PAR Laboratory Assignment
Lab 2: OpenMP programming model and analysis of overheads

E. Ayguadé, J. Corbalán, J. Morillo, J. Tubella and G. Utrera

Spring 2017-18

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

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"`.

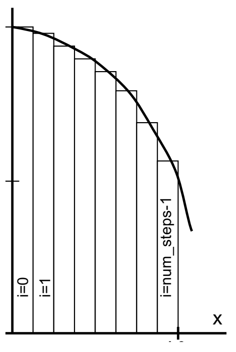
Session 1

A very practical introduction to OpenMP

This chapter has been prepared with the purpose of introducing the main constructs in the OpenMP extensions to the C programming language. You will go through a set of different code versions (some of them not correct) for the computation of number Pi in parallel. All files are in the `lab2/pi` directory.

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 parallelization 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 parallelize the sequential code we will proceed through a set of versions `"pi-vx.c"`, 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"`). 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 10000000 in the script), just by typing `"qsub -l execution ./submit-extrae.sh pi-vx"`.

1.2 Parallelization 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 parallelize the sequential code, `pi-v1.c` introduces the `parallel` construct, which creates/activates the team of threads. However, just adding `parallel` makes our parallel code incorrect; 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 all threads execute the body of the parallel region accesing (sharing) the loop control variable `i` and the temporary variable `x`.

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. 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. **Important:** DO NOT try to submit the script to instrument the execution of this version, it will take very long and generate an excessively large trace file, consuming your available disk quota. If you want to do, change the input value set in the `submit-extrae.sh` script to a lower value (remove 2 zeros for example).

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. Finally, the alternative solution in `pi-v6.c` makes use of the `reduction` clause. Reduction is a very common pattern where all threads accumulate values into a single shared variable, in this case variable `sum`. The compiler creates a private copy of the reduction variable which is used to store the partial result computed by each thread; and at the end of the region, the runtime 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-v6` with the previous two versions and version `pi-v3`.

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 distribute the iterations of the loop among the threads of the team in different ways. This is the `pi-v7.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. 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-v7.c` and `pi-v8.c` for static; `pi-v9.c` and `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 visualizing 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. **Important**: Reduce the input value in the `submit-extrae.sh` script before submitting for instrumentation (especially for `pi-v9`, otherwise you may generate an excessively large trace file, consuming your available disk quota.

1.2.4 The `single` work–sharing construct

1. Version `pi-v12.c` exemplifies the use of the `single` construct. We have modified the code and moved the `pi=step*sum` statement inside the parallel region. Since this statement should be executed by only one thread in the team, we have put it inside a `single` construct. Run the code, is it correct?

2. Unfortunately the result is not correct. Observe that the `reduction` clause is placed in the `parallel` construct and therefore when one of the threads enters in `single` it does not still see the aggregated contributions of the rest of threads. Version `pi-v13.c` solves the problem by appropriately placing the reduction clause in the `for` work–sharing construct. Check the result.

1.2.5 Tasking execution model

1. `pi-v14.c` manually divides the execution of the loop in two loops, each computing half of the total number of iterations. With this example we will exemplify the use of the `task` construct, defining a `task` to compute each loop. The `task` construct provides a way of defining a deferred unit of computation that can be executed by any thread in the team. Observe the clauses that define `shared` and `private` variables. We also need to add the `taskwait` construct to wait for the termination of all tasks generated. If you execute this version you will see that the code is not correct; observe that tasks are generated as many times as threads in the parallel region, due to the replication of the body in the `parallel` region. You can easily observe this behaviour with *Paraver*.

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. Check that the result is correct and visualize again with *Paraver* to verify that only two tasks are generated. Can you explain how the sharing of variable `sum` is protected? Why are we not using a `reduction` clause as we did in previous versions based on the use of *work-sharing* constructs?

3. `pi-v16.c` defines a new `task` for each iteration of the loop body. Observe the use of `firstprivate` to capture the value of the `i` variable at task creation time. This version is correct, but the performance is very bad due to the large overheads of task creation (task granularity too small, just one iteration of the loop). However when the number of iterations of the loop is unknown the `task` construct is necessary. **Important**: Reduce the input value in the `submit-extrae.sh` script before submitting for instrumentation of this version, otherwise you may generate an excessively large trace file, consuming your available disk quota.

