PAR Laboratory Assignment
Lab 1: Experimental setup and tools

E. Ayguadé, J. R. Herrero, P. Martínez-Ferrer,
J. Morillo, J. Tubella and G. Utrera

Spring 2021-22

Index

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

Session 1

Experimental setup

The objective of this laboratory session is to familiarise yourself with the hardware and software environment that you will use during this semester to do all laboratory assignments in PAR. From your local 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 commands necessary to open remote windows in your local desktop[2]. Once you have the account credentials, the first thing you should do is to change the password for your account using `"ssh -t parXXYY@boada.ac.upc.edu passwd"`[3].

Once you are logged in you will find yourself in `boada-1`, the login node for the whole machine where you can execute interactive jobs and from where you can submit execution jobs to the rest of the nodes in the machine. In fact, `boada` is composed of 9 nodes (named `boada-1` to `boada-9`), equipped with four different processor generations, as shown in the following table:

Node name	Processor generation	Interactive	Partition
`boada-1`	Intel Xeon E5645	Yes	interactive
`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
`boada-9`	Intel Xeon E5-1620 v4 + Nvidia K40c	No	cuda9

However, in this course you are going to use only nodes `boada-1` to `boada-4`, either interactively or through the `execution` queue, as explained in the next subsection. The rest of the nodes have restricted access and PAR users are not allowed to send jobs to their corresponding queues.

All nodes have access to a shared NAS (*Network-Attached Storage*) disk; you can access it through `/scratch/nas/1/parXXYY` (in fact this is your *home directory*, check by typing `pwd` in the command line). In addition, each node in `boada` has its own local disk which can be used to store temporary files non visible to other nodes; you can access it through `/scratch/1/parXXYY`.

All necessary files to do each laboratory assignment will be posted in `/scratch/nas/1/par0/sessions`. For the session today, copy `lab1modelfactors.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 lab1modelfactors.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, it

[1] You can also access from your laptop, booted with Linux, MacOS X or Windows, if a secure shell client is installed. For MacOS X you will need to have installed *XQuartz*. For Windows you will need to have installed both *putty* for secure shell (https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html) and *xming* for X11 (https://wiki.centos.org/HowTos/Xming); alternatively you can also use *MobaXterm* which integrates both secure shell and X11 (https://mobaxterm.mobatek.net/download.html).

[2] Use option `-Y` if you are connecting from a MacOS X laptop with *XQuartz*.

[3] The `passwd` command will be executed in `boada`. After entering the old password correctly twice you will be asked for your new password also twice.

is strongly recommended that you add this command line in the `.bashrc` file in your home directory[4], a file that is executed every time a new session is initiated.

In case you need to transfer files from `boada` to your local machine (laptop or desktop in laboratory room), or viceversa, you have to use the secure copy `scp` command. For example if you type the following command `"scp parXXYY@boada.ac.upc.edu:lab1/pi/pi_seq.c ."` in your local machine you will be copying the source file `pi_seq.c` located in directory `lab1/pi` of your home directory in `boada` to the current directory, represented with the `"."`, in the local machine, with the same name.

1.1 Node architecture and memory

The first thing you will do is to investigate the architecture of the available nodes in `boada`. To do this execute interactively 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. the number of sockets, cores per socket and threads per core in a specific node;
2. the amount of main memory in a specific node, and each NUMAnode;
3. the cache memory hierarchy (L1, L2 and L3), private or shared to each core/socket.

Fill in the table in the "Deliverable" section with the main characteristics of the node. Use the `"--of fig map.fig"` option for `lstopo` in order to get the drawing of the architecture of a node. 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` in order to include it in your deliverable for this laboratory assignment[5].

1.2 Execution modes: interactive vs queued

There are two ways to execute your programs in boada:

1. via a queueing system (in one of the nodes `boada-2` to `boada-4`);
2. interactively (in `boada-1` itself).

It is mandatory to use option 1 when you want to execute scripts that require several processors in a node, ensuring that your job is executed in isolation (and therefore reporting reliable performance results) and to avoid adding additional load to the interactive node accessed by all users; 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, scripts for both options (`submit-xxxx.sh` and `run-xxxx.sh`, respectively) will be provided:

- Queueing a job for execution: `"sbatch [-p partition] ./submit-xxxx.sh"` . Additional parameters may be specified, if needed by the script, after the script name. If you do not specify the name of the partition with `"-p partition"` your script will run on the `execution` partition by default. Use `"squeue"` to ask the system about the status of your job submission. You can use `"scancel"` followed by the job identifier to remove a job from the queueing system. Note that partition names associated to each node name are shown in the last column of the table above. After the execution in an available node associated to the specified partition, in addition to the files being generated by the script, two additional files will be created. Their name will have the script name followed by an `".e"` and an `".o"` and the job identifier. They will contain the messages sent to the standard error and standard output respectively during the execution of the job. You should check them to be sure results make sense.
- 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.

[4] Initially this file does not exist and you can create it with any editor.

[5] In the `boada` Linux distribution you can use `xpdf` to open pdf files and `display` to visualise graphics files. 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"`.

1.3 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 number Pi by computing an approximation of 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 can be approximated by dividing the area into small rectangles and adding up their area.

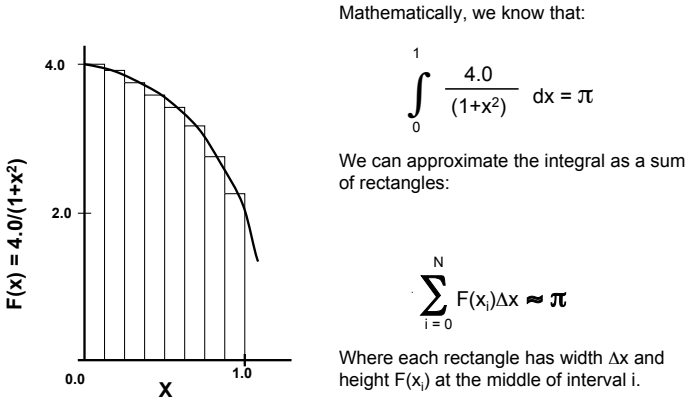

Figure 1.1: Pi computation

Figure 1.2 shows a simplified version of the code you have in `pi_seq.c`. The 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 computation.

Figure 1.3 shows the compilation and execution flow for a sequential program. You will always compile programs to generate binary executable files through a `Makefile`, with multiple targets that specify the rules to compile each program version; the appropriate `Makefile` will be provided in each assignment. In this course `icc` (the C front–end from the *Intel Compilers* collection) will be used to generate your binary files; you can type `"icc -v"` to know about which specific version of the compiler you are using.

In the following steps you will compile `pi_seq.c` using the `Makefile` and execute the binary generated interactively and through the queueing system using the `execution` queue, 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` followed by the `target` identified in order to generate the binary executable file.

2. Interactively execute the binary file generated to compute the number Pi using the `run-seq.sh` script with the appropriate arguments (executable name and number of iterations 1.000.000.000).

