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

E. Ayguadé, J. Corbalán, D. Jiménez,
J. I. Navarro, J. Tubella and G. Utrera

Fall 2015-16

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. From your PC/terminal booted with Linux you will access a multiprocessor server located at the Computer Architecture Department, establishing a connection to it 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. You can change the password for your account using `"ssh -t parXXYY@boada.ac.upc.edu passwd"`. Once you are managed to login, you will be placed in your "home" directory in boada, where you will copy all necessary files and do the different laboratory assignments.

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; the execution starts as soon as a node is available. When using option 2) your execution starts immediately but 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):

- Queueing a job for execution: `"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, remaining in the queue forever. In the script you can specify additional options to run the script, configure environment variables and launch the execution of 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.

- Interactive execution: `./run-xxxx.sh`. Additional parameters may be specified after the script name. Jobs interactively executed have a short time limit to be executed.

All files necessary to do this laboratory assignment are available in `/scratch/nas/1/par0/sessions`. Copy `lab0.tar.gz` from that location to your home directory in `boada.ac.upc.edu` and uncompress it with this command line: `"tar -zxvf lab0.tar.gz"`. Process the `environment.bash` file with this other command line `"source environment.bash"` in order to set all necessary environment variables. **Note:** since you have to do this every time you login in the account or open a new console window, we recommend that you add this command line in a `.profile` file in your home directory, which is executed when a new session is initiated.

1.1 Node architecture and memory

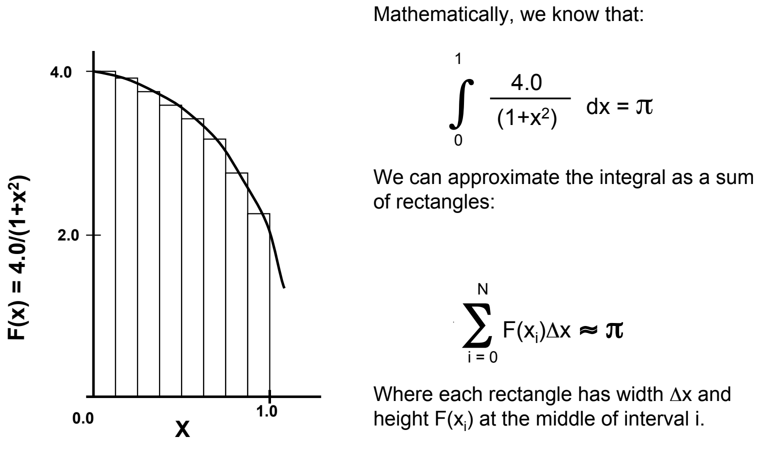
Execute the `lscpu`, `lstopo` and `"more /proc/meminfo"` commands to know:

1. the number of sockets, cores per socket and threads per core in a node of the machine;
2. the amount of main memory in a node of the machine, and each NUMAnode;
3. the cache memory hierarchy (L1, L2 and L3), private or shared to each core/socket.

Draw the architecture of each node based on the information generated by the tools above. The `"--of fig map.fig"` option for `lstopo` can be very useful for that purpose. Then you can use the `xfig` command to visualize the output file generated (`map.fig`) and export to a different format (PDF or JPG, for example) using `File` → `Export`[1].

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 `lab0/environment/pi` directory. `pi_seq.c` performs the computation of the pi number by computing the integral of the equation in Figure 1.1. 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.



