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

Mario Acosta, Eduard Ayguadé, Rosa M. Badia (Q2), Jesús Labarta,
Josep Ramón Herrero, Daniel Jiménez-González, Pedro J. Martínez-Ferrer (Q2),
Jordi Tubella, and Gladys Utrera

Spring 2024–2025

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Contents

Note. Each chapter in this document corresponds to a laboratory session (2 hours).

Chapter 1

A very practical introduction to OpenMP implicit tasks

This laboratory assignment introduces the principal OpenMP extensions to the C programming language.

Firstly, you will go through a set of different code versions (some of which are incorrect) for the parallel computation of pi (π). Secondly, you will be presented with a set of examples that will be helpful for practisising the main components of the OpenMP programming model. The session will end with an analysis of the overheads introduced by the different synchronisation constructs of OpenMP.

All the files necessary to do this laboratory assignment are available at:

`/scratch/nas/1/par0/sessions/lab2.tar.gz`

For today's session, copy the file `lab2.tar.gz` from that location into your home directory:

```
cp /scratch/nas/1/par0/sessions/lab2.tar.gz ~/
```

and, from your home directory, uncompress it with this command line:

```
tar -zxvf lab2.tar.gz
```

1.1 Computing pi

```
static long int num_steps = 100000;

void main () {
  double x, pi, 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;
}
```

Code 1: Computation of pi (π).

We already showed how to compute pi in the previous laboratory assignment, see Code 1. In order to distribute the work for the parallel version, each processor must be responsible for executing *some* `i` iterations of the loop. This parallelisation must also guarantee that there are no data races when accessing the variable `sum` that accumulates the contribution from each iteration.

In order to parallelise the sequential code, we will proceed through a set of versions `pi-v?.c`

inside the `lab2/pi` directory, being `?` the version number. We provide two entries inside the Makefile to compile them:

- `pi-v?-debug`,
- `pi-v?-omp`.

The binary generated via

```
make pi-v?-debug
```

prints which iterations are executed by each thread and the value of pi that is computed, which is useful to understand what the program is actually doing. You can run the script

```
./run-debug.sh pi-v?
```

to interactively execute the binary with a very small input value (by default set to 32 iterations in the script).

In order to time the parallel execution you need to compile using the second option

```
make pi-v?-omp
```

and then queue the execution

```
sbatch submit-omp.sh pi-v?
```

which uses a much larger input value (by default set to 10^7 iterations in the script). Note that both options (`debug` and `omp`) require you to specify the binary name after the script.

Finally, in order to instrument the execution with Extrae, which will help you to generate a execution trace and visualise the parallel execution with Paraver, you need to submit the corresponding script

```
sbatch submit-extrae.sh pi-v?
```

which sets the input to a smaller value (by default set to 10^5 iterations in the script).

Note that inside the `submit-omp.sh` and `submit-extrae.sh` scripts, the number of processors (i.e. OpenMP threads) to be used is by default set to 4. You can change both the number of iterations and the number of processors by adding two extra arguments when executing or submitting the script, like in the following example

```
sbatch submit-omp.sh pi-v? 1000000000 8
```

1.2 Code parallelisation with OpenMP

Firstly, you will learn how to define a parallel region to create a "team of threads" that will run in parallel. Each thread created in the parallel region executes a so-called "implicit task" that corresponds with the body of the parallel region. Secondly, you will see how to split the work among these implicit tasks (as many as threads in the parallel region) and how to guarantee the correct access to *private* and *shared* variables by using different synchronisation constructs.

1.2.1 The parallel region and the use of implicit tasks

We ask you to do the following:

1. Compile and run the initial sequential code `pi-v0.c` using both the `debug` and `omp` binaries to establish a reference for the obtained result (value of pi) and the execution time for the sequential version. Note that this initial version calls `omp_get_wtime` to measure the wall clock execution time.
2. Check whether the value of pi is different between the `run-debug.sh` (32 iterations) and `submit-omp.sh` (10^7 iterations) scripts.