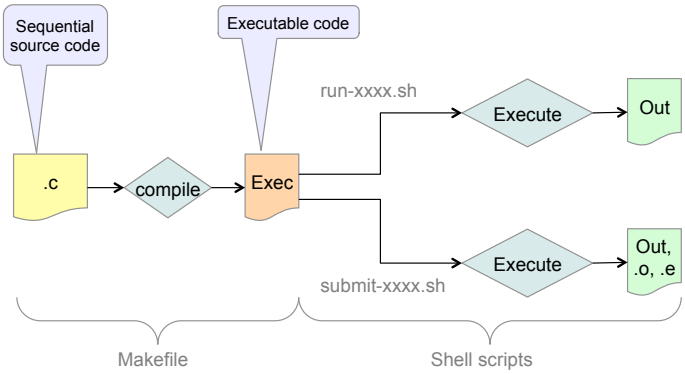

Figure 1.3: Compilation and execution flow for sequential programs.

The execution 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. Please also take a look at the `run-seq.sh` script to understand how the binary file is executed.

3. Submit the execution of the `submit-seq.sh` script to the `execution` partition using the `sbatch` command with the appropriate arguments (executable name and number of iterations 1.000.000.000). Use `squeue` to see that your script is queued. Look at the files generated and their content: the standard output and error of the script and the `time-pi_seq-boada-Y` file, being `Y` the node where the execution happened. Please also take a look at the `submit-seq.sh` script.

1.4 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 after this first 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*.

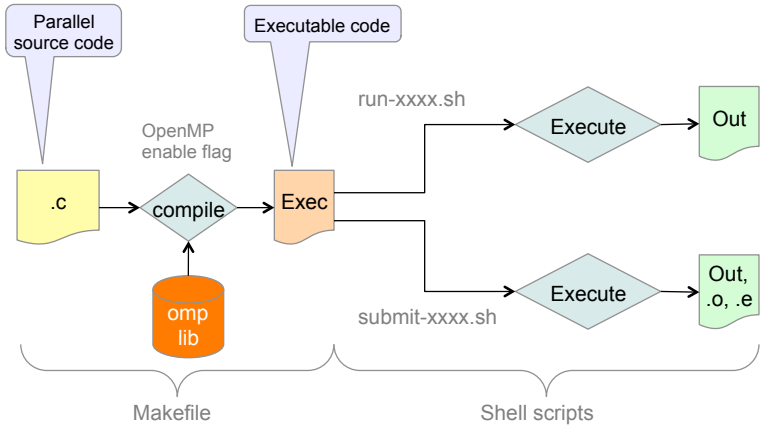

Figure 1.4: Compilation and execution flow for OpenMP.

1. Open the parallel version for the Pi computation that we provide in file `pi_omp.c`. Take a look at code and figure out what the new lines of code are doing.

2. Now open the `Makefile` file and identify the `target` you have to use to compile the *OpenMP* code. Observe that the only difference is the use of the `-fopenmp` compilation flag. Execute the command line `make` followed by the `target` identified in order to generate the binary executable file.

3. Interactively execute the *OpenMP* code with 1, 2, 4 and 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 the number of threads to use in *OpenMP* is specified.

4. Submit the execution of the `submit-omp.sh` script to the execution partition using the `sbatch` command, specifying the *OpenMP* code, the same number of iterations (1.000.000.000) and the number of threads (do it with 1, 2, 4 and 8 threads). Look at the `time-pi_omp-X-boada-Y` files, being `X` the number of threads used and `Y` the node where the execution happened.

5. Draw a table in your deliverable showing the user and system CPU time, the elapsed time, and the % of CPU used in the two scenarios (interactive and queued) and with the number of threads enumerated before. Do you observe a major difference between the interactive and queued execution? Include the table in the Deliverable commenting the results observed.

1.5 Strong vs. weak scalability

Finally in this section you are going to explore the *scalability* of the parallel version in `pi_omp.c` when varying the number of threads used to execute the parallel code. The scalability will be measured calculating the ratio between the sequential and the parallel execution times (this ratio is called *speed–up*). Two different scenarios are 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 the 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 the program is executed.

Two scripts are provided to analyse scalability, `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 parallel execution time and speed–up.

1. Submit the execution of the `submit-strong-omp.sh` script (no arguments are required to execute the script). The execution will take some time because several executions are done for each test (in order to get a minimum time), please be patient!. Use the ghostscript `gs` command to visualise the Postscript file generated[6]. Observe how the execution time and speed–up varies with the number of threads in the *strong* scaling scenario.

2. Change the value for `np_NMAX` in the `submit-strong-omp.sh` from 12 to 24 and execute again. Can you explain the behaviour observed in the scalability plot?

3. Submit the execution of the `submit-weak-omp.sh` script (no arguments are required to execute the script). Observe now how the parallel efficiency varies with the number of threads in the *weak* scaling scenario.

Include all the scalability plots generated in the Deliverable reasoning about the differences observed between the two scenarios.

[6] You can also convert the Postscript file to PDF using the `ps2pdf` command and use `xpdf` to visualise the resulting PDF file.

Session 2

Systematically analysing task decompositions with *Tareador*

This chapter introduces *Tareador*, an environment useful for analysing the potential parallelism that can be obtained when a certain *task decomposition* strategy is applied to your sequential code. With *Tareador* the programmer simply needs to identify which are the tasks in the task decomposition strategy that wants to be evaluated. Then *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.

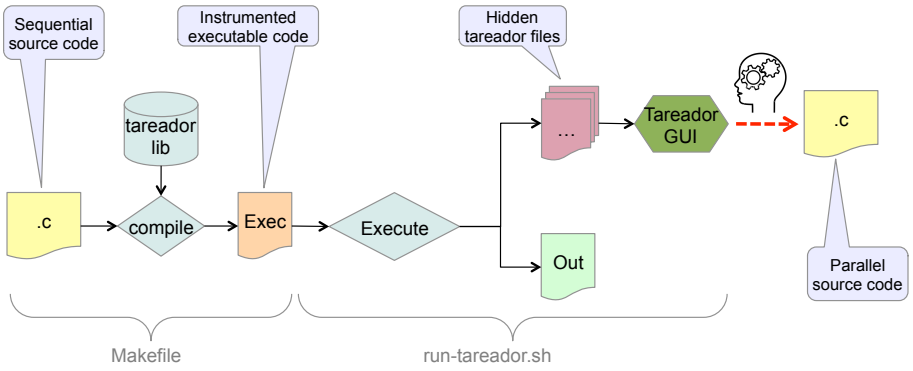

Figure 2.1: Compilation and execution flow for *Tareador*.

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.

In order to understand the possibilities of *Tareador*, you will use a program that computes the FFT *(Fast Fourier Transform)* of an input dataset in 3 directions (x, y and z), producing an output dataset that can be validated for correctness.

1. Go into the `lab1/3dfft` directory, open the `3dfft_tar.c` source code and identify the calls to the *Tareador* API, understanding the tasks that are initially defined. Also open the `Makefile` to understand how the source code is compiled and linked to produce the executable. Generate the executable by typing "`make 3dfft_tar`".

2. Execute the binary generated by typing `./run-tareador.sh 3dfft_tar`. Due to the instrumentation performed, the execution time may be several orders of magnitude higher than that of the original sequential code (warning presented to you in a window, just click `Ok` to continue the instrumented execution).

2.2 Brief *Tareador* hands–on

Next you will follow this short guided tour through some of the different options that *Tareador* offers to analyze the potential of task decomposition strategies.

1. The execution of the `run_tareador.sh` script opens a new window in which the task dependence graph is visualised (see Figure 2.2). 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 is 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. Simply zoom in and out to see the names of the tasks (the same that were provided in the source code) and the information reported for each node.

