PAR Laboratory Assignment
Lab 1: Experimental setup and tools

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

Spring 2017-18

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Contents

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

Session 1

Experimental setup

The objective of this first session is to familiarise yourself with the hardware and software environment that you will be using during the semester to do all laboratory assignments in PAR. From your PC/terminal booted with Linux[1] you will access `boada`, a multiprocessor server located at the Computer Architecture Department. To connect to it you will have to establish a connection using the secure shell command: `"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[2]. You should change the password for your account using `"ssh -t parXXYY@boada.ac.upc.edu passwd"`.

Once you are logged in you will find yourself in `boada-1`, the login node for the whole machine from where you can submit execution jobs to the rest of the nodes in the machine. In fact `boada` is composed of 8 nodes (named `boada-1` to `boada-8`), equipped with three different processor generations, as shown in the following table:

Node name	**Processor generation**	**Interactive**	**Queue name**
boada-1	Intel Xeon E5645	Yes	batch
boada-2 to 4	Intel Xeon E5645	No	execution
boada-5	Intel Xeon E5-2620 v2 + Nvidia K40c	No	cuda
boada-6 to 8	Intel Xeon E5-2609 v4	No	execution2

However, in this course we are going to mainly use nodes `boada-1` to `boada-4`, either interactively or through the `execution` queue, as explained later in this chapter.

Each node in `boada` has its own local disk; you can access it through `/scratch/1/parXXYY`. In addition, all nodes have access to a shared NAS (*Network-attached Storage*) disk; you can access it through `/scratch/nas/1/parXXYY` and this is your *home directory* (check by typing `pwd` in the command line). We will post all necessary files to do each laboratory assignment in `/scratch/nas/1/par0/sessions`. For the session today, copy `lab1.tar.gz` **from** that location **to** your home directory in `boada` and uncompress it at the root of your home directory with this command line: `"tar -zxvf lab1.tar.gz"`.

In order to set up all environment variables you have to process the `environment.bash` file now available in your home directory with `"source environment.bash"`. **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 the `.profile` file in your home directory, a file that is executed every time a new session is initiated.

1.1 Node architecture and memory

The first thing you will do is to execute the `lscpu` and `lstopo` commands in order to obtain information about the hardware in `boada-1` (which is identical to the other nodes `boada-2` to `boada-4`):

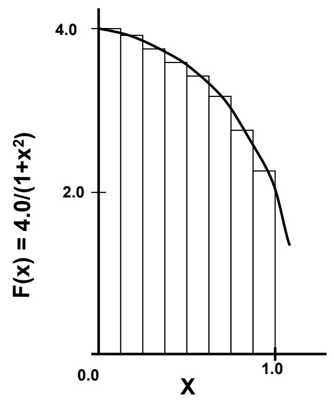
[1]You can also access from your laptop, booted with Linux, Windows or MacOS X, if a secure shell client is installed.
[2]Option `-Y` if you are connecting from a MacOS X laptop with XQuartz.

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 the 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 visualise the output file generated (`map.fig`) and export to a different format (PDF or JPG, for example) using `File` → `Export`[3].

1.2 Serial compilation and execution

Next you will get familiar with the compilation and execution steps for both sequential and parallel applications. You are going to use a very simple code, `pi_seq.c`, which you can find inside the `lab1/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

[3]In the boada Linux distribution you can use `display` to visualise 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 shows the compilation and execution flow for a sequential program. We will always compile programs to generate binary executable files through a `Makefile`, with multiple targets that specify the rules to compile each program version. In this course we are going to use `gcc` (the C front–end from the *GNU Compiler Collection*); you can type "`gcc -v`" to know about which specific version of the compiler you are using.

