PAR Laboratory Assignment
Lab 0: Experimental setup, tools and programming model

E. Ayguadé, J. R. Herrero, D. Jiménez,
J. Labarta, N. Navarro, J. Tubella and G. Utrera

Spring 2013-14

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Contents

Note: Each chapter in this document corresponds to a laboratory session (2 hours). There are two **deliverables**: one after session 3 and another after session 6. Your professor will open them at the Raco and set the appropriate delivery dates.

Session 1

Experimental setup

The objective of this chapter is to become familiar with the environment that will be used during the semester to do the laboratory assignments. All sessions will be done using your PC/terminal booted with Linux to access a multiprocessor server located at the Computer Architecture Department. You will need to establish a connection to the server using secure shell: `"ssh -X parXXYY@boada.ac.upc.edu"`, being `XXYY` the user number assigned to you. Option `-X` is necessary in order to forward the X11 and be able to open remote windows in your local desktop.

There are two ways to execute your programs: 1) via a queueing system or 2) interactively. We strongly suggest to use option 1) when you want to ensure that the execution is done in isolation inside a single node of the machine. Using option 2) your execution has a limit of time and will share resources with other programs and interactive jobs, not ensuring representative timing results. Usually, we will provide scripts for both options (`submit-xxxx.sh` and `run-xxxx.sh`, respectively).

In order to submit a job to the execution queue, use `"qsub -l execution submit-xxxx.sh"`. If you do not specify the name of the queue with `"-l execution"` your script will not be run. In the script you configure some environment variables for the application and the queue and run your program. Use `"qstat"` to ask the system about the status of your job submission. You can use `"qdel"` to remove a job from the queueing system. In addition, in order to test your codes with no time limit you may want to use the `batch` queue; jobs submitted to this queue are executed in the interactive node and therefore are not executed in isolation for timing purposes.

All files (`lab0.tar.gz` and `environment.bash`) necessary to do this laboratory assignment are available in `/scratch/nas/1/par0/sessions`. Copy them to your home directory in `boada.ac.upc.edu`, uncompress `lab0.tar.gz` with `"tar -zxvf lab0.tar.gz"` and process `environment.bash` with `"source environment.bash"` in order to set all necessary environment variables).

1.1 Node architecture and memory

Execute the `lscpu` and `"more /proc/meminfo"` commands to know:

1. the number of sockets, cores per socket and threads per core in a node of the machine;
2. the cache memory hierarchy (L1, L2 and L3), private or shared to each core/socket;
   - You can find more information about the cache memory hierarchy (and the machine) by exploring the `/sys/devices/system/cpu` directory.
3. the amount of main memory in a node of the machine.

1.2 Serial compilation and execution

For this first part of the laboratory assignment you are going to use the `pi_seq.c` source code, which you can find inside the `environment/seq` directory. `pi_seq.c` performs the computation of the pi number by computing the numerical integral of a certain function, doing a number of iterations to compute the integral.

1. Open the `Makefile` file, identify the `target` you have to use to compile the sequential code. Observe how the compiler is invoked and the options used (you can use `"man gcc"` to know their meaning). Execute the command line `"make target_identified"` in order to generate the binary executable file.

2. Interactively execute the binary generated to compute the pi number by doing 1.000.000.000 iterations using the `run-seq.sh` script, which returns the user and system CPU time, the elapsed time, and the % of CPU used (using GNU `/usr/bin/time`). In addition, the program itself also reports the elapsed execution time using `gettimeofday`. Look at the source code and identify the function invocations and data structures required to measure execution time.

3. Submit the execution to the queueing system using the `"qsub submit-seq.sh"` command. Use `"qstat"` to see that your script is queued but no run since you have not specified the `execution` queue. Identify your job-ID number in the `"qstat"` output and use `"qdel job-ID-number"` to remove it from the queue. Submit the execution to the queueing system using the `"qsub -l execution submit-seq.sh"` command and use `"qstat"` to see that your script is running. Look at `submit-seq.sh` script and the results generated (the standard output and error of the script and the `pi_seq_time.txt` file).

1.3 Compilation and execution of OpenMP programs

For all laboratory assignments in this course we are going to compile and execute parallel codes in `OpenMP`, the standard for parallel programming using shared-memory. `OpenMP` will be explained in more detail later in this same laboratory assignment; but by now, we will just see how to compile and execute parallel programs in `OpenMP`.

1.3.1 Compiling OpenMP programs

1. Go into the `environment/omp` directory, where you will find an `OpenMP` version of the code for doing the computation of pi (`pi_omp.c`[1]). Compile it with the same compilation options you used to compile `pi_seq.c` in the previous section (the target is already prepared in the `Makefile`). What is the compiler telling you? Is the compiler generating an executable file?

