PAR Laboratory Assignment
Lab 1: Experimental setup and tools

Ll. Àlvarez, E. Ayguadé, J. R. Herrero, J. Morillo, J. Tubella and G. Utrera

Fall 2018-19

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 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 is to 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 where you can execute interactive jobs and from where you can submit execution jobs to the rest of the nodes in the machine. `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 you are going to mainly use nodes `boada-1` to `boada-4`, either interactively or through the `execution` queue, as explained later in this chapter.

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 `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, it is strongly recommended 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.

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

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

copying the source file `pi_seq.c` inside directory `lab1/pi` of your home directory in `boada` to the current directory 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 different nodes in `boada`. To do this 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. the number of sockets, cores per socket and threads per core in a specific node of `boada`;
2. the amount of main memory in a specific node of `boada`, and each NUMAnode;
3. the cache memory hierarchy (L1, L2 and L3), private or shared to each core/socket.

Use the `"--of fig map.fig"` option for `lstopo` in order to get a drawing of the architecture for 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[3].

Next you should do the same for the rest of node types in `boada`. To do that you have to execute the two previous commands remotely using the `ssh` command on one of the nodes of each kind. For example type `ssh boada-6 "lstopo --of fig map-6.fig"` to remotely execute the `lstopo` command in `boada-6` and generate the output file `map-6.fig` in your home directory. With all this information you will be able fill in the table requested for the deliverable, which will be useful to compare side–by–side the different node types in `boada`.

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.

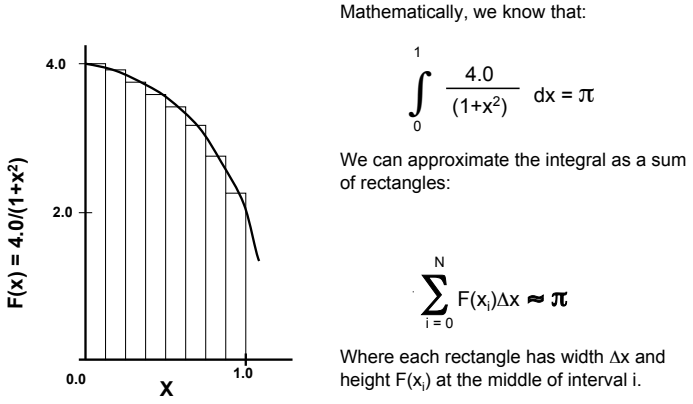

Figure 1.1: Pi computation

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.

[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"`.

```
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.

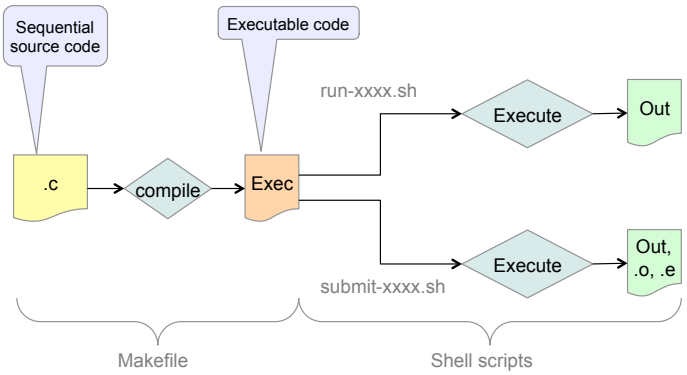

Figure 1.3: Compilation and execution flow for sequential program.

Once compiled, you will use two different ways to execute your programs: 1) via a queueing system (in one of the nodes `boada-2` to `boada-8`); or 2) interactively (in `boada-1` itself). It is strongly suggested 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, scripts for both options (`submit-xxxx.sh` and `run-xxxx.sh`, respectively) will be provided:

- Queueing a job for execution: `"qsub -l queue-name submit-xxxx.sh"`. Additional parameters may be specified after the script name. If you do not specify the name of the queue with `"-l queue-name"` your script will not be run, remaining in the queue forever. 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 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 target"` in order to generate the binary executable file.

2. Interactively execute the binary file generated to compute the pi number using the `run-seq.sh` script with the appropriate arguments (executable name and number of iterations 1.000.000.000). 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` queue using the `qsub` command with the appropriate arguments (executable name and number of iterations 1.000.000.000). Use `qstat` to see that your script is queued; however if it is not executed is because you have not specified the `"-l execution"` queue name; in this case Identify your job-ID number in the `qstat` output, use `"qdel job-ID-number"` to remove it from the queue and re-submit it using the `"qsub -l execution"` command, making sure it is running with `qstat`. Look at the files generated and their content: the standard output and error of the script and the `time-pi_seq-boada-{2-4}` file. Please also take a look at the `submit-seq.sh` script.

1.3 Compilation and execution of *OpenMP* programs

In this course you are going to use *OpenMP*, the standard for parallel programming using shared-memory, to express parallelism in the C programming language. *OpenMP* will be progressively introduced and explained along the different laboratory assignments. To start with, let's take a look at the parallel code that is provided for the computation of Pi. Go into the `lab1/pi` directory and edit the file `pi_omp.c`. The code looks similar to the code shown in Figure 1.4.

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

    step = 1.0/(double) num_steps;
    // omp_set_num_threads(8);
    #pragma omp parallel private(x) reduction(+: sum) // num_threads(8)
    {
        long int myid = omp_get_thread_num();
        long int howmany = omp_get_num_threads();
        for (long int i=myid; i<num_steps; i+=howmany) {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
    }
    pi = step * sum;
}
```