3. In a first attempt to parallelise the sequential code, `pi-v1.c` introduces the `parallel` construct, which creates/activates the team of threads. All the threads inside the team do replicate the execution of the body inside the parallel region(s) delimited by the open/close curly brackets.

4. Take into account that just adding the `#pragma omp parallel` clause makes the parallel code incorrect, as you can verify by executing the `debug` binary.

5. The parallel execution is incorrect because with OpenMP all the variables used inside the parallel region are *shared* by default unless declared *private* by either:

   - defining the variable inside the scope of the parallel region, or by
   - declaring the variable `private` in the same `parallel` construct.

6. In this particular code, the (shared) access to the loop control variable `i` and the temporary variable `x` causes the data races that make the parallel execution incorrect.

7. To *partially* fix the code, `pi-v2.c` adds the `private` clause to both variables `i` and `x`.

8. Execute the `pi-v2-debug` binary and verify that, when `i` is declared as private, each thread executes *all* the iterations of the loop, that is, we are not taking advantage of the parallelism.

9. To avoid the total replication of work from the previous version, `pi-v3.c` firstly calls `omp_get_num_threads` and secondly changes the `for` loop in order to distribute the iterations among the participating threads.

10. Compile and execute the `pi-v3-debug` binary: which iterations is each thread executing?

11. Note that the result is still incorrect due to yet another race condition. Perhaps you do not see the error when using a small input value so try to execute the `pi-v3-omp` binary instead.

12. Finally, can you guess what access to which variable is causing the data race in `pi-v3.c` and, therefore, the incorrect result of pi?

1.2.2 Using synchronisation to avoid data races

We ask you to carry out the following steps:

1. Compile and execute the `pi-v4-debug` binary and check the results. This versions uses the `critical` construct that provides a region of mutual exclusion, that is a region of code in which only one thread executes it at any given time.

2. Verify that this version is correct and also validate that, when submitting the execution of the `pi-v4-omp` binary, it introduces large synchronisation overheads.

3. Compare the execution times of both `pi-v4-omp` and `pi-v5-omp` binaries. The version `pi-v5.c` uses the `atomic` construct to guarantee the indivisible execution of read-modify-write instructions, which is much more efficient than the `critical` construct used in the previous version. In any case, `pi-v5.c` could still be largely improved (see below).

4. In order to reduce the amount of synchronisation needed to protect the access to the variable `sum`, version `pi-v6.c` defines a private copy for each thread named `sumlocal`, which is used to accumulate its partial contribution to the global variable `sum`.

5. This new variable `sumlocal` is initialised to zero (neutral value) and, before the thread finishes, it accumulates its value into the variable `sum` making use of either:

   - `atomic`, or
   - `critical`.

6. Check that the result is correct by executing the `pi-v6-debug` binary.

7. Check the scalability by submitting the `pi-v6-omp` binary. In the previous lab we checked that the execution time of the parallel version (using four processors) was close to one fourth of the original sequential time.

8. Version `pi-v7.c` makes use of the `reduction` clause that applies *automatically* the same code transformation that previously written by hand in `pi-v6.c`:
    - the compiler creates a private copy of the reduction variable that is used to store the partial result computed by each thread;
    - the compiler initialises this variable to the *neutral* value of the operator specified (zero in the case of `+`);
    - at the end of the parallel region, the compiler ensures that the shared variable is properly updated by combining the partial results computed by each thread using the operator specified (`+` in this case).
9. Run both `pi-v7-debug` and `pi-v7-omp` binaries and compare their execution times with the previous three versions.
10. Finally, `pi-v8.c` introduces the `barrier` synchronisation construct as a way to define a point where all the threads wait for each other to arrive.
11. Execute the `pi-v8-omp` binary and check the partial values of pi printed by each thread after the `barrier`. Where does the `reduction` operation actually take place?

1.2.3 Summary of the code versions

Table 1.1 summarises all the code versions used in the two previous sections, where changes accumulate from one version to the next.