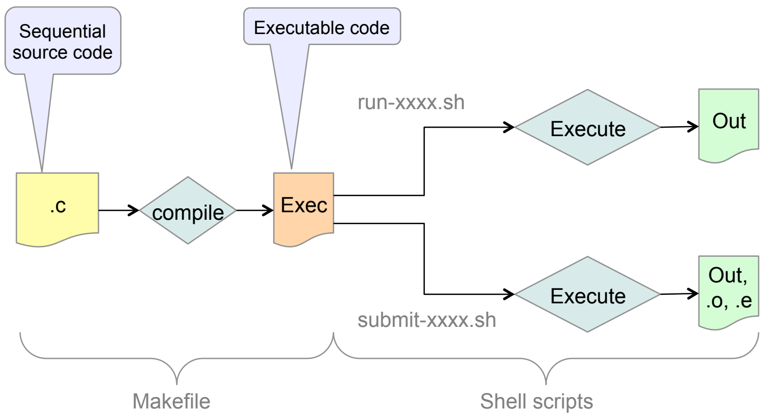


Figure 1.3: Compilation and execution flow for sequential program.

There are two ways to execute your programs: 1) via a queueing system (in one of the nodes `boada-2` to `boada-4`; or 2) interactively (in `boada-1` itself). 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.

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.
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 `time-pi_seq-boada{2-4}` 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 the next laboratory assignment, in this section we will see how to compile and execute parallel programs in `OpenMP`. Figure 1.4 shows the compilation and execution flow for an OpenMP program. The main difference with the flow shown in Figure 1.3 is that now the `Makefile` will include the appropriate compilation flag to enable `OpenMP`.

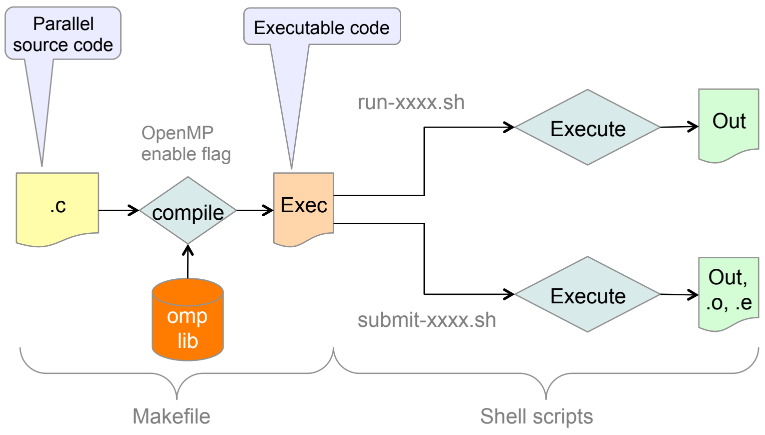


Figure 1.4: Compilation and execution flow for OpenMP.

1.3.1 Compiling OpenMP programs

1. In the same `lab1/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 section 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 for which your program is executed.

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!. Visualise[4] the plots generated and reason about how the speed–up changes with the number of threads in the two scenarios.

1.4 Do it yourself: putting it all together

To finish this session you have to measure the sequential execution time of `pi_seq.c` and the scalability of the `pi_omp.c` code on the other node types available in `boada`. In order to know the number of cores to use when submitting your jobs to each node type and to understand the differences you may observe, we suggest that you first investigate the architecture of the different types of nodes. Do the necessary changes in the script files already provided to evaluate the scalability in the total number of cores available in each node type, and create new ones in order to obtain the architectural information for each node type that is requested in the deliverable.

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

Session 2

Analysis of task decompositions using Tareador

In this chapter we introduce *Tareador*, an environment to analyse the potential parallelism that can be obtained when a certain parallelisation 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 2.1 shows the compilation and execution flow for *Tareador*, starting from the taskified source code.