Figure 1.4: *OpenMP* parallel version for `pi_omp.c`.

The first difference with the sequential version in Figure 1.2 is the `#pragma omp parallel` construct. Figure 1.5 shows the *fork–join* execution model that is fundamental in OpenMP; when the master thread executing the serial part of the program encounters a `parallel` construct, it spawns a team of threads, composed of itself and zero or more additional threads. The total number of threads in the team can be globally defined (using the `OMP_NUM_THREADS` environment variable, see later), dynamically during program execution (using the `omp_set_num_threads()` intrinsic function) or for each individual `parallel` region (using the `num_threads` clause). All threads join at the end of the parallel region in an implicit *barrier* and the master thread proceeds executing sequentially. As shown in Figure 1.5 parallel

regions can be nested, but this possibility will not be explored for now. In this fork-join model, all threads replicate the execution of an implicit task that contains all the code in the body of the `parallel` region. Inside the parallel region any thread is identified with an identifier, between 0 and the number of threads in the team minus 1. The identifier can be obtained by calling the `omp_get_thread_num()` intrinsic function and the number of participating threads by calling `omp_get_num_threads()`. *OpenMP* assumes that all threads share the access to variables by default (i.e. unless otherwise specified).

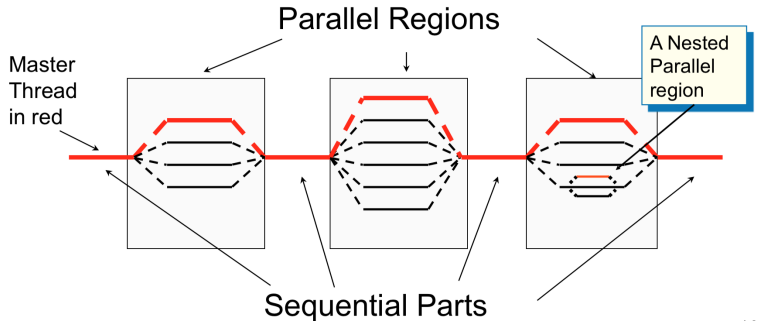

Figure 1.5: Fork–join execution model in OpenMP.

In the code shown in Figure 1.4 each thread executing the body of the parallel region computes the area for a subset of all the rectangles, each thread starting from the rectangle that corresponds to its identifier (stored in variable `myid`) and jumps as many rectangles as the number of threads in the team (stored in variable `howmany`) until all rectangles are computed. Observe that variable `i` is private to each thread in the team, since it is defined inside the body of the parallel region, as done with `myid` and `howmany`. This is needed in order to guarantee that each thread traverses the subset of assigned rectangles. The problem is with variable `sum`: each thread needs to accumulate the area of each computed rectangle in variable `sum`, which should be done ensuring that there are no simultaneous updates (in other words, threads update variable `sum` one at a time). In this version of the code this is done with the `reduction` clause, which specifies a variable (`sum` in this case) and an operator to apply (`+` in this case). With this clause the compiler defines a private per–thread copy of `sum`, initialized to the neutral value for the operation specified (i.e. zero for the addition), which is used by each thread to accumulate the area of its rectangles, and at the end updates the original shared variable `sum` avoiding data races (i.e. making sure that only one thread at a time updates the original `sum`). Each thread also declares a `private` copy of variable `x`. Why?