Table 1.1: Code versions with implicit tasks.

Version	Description of changes	Correct?
v0	Sequential code making use of `omp_get_wtime`	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 with `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	Use of `barrier` construct	Yes

1.2.4 Discussion at the laboratory

Now you will go through a set of simple examples that will be helpful to practise the components of the OpenMP programming model that have been used to parallelise the computation of pi:

1. hello,
2. hello,
3. how_many,
4. data_sharing,
5. datarace,
6. datarace.

For each example, try to answer the question(s) that you will find within the source code and ask your professor in case you have any doubts. The way to proceed with the exercises is the following:

1. Go to the excersises directory:

```
cd ~/lab2/openmp/day1
```

2. Open each C file in that directory (the examples are ordered), compile it and execute the binary (several executions may sometimes be needed).
3. For this, use the appropriate entry in the `Makefile` corresponding to each program, for example

   ```
   make 1.hello
   ```

4. Next, execute the generated binary interactively via

   ```
   ./1.hello
   ```

5. Or, if you want to set the number of OpenMP threads (e.g. 4)

   ```
   OMP_NUM_THREADS=4 ./1.hello
   ```

6. Answer the question(s) associated with each code.
7. If asked in the program, proceed with the required modifications and execute the program again.
8. In case you need to know more details about OpenMP constructs, we suggest you to either use the complete specification document or the short reference guide available at Atenea or at www.openmp.org.

1.3 Observing overheads

1.3.1 Synchronisation overheads

To finish this session, we ask you to compile and execute four parallel versions of the program that computes pi. Each version makes use of a different synchronisation mechanism to perform the update of the global variable `sum`. The following codes are available inside the `lab2/overheads` directory:

`pi_omp_critical.c` A `critical` region protects `sum`, ensuring *exclusive* access to it; this version is equivalent to `pi-v4`.

`pi_omp_atomic.c` It uses `atomic` to guarantee atomic (i.e. indivisible) access to the memory location where `sum` is stored; this version is equivalent to `pi-v5`.

`pi_omp_sumlocal.c` It employs a *per-thread* private copy of `sumlocal` followed by a global update at the end that uses a single `critical` region; this version is equivalent to `pi-v6`.

`pi_omp_reduction.c` A `reduction` clause is applied to the global variable `sum`; this version is equivalent to `pi-v7`.

Have a look at the four versions and make sure you that understand them: how many synchronisation operations, either `critical` or `atomic`, are executed by each version?

Notice that the function `difference` in versions `pi_omp_critical.c` and `pi_omp_atomic.c` returns the difference in time between:

- the *ideal* parallel execution time (sequential time divided by number of threads), and
- the *real* parallel execution when using each type of synchronisation.

This value is printed at the end of the execution. Moreover, when using either of these two first versions with a single thread, we do not create a parallel region to measure the overhead due to synchronisation.

In the case of the last two versions, i.e. `sumlocal` and `reduction`, both synchronisation overheads are indeed similar: version `pi_omp_sumlocal.c` implements *manually* the same synchronisation mechanism that is *automatically* done by the compiler with a reduction clause in `pi_omp_reduction.c`.

We ask you to carry out the following steps:

1. Run via the script `submit-seq.sh` the sequential version `pi_sequential.c` for 10^7 iterations.

2. Compile the four parallel versions using the appropriate entries in the `Makefile`.

3. Queue the execution of each generated binary using the `submit-omp.sh` script. This script requires the name of the binary, the number of iterations (i.e. 10^7), and the number of threads (i.e. 4).

4. Execute again the versions `pi_omp_critical.c` and `pi_omp_atomic.c` with a single thread and 10^7 iterations.

1.3.2 Discussion at laboratory

We ask you to answer the following questions:

- Do the four programs benefit from the use of several processors in the same way? Can you guess the reason for such behaviour?
- For versions `pi_omp_critical.c` and `pi_omp_atomic.c` and only one thread and 10^7 iterations, do you notice any major overhead in the execution time caused by the use of different synchronisation mechanisms?
- Quantify (in microseconds) the cost of each individual synchronisation operation (`critical` or `atomic`) when executing with one and four threads.

