PAR Laboratory Assignment
Lab 1: Experimental setup and tools

Mario Acosta, Eduard Ayguadé, Rosa M. Badia (Q2), Jesús Labarta,
Josep Ramón Herrero, Daniel Jiménez-González, Pedro J. Martínez-Ferrer (Q2),
Jordi Tubella, and Gladys Utrera

Fall 2024–2025

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Contents

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

Objectives

At the end of this laboratory you should be able to:

1. Discover the architecture of a node in our cluster environment and understand its main parameters.
2. Compile and execute sequential and OpenMP parallel codes in our cluster environment.
3. Use Tareador, an automatic tool to build the task dependency graph (TDG) for a parallel decomposition strategy.
4. Use Modelfactors, an automatic tool that reports the overall performance analysis for your parallel implementation in OpenMP.
5. Use Paraver, a graphical tool to perform a detailed analysis of the execution traces generated for your parallel OpenMP implementation.
6. Apply a parallelisation/optimisation methodology based on the use of the aforementioned tools:
    (a) Define a parallelisation strategy and analyse its potential parallelism using Tareador.
    (b) Once you have found the appropriate strategy you will:
        i. Implement an OpenMP parallel version of your code following the parallelization strategy discovered with Tareador.
        ii. Perform the overall scalability analysis of your parallel implementation, looking at the parallel fraction (*phi*), load balancing and parallelisation overheads, using Modelfactors.
        iii. Perform a detailed analysis of some execution traces using Paraver to better understand the factors that limit the performance/scalability of your parallel implementation.
        iv. Modify the parallel implementation if necessary and repeat the steps above.

Chapter 1

Experimental setup

The objective of this laboratory session is to familiarise yourself with the hardware and software environments that you will be using during this semester to carry out all the laboratory assignments in PAR.

1.1 Accessing the boada cluster

From a local Linux terminal you can access boada, a multiprocessor server located at the Departament d'Arquitectura de Computadors (DAC). However, to be able to access the boada server from a computer outside the UPC you will need to use a UPC VPN connection. To connect to boada you need to establish a connection using the secure shell (SSH) command:

```
ssh -X par????@boada.ac.upc.edu
```

being `????` the user number assigned to you. The option `-X` is necessary to forward the X11 commands opening remote graphical windows in your local desktop.

We recommend you to change the password of your account by typing:

```
ssh -t par????@boada.ac.upc.edu passwd
```

After entering the old password correctly twice, you will be asked to enter a new password also twice. Annex A explains how to access boada from your personal computer.

Once you log in to boada, you will find yourself in any of the interactive nodes, i.e. `boada-6` to `boada-8`, where you can execute *interactive* jobs and from where you can submit *execution* jobs to other nodes of the machine. In fact, boada is composed of several nodes (named `boada-1` to `boada-15`), equipped with different processor generations, as shown in Table 1.1:

Table 1.1: Nodes of the boada cluster.

Node name	Processor generation	Interactive	Partition
`boada-1 to 4`	Intel Xeon E5645	No	execution2
`boada-6 to 8`	Intel Xeon E5-2609 v4	Yes	interactive
`boada-9`	Intel Xeon E5-1620 v4 + Nvidia K40c	No	CUDA9
`boada-10`	Intel Xeon Silver 4314 + 4x Nvidia GeForce RTX 3080	No	CUDA
`boada-11 to 14`	Intel Xeon Silver 4210R	No	execution
`boada-15`	Intel Xeon Silver 4210R + ASUS AI CRL-G116U-P3DF	No	iacard

In this course you are going to use nodes `boada-6` to `boada-8` interactively and nodes `boada-11` to `boada-14` through the execution partition, as explained in subsequent sections.

All the nodes have access to a shared network attached storage (NAS) disk. You can access it through

```
cd /scratch/nas/1/par????
```

which corresponds to your *home* directory (also referred to as ~/).

