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

Ll. Àlvarez, E. Ayguadé, R. M. Badia, J. R. Herrero,
D. Jiménez, J. Morillo, J. Tubella and G. Utrera

Fall 2020-21

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"`.

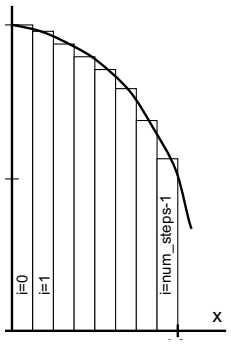
1

A very practical introduction to OpenMP (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 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 (by default set to 32 iterations in the script). In order to time the parallel execution you need to choose the second option `"make pi-vx-omp"` and queue the execution of the `submit-omp.sh` script, which is using a much larger input value (by default set to 100000000 iterations in the script). In both cases, after the script name you

only have to specify `pi-vx`. Finally, to instrument the execution with `Extrae` and visualise 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 (by default set to 100000 iterations in the script). In all these scripts, the number of processors 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 for execution the script (e.g. `sbatch submit-omp.sh pi-vx 1000000000 8`).

1.2 Parallelisation with OpenMP

In this section you will first learn how to define a parallel region to create a "team of threads" that will execute in parallel. Each one of the threads created in the parallel region will execute a so-called "implicit task" that corresponds with the body of the parallel region. You will also 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 (using different synchronisation constructs). Then in the next session you will go one step further into the so-called "explicit tasks", a much more versatile way to express parallelism in OpenMP. First, one of the threads (implicit task) in the team will be selected as the responsible for generating the explicits tasks that will be executed by the other threads (implicit tasks) in the team, learning 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 tasking model in OpenMP is.

1.2.1 Defining the parallel region and using implicit tasks

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. Execute both the `-debug` and `-omp` binaries to have a reference for the result that is obtained and the execution time for the sequential version.

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, as you will realise when executing the `-debug` binary; 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. By executing the `-debug` binary you will 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 in the previous version, `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. Compile and execute the `-debug` binary. Which iterations is each thread executing? Observe that the result is not correct due to a race condition (perhaps you don't see the error when using the small input value, but for sure you will see if you execute the `-omp` binary). The access to which variable is causing the data race, and therefore the incorrect result?

1.2.2 Using synchronisation 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. Compile and execute the `-debug` binary to check the result. This version should be correct although, as you will observe when submitting the execution of the `-omp` binary, it introduces large synchronisation 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 submitting the `-omp` binaries for execution. In any case, still the execution of `pi-v5.c` is very inefficient.

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`. Check that the result is correct by executing the `-debug` binary and check scalability by submitting the `-omp` binary; the execution time should be close to $1/4^{th}$ of the original sequential time.

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 in terms of execution time.

5. If you have not done it before, this would be a good time to submit the `Extrae` instrumented version and visualise the parallel execution with `Paraver`.

6. Finally `pi-v8.c` introduces the `barrier` synchronisation construct, as a way to define a point in your parallel where all threads wait for each other to arrive. Execute the `-omp` binary and observe the partial values for pi that each thread is printing after the thread `barrier`, once its computation loop. Where is the reduction operation specified in the `parallel` construct happening?

1.2.3 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?
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	Use of `barrier` construct	yes

1.3 Test your understanding

Next you will go through a set of 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 directory `lab2/openmp/Day1`. Open each file in the directory (the examples are ordered), compile and execute; then answer the question(s) in the questionnaire associated to it. If asked in the program, do the required modifications.

- In case you want (or need) to know more details about the `OpenMP` constructs, we suggest you either use the complete specification document or the short reference guide that you will find either at the *www.openmp.org* website or at 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).

1.4 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_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`.
- `pi_omp_reduction.c`: it makes use of the `reduction` clause applied to the global variable `sum`. This version is equivalent to `pi-v7`.

Take a look at the four different versions and make sure you understand them. How many synchronisation operations (`critical` or `atomic`) are executed in each version? Observe that function `difference` returns the difference in time between the ideal parallel execution time (sequential divided by number of threads used) and the real parallel execution when using each kind of synchronisation. This value is printed at the end of the execution.