1.3.1 Compiling and executing *OpenMP* programs

Now that you understand your first parallel program in *OpenMP*, let's compile and execute it in `boada`. Figure 1.6 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*.

1. Open again 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 that enables the use of *OpenMP*. Execute the command line `"make target"` in order to generate the binary executable file.

2. Interactively execute the *OpenMP* code with 1 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.

3. Use `submit-omp.sh` script to queue the execution of the *OpenMP* code with 1 and 8 threads and same number of iterations (1.000.000.000). Do you observe a major difference between the interactive and queued execution?

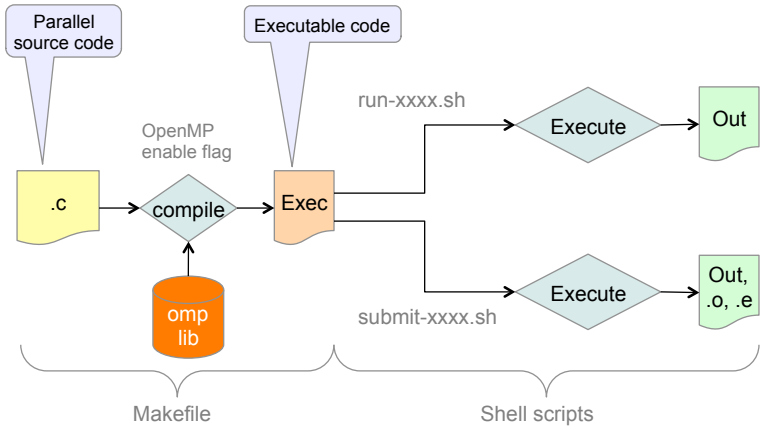

Figure 1.6: Compilation and execution flow for OpenMP.

1.4 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[4]. Observe how the execution time and speed–up varies with the number of threads in the *strong* scaling scenario.
2. Submit the execution of the `submit-weak-omp.sh` script (no arguments are required to execute the script). Observe now how the speed–up varies with the number of threads in the *weak* scaling scenario. Reason about the differences observed between the two scenarios.
3. Repeat the execution of the two scripts in the different node types in `boada`. Do you observe any significant differences among them?

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

Session 2

Systematically analysing task decompositions with *Tareador*

This chapter introduces *Tareador*, an environment to analyse 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.

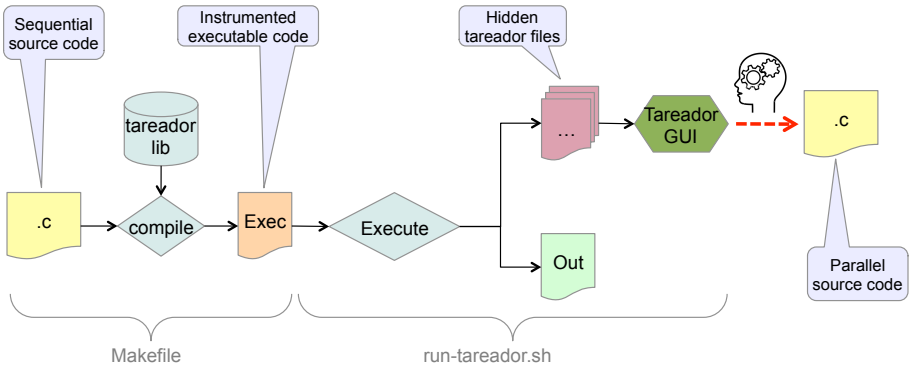

Figure 2.1: Compilation and execution flow for *Tareador*.

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`[1]. Due to the instrumentation performed, the execution time may take several orders of magnitude more than 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 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.