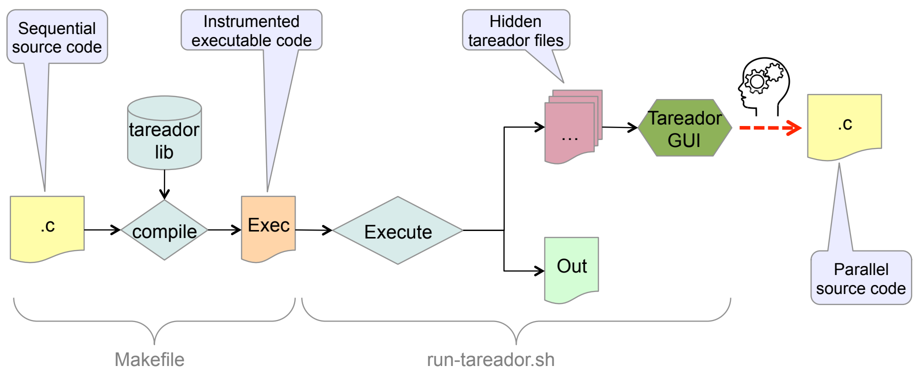


Figure 2.1: Compilation and execution flow for Tareador.

2.1 *Tareador* API

Tareador offers an API (*Application Programmer Interface*) 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.

2.2 Tareador hands–on

For the first part of this session a very simple program, `dot_product.c` available in the `lab1/dot_product` directory, that performs the computation of the dot product of two vectors of a given size. The program first initialises both vectors (`A` and `B`) and then calls function `dot_product` in order to perform the actual computation and compute `result`:

1. Go into the `lab1/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 visualised (see Figure 2.2, left). Each node of the graph represents a task: different shapes and colours 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 colours/patterns are used to represent different kind of dependences (blue: data dependences, green/yellow: control dependences).

3. *Tareador* allows you to analyse 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 2.2, 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 2.2, 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, you can simulate the execution of the initial task decomposition, for example with 4 processors, by clicking *View Simulation* in the main *Tareador* window. This will open a *Paraver* window showing the timeline for the simulated execution, similar to the one shown in Figure 2.2, right/bottom. In fact you will have to zoom into the initial part of the trace in order to visualise the same part of the trace; you can do this by clicking the left button in your mouse and selecting the zone you want to zoom. Colours are used to represent the different tasks (same colours that are used in the task graph). In the next laboratory session you will learn more details about the usage of *Paraver*; for example, you can activate the visualisation 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 timeline *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.