All the necessary files to carry out each laboratory assignment will be posted in

```
/scratch/nas/1/par0/sessions
```

For today's session, copy the file `lab1.tar.gz` from that location into your home directory:

```
cp /scratch/nas/1/par0/sessions/lab1.tar.gz ~/
```

and, from your home directory, uncompress it with this command line:

```
tar -zxvf lab1.tar.gz
```

Finally, in order to set up all the environment variables, you have to process the `environment.bash` file now available in your home directory with:

```
source ~/environment.bash
```

Note that you must load this environment *every time* you log in to boada. Alternatively, you can add this command into your `.bashrc` so that the environment will be automatically load every time you access the cluster. You can achieve this by executing the following command from your home directory:

```
echo ''source ~/environment.bash'' >> .bashrc
```

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 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.

In some cases, you will need to add -O option to be able to run scp:

```
scp -O parXXYY@boada.ac.upc.edu:lab1/pi/pi_seq.c .
```

due to diferent versions of the `scp` protocol.

1.2 Node architecture and memory

We propose you to investigate the architecture of the available nodes in boada. Execute the following command:

```
sbatch submit-arch.sh
```

within the `lab1/arch` directory.

The `sbatch` command above enqueues the `submit-arch.sh` script that executes the `lscpu` and `lstopo` commands to retrieve information about the hardware in one of the `execution` nodes (i.e., from `boada-11` to `boada-14`), thereby generating three files:

- `lscpu-boada-??`,
- `lstopo-boada-??`,
- `map-boada-??.fig`,

where `??` may be 11, 12, 13, or 14, referring to one of the execution nodes of boada. You can execute the command

```
xfig map-boada-??.fig
```

to visualise the output file and export to a different format (PDF or JPG, for example) using `File` → `Export` in order to keep this information. 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".

Alternatively, remember that you can also take screenshots by pressing the dedicated Prtsc keyboard button.

The three generated files will help you to figure out:

- the number of sockets, cores per socket, and threads per core in a specific node;
- the amount of main memory in a specific node, and each NUMA node;
- the cache memory hierarchy (i.e., L1, L2, and L3), private or shared to each core/socket.

We suggest you to fill in Table 1.2 that will help you understand some performance degradation that may occur when increasing the number of threads in future exercises.

Table 1.2: Architecture and memory hierarchy characteristics of any of the boada execution nodes boada-11 to boada-14.

Characteristic	Value
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)	

1.3 Execution modes: interactive vs. queued

There are two ways to execute your programs in boada:

Interactively. Using any of the login nodes, from boada-6 to boada-8, all of them limited to a maximum of 2 cores for parallel executions.

Via a queuing system. Using one of the execution nodes, from boada-11 to boada-14.

Interactive executions start immediately and do share resources with interative programs run by other users logged in to the system, thereby not ensuring representative timing results. In addition to this, the login nodes have a different architecture and memory hierarchy than the execution nodes. The corresponding *interactive* scripts are executed with the command (where ??? refers to a name):

```
./run-???.sh
```

You must use the second option for running programs that require several processors in a node, ensuring that your job is executed in isolation and therefore reporting reliable performance results. These executions do not overload the interactive/login nodes. Note also that *queued* scripts start as soon as an execution node is available. The corresponding *execution* scripts must be submitted as a SLURM job:

```
sbatch ./submit-???.sh
```

After submitting your job, you can run the command

```
squeue
```

to ask the system about the job status within the global queue (together with other users' jobs). You can check only the jobs associated with your account by typing the -u flag and your user identifier:

```
squeue -u par????
```