2. Figure out what is the option you have to add to the compilation line in order to be able to execute the `pi_omp.c` in parallel (using `"man gcc"`).

3. Generate the `OpenMP` executables of the `pi_omp.c` source code by adding the necessary compilation flag in the `Makefile`. Double check to be sure that the compiler has compiled `pi_omp.c` again with the new compilation flag provided.

1.3.2 Executing OpenMP programs

1. Interactively run the `OpenMP` executable with 8 threads using the `run-omp.sh` script. What is the `time` command telling you about the user and system CPU time, the elapsed time, and the % of CPU used? Take a look at the script to discover how do we specify the number of threads to use in `OpenMP`.

[1]Other versions also available, to be used in the next session.

2. Use `submit-omp.sh` script to measure the CPU time, elapsed time and % of CPU when executing the `OpenMP` program when using 8 threads. You should submit to the `execution` queue in order to execute in isolation in one of the nodes of the machine. Do you observe a major difference between the interactive and queued execution?

3. In order to plot the execution time with varying number of threads, we provide you with the `submit-plot-omp.sh` script which should be submitted to the queueing system. The script generates a plot in Postscript format with extension `.ps` that you can visualize with ghostscript `gs`. Submit the script specifying the execution from 1 (`np_NMIN`) to 12 (`np_NMAX`) threads (the execution will take some time, be patient!). Visualize the plot generated and reason about how the number of threads influence the execution time.

Session 2

Tracing the execution of programs

The objective of this chapter is to present you a environment (`Extrae`) to gather information about the execution of a parallel application in `OpenMP`. On one side `Extrae` provides an API (application programming interface) to manually define in the source code points where to emit events. On the other side, `Extrae` can also be used to transparently instrument the execution of `OpenMP` and gather information about the different states in the execution of a parallel program. When the instrumented binary is executed, a trace file (`.prv`, `.pcf` and `.row` files) with those states and events recorded is generated. Then, the `Paraver` trace browser (`wxparaver` should be in your path) will be used to visualize the trace and analyze the execution of the program.

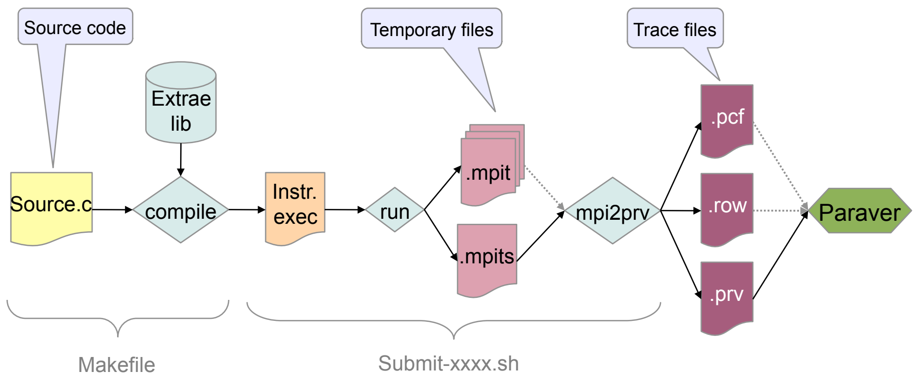


Figure 2.1: Compilation and execution flow for tracing.

2.1 Instrumentation API

Function `Extrae_event(int type, int value)` is used to emit an event in a certain point of the source code. Each event has an associated `type` and a `value`. For example we could use `type` to classify different kind of events in the program and `value` to differentiate different occurrences of the same `type`. In the instrumented codes that we provide you (`pi_omp_sum_local.c` and `pi_omp_critical.c` inside the `environment/omp` directory) this function is invoked to trace the entry and exit to different program regions. In particular, a constant value for the `type` argument is used (`PROGRAM` with value 1000) and different values for the `value` argument are used to trace the entry to different program regions, as shown below (defined in the `constants.h` file):

```
// Extrae Constants
#define  PROGRAM        1000
#define  END            0
#define  PI_COMPUTATION  1
```

```
#define  REST_MAIN       2
#define  TIMING          3
...
Extrae_event (PROGRAM, TIMING);
START_COUNT_TIME;
Extrae_event (PROGRAM, END);
...
```

In addition, two additional functions (`Extrae_init` and `Extrae_fini`) need to be invoked in order to use the `Extrae` instrumentation library.