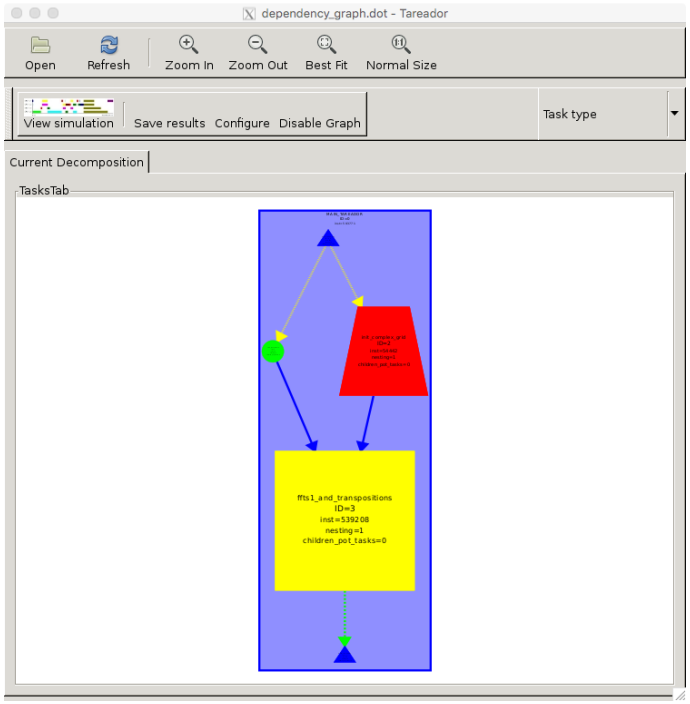

Figure 2.2: Task dependence graph for the initial task decomposition expressed in `3dftt_tar.c`.

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

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).

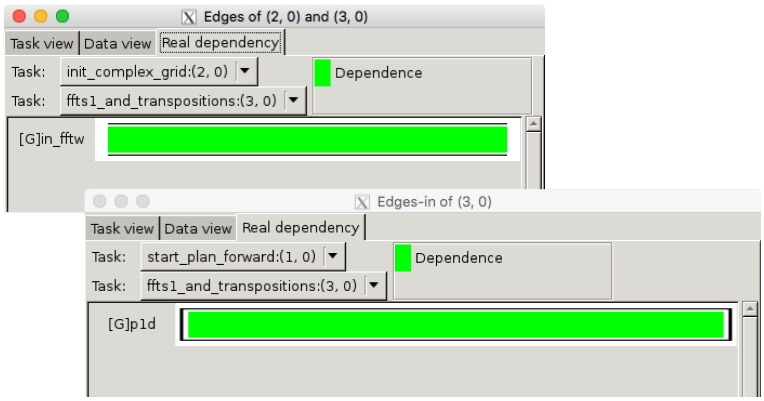

Figure 2.3: Visualisation of variables provoking data dependencies between tasks: for a specific *edge* or for all *edges–in* a specific task (the upper task chooser allows to select the task origin of the dependences).

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

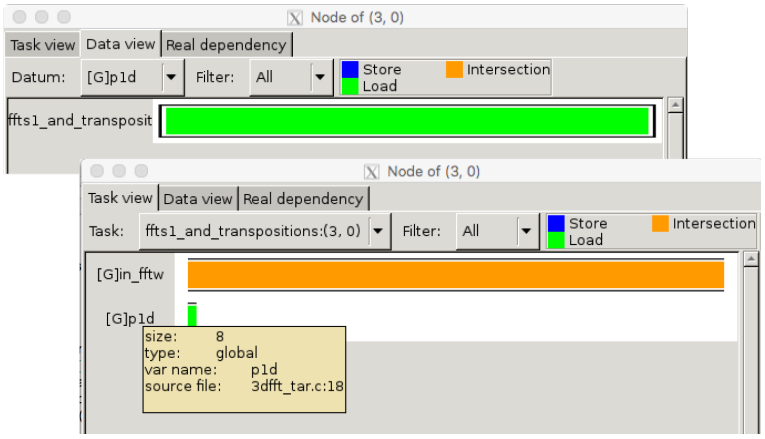

Figure 2.4: Visualisation of variables provoking data dependencies between tasks: for a specific *edge* or for all *edges–in* a specific task (the upper task chooser allows to select the task origin of the dependences).

will 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 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.