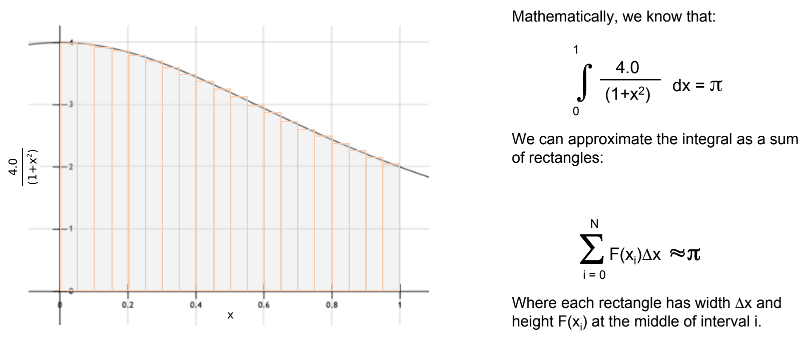
Finally, you can cancel a previously submitted job via

```
scancel <jobid>
```

where `<jobid>` refers to the job identifier. Every submitted execution generates two additional files that share the same script name followed by `.e` and `.o` together with the job identifier. Such files contain the messages sent to the standard error and output streams, respectively, which we encourage you to check.

1.4 Serial compilation and execution

The `pi_seq.c` code inside the `lab1/pi` directory computes the value of pi (π) by numerically integrating the equation $1/(1+x^2)$ within the interval $[0, 1]$. The *exact* integral equals $\pi/4$ and hence we use $4/(1+x^2)$ as depicted in Fig. 1.1. The integral is numerically *approximated* by the sum of N rectangles, being more accurate as this number increases.



Mathematically, we know that:

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

We can approximate the integral as a sum of rectangles:

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

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

Figure 1.1: Computation of pi (π).

Code 1 shows a simplified version of the `pi_seq.c` code. The variable `num_steps` defines the number of rectangles, whose area is computed at each iteration of the loop.

```c
static long int num_steps = 100000;

void main () {
  double x, pi, 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;
}
```

Code 1: Computation of π.

Figure 1.2 shows the compilation and execution flow for a sequential program. Program compilation and binary generation is done with a `Makefile` that declares multiple targets with specific rules. This course uses `icx` (Intel's oneAPI C compiler) to generate the binary files.

Now lets compile `pi_seq.c` using the `Makefile` and execute the binary generated interactively and through the queueing system:

1. Open the `Makefile` file and identify the `target` employed to compile the sequential code. Observe how the compiler is invoked.

2. Interactively execute the generated binary file using the command:

   ```
   ./run-seq.sh <name> <iterations>
   ```

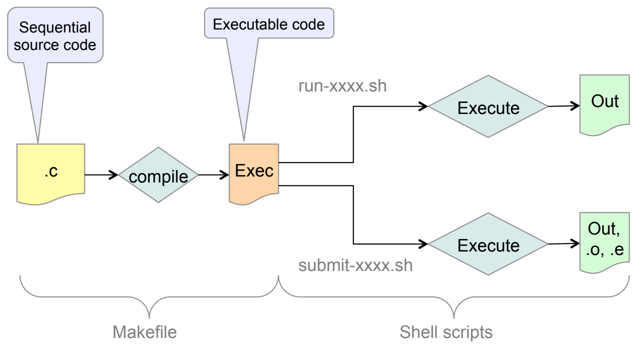


Figure 1.2: Compilation and execution flow for sequential programs.

with the corresponding executable name and a fixed number of iterations equal to 10^9. The execution returns the user and system CPU time, the elapsed execution time, and the percentage of CPU used.

3. Look at the source code and identify the function invocations and data structures used to measure the execution time.

4. Submit the execution script to the queuing system via:

   ```
   sbatch submit-seq.sh <name> <iterations>
   ```

   with similar arguments: the binary name and 10^9 iterations. Use the `squeue` command to check your job.

5. Open the generated standard output and error files `time-pi_seq-boada-??`, where `??` refers to the node where the execution took place.

Take a closer look at the `run-seq.sh` and `submit-seq.sh` scripts to understand how the binaries are executed.

1.5 Compilation and execution of OpenMP programs

OpenMP is the standard model for parallel programming using shared-memory. It allows to express parallelism in the C programming language. This section explains how to compile and execute parallel programs with OpenMP.