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 the number of consecutive iterations per task. Take a look at the code and try both options by commenting one or the other option. Visualize the execution with *Paraver* to verify when tasks are generated and executed.

1.3 Summary of code versions

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

Code	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	Critical construct to protect `sum`	yes
v5	Atomic construct to protect `sum`	yes
v6	Reduction on `sum`	yes
v7	Add `for` construct to distribute iterations of loop (default schedule: static)	yes
v8	Example of `schedule(static,1)`	yes
v9	Example of `schedule(dynamic)`	yes
v10	Example of `schedule(dynamic,1000)`	yes
v11	Example of `schedule(guided,10)`	yes
v12	Use of `single` construct	no
v13	`reduction` clause revisited	yes
v14	Use of `task` and `taskwait` constructs	no
v15	Use of `single` to have just one task generator	yes
v16	Fine-grained parallelization with tasks	yes
v17	Fine-grained parallelization with taskloop	yes

Session 2

OpenMP tutorial examples

This chapter has been prepared with the purpose of guiding you through a set of very simple examples that will be helpful to practice the main components of the OpenMP programming model, filling-in the questionnaire in Part I of the deliverable for this second laboratory assignment. In order to follow them, you will need:

- The set of files inside the `lab2/openmp` directory.
- The set of slides for *Short tutorial on OpenMP* available through "Atenea".

2.1 OpenMP basics

1. Get into the directory called `lab2/openmp/basics`. Open each of the codes in the directory (the examples are ordered) and answer the questions in the questionnaire associated to it; later you can compile (e.g. `"make 1.parallel"`) and run (e.g. `"./1.parallel"` or `"OMP_NUM_THREADS=4 ./1.parallel"` if you want to externally set the number of threads to be used) to check your answers.
2. Consult "`Part I: OpenMP Basics`" of the tutorial slides, if necessary.

2.2 Loop parallelism

1. Get into the directory `lab2/openmp/worksharing`. Open each of the codes in the directory (the examples are ordered) and answer the questions in the questionnaire associated to it; later you can compile and run to check your answers.
2. Consult "`Part II: Loop Parallelism in OpenMP`" of the tutorial slides, if necessary.

2.3 Task parallelism

1. Get into the directory `lab2/openmp/tasks`. The codes in this directory are a bit more complex than the previous ones. Programs `1.serial.c` and `2.parallel.c` define and dynamically create a linked list; afterwards, some computation is done for every node of the list in the function called `processwork`[1]. In the directory there is also a third code `3.taskloop.c` that will help you to understand when threads are available to execute tasks. Open each of these codes in the directory (the examples are ordered) and answer the questions in the questionnaire associated to it; later you can compile and run to check your answers.
2. Consult "`Part III: Task Parallelism in OpenMP`" of the tutorial slides, if necessary.

[1]Function `processwork` generates the ith number of the Fibonacci series where i is the value of the *data* field in a node in the list. The function also stores the identifier of the thread that did the computation.

Session 3

Measuring parallelization overheads

In this chapter you will measure some of the main overheads that need to be considered in the parallel execution of programs in OpenMP for a shared–memory architecture. For this chapter, you will need to go back into directory `lab2/overheads`. Experimental results and conclusions will contribute to Part II of the deliverable for this second laboratory assignment.

3.1 Thread creation and termination

First you will measure the overhead related with the creation and termination of threads in a parallel region. Threads are the execution entities offered by the operating system to support the execution of shared-memory parallel paradigms such as `OpenMP`.

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 numbers 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 execute the binary `pi_omp_parallel` with just one iteration and a maximum of 24 threads, first interactively `"./run-omp.sh pi_omp_parallel 1 24"` and then submitting to the execution queue using the `./submit-omp.sh` script with the same arguments. 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?

3.2 Task creation and synchronization

Second you will measure the overhead related with the creation of `task` and their synchronization at a `taskwait` point, the key elements in the `OpenMP` tasking model.

1. Open the `pi_omp_tasks.c` file and look at the changes done to the parallel version of pi. This new version is very similar to the previous one, but now creating tasks inside function `difference` by a single thread in the parallel region. Again 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 execute the binary `pi_omp_tasks` with just 10 iterations and one thread, first interactively `"./run-omp.sh pi_omp_tasks 10 1"` and then submitting to the execution queue using the `./submit-omp.sh` script with the same arguments. Do not expect a correct result for the value of Pi!