1. Edit one of the two source codes provided (`pi_omp_sum_local.c` and `pi_omp_critical.c`) to observe how the instrumentation is done and which code regions are identified.
2. Open the `Makefile` and identify the targets to compile both `OpenMP` programs. Observe that we specify the location of the `Extrae` include file (`IINCL=-I$(EXTRAE_HOME)/include`) and library (`-ILIBS=-L$(EXTRAE_HOME)/lib -lomptrace`). Make sure that the appropriate flag for compilation of `OpenMP` is applied to them. Compile both programs with `Makefile`.
3. Use the `submit-omp-i.sh` script to execute each binary generated and to generate the traces of their parallel execution. Notice that the name of the binary is specified inside the script file. The script will invoke your binary, which will call the `Extrae` library to emit evens at runtime; the script also invokes `mpi2prv` to generate the final trace (`.prv`, `.pcf` and `.row`). Execute them with 8 threads. In these executions we use a small number of iterations in order to avoid the generation of very large trace files. The script generates a filename whose name includes the size of the problem and the number of threads used for the execution.

2.2 Paraver hands on

In one of the laboratory sessions your professor will do a hands-on to show the main features of `Paraver`, a graphical browser of the traces generated with `Extrae`, and the set of configuration files to be used to visualize and analyze the execution of your program. A guide for this hands-on can be found in the *Intro2ParaverPAR.pdf* document available through the Raco. Traces for the two parallel versions should have already been generated as a result of the previous section. The configuration files that are necessary to use `Paraver` can be copied from `/scratch/nas/1/par0/sessions/cfgs.tar.gz` and are described in the following table.

User events	Timeline showing ...
APP_userevents	type 1000 events manually introduced by programmer
OpenMP	
OMP_in_barrier	when threads are in a barrier synchronization
OMP_in_lock	when threads are in/out/entering/exiting critical sections
OMP_in_schedforkjoin	when threads are scheduling work, forking or joining
OMP_parallel_functions	the parallel function each thread is executing
OMP_parallel_functions_duration	the duration for the parallel functions
Profiles showing ...	
APP_userevents_profile	the duration for type 1000 events
OMP_profile	the time spent in different OpenMP states (useful, schedforkjoin, barrier, ...)
OMP_critical_profile	the total time, percentage of time, number of instances or average duration spent in the three phases of the critical section
OMP_critical_duration_histogram	histogram of the duration of the different phases of the critical section

Session 3

Analysis of task decompositions using Tareador

In this chapter we will introduce *Tareador*, an environment to analyze the potential parallelism that could be obtained when a certain parallelization strategy (task decomposition) is applied to your sequential code. *Tareador* 1) traces the execution of the program based on the specification of potential tasks to be run in parallel, 2) records all static/dynamic data allocations and memory accesses in order to build the task dependence graph, and 3) simulates the parallel execution of the tasks on a certain number of processors in order to estimate the potential speed-up.

3.1 Using *Tareador*

Tareador offers an API to specify code regions to be considered as potential tasks:

```
tareador_start_task("name of task");
/* Code region to be a potential task */
tareador_end_task();
```

The `name of task` string specified in `tareador_start_task` identifies that task in the graph produced by `Tareador`. In order to enable the analysis with *Tareador*, the programmer must invoke:

```
tareador_ON();
...
tareador_OFF();
```

at the beginning and end of the program, respectively. Make sure both calls are always executed for any possible entry/exit points to/from your main program.

For the first part of this session you will use a very simple program that performs the computation of the dot product (`result`) of two vectors (`A` and `B`) of size 16. The program first initializes both vectors and then calls function `dot_product` in order to perform the actual computation:

1. Go into the `environment/dot_product` directory, edit the `dot_product.c` source code and identify the calls to the instrumentation functions mentioned above. Edit the `Makefile` to understand how the source code is compiled and linked to produce the executable. Generate the executable by doing ("`make dot_product`").

2. Execute the *Tareador* environment by invoking the `./run_tareador.sh` script[1]. This will open a new window in which the task dependence graph is visualized (see Figure 3.1, left). Each node of the graph represents a task: different shapes and colors are used to identify task instances generated from the same task definition and each one labeled with a task instance number. In addition, each node contains the number of instructions that the task instance has executed, as an indication of the task granularity; the size of the node also reflects in some way this task granularity. Edges in the

[1]This script simply invokes "`tareador_gui.py --lite`" followed by the name of the executable.

graph represent dependencies between task instances; different colors/patterns are used to represent different kind of dependences (blue: data dependences, green/yellow: control dependences).