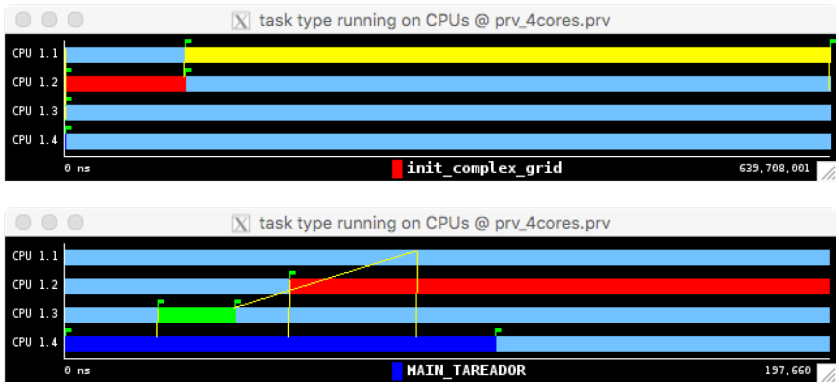

Figure 2.5: *Paraver* visualisation of the simulated execution with 4 processors, full view and after zooming into the initial part of the trace.

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–gained 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[2] 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?

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

Session 3

Understanding the execution of *OpenMP* programs

The first objective of this chapter is to go one step deeper in the *OpenMP* programming model by turning one of the task decompositions expressed in *Tareador* for the *3DFFT* code into a real *OpenMP* parallel program. The second objective 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.

3.1 *OpenMP* parallelization for `3DFFT`

Go into the `lab1/3dfft` directory. Let's take a look at the *OpenMP* parallelization for one of the loops in `3dfft_omp.c`, that follows the v5 task decomposition strategy explored in the previous chapter, shown in Figure 3.1.