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. Can you quantify (in microseconds) the cost of each individual synchronisation operation (`critical` or `atomic`) that is used?. You will have to write your conclusions 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 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

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. First, one of the threads (implicit task) in the team will be selected as the responsible for generating the explicits tasks that will be executed by the other threads (implicit tasks) in the team, learning 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 tasking model in OpenMP is.

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) 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. Default values for the number of iterations and number of processors are used in these scripts. Take a look at them to remember those values and how to change them at execution or submission time.

2.1.1 Tasking execution model and use of explicit tasks

1. In `pi-v9.c` we have changed the code so that the original loop is manually divided in two halves, each one computing half of the loop iterations. For each loop then an explicit task is defined, using the `task` construct. 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 of threads. Observe that each task defines a private copy of variables `i` and `x`. If you execute the `-debug` binary you will see two things: 1) each iteration of the loop is executed 4 times; and of course, 2) the value of pi is not correct. Can you guess why these two effects are happening? In order to give an answer to 1) think about which thread(s) is(are) generating tasks and how many explicit tasks are generated in total.

2. In order to solve problem 1) above, `pi-v10.c` makes use of the `single` construct so that only one thread generates the tasks; the others threads will skip the `single` construct and wait at its end;

in the meanwhile, they will look in the pool of tasks for tasks to execute. Compile and run the `-debug` binary for this version and check the result that is obtained. Why the result is still not correct?

3. In order to have the correct value for pi we need to do a couple of things in our program. First, the thread that generates the two tasks has to wait for them to complete before executing the statement "pi = step * sum"; observe that this statement needs a correct value for variable `sum`, which is jointly computed by the two tasks being generated. This will require task synchronisation. And second, the two tasks have to contribute with their partial result for variable `sum`; this will require the use of one of the constructs used in versions `v4` (critical), `v5` (atomic) or `v7` (reduction). Assuming that you would suggest the use of reduction (it was the most efficient option of the three since it minimises the number of task interactions), version `pi-v11.c` introduces the use of `taskgroup` and the `task_reduction` and `in_reduction` clauses. Take a look at the code and try to understand it: first, the `taskgroup` construct defines a task barrier at its end, so that the thread that is generating the tasks waits until they finish; second, the `taskgroup` construct also defines a reduction operation (`task_reduction(+: sum)` clause) that will occur at its end, gathering the contributions for variable `sum` from the 2 tasks generated; and third, each one of the tasks specifies that it will contribute to the reduction with its results for variable `sum` (with the `in_reduction(+: sum) clause`). Compile and run the `-debug` binary to check that the result is correct. Compile and run the `-omp` version to measure its scalability and check that it is close to the ideal case.

4. If you have not done it before, this would be a good time to submit the `Extrae` instrumented version and visualise the parallel execution of this code version with `Paraver`.

5. `pi-v12.c` provides an alternative version based on the use of `atomic` to protect the update of shared variable `sum`. In this case the code makes use of the `taskwait` construct to simply wait for the termination of the tasks: it forces the thread that entered into the `single` region and created the two tasks to wait for their termination before using the global value for `sum` that they have computed. Execute both the `-debug` and `-omp` binaries to validate the execution and observe again the large overhead introduced by the use of `atomic` (which would be much terrible if `critical` had been used).

6. `pi-v13.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 specification the OpenMP runtime can establish the appropriate execution order for tasks so that any task is not executed until all its dependences are satisfied. This is the case for the third task defined in `pi-v13.c`, which waits for the termination of the other two. This is replacing the use of `taskgroup` and the reduction clauses in `pi-v11.c` or the use of `taskwait` and `atomic` (or `critical`) in `pi-v12.c`. Execute both the `-debug` and `-omp` binaries to validate the execution and observe the overheads of using task dependences.

7. Again, this would also be a good time to submit the `Extrae` instrumented version and visualise with `Paraver` if there is any visible difference between the execution of this version and the execution of version `pi-v11.c`.