3. *Tareador* allows you to analyze the data that create the data dependences between nodes in the task graph. With the mouse on node `dot_product`, right click with the mouse and select *Dataview* → *Edges-in*. This will open a window similar to the one shown in Figure 3.1, right/top. In the *Task view* tab, you can see the variables that are read (i.e. with a load memory access, green color in the window) by the task selected (for example `dot_product`) and written (i.e. with a store memory access, blue color in the window) by any of the tasks that are source of a dependence with it (in this case, either `init_A` or `init_B` as selected in the chooser). In this tab, the orange color is used to represent data that is written by the source task and read by the destination task, i.e. a data dependence. For each variable in the list you have its name and its storage (G: global, H: heap – for dynamically allocated data, or S: stack – for function local variables); additional information is obtained by placing the mouse on the name (size and allocation) and when doing right click with the mouse on the bar that represents a data access (offsets inside the object in bytes). In the *Data view* tab you can see for each variable (selected in the chooser) the kind of access (store, load or both, using the same colors) performed by each task. Similarly you can choose *Dataview* → *Edges-out* to analyze data accesses for those edges going out of a task node (for example for the tasks initializing the vectors `init_A` or `init_B`).

4. Once you understand the data dependences and the task graph generated, simulate the execution of the initial task decomposition, for example with 4 processors, by clicking *View Simulation* in the main *Tareador* window. A *Paraver* window will appear showing the timeline for the simulated execution, similar to the one shown in Figure 3.1, right/bottom. Colors are used to represent the different tasks (same colors that are used in the task graph). You can activate the visualization of dependencies between tasks by selecting View → Communication Lines when you click the right button of the mouse.

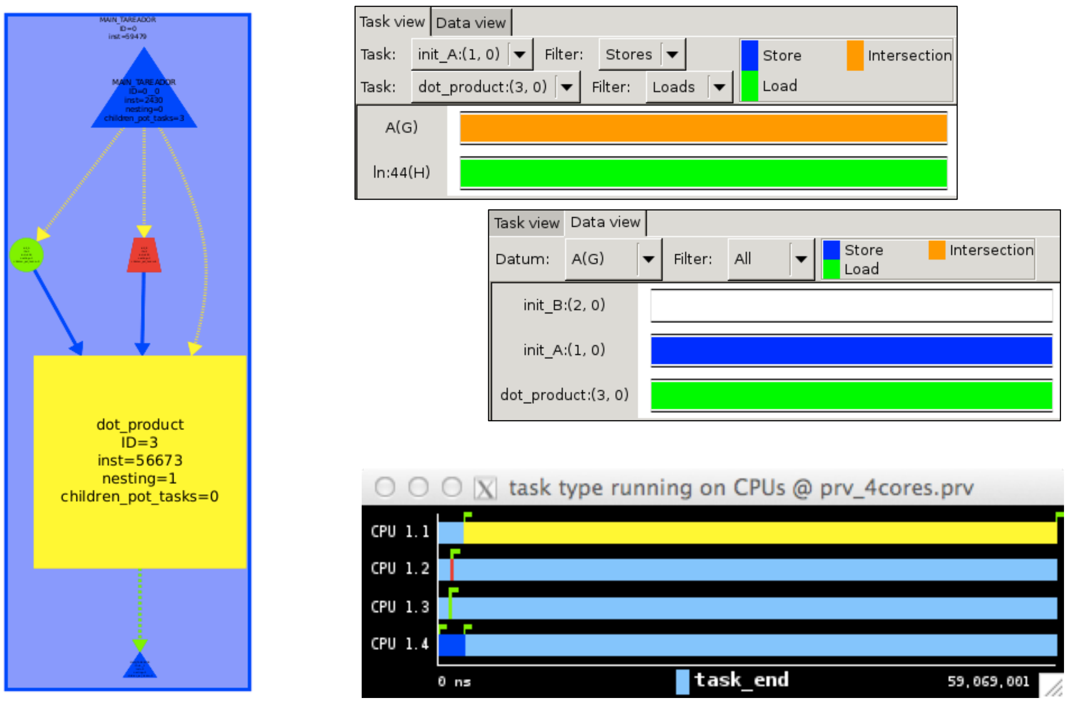


Figure 3.1: Left: Initial task dependence graph for `dot_product`. Right/Top: Visualization of data involved in task dependencies. Right/Bottom: *Paraver* visualization of the simulated execution with 4 processors.

Next you will refine the initial task decomposition in order to exploit additional parallelism inside the computation of the `dot_product` task:

5. Edit the source code `dot_product.c` and use `tareador_start_task` and `tareador_end_task` to identify as a potential task each iteration of the loop inside the `dot_product` function. Compile the source code and execute *Tareador* in order to visualize the task dependence graph and simulate the execution with 4 processors. Do you observe any improvement in the parallelism that is obtained?