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 1.1: Pi computation

Figure 1.2 shows a simplified version of the code you have in `pi_seq.c`. Variable `num_steps` defines the number of rectangles, whose area is computed in each iteration of the `i` loop.

```
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 1.2: Serial code for Pi

In the following steps you will compile `pi_seq.c` using the `Makefile` and execute it interactively and through the queueing system, with the appropriate timing commands to measure its execution time:

1. Open the `Makefile` file, identify the `target` you have to use to compile the sequential code. Observe how the compiler is invoked. Execute the command line `"make target_identified"` in order to generate the binary executable file.

[1]In the boada Linux distribution you can use `display` to visualize PDF and other graphic formats. You can also use the `"fig2dev -L pdf map.fig map.pdf"` command to convert from `.fig` to `.pdf`; look for alternative output graphic languages by typing `"man fig2dev"`.

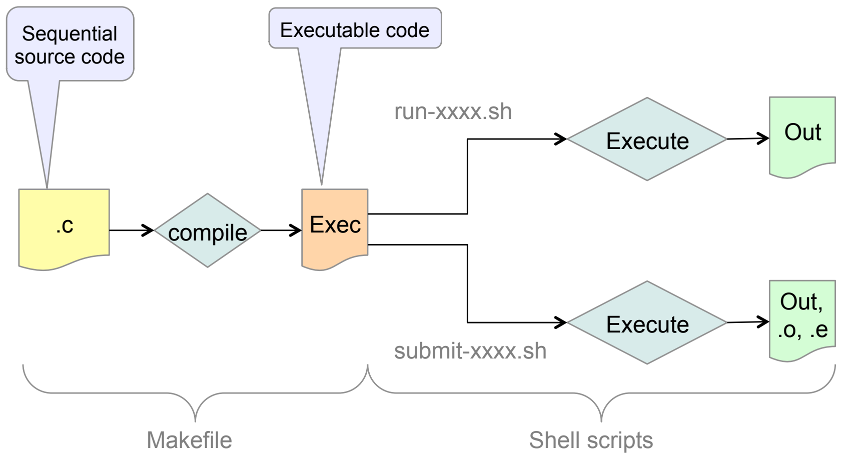


Figure 1.3: Compilation and execution flow for sequential program.

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; however it is not executed since you have not specified an `execution` queue name. 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

In this course we are going to use `OpenMP`, the standard for parallel programming using shared-memory, to express parallelism in the C programming language. Although `OpenMP` will be explained in more detail later in this same laboratory assignment, in this section we will see how to compile and execute parallel programs in `OpenMP`.

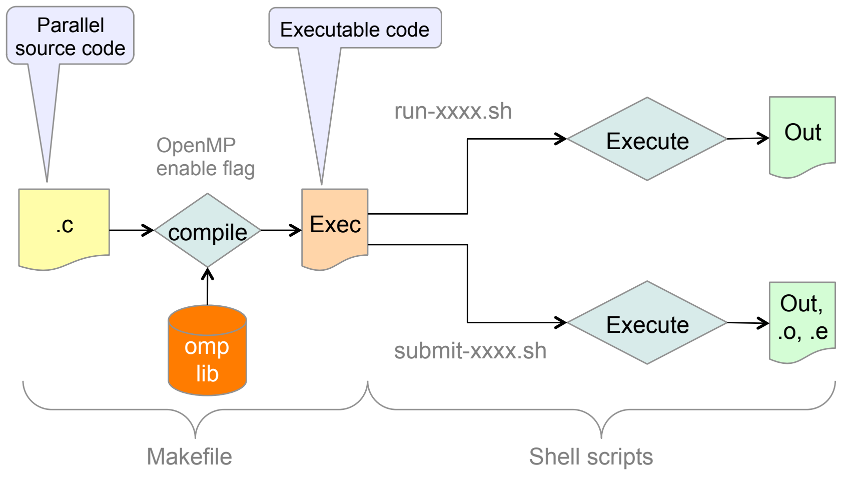


Figure 1.4: Compilation and execution flow for OpenMP.

1.3.1 Compiling OpenMP programs

1. In the same `lab0/environment/pi` directory you will find an `OpenMP` version of the code for doing the computation of pi in parallel (`pi_omp.c`). Compile the `OpenMP` code using the appropriate target in the `Makefile`). What is the compiler telling you? Is the compiler issuing a warning or an error message? 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 again the `OpenMP` executable of the `pi_omp.c` source code after 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 (i.e. `"‘pi_omp’ is up to date"` is not returned by `make`).

1.3.2 Executing OpenMP programs

1. Interactively execute the `OpenMP` code with 8 threads (processors) and same number of iterations (1.000.000.000) 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`.

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

1.3.3 Strong vs. weak scalability