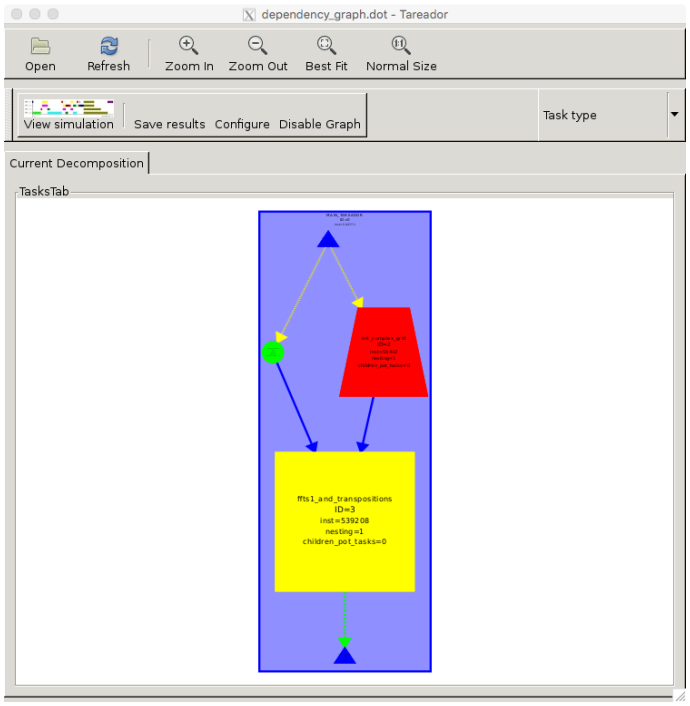

Figure 2.2: Task dependence graph for the initial task decomposition expressed in `3dftt_tar.c`.

2. 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). *Tareador* allows you to analyse the variables whose access provokes each data dependence between a pair of nodes: with the mouse on an edge (for example the edge going from the red task (`init_complex_grid`) to the yellow task (`ffts1_and_transpositions`), right click with the mouse and select *Dataview* → *edge*. This will open a window similar to the one shown in Figure 2.3. In the *Real dependency* tab, you can see the variable that causes that dependence (in this case the access to vector `in_fftw`. You can also right click with the mouse on a task (for example `ffts1_and_transpositions`) and select *Dataview* → *Edges-in*. This will open a window similar to the previous one again showing in the *Real dependency* tab the variables that cause the dependences for all the other tasks that are source of a dependence that sinks into the selected task (you can change the task that is source of the dependences in the upper selector). You can do the same for the edges going out of a task (selecting *Dataview* → *Edges-out* when clicking on top of a task).

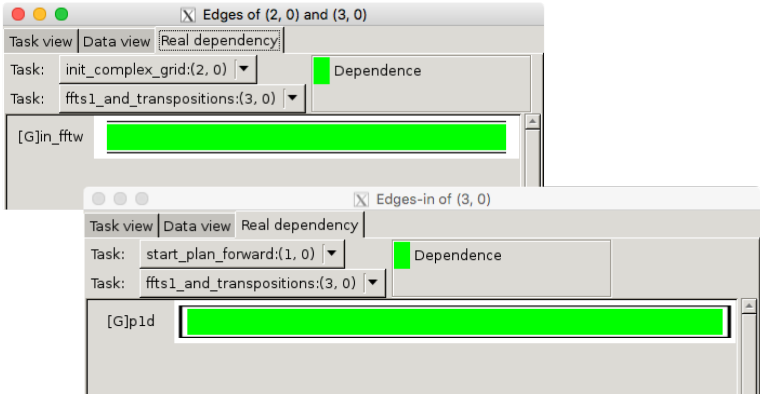

Figure 2.3: Visualisation of variables provoking data dependencies between tasks: for a specific *edge* or for all *edges–in* a specific task (the upper task chooser allows to select the task origin of the dependences).

3. For each node you can also analyse the variables that are accessed during the execution of the associated task. For example, with the mouse on node `ffts1_and_transpositions`, right click with the mouse and select *Dataview* → *node*. You can select either the *Task view* tab or the *Data view* tab in that window, as shown in Figure 2.4. 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, as in this case variable p1d), written (i.e. with a store memory access, blue color in the window) or both (orange color in the window, as in this case variable `in_fftw`). 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) that are performed by the task.

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

5. Once you understand the data dependences and the task graph generated, you can simulate the execution of the task graph in an ideal machine with a certain number of 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 2.5. Each horizontal line shows the task(s) executed by each processor (`CPU1.x`, with `x={1..4}`). Colours are used to represent the different tasks (same colours that are used in the task graph). The number on the lower-right corner of the window indicates the simulated execution time (in time units, assuming each instruction takes 1 time unit) for the parallel execution. In the next laboratory session you will go deep into the use of this tool, but for example you can zoom into the initial part of the timeline in order to visualise the same part of the trace that is shown in that figure; you can do

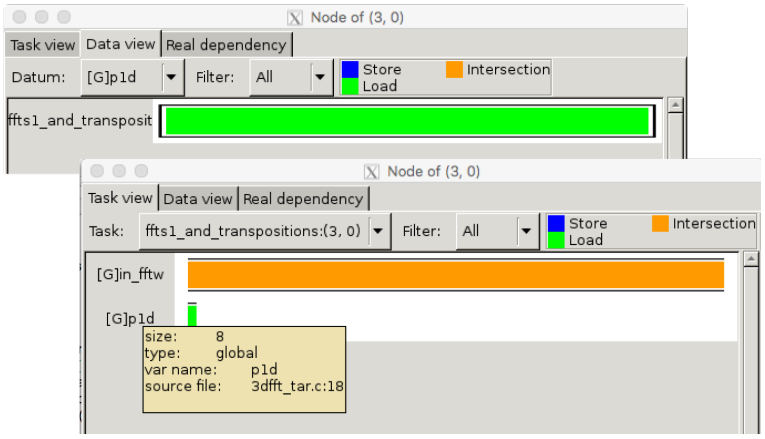

Figure 2.4: Visualisation of variables provoking data dependencies between tasks: for a specific *edge* or for all *edges–in* a specific task (the upper task chooser allows to select the task origin of the dependences).

this by clicking the left button in your mouse and selecting the zone you want to zoom. Yellow lines show task dependences (and creations). You can undo the zooms done by clicking *Undo zoom* or *Fit time scale* on top of the timeline *Paraver* window.

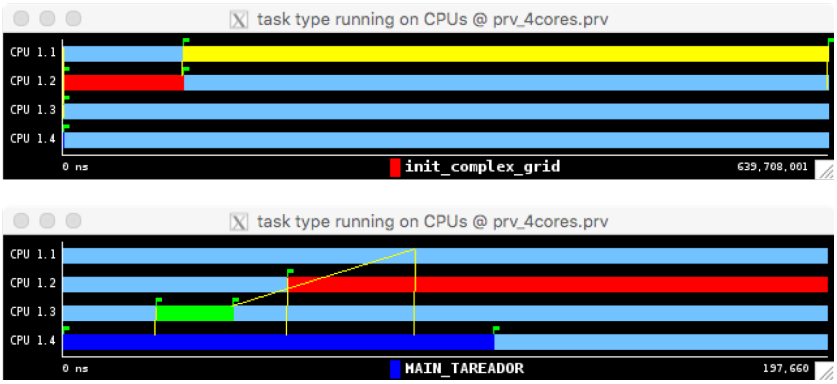

Figure 2.5: *Paraver* visualisation of the simulated execution with 4 processors, full view and after zooming into the initial part of the trace.