6. With the *Dataview* option in *Tareador* identify the variables that are sequentializing the execution of the new tasks and locate the instructions in the source code where these dependences are created. Once you know which variables are causing these dependences, you can temporarily filter their analysis by using the following functions in the *Tareador* API:

   ```
   tareador_disable_object(&name_var)
   // ... code region with memory accesses to variable name_var
   tareador_enable_object(&name_var)
   ```

   Insert these calls into the source code in order to disable the variables that you have identified. Compile and run *Tareador* again. Are you increasing the parallelism? Perform the simulation with different number of processors and observe how the execution time changes. How will you handle the dependences caused by the accesses to these variables?

7. You can save the task dependence graph generated by clicking the *Save results* button.

3.2 Exploring task decompositions

Go into the `environment/3dfft` directory, edit the `3dfft_seq.c` source code and identify the calls to the *Tareador* API, understanding the tasks that are initially defined.

1. Generate the executable and instrument it with the "`./run_tareador`" script. Analyze the task dependence graph and the dependences that are visualized, using the *Dataview* option in *Tareador*.

2. Next you will be refining the potential tasks with the objective of discovering more parallelism in `3dfft_seq.c`. You will incrementally generate four new task decompositions (named v1, v2, v3 and v4) as described in the following bullets. For the original and the four new decompositions compute T_1, T_∞ and the potential parallelism from the task dependence graph generated by *Tareador*, assuming that each instruction takes one time unit to execute.

   (a) Version v1: REPLACE[2] the task named `ffts1_and_transpositions` with a sequence of finer grained tasks, one for each function invocation inside it.

   (b) Version v2: starting from v1, REPLACE the definition of tasks associated to function invocations `ffts1_planes` with fine-grained tasks defined inside the function body and associated to individual iterations of the `k` loop, as shown below:

   ```
   void ffts1_planes(fftwf_plan p1d, fftwf_complex in_fftw[][N][N])
   {
       int k,j;

       for (k=0; k<N; k++)  {
        tareador_start_task("ffts1_planes_loop_k");
        for (j=0; j<N; j++)
          fftwf_execute_dft( p1d, (fftwf_complex *)in_fftw[k][j][0],
                                  (fftwf_complex *)in_fftw[k][j][0]);
        tareador_end_task();
       }
   }
   ```

[2]REPLACE means: 1) remove the original task definition and 2) add the new ones.

(c) Version v3: starting from v2, REPLACE the definition of tasks associated to function invocations `transpose_xy_planes` and `transpose_zx_planes` with fine-grained tasks inside the corresponding body functions and associated to individual iterations of the `k` loop, as you did in version v2 for `ffts1_planes`.

(d) Version v4: starting from v3, propose which should be the next task(s) to decompose with fine-grained tasks?. Modify the source code to instrument this task decomposition.

For version v4, simulate the parallel execution for 2, 4, 8, 16 and 32. For the original (`seq`) decomposition, simulate its execution time with just 1 processor; the time reported in the trace will be used to compute the speed-up obtained by v4 when using different numbers of processors, as requested in the first deliverable.

First deliverable

Deliver a document in PDF format (other formats will not be accepted) containing the answers to the following questions. 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. Only one file has to be submitted per group through the Raco website.

Node architecture and memory

1. Describe (better if you do a simple drawing) the architecture of the computer in which you are doing this lab session (number of sockets, cores per socket, threads per core, cache hierarchy size and sharing, and amount of main memory).

Timing sequential and parallel executions

2. Indicate the library header where the structure `struct timeval` is declared and which are its fields.

3. Plot the execution time when varying the number of threads for `pi_omp.c`. Reason about how the number of threads influence on the execution time.

Tracing parallel executions

4. From the two instrumented OpenMP versions (`pi_omp_sum_local.c` and `pi_omp_critical.c`), show a profile of the % of time spent in the different OpenMP states. Reason about the differences observed and guess which is the construct in the source code that is the cause of these differences.

5. For the two OpenMP versions provided, fill in the following table with the time elapsed in each part of the code and the total elapsed time of the instrumented code (using 8 threads).

	REST_MAIN	PI_COMPUTATION	TIMING
`pi_omp_sum_local.c`			
`pi_omp_critical.c`			

Visualizing the task graph and data dependences

6. Include the source code for function `dot_product` in which you show the instrumentation that has been added in order to identify tasks and filter the analysis of variable(s) that cause the dependence(s).

7. Capture the task dependence graph and execution timeline (for 8 processors) for that task decomposition.

Analysis of task decompositions

8. Complete the following table for the initial and different versions generated for `3dfft_seq.c`.

Version	T_1	T_∞	**Parallelism**
seq			
v1			
v2			
v3			
v4			

9. With the results from the parallel simulation with 2, 4, 8, 16 and 32 processors, draw the execution time and speedup plots for version `v4` with respect to the sequential execution (that you can estimate from the simulation of the initial task decomposition that we provided in `3dfft_seq.c`, using just 1 processor).