8. Finally, `pi-v14.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 (and different values). For example, when `num_tasks(8)` is used, in the `-debug` version try to see which thread executes each task and how many tasks each thread executes; run several times if necessary. Compile and run the `-omp` version to measure its scalability and check that it is close to the ideal case.

9. Well, a final point where it would be worth to submit the `Extrae` instrumented version and visualise with `Paraver` if there is any visible difference between the execution of this last version and the execution of versions `pi-v11.c` and `pi-v13c`.

2.1.2 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?
v9	Use of `task` construct	no
v10	Use of `single` construct to have 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 dependences (`depend` clause)	yes
v14	Use of `taskloop` to generate tasks from loop iterations	yes

2.2 Test your understanding

Next you will go through a set of 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 directory `lab2/openmp/Day2`. Open each file in the directory (the examples are ordered), compile and execute; then answer the question(s) in the questionnaire associated to it. If asked in the program, do the required modifications.
- In case you want (or need) to know more details about the `OpenMP` constructs, we suggest you either use the complete specification document or the short reference guide that you will find either at the *www.openmp.org* website or at Atenea.

2.3 Observing overheads

To finish with this section we propose you to measure the overheads related with the creation of `parallel` regions, `task` creation and synchronisation. 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. Function `difference` returns 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. `"sbatch ./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 synchronisation

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. `"sbatch ./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 in *Atenea* and set the appropriate dates for the delivery. Only one file has to be submitted per group.

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.

Day 1: Parallel regions and implicit tasks

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 `"OMP_NUM_THREADS=8 ./3.how_many"`

1. What does `omp_get_num_threads` return when invoked outside and inside a parallel region?
2. Indicate the two alternatives to supersede the number of threads that is specified by the `OMP_NUM_THREADS` environment variable.
3. Which is the lifespan for each way of defining the number of threads to be used?

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)

5.datarace.c

1. Is the program executing correctly? Why?
2. Propose two alternative solutions to make it correct, without changing the structure of the code (just add directives or clauses). Explain why they make the execution correct.
3. Write an alternative distribution of iterations to implicit tasks (threads) so that each of them executes only one block of consecutive iterations (i.e. N divided by the number of threads.

6.datarace.c

1. Is the program executing correctly? Why?
2. Propose two alternative solutions to make it correct, without changing the structure of the program (just using directives or clauses) and never making use of `critical`. Explain why they make the execution correct.

7.datarace.c

1. Is the program executing correctly? If not, explain why it is not providing the correct result for one or the two variables (`countmax` and `maxvalue`)
2. Write a correct way to synchronise the execution of implicit tasks (threads) for this program.

8.barrier.c

1. Can you predict the sequence of `printf` in this program? Do threads exit from the `#pragma omp barrier` construct in any specific order?

Day 2: explicit tasks

1.single.c

1. What is the `nowait` clause doing when associated to `single`?
2. Then, can you explain why all threads contribute to the execution of the multiple instances of `single`? Why 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 tasks are executed in parallel and each iteration of the while loop is executed only once.
3. What is the `firstprivate(p)` clause doing? Comment it and execute again. What is happening with the execution? Why?

3.taskloop.c

1. Which iterations of the loops are executed by each thread for each task `grainsize` or `num_tasks` specified?
2. Change the value for `grainsize` and `num_tasks` to 5. How many iterations is now each thread executing? How is the number of iterations decided in each case?
3. Can `grainsize` and `num_tasks` be used at the same time in the same loop?
4. What is happening with the execution of tasks if the `nogroup` clause is uncommented in the first loop? Why?

4.reduction.c

1. Complete the parallelisation of the program so that the correct value for variable `sum` is returned in each `printf` statement. Note: in each part of the 3 parts of the program, all tasks generated should potentially execute in parallel.

5.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), trying to achieve the same potential parallelism that was obtained when using `depend`.
3. Rewrite the program using only `taskgroup` as task synchronisation mechanism (no `depend` clauses allowed), again trying to achieve the same potential parallelism that was obtained when using `depend`.

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.