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

Although not useful for this code, you could disable the analysis of certain variables in the program 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 remove all the dependences caused by the selected variable in the given code region. For example, if you decide to disable the analysis of variable `in_fftw` you will observe in the new task dependence graph that tasks `init_complex_grid` and `ffts1_and_transpositions` can go in parallel.

2.3 Exploring new task decompositions for `3DFFT`

Once you are familiar with the basic features in *Tareador*, and motivated by the reduced parallelism obtained in the initial task decomposition (named v0 from now on), you will proceed refining the initial

tasks with the objective of discovering more parallelism. You will incrementally generate five new finer–grained task decompositions (named v1, v2, v3, v4 and v5) as described in the following bullets. For each task decomposition compute T_1, T_∞ and the potential parallelism ($T_1 \div T_\infty$) from the task dependence graph generated by *Tareador*, assuming that each instruction takes one time unit to execute. You can obtain T_∞ by simulating the execution of the graph with a sufficiently large number of processors.

1. Version v1: REPLACE[1] the task named `ffts1_and_transpositions` with a sequence of finer grained tasks, one for each function invocation inside it.

2. 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");
    }
}
```

   For this version pay special attention to the data dependences that appear in the task dependence graph. For example analyze the *Edges-in* for one of the transposition tasks, making sure you understand what is reported by *Tareador*.

3. 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`. Again, make sure you understand what is causing data dependences.

4. Version v4: starting from v3, REPLACE the definition of task for the `init_complex_grid` function with fine-grained tasks inside the body function. For this version v4, also simulate the parallel execution for 1, 2, 4, 8, 16 and 32 processors, drawing a graph or table showing the potential strong scalability. What is limiting the scalability of this version v4?

5. Version v5: finally create a new version in which you explore even finer-grained tasks. Due to the large number of tasks, *Tareador* may take a while to compute and draw the task dependence graph. Please be patient! Again, simulate the parallel execution for 1, 2, 4, 8, 16 and 32 processors, completing the previous graph or plot with the results obtained for version v5. According to the results, is it worth going to this granularity level? When?

Important: In the Deliverable you will have to include all the task dependence graphs obtained, the table for the T_1, T_∞ and the potential parallelism metrics for all the versions explored and the scalability plots for versions v4 and v5, commenting how performance is improving in the process. You should also include the relevant part(s) of the code to understand why v5 is able to scale to a higher number of processors compared to v4.

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

Session 3

Understanding the execution of *OpenMP* programs

The objective of this chapter is to present you the *Paraver* environment that will be used to gather information about the execution of a parallel application in *OpenMP* and visualise it. Figure 3.1 shows the complete compilation and execution flow that needs to be taken in order to trace the execution of a parallel *OpenMP* program. 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 this course only uses *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.

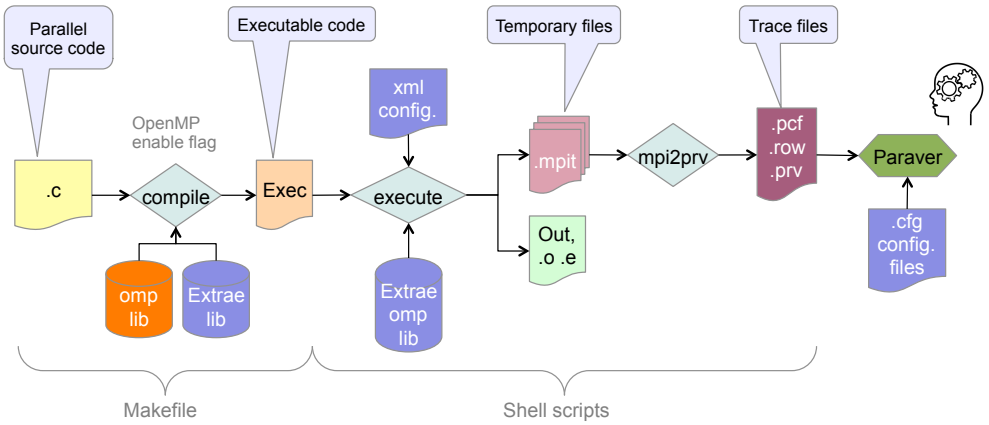

Figure 3.1: Compilation and execution flow for tracing.

For this chapter we provide you with an *OpenMP* implementation for the *3DFFT* code which implements a task decomposition with the same task granularity as version *v5* explored in the previous chapter.

1. Go into the `lab1/3dfft` directory. Open the `3dfft_omp.c` file in that directory and look for the lines containing *OpenMP* pragmas. We will study *OpenMP* in the next laboratory assignment, but for now just a bit of explanation of what you have here in the program. With '`#pragma omp parallel`' (for example in line 47) we are simply defining a parallel region, that is a region of code to be executed by a number of threads (processors); each thread will execute the body of the

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

`parallel` construct in a replicated way (what we will call the *implicit task*). With '`#pragma omp single`' (line 48) we are telling that only one of the threads in the parallel region will continue with the execution of the body of the `single` construct; the other threads will remain idle waiting at the end of the `single` construct for additional tasks to be executed. And finally, the thread that entered into `single` will encounter '`#pragma omp taskloop`' (line 50), indicating that the for loop that follows will be divided in a number of so called *explicit tasks*, each one to be executed by any of the idle threads. When the thread that entered into the `single` finishes with the generation of all tasks in the `taskloop`, it will also participate in the execution of the *explicit tasks* that it generated. When all *explicit tasks* are executed, the `parallel` region will be finished returning to serial execution until a new parallel region is found, if any in the code. In our program, three parallel regions are initially defined[2].

2. Compile both the `3dfft_seq.c` and `3dfft_omp.c` programs using the appropriate entry in the Makefile.

3. Execute the binary generated by launching to the queue the `submit-omp.sh` script, which receives the name of the executable file and the number of threads to use in the parallel execution. Execute with 1 and 8 threads. Observe that the execution generates a text file with information about the time taken by the different parts of the program: *FFT Plan Generation*, *Init Complex Grid* and *Execution*. Is the scalability obtained appropriate? In order to help you to better answer to this question you will be introduced to an environment to analyse the performance of a parallel application.

4. Now open the `submit-extrae.sh` script to see how the parallel binary is executed and traced. The script invokes your binary, which will use the *Extrae* library (by using the `LD_PRELOAD` mechanism) to emit events at runtime; the script also invokes `mpi2prv` to generate the final trace (`.prv`, `.pcf` and `.row`) and removes all intermediate files.

5. Submit the execution of the `3dfft_omp` binary using the `submit-extrae.sh` script, which as before receives the name of the binary file to execute and the number of threads to use. For now do the execution with 8 threads. Make sure that the trace is generated before opening it with *Paraver* and follow the hands-on guided tour in the next section.

3.1 Short *Paraver* hands–on

In this guided tour you will learn 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.

3.1.1 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*, shown in Figure 3.2 (left).

2. Load trace: From the main menu, select *"File → Load Trace"*, and navigate through the directory structure until you find the trace file (`.prv`) generated from the instrumented execution of the `3dfft_omp` binary. 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 about the parallel execution in the trace.