Finally, in this last part of the chapter you are going to explore the scalability of the `pi_omp.c` code when varying the number of threads used to execute the parallel code. To evaluate the scalability the ratio between the sequential and the parallel execution times will be computed. Two different scenarios will be considered: *strong* and *weak* scalability.

- In *strong* scalability the number of threads is changed with a fixed problem size. In this case parallelism is used to reduce the execution time of your program.

- In *weak* scalability the problem size is proportional to the number of threads. In this case parallelism is used to increase the problem size that is executed on your program.

We provide you with two scripts, `submit-strong-omp.sh` and `submit-weak-omp.sh`, which should be submitted to the queueing system. The scripts execute the parallel code using from 1 (`np_NMIN`) to 12 (`np_NMAX`) threads. The problem size for strong scalability is 1.000.000.000 iterations; for weak scalability, the initial problem size is 100.000.000 which grows proportionally with the number of threads. As a result the script generates a plot (in Postscript format) showing the resulting execution time and speed–up. The execution will take some time because several executions are done for each test (in order to get a minimum time), please be patient!. Visualize[2] the plots generated and reason about how the speed–up changes with the number of threads in the two scenarios.

[2]In the boada Linux distribution you can use the ghostscript `gs` command to visualize Postscript files or convert the files to PDF using the `ps2pdf` command and use `display` to visualize PDF files.

Session 2

Tracing the execution of programs

The objective of this chapter is to present you the environment that will be used to gather information about the execution of a parallel application in OpenMP and visualize it. The environment is mainly composed of Extrae and Paraver. Extrae provides an API (application programming interface) to manually define in the source code points where to emit events. Additionally, Extrae transparently instruments the execution of OpenMP collecting information about the different states in the execution of a parallel program and the values of the hardware counters available in the architecture, reporting information about the processor and memory accesses. After program execution, a trace file (.prv, .pcf and .row files) is generated containing all the information collected at execution time. Then, the Paraver trace browser (wxparaver command) will be used to visualize the trace and analyze the execution of the program.

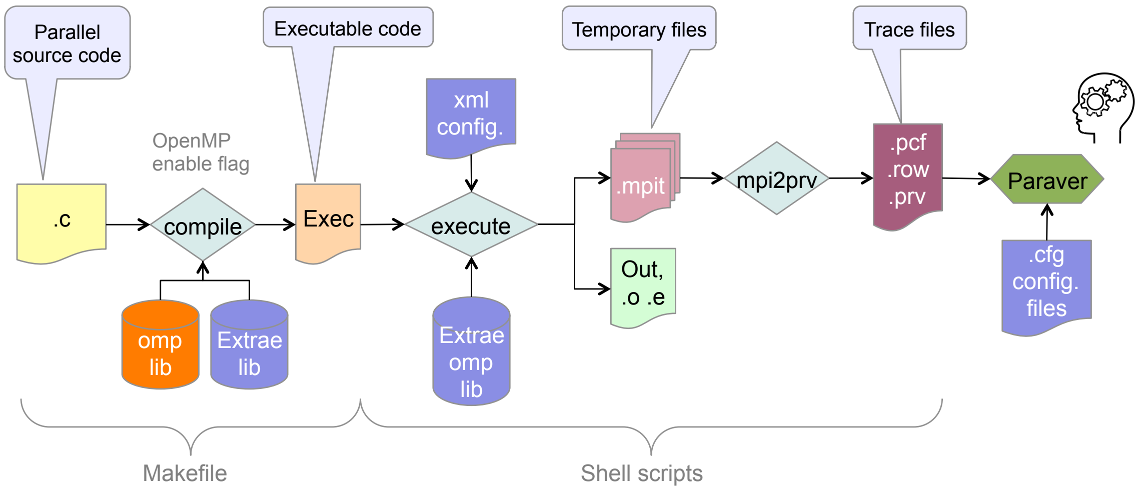


Figure 2.1: Compilation and execution flow for tracing.

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 inside the lab0/environment/pi directory this function is invoked to trace the entry and exit to different parts of the program (serial and parallel 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 source code):