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

3.3 Task vs. taskloop

Third you will compare the overheads of `task` and `taskloop`. Both versions generate tasks with the same granularity and the same number of tasks. `taskloop`has an implicit synchronization at the end (`taskgroup` that is equivalent in this case to the `taskwait`). Therefore the comparison should be fair.

1. Open the `pi_omp_taskloop.c` file and look at the changes done with respect to the previous version just using `task`.

2. Compile using the appropriate target in `Makefile` and execute the binary `pi_omp_taskloop` with just 10 iterations and one thread, first interactively `"./run-omp.sh pi_omp_tasks 10 1"` and then submitting to the execution queue using the `./submit-omp.sh` script with the same arguments. Do not expect a correct result for the value of Pi!

3. How does the overhead of creating/synchronizing tasks varies with the number of tasks created in both cases, with `task` and `taskloop`?

3.4 Thread synchronization: `critical`

Fourth you will measure the overhead related with the use of `critical` as one of the mechanisms for synchronization in `OpenMP`.

1. `pi_omp_critical.c` and `pi_omp.c` contain two different `OpenMP` parallelizations of the Pi computation. In the first one a `critical` region is used to protect every access to `sum`, ensuring exclusive access to it. In the second one, a "per–thread" private copy `sumlocal` is used followed by a global update at the end using only one `critical` region. Take a look at both codes and make sure you understand the differences. How many critical sections are executed in each case?

2. Use the `Makefile` to compile both programs and execute them with just 1 thread. Compare with the sequential execution time of `pi_seq.c` for 100.000.000. Can you explain the differences, if any? Do an estimation of the order of magnitude for the overhead of executing a `critical` region.

3. Next execute with 8 threads. Do both programs benefit from the use of several processors in the same way? Can you guess the reason for this behavior?

4. In order to understand the numbers obtained and the sources of the overhead observed, you will generate a `Paraver` trace for a small number of iterations of the program (100.000 iterations), for 1 and 8 processors, for the execution of `pi_omp_critical.c` (look at the `Makefile` in order to use the appropriate target). Observe how the program proceeds through three `lock` phases for every `critical` region. Use `Paraver` and the appropriate configuration file to measure how much it takes, on average, each of these phases with 1 and 8 threads. Can you identify at least three reasons that justify the large performance degradation (if any) observed?

3.5 Atomic memory access

Next you will measure the overhead related with the use of `atomic` as another mechanism that can be used to protect the access to shared variables in `OpenMP`.

1. `pi_omp_atomic.c` contains a different `OpenMP` parallelization that makes use of `atomic` to guarantee atomic (indivisible) access to memory. Take a look at the code and compare with `pi_omp_critical.c`.

2. Use the `Makefile` to compile this new version and execute it with 1 and 8 threads. Can you estimate the order of magnitude for the overhead of executing `atomic`?

3.6 False data sharing

Finally, you will measure the impact of excessive (unnecessary) data sharing (*false sharing*) and use a technique to appropriately eliminate it.

1. `pi_omp_sumvector.c` provides yet another version for the program computing Pi that avoids the use of `critical` regions. Open the file and explain the purpose of vector `sumvector`. What does each element of this vector store? Why just after the parallel region the values in this vector are added?

2. Use the `Makefile` to compile it and execute by submitting the appropriate script with a size of 100.000.000 (and 1.000.000.000) and 1 and 8 threads. Compare the execution times, for 1 and 8 threads, with the execution times for `pi_omp.c`. Even totally avoiding synchronizations, why the execution time is worse? Is the use of vector `sumvector` the cause of that difference?

3. The answer to the last question is "yes". `pi_omp_sumvector.c` suffers of what is called "false sharing". False sharing occurs when multiple threads modify different memory addresses that are stored in the same cache line. When multiple threads update these independent memory locations, the cache coherence protocol forces other threads to update/invalidate their caches. To avoid false sharing we need to make sure that threads do not share cache lines. To that end, `pi_omp_padding.c` provides a new version in which padding is used (i.e. elements accessed by each thread reside in different cache lines). Look at the source code and explain the way padding is done in this code. Compile the program using the `Makefile`, and execute. Is the execution time similar, better or worse than `pi_omp.c`? Compute the additional average latency for each individual access to memory when your program suffers from false sharing?