3. Once the file is loaded, click on the *New single timeline window* box (top left icon in *Main Window*). A new window, similar to the one shown in Figure 3.2 (top-right), appears showing a timeline with the *state* (encoded in colour) of each thread (vertical axis) in the parallel program. The horizontal axis represents time, advancing from left to right.

[2]There is a fourth parallel region that will be activated at the end of this chapter.

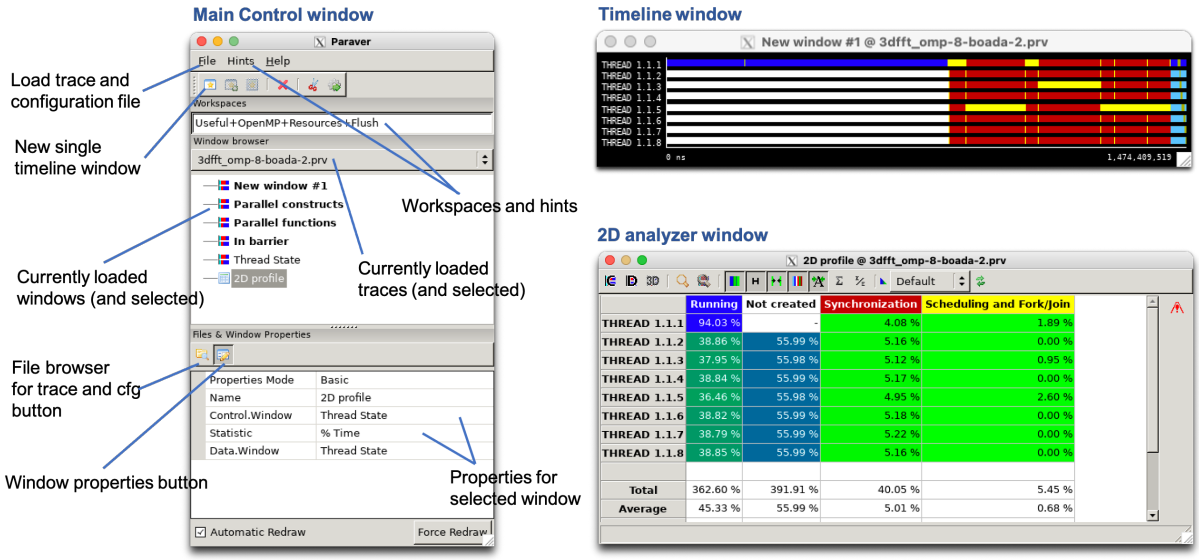

Figure 3.2: Paraver Main Window, Timeline and 2DAnalyzer windows.

- Colours: While moving the mouse over the window, a textual description of the meaning of each colour is shown (at the bottom of the same window): light blue (*idle*), dark blue (*running*), red (*synchronisation*), white (*not created*), yellow (*scheduling and fork-join*), ... It is important to be aware that the meaning of each color is specific to each window. Through this hands-on you will see different timeline windows each of them displaying a different information with its own colouring table. Click with the right button of mouse at any point inside the window and select *Info Panel* and then the *Colors* tab to see the colouring table for the window.
- Textual information: Double click with the left button of your mouse on any point in the red or yellow areas in the timeline. This action will activate the *What/Where* tab of the Info Panel, showing in textual form information at the selected point in the trace (thread and time where you have clicked, thread state at this point and how long the time interval with that state is). With *"Right Button → Info Panel"* in the trace window you can decide to show or hide this panel.

With this first window it is interesting to see how the parallel execution model in *OpenMP* programs is visualised: a master thread executing the sequential part of the program (dark blue with all the other threads not yet created, shown in white) until the parallel region is reached; at that moment threads are created and the already mentioned *explicit tasks* are created and executed, synchronising when necessary. Once the parallel region is finished, only the master continues its execution (dark blue) with all other threads remaining idle (light blue). In order to go deep in understanding how the program is executed you need to learn how to zoom in the trace in order to see a more datalled behaviour.

But before continuing into zooming, it is important to understand at this point that a single pixel of the window represents a time interval, and in that time interval the thread may change its state multiple times. So in this case, since *Paraver* can only use a single colour to paint each pixel, it has to choose one to summarise those multiple states the thread is. So don't get fooled by the colours shown and extract premature conclusions (for example, "it seems that this thread is only synchronising and not running useful code since it is all time in red" or "None of the threads inside the parallel region seems to be running useful code").

4. The current timeline window shows the execution from time zero to the total execution time (shown on the right bottom in nanoseconds). You can zoom in the trace, selecting areas that you want to further view in detail, and zoom out to go to lower levels of detail in the trace:

- 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.
- *Fit time scale* can be used to return to the initial view of the complete execution.

Now that you know how to zoom in and out, take some additional time to go deeper in understanding how the fork–join model in *OpenMP* works: zoom into the beginning of the first parallel region (yellow part that is setting up the threads and giving them work to do). You can observe the dark blue bursts representing the execution of tasks and the red bursts in between representing synchronisations, with less frequent yellow bursts that is when tasks are being generated. Play a little bit more zooming in and out to observe how threads transition states without trying to understand what really happens, we will analyse all this in more detail later in this guided tour.

3.1.2 Flags and configuration files

Flags are the other elements in the trace that provide information about the parallel execution; they have two properties: *type* and *value*. The *type* is used to encode the kind of event the flag contains while the *value* gives the specific value for the event. 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 (e.g. start/end of parallel region, function executed by *explicit tasks*, ...). Click on one of the flags and enable the *Event* tick box on the *What/where* tab of the *Info Panel*. Sometimes you need to click several times to select the pixel where the flag is painted (or click the *Prev./Next* tick box to show the information around clicked pixel). Flags are also useful to differentiate different bursts in what may look like a simple burst in the timeline.

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. Instead we will be using a set of pre-defined configuration files that extract the appropriate semantics out of the flags in the trace; configuration files are the simplest way to do the analysis of a trace, and specifically of the *Flags*. Next you will use some of them available in different sub-directories inside the `cfgs` directory in your home directory. To load a configuration file, from the main menu, select *"File → Load Configuration"*.

1. For example, configuration file `OMP_parallel_constructs.cfg` (in `cfgs/OpenMP`) identifies when `parallel` constructs are executed. For this window, red means when the master thread enters into a parallel region. Observe that in this trace only one thread encounters the parallel region and that there are 9 parallel regions executed (as delimited by the flags): 2 of these 9 regions are "fake" parallel regions intentionally introduced to delimit the start and end of the main program (to identify the part of the trace that is related with initialization of libraries, including *Extrae*, which is not of our interest for the evaluation of performance). To know more about the parallel regions in our program ...

2. Configuration file `OMP_implicit_tasks.cfg` (also in `cfgs/OpenMP`) identifies the functions that encapsulate the code regions that each thread executes when a parallel region is found (what we have called the "implicit task" associated to the parallel region). All threads contribute to the execution of the parallel region that was encountered by a single thread. The semantics for this window associates different colours to visualise different implicit tasks. The textual information shows the line number in the source file associated to the `parallel` construct. You could check the line number in the original source code to see which one of the 3 parallel regions identified in the code is executed. Can we know how much time it takes for a thread to execute these implicit tasks? Let's see ...

3. Configuration file `OMP_implicit_tasks_duration.cfg` (also in `cfgs/OpenMP`) shows the duration of the implicit tasks executed by threads in the parallel regions. A gradient coloured timeline is used to show the semantics of the window, painting each pixel in the window with a colour, from light green (low value for the duration) to dark blue (high value for the duration). By moving the

mouse arrow on top of the bursts *Paraver* will tell you their duration. Of course, clicking in one of them you can also have this information; but the gradient coloured window gives you a more global picture of parallel region durations in the same parallel region and across parallel regions.