```
// Extrae Constants
#define  PROGRAM    1000
#define  END        0
#define  SERIAL     1
#define  PARALLEL   2
...
Extrae_event (PROGRAM, PARALLEL);
....
Extrae_event (PROGRAM, END);
...
```

2.2 Trace generation

Next you are going to generate the trace that will be later visualized with `Paraver`.

1. Open any of the source codes provided (`pi_seq.c` or `pi_omp.c`) to observe how the previous instrumentation API is used to identify code regions in the code.
2. Open the `Makefile` and identify the targets to compile the instrumented versions of the both codes. 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 those two programs with `Makefile`.
3. Open the `submit-seq-i.sh` and `submit-omp-i.sh` scripts to see how the sequential and parallel binaries are executed and traced. Notice that the name of the binary, number of iterations and number of threads to use are specified inside the script file. The script invokes your binary, which will use the `Extrae` library (by using the `LD_PRELOAD` mechanism) to emit evens at runtime; the script also invokes `mpi2prv` to generate the final trace (`.prv`, `.pcf` and `.row`) and removes all intermediate files.
4. Submit for execution both `submit-seq-i.sh` and `submit-omp-i.sh` scripts. The execution of the scripts will generate a trace file whose file name includes the name of the executable, the size of the problem[1] and the number of threads used for the execution, in addition to the standard output and error output files that you can use to check if the execution and tracing has been correct.

2.3 Paraver hands on

In this laboratory session 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.

Configuration files are available in your home directory inside the `cfgs` directory. Some of them are used to visualize timelines showing different aspects of the parallel execution, as briefly described in the following table.

[1]Use 100.000.000 iterations for both the sequential and parallel execution.

User events	Timeline showing ...
APP_userevents	type 1000 events manually introduced by programmer

OpenMP events	Timeline showing ...
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

Other configuration files compute statistics (profiles) about the parallel execution, as briefly described in the following table.

User events	Profile showing ...
APP_userevents_profile	the duration for type 1000 events

OpenMP events	Profile showing ...
OMP_profile	the time spent in different OpenMP states (useful, scheduling/fork/join, synchronization, ...)
OMP_critical_profile	the total time, percentage of time, number of instances or average duration spent in the three phases of a critical section
OMP_critical_duration_histogram	histogram of the duration of the different phases of the critical section

Later in the course we will use other configuration files to show timelines related with the creation and execution of tasks and their synchronization constructs.

OpenMP events	Timeline showing ...
OMP_tasks/task_instantiation	when tasks are created
OMP_tasks/task_execution	when tasks are executed
OMP_tasks/useful_task_execution	useful time during task execution
OMP_tasks/taskid_instantiation	task identifier at instatiation time
OMP_tasks/taskid_execution	task identifier at execution time
OMP_tasks/taskwait	when taskwait synchronization constructs are executed
OMP_tasks/in_taskgroup	taskgroup regions
OMP_tasks/taskgroup	when taskgroup synchronization is executed