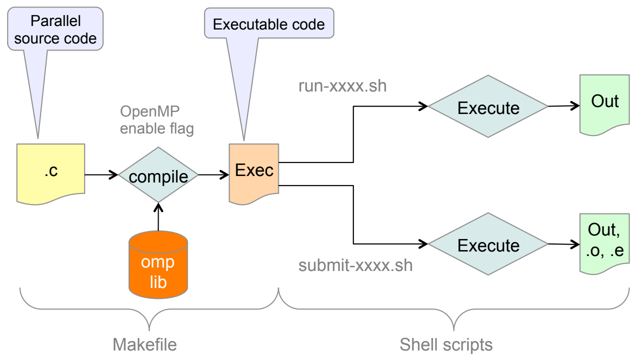


Figure 1.3: Compilation and execution flow for parallel programs using OpenMP.

Figure 1.3 shows the compilation and execution flow for an OpenMP program. The main difference with respect to Fig. 1.2 is that now the `Makefile` includes the appropriate compilation flag that enables OpenMP.

We propose you to carry out the following steps:

1. Open the parallelised code `pi_omp.c` and figure out what the new added lines of code do.

2. Open the `Makefile` and identify the `target` associated with the OpenMP code compilation (the only difference is the `-qopenmp` compilation flag).
3. Take a look at the script to learn how to specify the number of threads to OpenMP.
4. Interactively execute the OpenMP code with 1, 2, 4, 8, 16, and 20 threads using 10^9 iterations:

   ```
   ./run-omp.sh <name> <iterations> <threads>
   ```

5. What is the `time` command telling you about the user and system CPU time, the elapsed time, and the percentage of CPU used?
6. Submit the execution script to the queuing system via:

   ```
   sbatch submit-omp.sh <name> <iterations> <threads>
   ```

   specifying the name of OpenMP binary, 10^9 iterations, and an increasing number of threads: 1, 2, 4, 8, 16, and 20. Look at all the generated files `time-pi_omp-¿¿-boada-??`, being `¿¿` the number of threads and `??` the execution node.

We suggest you to fill in Table 1.3 to compare the differences between interactive and queued executions.

Table 1.3: Interactive vs. queued execution times.

Number	Interactive timing information				Queued timing information			
of threads	user	system	elapsed	% CPU	user	system	elapsed	% CPU
1								
2								
4								
8								
16								
20								

1.6 Strong vs. weak scalability

This section explores the *scalability* of the parallel version implemented in `pi_seq.c` for an increasing number of threads. It is measured as the ratio between the sequential and the parallel execution times. This ratio is often referred to as the *speedup*.

Two scalability scenarios are considered:

Strong. The number of threads increases while the problem size is kept fixed; that is, the parallelisation is employed to reduce the execution time of the program.

Weak. The problem size is proportional to the number of threads; that is, the parallelisation is employed to increase the problem size for which de program is executed.

We provide two scripts that analyse the scalability of `pi_seq.c`. They execute the parallel code using `np_NMIN=1` to `np_NMAX=20` threads. The problem size is set to 10^9 and 10^8 for strong and weak scalability scenarios, respectively. For the latter, the problem size grows proportionally to the number of threads, as previously explained. Finally, each script generates a plot showing the parallel execution time and the speedup.

We ask you to follow the next steps:

1. Submit the strong scalability script (no arguments are required):

   ```
   sbatch submit-strong-omp.sh
   ```

   and watch the state of your submission via the `squeue` command.
2. Convert the generated Postcript file to PDF via the `ps2pdf` command and visualise it with the `xpdf` command (replace `???` by the corresponding name):

```
ps2pdf pi_omp-???.ps
xpdf pi_omp-???.pdf
```

and observe how the execution time and speedup varies with the number of threads.

3. Set `np_NMAX=40` within `submit-strong-omp.sh` and repeat the previous steps. Can you explain the behaviour observed in the scalability plot?

4. Submit the weak scalability script (no arguments required):

   ```
   sbatch submit-weak-omp.sh
   ```

   and observe how the parallel efficiency varies with the number of threads.

Remember to take screenshots by pressing the dedicated `Prtsc` keyboard button.

1.7 Discussion at the laboratory

These questions are to be discussed during or at the end of the laboratory with your professor:

Do you observe any major differences between *interactive* and *queued* executions in Table 1.3?