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

    for (k=0; k<N; k++)
       #pragma omp parallel
       #pragma omp for schedule(static, 1)
       for (j=0; j<N; j++)
          fftwf_execute_dft( p1d, (fftwf_complex *) in_fftw[k][j][0],
                                  (fftwf_complex *) in_fftw[k][j][0] );
}
...
int main (int argc, char *argv[]) {
    ...
    ffts1_planes(p1d, in_fftw);
    ...
}
```

Figure 3.1: *OpenMP* parallelization for one of the loops in `3dftt_omp.c`.

The nested loop in function `ffts1_planes` repeatedly invokes the computation of the DFT for a vector, `fftwf_execute_dft`, for each plane `k` and column `j`. From the analysis in the previous chapter you already know that all these invocations can be executed in parallel. In order to execute in parallel, the *OpenMP* `parallel` construct you are already familiar with is used. This construct creates a team with a certain number of threads, each executing an *implicit task* that includes the execution of the `j` loop. In order to avoid the replicated execution of this loop, the `#pragma omp for` *work–sharing* construct is introduced. This work–sharing distributes the iterations of the `j` loop among the participating threads (i.e. among the implicit tasks), according to the distribution strategy specified in the `schedule` clause, in this case `(static, 1)`. This `(static, 1)` strategy performs exactly the distribution of iterations

what was realised in Figure 1.4 for Pi: each thread starts from (0+`omp_get_thread_num()`) and jumps `omp_get_num_threads()` iterations until the last iteration is reached. All threads in the team perform an implicit *barrier* synchronization at the end of the `for` work–sharing construct, waiting each other until the last one finishes. Loop control variable `j` is implicitly privatized by the *OpenMP* `for` construct itself. so that each thread uses its own copy to traverse the assigned iterations.

Figure 3.2 illustrates the different schedule kinds in *OpenMP* applied to a loop with N iterations (0..N-1); try to understand them. In static schedules iterations are assigned in chunks in a pre-determined order (thread order) while in dynamic schedules (including the so-called `guided`) chunks of iterations are dynamically assigned, as soon as a thread requests the execution of the next chunk to execute.

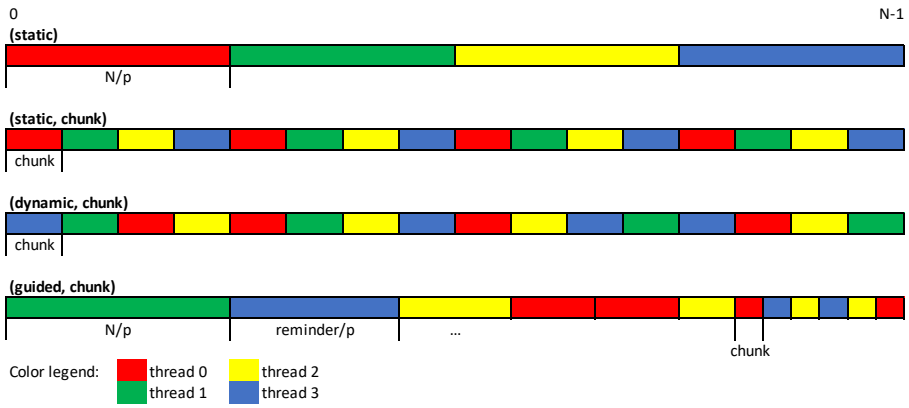

Figure 3.2: Iteration scheduling strategies in *OpenMP*.

1. Take a look at the complete `3dfft_omp.c` file in order to see the different loops that are parallelised inside the scope of the parallel regions.

2. Compile the `3dfft_omp.c` program using the appropriate entry in the Makefile.

3. Execute the binary generated using 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. The script generates a text file with information about the time taken by the different parts of the program. 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.

3.2 Generation of a trace with *Extrae*

Figure 3.3 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.

1. Open the `submit-omp-i.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 evens at runtime; the script also invokes `mpi2prv` to generate the final trace (`.prv`, `.pcf` and `.row`) and removes all intermediate files.

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

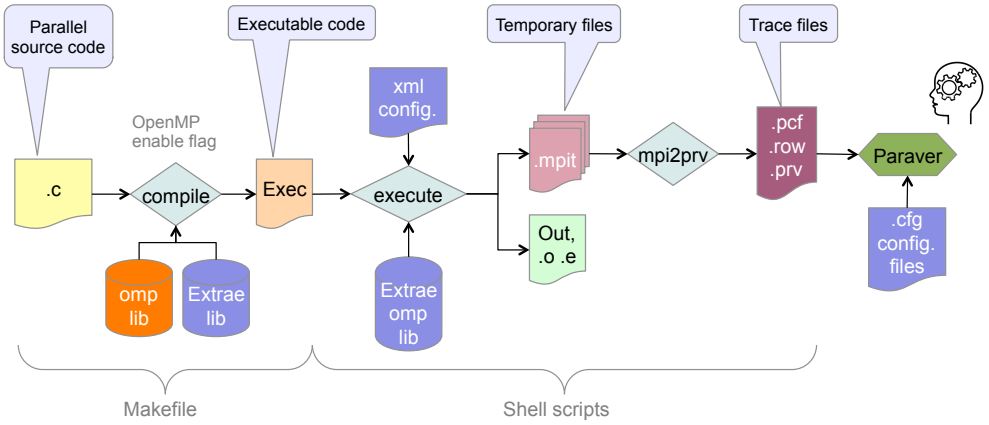

Figure 3.3: Compilation and execution flow for tracing.

2. Submit the execution of `3dfft_omp.c` using the `submit-omp-i.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.

3.3 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.

3.3.1 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 3.4 (left).

2. Load trace: From the main menu, select *"File → Load Trace"*, and select the trace generated from the instrumented execution of the parallel `3dfft_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 in *Main Window*). A new window, as the one shown in Figure 3.4 (top-right), appears showing a timeline with the activity (state, encoded in colour) of each thread (vertical axis) in the parallel program. The horizontal axis represents time, from left to right.

   - 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.
   - 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 colour 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.

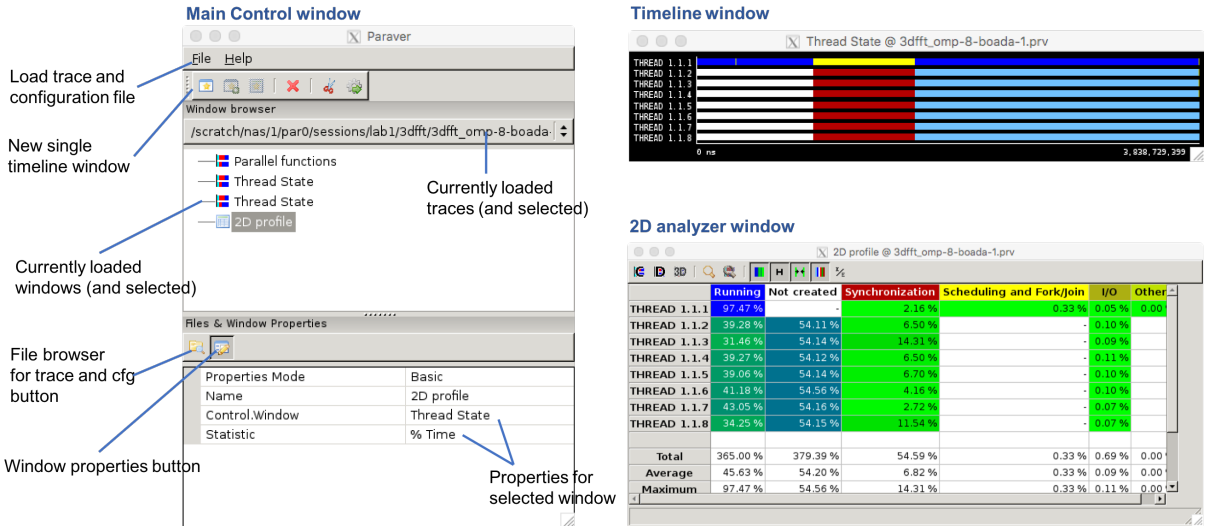

Figure 3.4: Paraver Main Window, Timeline and 2D Analyzer windows.