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 questions. 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:

- Please, follow the same recommendations that we made for the previous deliverable.
- 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. As part of the document, you can include any code fragment you need to support your explanations.

Part I: 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. Sometimes you may need to execute several times in order to see the effect of data races in the parallel execution.

A) Basics

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: Assuming the `OMP_NUM_THREADS` variable is set to 8 with `"export OMP_NUM_THREADS=8"`

1. Is the execution of the program correct? (i.e., prints a sequence of `"(Thid) Hello (Thid) world!"` being `Thid` the thread identifier) Which data sharing clause should be added to make it correct?.
2. Are the lines always printed in the same order? Could the messages appear intermixed?

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. If the `if(0)` clause is commented in the last parallel directive, how many `"Hello world ..."` lines are printed on the screen?

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` and `firstprivate`)?
2. What needs to be changed/added/removed in the first directive to ensure that the value after the first parallel is always 8?.

5.parallel.c

1. How many messages the program prints? Which iterations is each thread executing?
2. Change the for loop to ensure that its iterations are distributed among all participating threads.

6.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.

7.barrier.c

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

B) Worksharing

1.for.c

1. How many and which iterations from the loop are executed by each thread? Which kind of `schedule` is applied by default?
2. Which directive should be added so that the first `printf` is executed only once by the first thread that finds it?.

2.schedule.c

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

3.nowait.c

1. How does the sequence of `printf` change if the `nowait` clause is removed from the first `for` directive?
2. If the `nowait` clause is removed in the second `for` directive, will you observe any difference?

4.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?.

5.ordered.c

1. Can you explain the order in which `printf` appear?
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?

6.doacross.c

1. In which order are the "Outside" and "Inside" messages printed?
2. In which order are the iterations in the second loop nest executed?
3. What would happen if you remove the invocation of `sleep(1)`. Execute several times to answer in the general case.

C) Tasks

1.serial.c

1. Is the code printing what you expect? Is it executing in parallel?

2.parallel.c

1. Is the code printing what you expect? What is wrong with it?
2. Which directive should be added to make its execution correct?
3. What would happen if the `firstprivate` clause is removed from the task directive? And if the `firstprivate` clause is ALSO removed from the `parallel` directive? Why are they redundant?
4. Why the program breaks when variable `p` is not `firstprivate` to the task?
5. Why the `firstprivate` clause was not needed in `1.serial.c`?

3.taskloop.c

1. Execute the program several times and make sure you are able to explain when each thread in the threads team is actually contributing to the execution of work (tasks) generated in the `taskloop`.

Part II: Parallelization overheads

1. Which is the order of magnitude for the overhead associated with a `parallel` region (fork and join) in `OpenMP`? Is it constant? Reason the answer based on the results reported by the `pi_omp_parallel.c` code.

2. Which is the order of magnitude for the overhead associated with the creation of a `task` and its synchronization at `taskwait` in `OpenMP`? Is it constant? Reason the answer based on the results reported by the `pi_omp_tasks.c` code.

3. Based on the results reported by the `pi_omp_taskloop.c` code, If you have to generate tasks out of a loop, what seems to be better: to use `task` or `taskloop`? Try to reason the answer.

4. Which is the order of magnitude for the overhead associated with the execution of `critical` regions in `OpenMP`? How is this overhead decomposed? How and why does the overhead associated with `critical` increase with the number of processors? Identify at least three reasons that justify the observed performance degradation. Base your answers on the execution times reported by the `pi_omp.c` and `pi_omp_critical.c` programs and their `Paraver` execution traces.

5. Which is the order of magnitude for the overhead associated with the execution of `atomic` memory accesses in `OpenMP`? How and why does the overhead associated with `atomic` increase with the number of processors? Reason the answers based on the execution times reported by the `pi_omp.c` and `pi_omp_atomic.c` programs.

6. In the presence of false sharing (as it happens in `pi_omp_sumvector.c`), which is the additional average time for each individual access to memory that you observe? What is causing this increase in the memory access time? Reason the answers based on the execution times reported by the `pi_omp_sumvector.c` and `pi_omp_padding.c` programs. Explain how padding is done in `pi_omp_padding.c`.