Look at the *weak* and *strong* scalability plots generated and reason about the obtained performances.

Chapter 2

Systematically analysing task decompositions with Tareador

In this laboratory session, you will learn how to use Tareador, a tool that analyses the potential parallelism that can be obtained when a given *task decomposition* strategy is applied to your sequential code. This way the programmer simply needs to identify which are the tasks in that strategy that wants to evaluate.

These are the main features of Tareador and Tareador GUI:

- Tareador traces the execution of the program based on the specification of potential tasks to be run in parallel;
- Tareador records all static/dynamic data allocations and memory accesses in order to build the task dependency graph (TDG);
- Tareador GUI uses Tareador output information to simulate the parallel execution of the tasks on a certain number of processors in order to estimate the potential speedup.

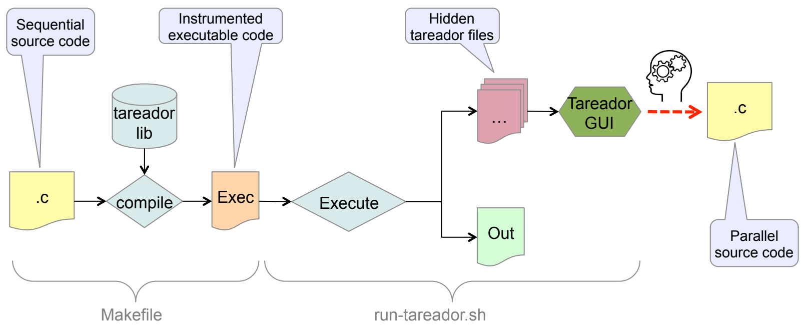


Figure 2.1: Compilation and execution flow for Tareador.

Figure 2.1 shows the compilation and execution flow for Tareador, starting from the taskified source code.

2.1 Tareador API

The Tareador application programmer interface (API) allows 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");
```

where 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();
/* Main program */
tareador_OFF();
```

at the beginning and at the end of the program, respectively.

From now onwards you will use a program that computes the fast Fourier transform (FFT) of an input dataset in three directions (x, y, and z), producing an output dataset that can be validated for correctness. We ask you to follow the next steps:

1. Go into the `lab1/3dfft` directory, open the `3dfft_tar.c` source code and identify the calls to the Tareador API; try to understand the tasks that have been initially defined.

2. Open the `Makefile` to understand how the source code is compiled and linked to produce the executable.

3. Generate the executable by running:

   ```
   make 3dfft_tar
   ```

4. Execute the binary generated with:

   ```
   ./run-tareador.sh 3dfft_tar
   ```

   Note that, due to the instrumentation overhead, the execution time may be several orders of magnitude higher than that of the original sequential code.

5. A window with a warning message will appear, just click on the `Ok` button to continue with the instrumented execution.

2.2 Brief Tareador hands-on

Now you will go through some of the different options that Tareador offers to analyse the potential of task decomposition strategies.