Spend some time to understand how the parallel execution model in *OpenMP* programs is visualised: a master thread executing the sequential part of the program (with all the other threads not yet created) until the parallel region is reached; at that moment threads are created and start executing and synchronising. Once the parallel region is finished, only the master continues its execution with all other threads remaining idle; but to see all this you need to learn how to zoom 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 your time to understand how the fork–join model in *OpenMP* works: zoom into the beginning of the parallel region (yellow part that is setting up the threads and giving them work to do), the red part between dark blue bursts representing the implicit barrier at the end of each `for` work–sharing and parallel region termination (with implicit barrier in red and yellow to indicate the closing of the region), ...

4. *Flags* are the other elements in the trace that provide information about the parallel execution. 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. parallel region and work–sharing constructs). Click on one of the flags and enable the *Event* tick box on the *What/where* tab. Flags are also 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.

5. 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.

   - For example, configuration file `OMP_parallel_constructs.cfg` (in `cfgs/OpenMP`) can be used to identify when `parallel` constructs are executed. To load this configuration file, from the main menu, select *"File → Load Configuration"*. For this window, red means when the master thread enters into a parallel region.

***Paraver* cfg file**	**Timeline showing ...**
OMP_parallel_constructs	when a `parallel` construct is executed
OMP_parallel_functions	the function each thread executes in a parallel region
OMP_parallel_functions_duration	the duration for the function executed in a parallel region
OMP_worksharing_constructs	when threads are in a worksharing construct (`for` or `single`)
OMP_worksharings_duration	the duration of worksharing regions
OMP_in_barrier	when threads are in a barrier synchronization
OMP_in_schedforkjoin	when threads are scheduling work, forking or joining
OMP_in_critical	when threads are in/out/entering/exiting critical sections
***Paraver* cfg file**	**Profile showing ...**
OMP_state_profile	the time spent in different OpenMP states (useful, scheduling/fork/join, synchronization, ...)

Table 3.1: First set of configuration files to support analysis in *Paraver*– upper part: timeline views; lower part: statistical summaries.

- Configuration file `OMP_parallel_functions.cfg` (also in `cfgs/OpenMP`) can be used to identify when threads in a team execute the implicit task associated to the parallel constructs shown with the previous configuration file. In this window, different colors are used here to visualise different parallel functions. The textual information shows the line number in the source file associated to the `parallel` construct.
- Open the `OMP_in_barrier.cfg` configuration file to visualise the synchronisation activity (in implicit barriers at the end of parallel regions in this case) in the parallel execution.
- Open the `OMP_in_schedforkjoin.cfg` configuration file to visualise when the master thread is forking and joining the team of threads in each `parallel` construct.

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.

6. 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.

3.3.2 Profiles

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

1. Load configuration file `OMP_state_profile.cfg`. A table pops up, as the one shown in Figure 3.4 (bottom-right), 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:
   - *Time*: to show the total time spent on each state, per thread.
   - *% 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 is called 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 that was opened at the beginning. The values of the control window determine to which column is a given statistic accumulated/accounted. Any of the opened timelines can be selected in *Control Window* selector in the *Main Window*, for example the one associated to `OMP_parallel_functions.cfg` or to `OMP_in_barrier.cfg`. With that you can answer questions like "how many times a certain parallel construct is executed?", "how much time or which percentage of time each thread has been spent waiting in barriers?", ...

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.

3.4 Improving the parallelization of `3DFFT` using *Paraver*