Chapter 2

A very practical introduction to OpenMP explicit tasks

As you did in the previous session, you will first go through a set of different code versions (some of which are incorrect) for the parallel computation of pi using, this time, the tasking model. After this, you will be presented with a set of simple examples that will be helpful for practising the main components of the OpenMP programming model. The session will end with an analysis of the overheads related to the creation of parallel regions and tasks in OpenMP.

2.1 Code parallelisation with OpenMP

In this section you will go one step further into the so-called "explicit tasks", a much more versatile way to express parallelism in OpenMP. Firstly, you will learn how one of the threads (implicit task) in the team is selected as the responsible for generating the explicits tasks that will be executed by the other threads in the team. Secondly, you will learn the simplest way to synchronise these explicit tasks through task barriers. In subsequent sessions you will dive deeper into nested explicit tasks to see how powerful the OpenMP tasking model is.

Use the two entries provided in the Makefile in order to compile the different versions that will be explored in this session (i.e. `make pi-v?-debug` and `make pi-v?-omp`) where `?` denotes the version number. Execute the binaries generated using the `run-debug.sh` (interactive execution) and `submit-omp.sh` (execution queue) scripts. You can also instrument the execution with Extrae and visualise the parallel execution with Paraver by submitting the `submit-extrae.sh` script. Default values for the number of iterations and number of processors are used in these scripts. Take a look at those values and learn how you can change them at execution or submission time.

2.1.1 Tasking execution model and the use of explicit tasks

We ask you to do the following steps:

1. Open the file `pi-v9.c` and verify that we have changed the code so that the original loop is manually divided in two halves, each one computing half of the loop iterations:
   - each loop defines an explicit task via the `task` construct;
   - when a thread encounters a `task` construct, it generates a task and places it in a pool of tasks;
   - any thread in the team can execute a task;
   - 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 of threads;
   - notice that each task defines a private copy of variables `i` and `x`.
2. Execute the `debug` binary and verify that:

(a) each iteration of the loop is executed four times, and

(b) the computed value of pi is not correct.

3. Can you guess why these two anomalies are happening? In order to give an answer to the first one (Step 2a), think about which thread(s) is(are) generating tasks and how many explicit tasks are generated in total.

4. To solve Step 2a above, `pi-v10.c` makes use of the `single` construct so that:
   - only *one* thread (implicit task) generates *all* the (explicit) tasks;
   - the remaining threads skip the `single` construct and wait at its end;
   - meanwhile, these threads are constantly looking in the pool of tasks for (explicit) tasks to execute.

5. Compile and run the `debug` binary for this version and check the obtained result. Why is it still incorrect?

6. Take into account that, in order to calculate a correct value of pi (e.g. as with `pi-v11.c`), a couple of things must be considered:
   - firstly, the thread that generates the two tasks has to wait for them to finish before executing the statement `pi = step*sum`;
   - notice that this statement needs a correct value for `sum`, which is jointly computed by the two previously generated tasks;
   - secondly, these two tasks have to contribute with their partial result to variable `sum`;
   - this contribution needs one of the synchronisation mechanisms shown in the previous session:
     - `critical` (version `v4` of the pi program),
     - `atomic` (version `v5` of the pi program),
     - `reduction` (version `v7` of the pi program).

7. Assuming that you would suggest the use of `reduction` (as it was the most efficient option), version `pi-v11.c` introduces the use of `taskgroup` as well as the `task_reduction` and `in_reduction` clauses. Have a look at the code and try to understand it:
   - firstly, the `taskgroup` construct defines a task barrier at its end, so that the thread that is generating the tasks waits until they have finished;
   - secondly, the `taskgroup` construct also defines the reduction operation, with the clause `task_reduction(+:sum)`, that will occur at its end, gathering the contributions for variable `sum` from the two previously generated tasks;
   - thirdly, each task specifies that it contributes to the reduction with its local results for variable `sum` with the clause `in_reduction(+:sum)`.