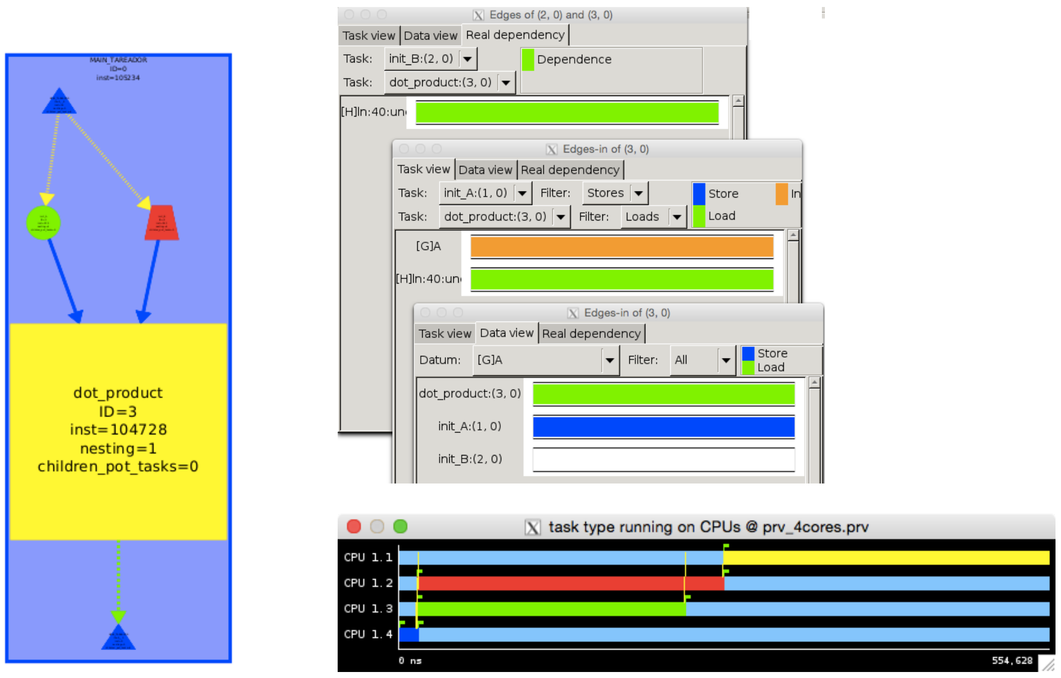


Figure 2.2: Left: Initial task dependence graph for `dot_product`. Right/Top: Visualisation of data involved in task dependencies. Right/Bottom: *Paraver* visualisation of the simulated execution with 4 processors, after zooming into the initial part of the trace.