3.4.1 Initial version

Using *Paraver* and the subset of options you know about it, obtain the following metrics for the initial parallel version in `3dfft_omp.c`:

1. Execution times T_{seq} and T_{par} and the *parallel fraction* $\phi = T_{par} \div (T_{seq} + T_{par})$. In the sequential program T_{seq} corresponds with the time spent in those parts of the program that can NOT be parallelized, while T_{par} corresponds with the time spent in those parts of the program that can be parallelized; in fact, $T_1 = T_{seq} + T_{par}$. You can approximate the value for these two metrics applying the appropriate configuration file to trace generated from the Extrae–enabled execution of the parallel version executed on a single processor.

2. Execution time T_8 and Speed–up S_8 when the program is executed with 8 processors. Obtain the profile with the percentage of time spent in the different *OpenMP* states.

3. Obtain the value for the same metrics (T_1, T_8 and S_8) directly using the time information reported by the non–instrumented version of the program, comparing with the times that you see in *Paraver*. Observe that *Paraver* is introducing some extra time needed to initialise the instrumentation (in fact up to a point in the trace shown with a small state burst labeled "Others"), but then the behaviour is reflected accurately with no much overhead.

4. Obtain the strong scalability plot by submitting the execution of the `submit-strong-omp.sh` script. How far is the speed–up achieved from the ideal S_∞ that you can compute from the previous value of ϕ?

3.4.2 Improving ϕ

Which function in the program is causing the low value for ϕ? Add the necessary pragmas in `3dfft_omp.c` in order to execute this function also in parallel. Recompile and execute, obtaining the new values for ϕ, S_8 and the new strong scalability plot. How far from the ideal S_∞ is the scalability plot that you have obtained?.

3.4.3 Reducing parallelization overheads

Observe that for each `parallel` region the overhead of creating the team of threads has to be paid. Rewrite the pragmas in your program in order to reduce this overhead, minimising the number of `parallel` constructs you use. Recompile and execute, obtaining the new values for ϕ, S_8, *OpenMP* state profile and the new strong scalability plot.

3.4.4 Reducing work–distribution overheads

What would happen if we move the `omp for` pragmas in the current version of your program one level up, i.e. instead of applying to the `j` loop they apply to the `k` loop. Rewrite the pragmas in your program, recompile and execute, obtaining the new values for ϕ, S_8, *OpenMP* state profile and the new strong scalability plot. Why the performance is improving?

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 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.

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 different node types available:

	`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)			

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

Briefly explain what strong and weak scalability refer to. Exemplify your explanation using the execution time and speed–up plots that you obtained for `pi_omp.c` on the different node types available in `boada`. Reason about the results that are obtained.

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 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/or 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 that help the reader to understand why v5 is able to scale to a higher number of processors compared to v4, capturing the task dependence graphs that are obtained with *Tareador*.

Understanding the parallel execution of *3DFFT*

In this final section of your report you should comment about the actual parallel performance of *3DFFT* when parallelised using *OpenMP*. Try to make a coherent history that shows how you optimised the code with the aim of increasing the parallel fraction of the program, reducing parallelisation overheads and improve load balancing. Accompany your explanations with the results reported in the following table which you obtained during the laboratory session. It is very important that you include the relevant *Paraver* captures (timelines and profiles of the % of time spent in the different *OpenMP* states) to support your explanations too.

Version	ϕ	S_∞	T_1	T_8	S_8
initial					
improved ϕ					
improved parallel overheads					
improved work–distribution overheads					

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