4. What about the `#pragma omp single` construct that we saw in the source code? How can we see it in the trace? Just load the `OMP_worksharing_constructs.cfg` configuration file (also in `cfgs/OpenMP`) and look at the new timeline. In each parallel region you will see that only one of the threads has the green burst active all the time; this is the one that entered into the `single` region. For the others you will see that only have a very short green bar at the beginning to check if the `single` region has already been taken by someone; you can zoom to see this in more detail.

The upper part in Table 3.1 lists these and some other configuration files that are available in your home directory inside the `cfgs` directory for doing this kind of analysis. We will use them later in this tour or in other laboratory sessions.

***Paraver* cfg file**	**Timeline showing ...**
OMP_parallel_constructs	when a `parallel` construct is executed
OMP_implicit_tasks	the implicit task each thread executes in a `parallel` region
OMP_implicit_tasks_duration	the duration of the implicit task executed in a parallel region
OMP_worksharing_constructs	when threads are in worksharings (in this course, only `single`)
OMP_in_barrier	when threads are in a `barrier` synchronization
OMP_in_critical	when threads are in/out/entering/exiting `critical` synchronisations
***Paraver* cfg file**	**Profile showing ...**
OMP_state_profile	a profile of the thread states (useful, scheduling, synchronization, ...)
OMP_critical_profile	a profile of the different phases a thread goes on when synchronising in a `critical` construct
OMP_implicit_tasks_profile	a profile of the implicit tasks
OMP_implicit_tasks_duration_3DH	histograms with the durations of implicit tasks

Table 3.1: First set of basic configuration files to support analysis in *Paraver*– upper part: timeline views; lower part: statistical summaries in the form of profiles (histograms). Additional ones in the *cfgs* directory.

3.1.3 What a mess with so many *Paraver* windows!

Now that you have several *Paraver* windows open, it is important to be able to make them show the same time span or group them so that zooming in and out happens for some of them at once. In this way you will have a more coherent visualisation, as an alternative to killing/hiding the windows.

1. Aligning 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. To practise this, just take two windows already open, for example the initial state timeline window and the one opened with `OMP_implicit_tasks.cfg`. Right-click inside one of the two windows (the source or reference window) and select *Copy*; then on the target window, right-click and select *"Paste → Default"* (or separately *"Paste → Size"* and *"Paste → Time"*). Both windows will then have 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.

2. You can also synchronise several windows by adding them to the same group. For example select *Synchronise → 1* after a Right-click with your mouse on the `OMP_implicit_tasks.cfg` timeline window; this adds the window to the group named *1*. Repeat the `OMP_implicit_tasks_duration.cfg` timeline window. Once synchronised they will continue aligned after zooming, undoing or redoing zoom. With the two windows synchronised it is easy, for example, to correlate the duration of tasks with the implicit task they belong to, allowing us to observe if parallel regions take the same time and observe if there is some work unbalance among threads in any of them. You can always un-synchronise a window by un-selecting again *Synchronise* after a Right-click with your mouse on the timeline window. You can create new groups of synchronised windows at your convenience.

3.1.4 Explicit tasks

Implicit tasks encapsulating the code region executed in parallel constructs are not the only kind of tasks we will have in our *OpenMP* parallel programs. Other tasks can be created during the execution of parallel regions by any thread to give work to another thread to execute. These tasks will be named *explicit tasks* and they will be created when the `task` or `taskloop` (this is the case for the `3dfft_omp.c` program we are looking at) constructs are used in our program. Table 3.2 adds some new configuration files to the ones you just used in the previous subsection, all of them inside the `cfgs/OpenMP/OMP_tasks` directory. Let's play with some of them, others for later.

1. Open `OMP_explicit_tasks.cfg` to visualise when *explicit tasks* are created and when are executed. Synchronise the two windows that just opened and do a bit of zooming in to see the tasks in the different parallel regions (flags are here useful to delimit each one of the tasks, plenty of them!, all generated by the `taskloop` constructs that we have already seen in the program). Observe that each burst provides information about the line number in the source code where the tasks are defined. Open the *OpenMP* source code to see where these tasks are defined. You will see that each of them is associated to a particular `taskloop` construct.

2. Open `OMP_taskloop.cfg` configuration file to visualise the `taskloop` constructs that the previous seen tasks belong to. Synchronise the new window with the previous two windows to see when the thread that is executing the `taskloop` construct waits for the termination of all the tasks that have been generated (while waiting it also contributes with the execution of the generated tasks).

3. Finally open the `OMP_explicit_tasks_duration.cfg` configuration file to visualise the gradient coloured timeline showing the duration of the tasks executed by the different threads. You will be able to observe the task granularities generated in the different `taskloop` constructs and observe, for example, if the explicit tasks generated from the same `taskloop` have the same duration or not.

***Paraver* cfg file**	**Timeline showing ...**
OMP_explicit_tasks	when *explicit tasks* are created and executed (file and line inside file)
OMP_explicit_tasks_duration	duration of *explicit task* functions executed
OMP_taskloop	when the `taskloop` construct is executed
OMP_taskwait	when a `taskwait` construct is executed
OMP_in_taskgroup	when a task is starting or waiting in a `taskgroup`
***Paraver* cfg file**	**Profile showing ...**
OMP_explicit_tasks_3DH	a profile of task creations and executions
OMP_explicit_tasks_duration_3DH	histograms with durations of the *explicit tasks* executed

Table 3.2: Second set of basic configuration files to support tasking analysis in *Paraver*– upper part: timeline views; lower part: statistical summaries in the form of profiles (histograms). Others available in the *cfgs/OMP_tasks* directory.

3.1.5 More than timelines ... profiles and histograms!

The analysis above went directly to the detailed timeline. Usually a less detailed averaged statistic analysis is sufficient to identify performance issues and have a summarised view of the behaviour of an application. *Paraver* provides the *2DAnalyzer* mechanism to obtain such profiles and histograms.

1. Load configuration file `OMP_state_profile.cfg` (in *cfgs/OpenMP*). A table pops up, as the one shown in Figure 3.2 (bottom-right), with one row per thread and one column per thread state (*Running*, *Synchronization*, *Scheduling and Fork/Join*, ...). Each cell value shows the percentage of time spent by a thread in a specific state. Observe that all threads, except the the first one, spent some time in *Not created* and that the percentage spent in *Synchronization* is significant compared to *Running*. The later means that the efficiency of thread utilisation is low for this version of the code (don't worry, you will solve this in the next section). To see a different statistic change the *Statistic* selector in the *Window Properties* panel of the *Paraver Main Window*. Other interesting metrics at this time may be:

