why they make the execution correct.

2.barrier.c

1. Can you predict the sequence of messages in this program? Do threads exit from the barrier in any specific order?

3.ordered.c

1. Can you explain the order in which the "Outside" and "Inside" messages are printed?
2. How can you ensure that a thread always executes two consecutive iterations in order during the execution of the ordered part of the loop body?

D) Tasks

1.single.c

1. Can you explain why all threads contribute to the execution of instances of the single work-sharing construct? Why are those instances appear to be executed in bursts?

2.fibtasks.c

1. Why all tasks are created and executed by the same thread? In other words, why the program is not executing in parallel?
2. Modify the code so that the program correctly executes in parallel, returning the same answer that the sequential execution would return.

3.synchtasks.c

1. Draw the task dependence graph that is specified in this program
2. Rewrite the program using only `taskwait` as task synchronisation mechanism (no `depend` clauses allowed)

4.taskloop.c

1. Find out how many tasks and how many iterations each task execute when using the `grainsize` and `num_tasks` clause in a `taskloop`. You will probably have to execute the program several times in order to have a clear answer to this question.
2. What does occur if the `nogroup` clause in the first `taskloop` is uncommented?

3.2 Observing overheads

Please explain in this section of your deliverable the main results obtained and your conclusions in terms of overheads for `parallel`, `task` and the different synchronisation mechanisms. Include any tables/plots that support your conclusions.