Next let's 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 visualise 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 sequentialising 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)
   ```

   With this mechanism you can remove certain dependences from the analysis, assuming that you will use the appropriate mechanisms in the parallel code to actually remove them. 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?

2.3 Exploring task decompositions

Once you are familiar with the basic features in *Tareador*, go into the `lab1/3dfft` 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.sh` script. Analyse the task dependence graph and the dependences that are visualised, 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 processors. 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. Draw the execution time and speedup, with respect to the sequential execution, plots for version `v4`, as requested in the first deliverable.

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

Session 3

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 visualise it. The complete compilation and execution flow for tracing is shown in Figure 3.1. 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. However in this course we will only use `Extrae` to transparently instrument the execution of `OpenMP` binaries, by collecting information about the status of each thread and different events related with the execution of the parallel program[1]. The `Extrae` library is appropriately set in the scripts that launch instrumented executions. 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 visualise the trace and analyse the execution of the program.

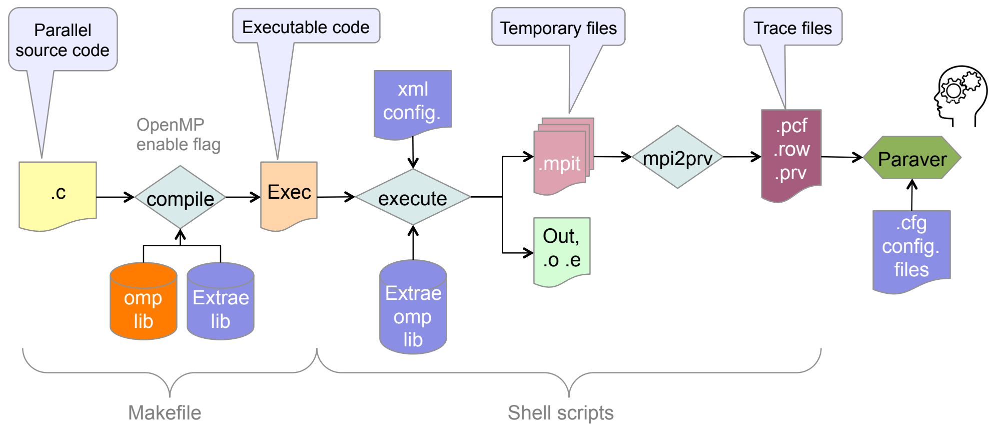


Figure 3.1: Compilation and execution flow for tracing.

3.1 Paraver hands–on

In this guided tour you will learn how the generate traces and the basic features of `Paraver`, a graphical browser of the traces generated with `Extrae`, together with the set of configuration files to be used to visualise and analyse the execution of your program. Go to the `lab1/pi` directory.

[1]`Extrae` also collects the values of hardware counters available in the architecture that report information about the processor activity and memory accesses

3.1.1 Trace generation

The first thing that needs to be done is the generation of the tracefile:

1. Open the `submit-omp-i.sh` script to see how the parallel binary is 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.

2. Submit for execution the `submit-omp-i.sh` script. The execution of the script will generate a trace file whose file name includes the name of the executable, the size of the problem[2], the number of threads used for the execution and the name of the node where the trace has been obtained, in addition to the standard output and error output files that you can use to check if the execution and tracing has been correct.

3.1.2 *Paraver* guided tour

Timelines: navigation and basic concepts

1. Launch *Paraver* by typing `wxparaver` in the command line (it should be in the path if you have already sourced the environment.bash file). This will open the so called *Main Window.*

2. Load trace: From the main menu, select *"File → Load Trace"*, and select the trace generated from the instrumented execution of the parallel `pi_omp.c` code. Alternatively, traces can be located through the browser at the bottom of the *Main Window*: double clicking on a `.prv` file will load it. For the purposes of this guided tour, traces mainly contain two types of records: *states* and *flags.* These two kind of records are used by *Extrae* to inject information in the trace.

3. Once the file is loaded, click on the *New single timeline window* box (top left icon). A new window appears showing a timeline with the activity (state, encoded in color) of each thread (vertical axis) in the parallel program. The horizontal axis represents time, from left to right.

   - Colors: While moving the mouse over the window, a textual description of the meaning of each color is shown (at the bottom of the same window): light blue (*idle*), dark blue (*running*), red (*synchronisation*), yellow (*scheduling and fork-join*), ... You can also see a table indicating the meaning of each color by clicking the right button in your mouse (anywhere inside the window) *"Right Button → Info Panel"*, then select the *Colors* tab. It is important to be aware that the meaning of each color is specific to each window. Through the tutorial you will see different timeline windows each of them displaying a different information with its own coloring table.
   - Textual information: Double click with the left button in your mouse on any point in the window. It will list in textual form the actual value at the point selected and how long the time interval with that color is. The text display will be in the *What/Where* tab of the Info Panel. *"Right Button → Info Panel"*, can be used to hide the lower info panel.
   - Zoom: Click with the left button of the mouse to select the starting time of the zoomed view, drag the mouse over the area of interest, and release the mouse to select the end time of the zoomed view.
   - *Undo Zoom* and *Redo Zoom* commands are available on the right button menu. You can do and undo several levels of zooming.
   - The Control-Zoom option will let you select a subset of threads. This is useful when analysing runs with many processes and you want to concentrate on a few of them. Hold down the *Control* or *CTRL* key on the keyboard, and using the mouse, identify a rectangular area by clicking on top left corner of the desired area with the left mouse button and dragging and releasing the button on the bottom right corner.

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