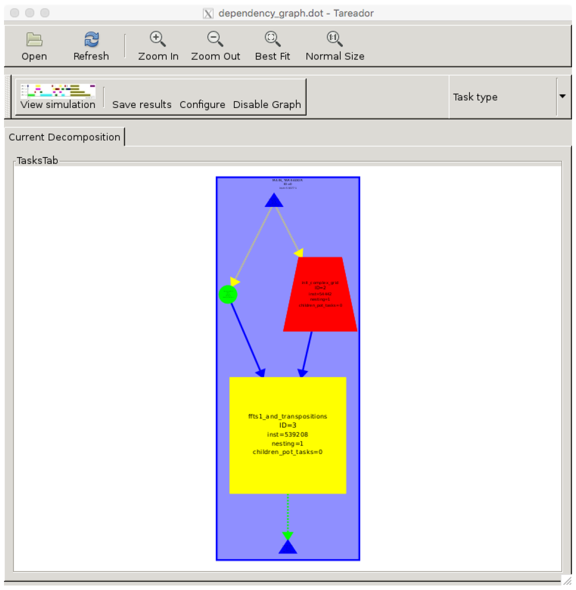
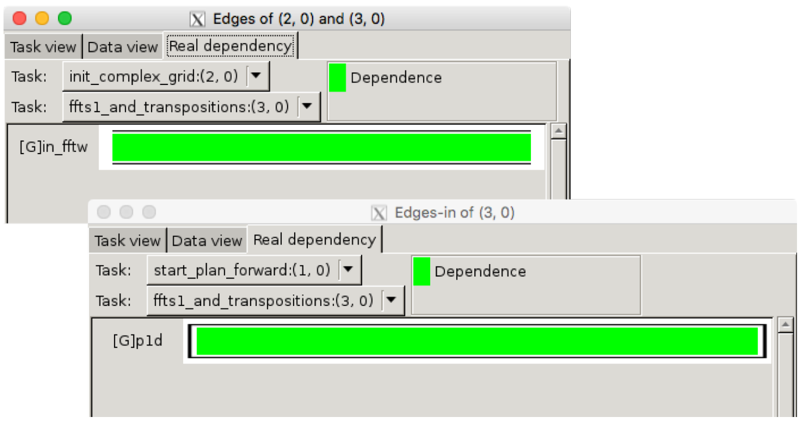


Figure 2.2: Task dependency graph (TDG) for the initial task decomposition expressed in `3dftt_tar.c`.

The execution of the `run_tareador.sh` script opens a new window in which the TDG 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;
- 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 granularity.

You can 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.



Task: init_complex_grid:(2, 0)
Task: ffts1_and_transpositions:(3, 0)
🟩 Dependence
[G]in_fftw

Task: start_plan_forward:(1, 0)
Task: ffts1_and_transpositions:(3, 0)
🟩 Dependence
[G]p1d

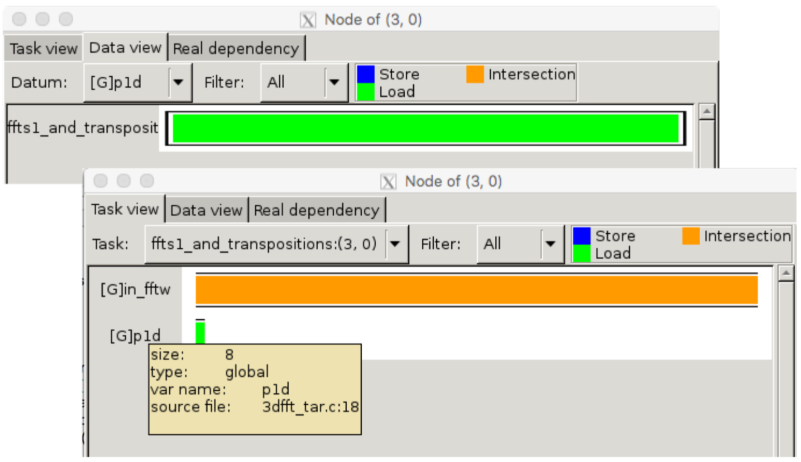
Figure 2.3: Variables provoking data dependencies between tasks for a specific or all edges: real dependency tab.

Edges in the graph represent dependencies between task instances. Different colours/patterns are used to represent different types of dependencies: e.g., blue colour for *data* dependencies and green/yellow for *control* dependencies. Moreover, Tareador allows you to analyse the variables whose access provokes data dependencies between a pair of nodes. Try the following:

1. Move the mouse over an edge. e.g., the edge going from the red task (`init_complex_grid`) to the yellow task (`ffts1_and_transpositions`).
2. Right click with the mouse and select `Dataview` → `edge`. This will open a window similar to the one shown in Fig. 2.3.
3. In the `Real dependency` tab, you can see the variable that causes that dependency (the access to the array `in_fftw`).
4. Alternatively, right click on a task (e.g., `ffts1_and_transpositions`) and select `Dataview` → `Edges-in`. This will open a window similar to the previous one.
5. You can do the same for the edges going out of a task by selecting `Dataview` → `Edges-out` when clicking on top of a task.