- *Time*: to show the total time spent on each state, per thread.
- *# Instances*: to count the number of times each state occurs.
- *Average Duration*: to show the average time a thread is in each state.

2. The previous profile and metrics are computed based on a single timeline window, in this case the initial timeline that you opened at the beginning showing the thread state along time. You can create such a profile for any of the timelines that you have opened, by simply clicking on the desired timeline window and clicking the *New Histogram* button in the *Main Window*, for example the one associated to `OMP_implicit_tasks.cfg` or `OMP_taskloop.cfg`. With that you can answer questions like "how many times a certain parallel construct is executed?", "how many `taskloop` constructs are executed and how much time their execution takes to complete?", ...

3. Finally, *Paraver* provides a *3DAnalyzer*, adding a new window to the previous 2D analysis capabilities. To better see this option, load `OMP_explicit_tasks_3DH.cfg`. A new window is opened showing a profile with information about tasks instantiated or executed by each thread; you have to select one of these two options in the *Window Properties* → *3D.Plane*. In the horizontal axis of the histogram you have the identification of the *explicit task*. Cells of the profile show the per-thread values for the metric selected in the *Window Properties* → *Statistic* selector (in the capture shown in Figure 3.3 the number of explicit tasks executed).

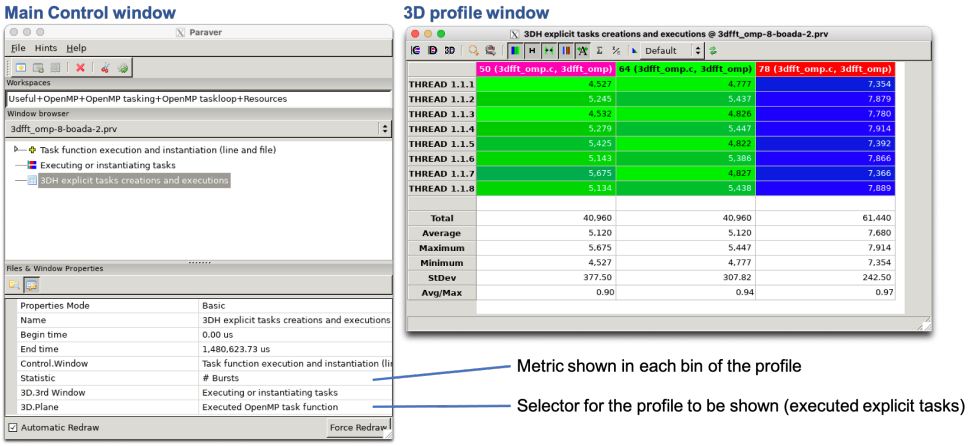

Figure 3.3: 3DAnalyzer window showing a profile for the explicit tasks in the program.

4. Similarly, you can load `OMP_explicit_tasks_duration_3DH.cfg` to see a more complex histogram, in this case showing the duration of tasks executed. A new window is opened showing a histogram with the durations of the tasks executed by each thread (in vertical axis); the specific *explicit task* for which the histogram is displayed can be selected in *Window Properties* → *3D.Plane*. In the horizontal axis the histogram shows task duration ranges. Bins of the histogram ara coloured following a gradient (from dark blue with high value to light green with a low value); the value in each bin corresponds to the metric selected in the *Statistics* selector in the *Window Properties* panel that (initially the total time spent executing tasks with that specific task duration range, as shown in Figure 3.4).

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. Select *Fit time scale* to compute the profile for the whole trace.

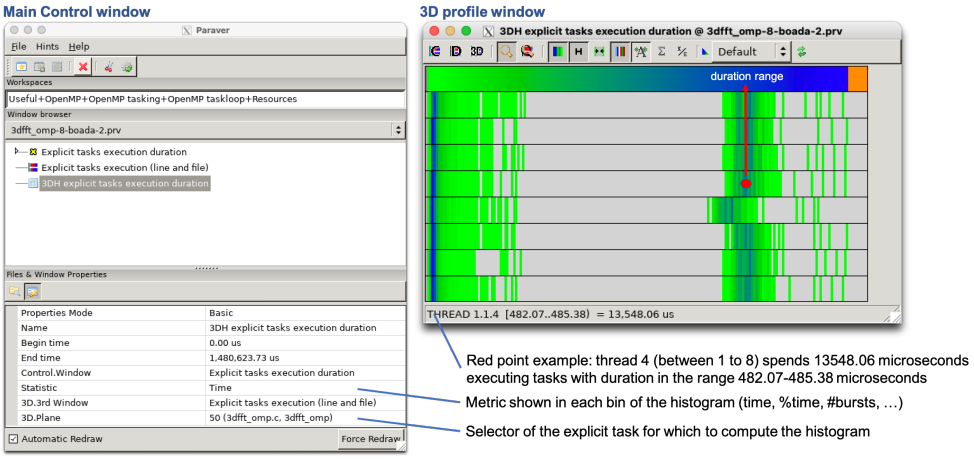

Figure 3.4: 3DAnalyzer window showing a histogram explicit tasks durations.

3.1.6 Can you make my life a bit easier, please?

Well, we can help you in two different ways. The first one is in finding and using the most common configuration files. For that *Paraver* provides *hints* (also named *workspaces*) that are loaded based on the flags available in the trace. The hints are grouped in different categories and available from the main menu in the *Main Window*. Take a look at them and try its correspondence with the configuration files provided in Tables 3.1 and 3.2. You will see that there are other hints that may also be useful (such as the instantaneous parallelism in the application or the mapping of cores to threads), just in case you still want to try more at this time.

The second way is by providing a summary of the most relevant performance metrics, cooked in some way to help you drive into the optimisation process of a parallel application. On this side we offer a Python program called `modelfactors.py` that receives a collection of traces to analyse. But we will propose to encapsulate its use in a script already prepared to generate the necessary traces for the strong scaling analysis of your code and do the analysis. We will indicate in the next section how to execute it and the required arguments. By now, we will just explain you the information reported in the text file `modelfactors.out` generated:

- In a first table the tool provides a summary of program execution metrics for the entire program and for the different number of threads p analysed: elapsed execution time (in seconds, let's name it T_p), Speed–up (i.e. $S_p = T_1 \div T_p$) and Efficiency (i.e. $E_p = S_p \div p$). With this information you can observe the scalability of your code. If not happy with the reported results, you can go to the next table in the summary.

- In a second table the tool provides a deep dive into the Efficiency metric, but only for the *parallel fraction* (ϕ) of your program, that is, the part of the program that can be (or it is currently) parallelised. This is a good metric to realise if the inefficiencies come from a large serial part in your program that has not yet been optimised or simply can not be optimised. For the parallel fraction of the code the tool reports two measures that contribute to its efficiency:

  - *Parallelisation strategy efficiency*, which is related to two main aspects: load balancing and overheads due to work generation and synchronisation in the critical path of the execution. The higher and close to 100% these two values are the better. With these two metrics it should be enough to know what to do next in your analysis and optimisation.
  - *Scalability for computation tasks*, which is related to how the processors in your system are actually executing the computation tasks, in terms of total number of instructions executed, number of instructions executed per cycle of operation, and frequency (number of cycles of operation per second) of the processor. The value of these metrics for p threads are relative to 1 thread and again, the closer to 100% the better. But they can be lower for example if in the parallel execution data locality in cache memories is impacted.

3.2 Obtaining parallelisation metrics for 3DFFT using modelfactors

3.2.1 Initial version

Now it is your turn! Starting from the initial version of `3dfft_omp.c` code, you will apply two performance optimisations. In addition to using *Paraver* and the configuration files to obtain the metrics and timelines that will help you to understand what the code optimisations do and the effect they have in the execution time, we also propose to use the `modelfactors` application. You will be generating multiple sets of traces, so make sure to save them with different names so that you can open and analyse them at any time. For the initial code version that we have been working on:

1. Submit the execution of the `submit-strong-extrae.sh` script indicating the name of the binary program `3dfft_omp`. This will take some time (minutes) since the script is tracing the execution with 1, 2, 4, 6, 8, 10 and 12 threads and performs the analysis with `modelfactors`; so monitor the status of the job in the queue to check when it is finished. Once finished check that the directory `3dftt_omp-strong-extrae` exists, and inside it the `modelfactors.out` file. Is the scalability appropriate? Which is the parallel fraction (ϕ) for this version of the program? Is the efficiency for the parallel regions appropriate? Which is the factor that is negatively influencing the most?

2. Open the trace generated for 8 threads and validate the diagnostic by `modelfactors`. Also obtain the histograms with the durations of the different **implicit tasks**. They will be useful to compare with the durations after the optimisation performed in the next subsection.