- To measure time between any two points in the trace: Use the Shift-Zoom combination to activate the timing. The time and the interval between the two selected points of the trace is displayed in the *Timing* tab of the *Info Panel*.
- *Flags*: Right-click with your mouse on the window, and select the *"View → Event Flags"* checkbox. In this trace flags appear to signal the entry and exit points of different OpenMP activities. Flags are useful to differentiate different bursts in what may look like a simple burst. Selecting the visualisation of a subset of flags (type and value) and giving them a specific semantic interpretation is possible through the *Main Window* (selecting the second icon in the *Files & Windows* properties panel); however this is not covered in this guided tour.

4. Configuration files are the simplest way to do the analysis of a trace. Next we will use some of them available in different sub-directories inside the `cfgs` directory in your home directory.

   - For example, configuration file `OMP_parallel_functions.cfg` (in `cfgs/OpenMP`) can be used to identify different parallel regions in the trace. To load it, from the main menu, select *"File → Load Configuration"*. Colors are used here to visualise different parallel regions (in our example only one). The textual information shows the name given by the compiler to the outlined routine that is called by each thread to execute the body of the parallel OpenMP construct. Different colors would correspond to different parallel regions in the program.
   - Load `OMP_parallel_functions_duration.cfg` in order to visualise the duration of the parallel bursts identified with the previous configuration file. A gradient coloring scheme is used in this window: between light green representing a small duration and dark blue representing a long one. Moving the mouse over the window will show at the bottom of the timeline the range of durations represented by the color the mouse goes over. You can Right-click with your mouse on the window, and select the *"Fit Semantic Scale → Fit Both"* to automatically adjust the gradient to the values in the window. Double click with the Left Button on any point in the window: this will list in textual form the actual duration at the point selected (and vicinity if time scale too coarse). The text display will be in the *What/Where* tab of the *Info Panel*.
   - Open the `OMP_in_barrier.cfg` and `OMP_in_lock.cfg` configuration files to visualise the synchronisation activity (in barriers and critical regions, respectively) in this parallel execution. Observe the different states when accessing locks.

5. Aligning and synchronizing windows: In *Paraver* every timeline window represents a single metric or view for all selected threads and time span. It is possible to align two timelines by making them display the exact same threads and time span. For doing so just right-click and select *Copy*, on the source (reference) window and then on the target window, right-click and select *"Paste → Default"* (or separately *"Paste → Size"* and *"Paste → Time"*). Both windows will then be of the same size and represent different views (metrics) for the same part of the trace. If you put one above the other there is a one to one correspondence between points in vertical. You can also synchronise several windows, by selecting *Synchronise* after a Right-click with your mouse on the timeline window; repeat the process for all the windows you want to synchronise. Once synchronised they will continue aligned after zooming, undoing or redoing zoom.

The upper part in Table 3.1 lists the configuration files that are available in your home directory inside the `cfgs` directory for doing this kind of analysis.

Profiles

The above analysis went directly to the detailed timeline, but a less detailed averaged statistic analysis can often be sufficient to identify problems and gives a summarised view of the behaviour of an application. *Paraver* provides one mechanism, named *2DAnalyzer*, to obtain such profiles for the desired region of a trace.

1. Load configuration file `OMP_state_profile.cfg`. A table pops up with one row per thread and one column per OpenMP state (*Running*, *Synchronization*, *Scheduling and Fork/Join*, ...). Each cell value shows the absolute time spent by a thread in a specific state. To see a different statistic change the *Statistic* selector in the *Main Window*. Interesting options at this time may be:

OpenMP events	Timeline showing ...
OMP_parallel_functions	the parallel function each thread is executing
OMP_parallel_functions_duration	the duration for the parallel functions
OMP_worksharings	when threads are executing worksharing regions (for or single)
OMP_worksharings_duration	the duration of worksharing regions
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_in_ordered	when threads are executing ordered sections inside loops

OpenMP events	Profile showing ...
OMP_state_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 different phases of a critical section
OMP_critical_duration_histogram	histogram of the duration of the different phases of the critical section
OMP_ordered_profile	the total time, percentage of time, number of instances or average duration spent in the different phases of an ordered section

Table 3.1: Upper part: configuration files to support analysis at the timeline level. Lower part: configuration files for statistical summaries.

- *% of Time*: to show the percentage of the total time spent on each state, per thread.
- *# Instances*: to count the number of times each state occurs.
- *Average Duration*: to compute the average duration of each state.

2. All the above statistics are computed based on a single timeline window, which we call the *Control Window* and which can be popped up by clicking on the control window icon in the top left corner of the window. In this example, you will see that it is the initial timeline we opened at the beginning. The values of the control window determine to which column is a given statistic accumulated/accounted.