Session 4

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 `openmp/pi` directory.

4.1 Computing number Pi

As an example we will use a program that computes the number pi by solving the equation in Figure 4.1. The equation can be solved by computing the area defined by the function, which at its turn it can be approximated by dividing the area into small rectangles and adding up its area. Figure 4.2 shows the sequential code for pi computation: To distribute the work for the parallel version each processor will be responsible for computing some rectangles (in other words, to execute some iterations of the `i` loop). It should be guaranteed that all processors have computed their `sum` before combining it into the final value.

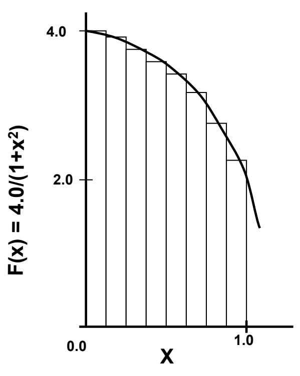
In order to parallelize the sequential code we will proceed through a set of versions `"pi-vx.c"`, being `x` the version number. We provide two different entries in the `Makefile` to compile them: `"make pi-vx-omp"` and `"make pi-vx-omp-i"`; the first one will generate the non–instrumented binary for regular execution while the second one will generate an instrumented binary that will generate a *Paraver* trace. We will execute the non–instrumented binary with a very small input (e.g. `./run-omp.sh pi-vx-omp 16`) to check which iterations are executed by each thread and the value of pi that is computed. For the instrumented binary we will do executions with a larger number of iterations (e.g. `./run-omp-i.sh pi-vx-omp-i 100000`) to visualize the trace generated and observe the parallel behavior.

4.2 Parallelization with OpenMP

1. Compile and run (both non–instrumented and instrumented) 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. Compile both `pi-v0-omp` and `pi-v0-omp-i` and execute. 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 the team of threads (or reuses them if they have been created before). In OpenMP all variables are shared by default and usually some of them would need to be privatized. This code is NOT correct: all threads execute the body of the parallel region and the loop control variable is shared.

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.



Mathematically, we know that:

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

We can approximate the integral as a sum of rectangles:

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

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

Figure 4.1: Pi computation

```
static long num_steps = 100000;
void main ()
{
         int i;
         double x, pi, step, sum = 0.0;

         step = 1.0/(double) num_steps;

         for (i=1;i<= num_steps; i++) {
                 x = (i-0.5)*step;
                 sum = sum + 4.0/(1.0+x*x);
         }
         pi = step * sum;
}
```

Figure 4.2: Serial code for Pi

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. Now iterations are distributed but the result is not correct due to a race condition. Which variable is causing it? In any case, observe with *Paraver* how the execution time of the loop is reduced.

5. Next version `pi-v4.c` uses the `critical` construct to provide a region of mutual exclusion where only one thread can be working at any given time. An alternative implementation could use the lock mechanism provided by the OpenMP runtime. This version should be correct, although the execution time is excessively large due to synchronization overheads, as observed with *Paraver*.

6. 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`. Run it and compare the time with the previous version. *Paraver* does not visualizes the execution of `atomic` regions.

7. An alternative solution is used in `pi-v6.c`: the `reduction` clause. Reduction is a very common pattern where all threads accumulate values into a single variable. The compiler creates a private copy of the reduction variable and at the end of the region, the runtime ensures that the shared variable is properly updated with the partial values of each thread, using the specified operator. Run it and compare the time with the previous version. Observe with *Paraver* the execution timeline.

8. Next we will use the `for` construct to distribute the iterations of the loop among the threads of

the team. 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.

9. The for construct accepts an schedule clause to determine which iterations are executed by each thread. There are three different options as schedule: (i) static (ii) dynamic and (iii) guided. An explanation of each schedule and its options can be found in the slides from page 52 to 54. Review how each schedule works and use the provided examples to try out the different schedules: pi-v7.c and pi-v8.c for static; pi-v9.c and pi-v10.c for dynamic; and pi-v11.c for guided. Observe with Paraver the overheads introduced by dynamic and guided due to the *scheduling* points where iterations are assigned to threads.

10. In version pi-v12.c we have artificially added a new parallel region to do the final computation of pi=step*sum. The version is still correct because the replicated execution of that instruction produces the same result. In this version we show how to decide inside the program the number of threads to be used in a parallel region: num_threads clause and omp_set_num_threads intrinsic. Use *Paraver* to visualize how many threads are used in each parallel region.