Finally, other configuration files related with *Tareador* will be introduced in forthcoming laboratory sessions.

2.4 Visualizing and analyzing the trace

Using `Paraver` and the appropriate configuration files, answer the following questions:

1. Open the trace generated for `pi_seq.c` and visualize the entry and exit to the serial and potential parallel regions in this code. Compute the *parallel fraction* ϕ from this timeline.
2. Open the trace generated for `pi_omp.c` and try to understand how the parallel execution evolves through the different execution states (fork/join, scheduling, useful work, different sorts of synchronization, ...). Then use the appropriate configuration file to display a table with the % of time spent in the different OpenMP states ONLY during the execution of the parallel region (not considering the sequential part before and after) when using 8 threads.

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.

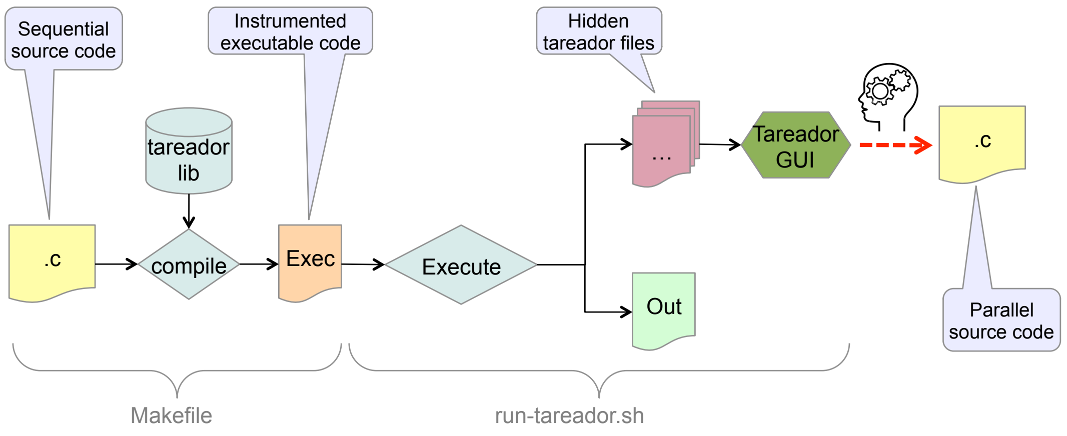


Figure 3.1: Compilation and execution flow for Tareador.

3.1 Using *Tareador*

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

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

The string `NameOfTask` 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 `lab0/environment/dot_product` directory, open the `dot_product.c` source code and identify the calls to the instrumentation functions mentioned above. Open 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 `./run_tareador.sh dot_product`[1]. This will open a new window in which the task dependence graph is visualized (see Figure 3.2, 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 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 an edge (for example the edge going from the red task (`init_B`) to the yellow task (`dot_product`), right click with the mouse and select *Dataview* → *edge.* This will open a window similar to the one shown in Figure 3.2, right/top. In the *Real dependency* tab, you can see the variables that cause that dependence. Also you can right click with the mouse on a task (for example `dot_product`) and select *Dataview* → *Edges-in.* This will open a window similar to the one shown in Figure 3.2, right/middle. 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.

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

5. 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 automatically appear showing the timeline for the simulated execution, similar to the one shown in Figure 3.2, right/bottom (after zooming into the initial part of the trace). 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.

6. You can save the timeline for the simulated parallel execution by clicking *Save* → *Save image* on top of the *Paraver* window.

[1]This script `run_tareador.sh` simply invokes "`tareador_gui.py --llvm --lite`" followed by the name of the executable provided as argument in the invocation.

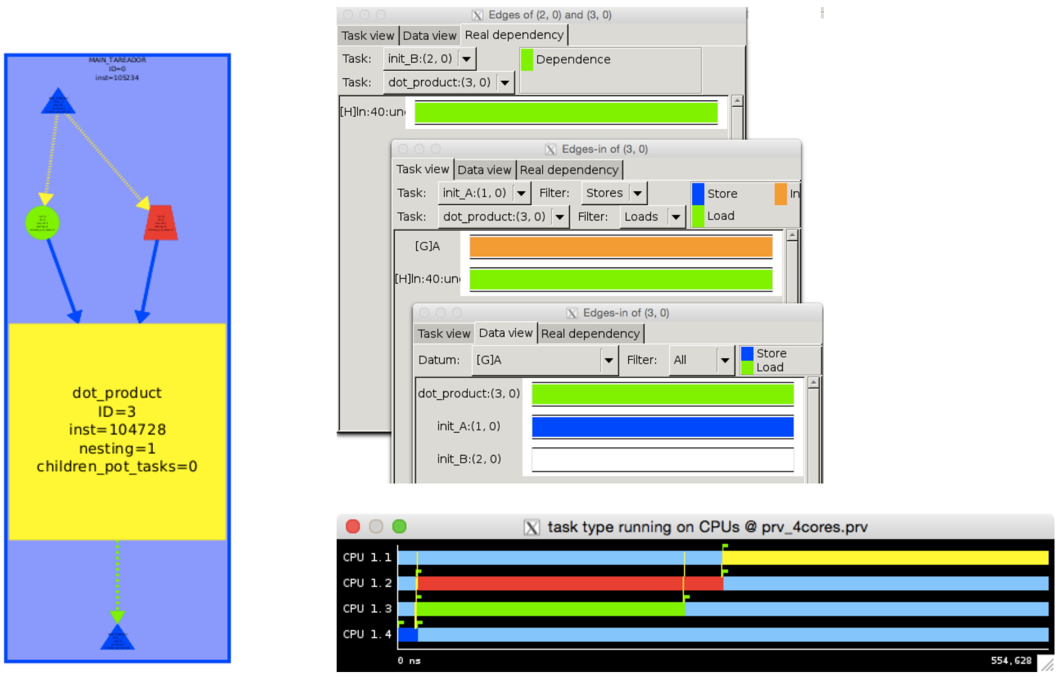


Figure 3.2: Left: Initial task dependence graph for `dot_product`. Right/Top: Visualization of data involved in task dependencies. Right/Bottom: *Paraver* visualization of the simulated execution with 4 processors, after zooming into the initial part of the trace.

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?

3.2 Exploring task decompositions

Go into the `lab0/environment/3dfft` directory, open 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("ffts1_planes_loop_k");
    }
}
```