8. Compile and run the `debug` binary to check that the result is correct.

9. Compile and run the `omp` version to measure its scalability and verify that it is close to the ideal case.

10. If you have not done it before, this would be a good time to submit the Extrae instrumented version:

    ```
    sbatch submit-extrae.sh pi-v11 10000000 4
    ```

11. Visualise the parallel execution of this version with Paraver. For this, we have prepared the configuration file `pi-v11.cfg` with all the Paraver timelines configured for you. Do not forget to execute the command `Fit time scale` to view the entire execution and later zoom in the interesting parts.

12. Code `pi-v12.c` provides an *alternative* version based on the `atomic` clause that protects the update of the shared variable `sum`. Notice that the code makes use of the `taskwait` construct to wait for the termination of the tasks: it forces the thread entering the `single` region and creating the two tasks to wait for their termination before accessing the global variable `sum`.

13. Execute both the `pi-v12-debug` and `pi-v12-omp` binaries to validate the execution and verify the large overhead introduced by `atomic` (which would be even worse by using `critical`, as you may have guessed).

14. Code `pi-v13.c` introduces the use of data *dependencies* between tasks:

    - the `depend` clause specifies the variables that are `in`, `out`, or `inout` to the task, that is, "read", "written", or "both", respectively;
    - these annotations allow the OpenMP runtime to establish the appropriate task execution order so that no task can be executed until all its dependencies are satisfied;
    - this is precisely the case for the third task defined in `pi-v13.c`, which waits for the termination of the other two;
    - data dependencies replace the use of `taskgroup` in conjunction with the reduction clauses in `pi-v11.c`, as well as the `taskwait` with `atomic` (or `critical`) in `pi-v12.c`.

15. Execute both the `pi-v13-debug` and `pi-v13-omp` binaries to validate the execution and verify the new overheads associated with task dependencies.

16. This would be an excellent time to submit the Extrae instrumented version:

    ```
    sbatch submit-extrae.sh pi-v13 10000000 4
    ```

17. Visualise it with Paraver and check whether there are any visible differences between the execution of `pi-v11-omp` and `pi-v13-omp`. We have prepared the configuration file `pi-v13.cfg` with all the Paraver timelines configured for you. Do not forget to execute the command `Fit time scale` to view the entire execution and later zoom in the interesting parts.

18. Finally, `pi-v14.c` makes use of the `taskloop` construct to generate a task for a certain number of consecutive iterations, controlled with either:

    - `num_tasks` that specifies the number of tasks to generate, or
    - `grain_size` that controls the number of consecutive iterations per task.

19. Take a look inside `pi-v14.c` and try both clauses by commenting one option (or the other) and specifying different values. For example, with `num_tasks(8)` and the `debug` option try to identify which thread executes each task and how many tasks each thread executes (run several times if necessary).

20. Compile and run the `pi-v14-omp` version to check that its scalability remains close to the ideal case.

21. Submit the corresponding Extrae script:

    ```
    sbatch submit-extrae.sh pi-v14 10000000 4
    ```

22. Visualise it with Paraver and check if there is any visible difference between the execution of this last version and that of `pi-v11-omp` or `pi-v13-omp`. We have prepared the configuration file `pi-v14.cfg` with all the Paraver timelines configured for you. Do not forget to execute the command `Fit time scale` to view the entire execution and later zoom in the interesting parts.

2.1.2 Summary of the code versions

Table 2.1 summarises all the code versions used in the previous section, where changes accumulate from one version to the next.

Table 2.1: Code versions with explicit tasks.