11. Version pi-v13.c artificially divides the computational loop in two loops and forces each one of the two new loops to be executed by just two threads. Observe with *Paraver* how the execution of the two loops is serialized. Why? Version pi-v14.c makes use of the nowait clause in an attempt to execute both loops in parallel, but still their execution is sequentialized. Why? Version pi-v15.c changes the loop scheduling to dynamic, achieving the desired behavior, as observed with *Paraver*. Why?

12. Version pi-v16.c exemplifies the use of the single construct. We have modified the code and moved the instruction pi=step*sum inside the parallel region. To make sure that only one thread of the team will execute the structured block we must use the single construct. Run the code, is it correct?

13. Unfortunately the result is not correct because there is still the nowait clause after the second loop and the implicit barrier at the end of this loop has been removed. We can force the required synchronization by removing the second nowait or introducing a barrier construct. Compile and run version pi-v17.c; the code is still not correct. Why?

14. Version pi-v18.c finally solves the problem by appropriately placing the reduction clause in each one of the loops. Check the result.

15. pi-v19.c exemplifies the use of the **task** construct, defining a task to compute half of the total number of iterations. 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 atomic to protect the data race and the task wait construct to wait for the termination of all tasks generated. This version is not correct since tasks are generated as many times as threads in the parallel region, due to the replication of the body in the parallel region. Observe this with *Paraver*. Version pi-v20.c makes use of the single construct to make the parallelization correct; visualize again with *Paraver* to verify it.

16. Finally pi-v21.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: observe with *Paraver* that the task granularity is too fine and one of the threads is busy all time just generating tasks for the rest of the threads. In the case of the pi program, using the for construct is the more effective way of parallelization; however when the number of iterations of the loop is unknown the task construct is necessary.

4.3 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	Defining the number of threads: `omp_set_num_threads` and `num_threads`	yes
v13–15	Use of `nowait` clause	yes
v16	Use of `single` construct	no
v17	Use of `barrier` construct	no
v18	`reduction` clause revisited	yes
v19	Use of `task` and `taskwait` constructs	no
v20	Use of `single` to have just one task generator	yes
v21	Finer grained parallelization with tasks	yes

Session 5

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. In order to follow them, you will need:

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

5.1 OpenMP basics

1. Get into the directory called `openmp/basics`. The examples are ordered. Look at each code and try to answer the questions that are included in the source code; later you can compile and run to check your answers.
2. Consult "`Part I: OpenMP Basics`" of the tutorial slides, if necessary.

5.2 Loop parallelism

1. Get into the directory `openmp/worksharing`. Look at the *1.for.c* and *2.collapse.c* codes and predict the output before running the program; later you can compile and run to check your answers.
2. Consult "`Part II: Loop Parallelism in OpenMP`" of the tutorial slides, if necessary.

5.3 Task parallelism

1. Get into the directory `openmp/tasks`. Look at the serial version in *linked_serial.c*. It implements a linked list which is traversed and some computation is done for every node of the list in the function called `processwork`. The computation consists on generating the *i*th number of the Fibonacci series where *i* is the *data* value of the node. Run the code and see how it works.
2. *linked_v1.omp.c* presents a parallel version of the same code using the **task** construct. This code is not correct, try to run it and see what happens. The problem with tasks is that they need to capture the value of the data they need when they are created. In this case this was not done and when the task was accessing *p* it was probably at the end of the list so it was pointing to *NULL*.
3. Next version (*linked_v2.omp.c*) uses the **firstprivate** clause to capture *p* at the time the task is created. Run and check the result.
4. Consult "`Part III: Task Parallelism in OpenMP`" of the tutorial slides, if necessary.

Session 6

Measuring parallelization overheads

In this chapter you will measure some of the main overheads that need to be considered in the parallel execution for a shared–memory architecture. For this chapter, you will need to go into directory `overheads`.

6.1 Thread creation and termination

First you will measure the overhead related with the creation and termination of threads, execution entities offered by the operating system to support the execution of shared-memory parallel paradigms such as `OpenMP`. For this part, go into the `overheads/threadcreation` directory.

1. Edit the `ompparallel.c` file and look for the `#pragma omp parallel`, which is used in `OpenMP` to activate a parallel region. As can be seen, this program iteratively creates parallel regions using different numbers of threads (2 to `NUMTHREADS`). Each thread invokes function `do_something` in order to perform some computations. Compile using `"make ompparallel"` and execute the binary with `qsub -l execution submit-omp.sh` (the name of the executable is specified in the script). The reported execution times represent the overhead associated with the parallel region in `OpenMP`. Do you observe any relationship between the number of threads and the overhead associated to the activation of parallel regions? Which is the order of magnitude for that overhead?
2. Compile the `Extrae`-enabled version of the `ompparallel` using `"make ompparallel-i"` and execute the instrumented version by submitting the (`submit-omp-i.sh`) script. Open the trace generated with `wxparaver` and look at the overhead associated with parallel regions (yellow bursts labeled with `"Scheduling and Fork/Join"` at the start and end) for different number of threads. For each number of threads, do you observe any significant difference between the first parallel region and the rest?