   (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.

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

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. Draw and briefly describe 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. Describe what do you need to add to your program to measure the elapsed execution time between a pair of points in the program, clearly indicating the library header file that needs to be included, the library functions that need to be invoked, the data structure and its fields.

3. Plot the speed–up obtained when varying the number of threads (strong scalability) and problem size (weak scalability) for `pi_omp.c`. Reason about how the scalability of the program.

Tracing sequential and parallel executions

4. From the instrumented version of `pi_seq.c`, and using the appropriate `Paraver` configuration file, obtain the value of the *parallel fraction* ϕ for this program when executed with 100.000.000 iterations.

5. From the instrumented version of `pi_omp.c`, and using the appropriate `Paraver` configuration file, show a profile of the % of time spent in the different OpenMP states ONLY during the execution of the parallel region (not considering the sequential part before and after) when using 8 threads and for 100.000.000 iterations.

Visualizing the task graph and data dependences

7. Include the source code for function `dot_product` in which you show the *Tareador* instrumentation that has been added to study the potential parallelism in the code. This instrumentation has to appropriately define tasks and filter the analysis of variable(s) that cause the dependence(s).

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

Analysis of task decompositions

9. 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			

10. 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 `lab0/openmp/pi` directory.

4.1 Computing number Pi

Figures 1.1 and 1.2 explained 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 other words, to execute some iterations of the `i` loop). It should be guaranteed that all processors have computed their contribution to `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.

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, filling-in the questionnaire in Part I of the second deliverable. In order to follow them, you will need:

- The set of files inside the `lab0/openmp` directory.
- The set of slides for *Short tutorial on OpenMP* available through the "Raco".

5.1 OpenMP basics

1. Get into the directory called `lab0/openmp/basics`. Open each one of the codes in the directory (the examples are ordered) and answer the questions in the questionnaire associated to it; later you can compile (e.g. `"make 1.parallel"`) and run (e.g. `"./1.parallel"` or `"OMP_NUM_THREADS=4 ./1.parallel"` if you want to externally set the number of threads to be used) to check your answers.
2. Consult "`Part I: OpenMP Basics`" of the tutorial slides, if necessary.

5.2 Loop parallelism

1. Get into the directory `lab0/openmp/worksharing`. Open each one of the codes in the directory (the examples are ordered) and answer the questions in the questionnaire associated to it; later you can compile and run to check your answers.
2. Consult "`Part II: Loop Parallelism in OpenMP`" of the tutorial slides, if necessary.

5.3 Task parallelism

1. Get into the directory `lab0/openmp/tasks`. This code is a bit more complex than the previous ones: the program defines and dynamically create a linked list; afterwards, some computation is done for every node of the list in the function called `processwork`[1]. Open each one of the codes in the directory (the examples are ordered) and answer the questions in the questionnaire associated to it; later you can compile and run to check your answers.
2. Consult "`Part III: Task Parallelism in OpenMP`" of the tutorial slides, if necessary.

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

Session 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 back into directory `lab0/overheads`. Experimental results and conclusions will contribute to Part II of the second deliverable.

6.1 Thread creation and termination

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

1. Open the `pi_omp_overhead.c` file and look at the changes done to the parallel version of pi. This new version iteratively executes the main core of the pi computation using different numbers of threads (2 to `NTHREADS`, using the `OpenMP` intrinsic function `omp_set_num_threads` to change the number of threads). For each number of threads, the execution is done `NUMITERS` times in order to average the execution time of one iteration. For each number of threads, the program measures the execution time and prints something related with it (try to understand what is printed there).
2. Compile using the appropriate target in `Makefile` and execute the binary `pi_omp_overhead` with just one iteration using `submit-omp-overhead.sh`. Do not expect a correct result! 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?

6.2 Thread synchronization: `critical`

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

1. `pi_omp_critical.c` and `pi_omp.c` contain two different `OpenMP` parallelizations of the Pi computation. In the first one a `critical` region is used to protect every access to `sum`, ensuring exclusive access to it. In the second one, a "per–thread" private copy `sumlocal` is used followed by a global update at the end using only one `critical` region. Take a look at both codes and make sure you understand the differences. How many critical sections are executed in each case?
2. Use the `Makefile` to compile both programs and execute them with just 1 thread. Compare with the sequential execution time of `pi_seq.c` for 100.000.000. Can you explain the differences, if any? Do an estimation of the order of magnitude for the overhead of executing a `critical` region.
3. Next execute with 8 threads. Do both programs benefit from the use of several processors in the same way? Can you guess the reason for this behavior?