3. To apply the analysis to a subset of the trace, zoom on any of the timelines to the time region you are interested on. Right-click and select *Copy* on this window and right-click and select *Paste* → *Time* on the table. The analysis will be repeated just for the selected time interval.

4. Load configuration file `OMP_critical_profile.cfg` to see the same statistics (total time, percent of time, number of instances or average duration) for the three phases of the critical section implementation.

The lower part in Table 3.1 briefly describes the configuration files that are available to compute statistics (profiles) about the parallel execution. Later in the course we will use other configuration files to show timelines related with the creation and execution of tasks and their synchronisation constructs. Finally, other configuration files related with *Tareador* that you already used in the previous session.

3.2 Navigate yourself: back to the `3DFFT` example

Let's put some of these ideas into practise. To do so, go to the `lab1/3dfft` directory.

1. Compile the `3dfft_omp.c` program using the appropriate entry in the Makefile and submit its execution using the `submit-omp-i.sh` script. Make sure that the trace is generated before opening it with *Paraver*.

2. Using `wxparaver` with the appropriate configuration files, try to answer the following questions:

- How many parallel regions have been defined by the programmer in `3dff_omp.c`?
- And how many work–sharing constructs inside each parallel region?
- Is there a serial part dominating the execution of the parallel program? How much does this part represents of the total execution? Which would be the *parallel fraction* ϕ for the program? And the ideal S_∞?

3. Next you will uncomment the pragmas that appear in lines 29 and 109 in the source code in order to increase the ϕ. To do that edit the `3dfft_omp.c` file, uncomment the lines and recompile. Repeat the same questions above and recompute the new value for ϕ.

4. Finally, obtain a profile of the % of time spent in the different OpenMP states when using 8 threads. Try to reason what is shown in the profile. Why there is time spent in synchronization? Where these synchronizations are happening? Why there is time spent in scheduling and fork/join? Where these activities are happening? Is there any load unbalance among threads? How do you observe it?

Deliverable

After the last session for this laboratory assignment, and before starting the next one, you will have to deliver a **report** in PDF format (other formats will not be accepted) containing the answers to the following questions. Your professor will open the assignment at the Raco website and set the appropriate delivery dates for the delivery. Only one file has to be submitted per group through the Raco website.

Important: In the front cover of the document, please clearly state the name of all components of the group, the identifier of the group (username parXXYY), title of the assignment, date, academic course/semester, ... and any other information you consider necessary.

As part of the document, you can include any code fragment, figure or plot you need to support your explanations. In case you need to transfer files from boada to your local machine (laptop or desktop in laboratory room), or viceversa, you can use the secure copy `scp` command. For example `"scp parXXYY@boada.ac.upc.edu:lab1/foobar.txt local/directory/."` executed in your local machine to copy file `foo.txt` inside directory `lab1` in your home directory of boada to directory `local/directory/` with the same name.

Node architecture and memory

1. Complete the following table with the relevant architectural characteristics of the different node types available in `boada`:

	`boada-1` to `boada-4`	`boada-5`	`boada-6` to `boada-8`
Number of sockets per node			
Number of cores per socket			
Number of threads per core			
Maximum core frequency			
L1-I cache size (per-core)			
L1-D cache size (per-core)			
L2 cache size (per-core)			
Last-level cache size (per-socket)			
Main memory size (per socket)			
Main memory size (per node)			

2. Include in the document the architectural diagram for one of the nodes `boada-1` to `boada-4` as obtained when using the `lstopo` command.

Timing sequential and parallel executions

3. Plot the execution time and speed–up that is obtained when varying the number of threads (strong scalability) and problem size (weak scalability) for `pi_omp.c` on the different node types available in `boada`. Reason about the results that are obtained.

Analysis of task decompositions with *Tareador*

4. Include the relevant(s) part(s) of the code to show the new task definition(s) in v4 of `3dfft_seq.c`. Capture the final task dependence graph that has been obtained after version v4.

5. Complete the following table for the initial and different versions generated for `3dfft_seq.c`, briefly commenting the evolution of the metrics with the different versions.

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

6. 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). Briefly comment the scalability behaviour shown on these two plots.

Tracing the execution of parallel programs

7. From the analysis with `Paraver` that you have done for the complete parallelization of `3dfft_omp.c`, explain how have you computed the value for ϕ, the *parallel fraction* of the application. Please, include any `Paraver` timeline that may help to understand how you have performed the computation of ϕ.

8. Show and comment the profile of the % of time spent in the different OpenMP states for the complete parallelization of `3dfft_omp.c` on 4 threads.