6.2 Thread synchronization

Second you will measure the overhead related with the use of `critical` as one of the mechanisms for synchronization in `OpenMP`. The files required to follow this part of the laboratory session are in the `overheads/synchronization` directory.

1. The `dotprod_serial.c` is the sequential version of a program that computes the dot product of two vectors. It will be used as reference to check the result of the dot product and its execution time. Use the `Makefile` to compile the sequential `dotprod_serial.c` program and execute with `qsub -l execution submit-seq.sh`.
2. The `dotprod_mutex_everytime.c` file contains the `OpenMP` parallelization of the dot product example. A `critical` region is used to protect the access to shared data, ensuring exclusive access to it. Take a look at the code and identify the shared data and the `critical` section. Use the `Makefile` to compile this program and execute with `qsub -l execution submit-omp.sh`(make

sure the name of the executable file in the script is appropriate). The script executes there program with 1 and 8 threads and writes the result in a `*_time.txt` file. Although the program is parallelized using 8 threads, the execution time is larger that sequential. Why?

3. The `dotprod_mutex.c` file contains a much efficient solution to protect the access to the shared variable. It is based on the use of a private "per–thread" copy followed by a global update at the end using only one `critical` region. Use the `Makefile` to compile this program and execute it by submitting the `submit-omp.sh` script. Does the program benefits from the use of several processors? From the execution times for the two previous codes, can you estimate the average cost of each mutual exclusion region in the `dotprod_mutex_everytime.c`?

4. In order to understand the numbers obtained and the sources of overhead in this example, you will generate `Paraver` traces, for 1 and 8 processors, for the execution of `dotprod_mutex_everytime.c` (look at the `Makefile` in order to see the appropriate target). Observe how the program proceeds through the 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. Do you get an intuition of the different sources of overhead that appear in this program?

6.3 Data sharing between threads

Finally, you will measure the impact of excessive (unnecessary) data sharing. The files required to follow this part of the laboratory session are in the `overheads/falsesharing` directory.

1. In file `dotprod_vectorsum.c` we provide you with a version that avoids the use of locks to guarantee the correctness of the program. Look at the code and understand how it works. Use the `Makefile` to compile it and execute by submitting the appropriate script. The result should be correct but the execution time is worse than the `dotprod_mutex.c` version. Compare the execution times for 1 and 8 threads for the `dotprod_mutex.c` and `dotprod_vectorsum.c`.

2. `dotprod_vectorsum.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, the `dotprod_vectorsum_padding.c` file 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 understand how padding is done. Compile the program using the `Makefile`, and execute. Is the execution time similar, better or worse than `dotprod_mutex.c`?.

3. Compile the `Extrae`-enabled versions of the two previous files and submit their execution with the `submit-omp-i.sh` script. Open the traces generated and load the `false_sharing.cfg` configuration file for each one. This configuration file opens 4 timelines showing the total number of cycles, total number of instructions executed, number of L3 accesses and number of snoop requests (i.e. requests to maintain coherent copies for shared cache lines). Do you observe any significant difference between the two traces?

Second deliverable

Deliver a report in PDF format (other formats will not be accepted) containing the answers to the following questions. Please, follow the same recommendations that we made for the previous deliverable. Only one file has to be submitted per group through the Raco website.

Parallelization overheads

1. Which is the overhead associated with the activation of a `parallel` region in `OpenMP`? Is it constant? Reason the answer based on the results reported by the `ompparallel` code and the the trace visualized with Paraver.

2. Which is the minimum overhead associated with the execution of `critical` regions in `OpenMP`?

3. In the presence of lock conflicts and true sharing (as it happens in `dotprod_mutex_everytime`), how the overhead associated with `critical` increases with the number of processors? How this overhead is decomposed? Reason the answer based on the results visualized with Paraver.

4. In the presence of false sharing (as it happens in `dotprod_vectorsum`), which is the additional average access latency that you observe to memory? Which causes this increase in the memory access time? Reason the answer based on the results visualized with Paraver.

Execution time and speed–up

5. Complete the following table with the execution times of the different versions of `dotprod` that we provide to you. The speed–up has to be computed with respect to the execution of the serial version. For each version and number of threads, how many executions have you performed?

version	**1 processor**	**8 processors**	**speed-up**
dotprod_serial		–	1
dotprod_mutex_everytime			
dotprod_mutex			
dotprod_vectorsum			
dotprod_vectorsum_padding			