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

6.3 Thread synchronization: `locks`

Third you will measure the overhead related with the use of another mechanisms for synchronization in `OpenMP`: `lock` and the associated `omp_set_lock` and `omp_unset_lock`.

1. `pi_omp_lock.c` is an alternative implementation for the previous `pi_omp_critical.c` in which we make use of `locks` to protect every access to `sum`, ensuring exclusive access to it. Take a look at this new version and make sure you understand how `locks` are being used.
2. Use the `Makefile` to compile this new program and execute them with just 1 thread. Compare with the execution times obtained in the previous section when using `critical`. Do an estimation of the order of magnitude for the "minimum" overhead of executing a pair `set_lock/unset_lock`.
3. Next execute with 8 threads. Compare with the execution times obtained in the previous section when using `critical`. Can you guess why the behavior now is even worse than with `critical`?

6.4 Atomic memory access

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

1. `pi_omp_atomic.c` contains a different `OpenMP` parallelization that makes use of `atomic` to guarantee atomic (indivisible) access to memory. Take a look at the code and compare with `pi_omp_critical.c`.
2. Use the `Makefile` to compile this new version and execute it with 1 and 8 threads. Can you estimate the order of magnitude for the overhead of executing `atomic`? Does it change with the number of threads? Why?

6.5 False data sharing

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

1. `pi_omp_sumvector.c` provides a new version for the program computing Pi that avoids the use of `critical` regions. Open the file and explain the purpose of vector `sumvector`. What does each element of this vector store? Use the `Makefile` to compile it and execute by submitting the appropriate script. Compare the execution times, for 1 and 8 threads, of both versions `pi_omp_sumvector.c` and `pi_omp.c`. Is the use of vector `sumvector` the cause of that difference?
2. `pi_omp_sumvector.c` suffers of what is called "false sharing". False sharing occurs when multiple threads modify different memory addresses that are stored in the same cache line. When multiple threads update these independent memory locations, the cache coherence protocol forces other threads to update/invalidate their caches. To avoid false sharing we need to make sure that threads do not share cache lines. To that end, `pi_omp_padding.c` provides a new version in which padding is used (i.e. elements accessed by each thread reside in different cache lines). Look at the source code and explain the way padding is done in this code. Compile the program using the `Makefile`, and execute. Is the execution time similar, better or worse than `pi_omp.c`? Which is the additional average access latency that you observe to memory when your program suffers from false sharing?

Second deliverable

Deliver a report in `PDF` format (other formats will not be accepted) containing your answers to the OpenMP Questionnaire and the results and conclusions from the evaluation of overheads for the main OpenMP constructs. 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.

Part I: OpenMP questionnaire

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

A) Basics

1.hello.c

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

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

1. Is the execution of the program correct? Which data sharing clause should be added to make it correct?.
2. Are the lines always printed in the same order? Could the messages appear intermixed?

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

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

4.data_sharing.c

1. Which is the value of variable `x` after the execution of each parallel region with different data-sharing attribute (`shared`, `private` and `firstprivate`)?
2. What needs to be changed/added/removed in the first directive to ensure that the value after the first parallel is always 8?.

5.parallel.c

1. How many messages the program prints? Which iterations is each thread executing?
2. What needs to be changed in the directive to ensure that each thread executes the appropriate iterations?.

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

1. Is the program always executing correctly?
2. Add two alternative directives to make it correct. Which are these directives?

7.barrier.c

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

B) Worksharing

1.for.c

1. How many iterations from the first loop are executed by each thread?
2. How many iterations from the second loop are executed by each thread?
3. Which directive should be added so that the first `printf` is executed only once by the first thread that finds it?.

2.schedule.c

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

3.nowait.c

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

4.collapse.c

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

C) Tasks

1.serial.c

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

2.parallel.c

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

Part II: Parallelization overheads

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

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

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

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

5. Complete the following table with the execution times of the different versions for the computation of Pi that we provide to you in this first laboratory assignment when executed with 100.000.000 iterations. 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**
pi_seq.c		–	1
pi_omp.c (sumlocal)			
pi_omp_critical.c			
pi_omp_lock.c			
pi_omp_atomic.c			
pi_omp_sumvector.c			
pi_omp_padding.c			