3. Finally obtain the strong scalability plot by submitting the execution of the `submit-strong-omp.sh` script, commenting the trend in that plot and checking with the values reported by `modelfactors`. How far is the real speed–up S_8 achieved from the ideal speed–up S_8 that you can compute from the value of ϕ that you computed?

Before continuing, make sure to rename the `3dfft_omp-strong-extrae` directory since it will be regenerated with new executions. Do this each time you work on another version if you need to keep the previous ones. However, beware that you can run out of disk quota. Therefore it's advisable to keep the information you regard as useful in another directory and remove all the other files.

3.2.2 Improving ϕ

Which function in the program is causing the low value for ϕ? Open the `3dfft_omp.c` file and uncomment the pragmas that will allow the parallel execution of the code inside function `init_complex_grid`. At this point only uncomment the innermost `taskloop` together with `parallel` and `single`.

1. Recompile and submit for execution the `submit-strong-extrae.sh` script, obtaining the new values for all the previous metrics. Have the speed–up and efficiency metrics improved? What about the new value for ϕ? Is the efficiency for the parallel regions appropriate? Which is the factor that is now negatively influencing the most?

2. Obtain the thread state profile to see the time (or % of time) threads spend doing useful work (dark blue) or synchronizing their execution (red) to validate your conclusions from `modelfactors`.

3. At this point we recommend that you visualise the traces for the two versions simultaneously, for example visualising at the same time the initial timeline window for both of them. In order to see the net effect of this optimization (reduction of execution time), click *Copy* in the window for the first version and *Paste* $\rightarrow$ *Time* in the second window. For the Deliverable capture the *Paraver* timeline(s) window(s) that allow to make the appropriate comparison and explain your conclusions concisely.

4. Obtain the strong scalability plot by submitting the execution of the `submit-strong-omp.sh` script, commenting the trend in that plot and checking with the values reported by `modelfactors`.

5. Finally, obtain the histograms with the durations of the different **implicit tasks**. Do you observe any major difference with respect to the ones obtained before? What do you think has provoked this difference?

3.2.3 Reducing parallelisation overheads

To finish with this session, we will explore the possibility of increasing the granularity of tasks by simply commenting the innermost `taskloop` in each parallel region and uncommenting the outer one. This should reduce parallelisation overheads and as a consequence have some impact in the overall performance obtained.

1. Recompile and submit for execution the `submit-strong-extrae.sh` script, obtaining the new values for all the previous metrics. Have the speed–up and efficiency metrics improved? Is the efficiency for the parallel regions appropriate? Obtain the strong scalability plot by submitting `submit-strong-omp.sh`.

2. As before, visualize the traces for the three versions simultaneously and make sure that all of them are with the same temporal scale. For the Deliverable capture the *Paraver* timeline(s) window(s) that allow to make the comparison of the tree versions and see the impact on the overall execution time. Remember to explain your conclusions briefly.

3. Obtain again the new profile for thread state and compare with the obtained before.

4

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) describing the results and conclusions that you have obtained when doing the assignment. As part of the document, you will have to include any code fragment, figure or plot you need to support your explanations. Your professor will open the assignment in *Atenea* and set the appropriate delivery dates for the delivery. Only one file has to be submitted per group.

Important: The quality of the reports that you deliver will also contribute to the grade of the laboratory assignment. It is important that you include all conclusions from the analysis of the parallel codes, the behaviour of their serial/parallel execution and as well as the performance obtained based on evidences obtained with the tools provided. Use plots and tables to provide data which supports your explanations. Note that plots and tables should have a caption and be referenced within the text using the plot or table number. 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.

Node architecture and memory

Describe the architecture of the `boada` server. To accompany your description, you should refer to the following table summarising the relevant architectural characteristics of the nodes you will be using during this course:

	`boada-1` **to** `boada-4`
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)	

Also include in the description the architectural diagram for one of the nodes `boada-1` to `boada-4` as obtained when using the `lstopo` command. Appropriately comment whatever you consider appropriate.

Strong vs. weak scalability

Explain the differences that you have observed between interactive and queued execution for `pi_omp.c`, filling in table below to show the user and system CPU time, the elapsed time, and the % of CPU used

in these two scenarios and with 1, 2, 4 and 8 threads.

# threads	Interactive				Queued			
	user	system	elapsed	% of CPU	user	system	elapsed	% of CPU
1								
2								
4								
8								

Also explain what strong and weak scalability refer to, exemplifying your explanation with the execution time and speed–up plots that you obtained for `pi_omp.c`.

Analysis of task decompositions for *3DFFT*

In this part of the report you should summarise the main conclusions from the analysis of task decompositions for the *3DFFT* program. Backup your conclusions with the task dependence graphs and the following table properly filled in with the information obtained in the laboratory session for the initial and different versions generated for `3dfft_tar.c`, briefly commenting the evolution of the metrics.

Version	T_1	T_∞	**Parallelism**
seq			
v1			
v2			
v3			
v4			
v5			

For versions v4 and v5 of `3dfft_tar.c` perform an analysis of the potential strong scalability that is expected. For that include a plot with the execution time and speedup when using 1, 2, 4, 8, 16 and 32 processors, as reported by the simulation module inside *Tareador*. You should also include the relevant(s) part(s) of the code to understand why v5 is able to scale to a higher number of processors compared to v4.

Understanding the parallel execution of *3DFFT*

In this final section of your report you should comment the parallel performance evolution that you observed for the three *OpenMP* parallel versions of *3DFFT*. Support your explanations with the results reported in the following table (which you obtained during the laboratory session) and the *Paraver* windows that you captured showing the most appropriate and strictly necessary execution timelines for all the versions of the code, all in the same time scale, as well as profiles/histograms. Compare them with the information provided by the `modelfactors` application.

Version	ϕ	**ideal** S_8	T_1	T_8	**real** S_8
initial version in `3dfft_omp.c`					
new version with improved ϕ					
final version with reduced parallelisation overheads					

Finally you should comment about the (strong) scalability plots (execution time and speed–up) that have obtained when varying the number of threads for the three parallel versions.