Version	Description of changes	Correct?
v9	Use of `task` construct	No
v10	Use of `single` construct for a single task generator	No
v11	Use of `taskgroup` and reductions (`task_reduction` and `in_reduction`)	Yes
v12	Use of `taskwait` and `atomic`	Yes
v13	Use of `task` with data dependencies (`depend` clause)	Yes
v14	Use of `taskloop` to generate tasks from loop iterations	Yes

2.1.3 Discussion at the laboratory

Now you will go through a set of simple examples that will be helpful to practise the components of the OpenMP programming model that have been used to parallelise the computation of pi:

1. single,
2. fibtasks,
3. taskloop,
4. reduction,
5. synchtasks.

For each example, try to answer the question(s) that you will find within the source code and ask your professor in case you have any doubts. The way to proceed with the exercises is the following:

1. Go to the excersises directory:

   ```
   cd ~/lab2/openmp/day2
   ```

2. Open each C file in that directory (the examples are ordered), compile it and execute the binary (several executions may sometimes be needed).

3. For this, use the appropriate entry in the `Makefile` corresponding to each program, for example

   ```
   make 1.single
   ```

4. Next, execute the generated binary interactively via

   ```
   ./1.single
   ```

5. Or, if you want to set the number of OpenMP threads (e.g. 4)

   ```
   OMP_NUM_THREADS=4 ./1.single
   ```

6. Answer the question(s) associated with each code.

7. If asked in the program, proceed with the required modifications and execute the program again.

8. In case you need to know more details about OpenMP constructs, we suggest you to either use the complete specification document or the short reference guide available at Atenea or at https://www.openmp.org.

2.2 Observing overheads

2.2.1 Thread creation and termination

To finish this section we propose you to measure the overheads related to the creation of `parallel` regions as well as `task` creation and synchronisation. We ask you to carry out the following steps and also take note of all the obtained results in order to draw your conclusions about the

overheads associated with these OpenMP constructs (the codes are available at the `lab2/overheads` directory):

1. Open the `pi_omp_parallel.c` file and look at the changes done to the parallel version of pi.
2. The function `difference` returns the difference in time between the sequential and parallel execution when using a certain number of threads, by employing the OpenMP intrinsic function `omp_set_num_threads` to change the number of threads.
3. The execution of both the sequential and parallel versions is repeated a total of `NUMITERS` times in order to average the execution time of one iteration.
4. In the `main` program, a loop iterates over the number of threads (from `2` to `max_threads`), which is one of the input arguments of the execution.
5. The program also prints the difference in time reported by `difference`, which is the overhead introduced by the `parallel` construct (try to understand what is printed there).
6. Compile the code using the appropriate target in the Makefile and submit the execution of the generated binary `pi_omp_parallel` with one iteration and twenty threads (do not expect to obtain the correct value of pi):

   ```
   sbatch submit-omp.sh pi_omp_parallel 1 20
   ```

2.2.2 Task creation and synchronisation

Like in the previous section, please follow the steps described below and take note of the results in order to draw your conclusions:

1. Open the `pi_omp_tasks.c` file. This version creates tasks inside the function `difference` by a single thread in the parallel region.
2. The function `difference` measures the difference between the sequential execution and the version that creates the tasks.
3. Each version is repeated a total of `NUMITERS` times in order to average the execution time of one iteration.
4. In the `main` program, a loop iterates over the number of tasks that we want to generate (from `MINTASKS` to `MAXTASKS`).
5. The program also prints the difference in time reported by `difference`, which is the overhead introduced by both the `task` and `taskwait` constructs (try to understand what is printed there).
6. Compile the code using the appropriate target in the Makefile and submit the execution of the generated binary `pi_omp_tasks` with ten iterations and one thread (do not expect to obtain the correct value of pi):

   ```
   sbatch submit-omp.sh pi_omp_tasks 10 1
   ```

2.2.3 Discussion at the laboratory

We ask you to answer the following questions:

- How does the overhead of creating/terminating threads change 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?
- How does the overhead of creating/synchronising (explicit) tasks change with the number of tasks created?
- Which is the order of magnitude for the overhead of creating/synchronising each individual (explicit) task?