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`, carry out the following steps:

1. 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 Fig. 2.4.
2. In the `Task view` tab you can see the variables that are:
   - read (i.e., with a `load` memory access) in green colour in the window, as in this case variable `p1d`;
   - written (i.e., with a `store` memory access) in blue colour in the window;
   - both read and written (i.e., `intersection`) in orange, as it is the case for `in_fftw`.
3. For each variable in the list you have its name and its storage:



```
size:        8
type:        global
var name:    p1d
source file: 3dfft_tar.c:18
```

Figure 2.4: Variables provoking data dependencies between tasks for a specific or all edges: dataview and taskview tabs.

- `G` for global;
- `H` for the heap or dynamically allocated data;
- `S` for the stack or function local variables;

and additional information (size and allocation) is obtained by placing the mouse on the name.

4. 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.
5. Finally, you can save the TDG by clicking on the `Save results` button in the main Tareador window.
6. Alternatively, you can simply take a screenshot by pressing the `Prtsc` keyboard button.

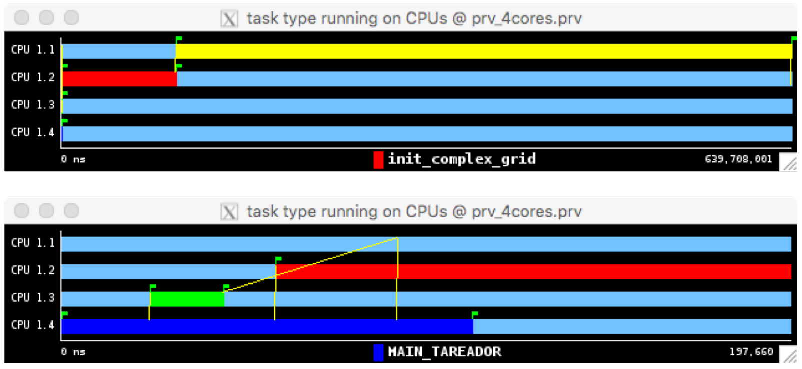


Figure 2.5: Paraver trace of the simulated execution with 4 processors, full view and after zooming into the initial part of the trace.

You can also simulate the execution of the task graph in an *ideal* machine with a certain number of processors by clicking on `View Simulation` in the main window of Tareador. This will open a Paraver window similar to Fig. 2.5 showing the timeline for the simulated execution with the following characteristics:

- each horizontal line shows the task(s) executed by each processor (`CPU1.x`, with `x={1,2,3,4}`);
- colours represent different tasks and match those of the TDG;
- the number on the lower-right corner of the window indicates the simulated execution time (in time units, assuming that each instruction takes one time unit) for the parallel execution;
- yellow lines show task dependencies (and task creations).

You can zoom in the *initial* part of the timeline by pressing the left button of your mouse and selecting the zone you want to zoom. You can undo the zooms done by clicking `Undo zoom` or `Fit time scale` on top of the timeline windows of Paraver. You can save the timeline for the simulated parallel execution by clicking `Save` → `Save image` on top of the timeline Paraver window; alternatively, you can make a screenshot.

2.3 Disabling Tareador objects

Although not useful for this code, you can disable the analysis of certain variables in the program using the following functions of 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 dependencies caused by the selected variable in a specific code region. For example, if you disable the analysis of the variable `in_fftw`, then you will observe that in the new TDG the tasks `init_complex_grid` and `ffts1_and_transpositions` are executed in parallel.

2.4 Exploring new task decompositions

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 decomposition (named v1, v2, v3, v4 and v5) as described in the following bullets. For each tasks 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. Note that T_∞ can be approximated with a simulation using the maximum number of processors. We advise you to save and compare all the TDGs and fill in Table 2.1.

Now it is time to *refine* the original tasks defined in the FFT benchmark with the objective of exploiting more parallelism. You will incrementally generate five additional fine-grained task decompositions (v1, v2, v3, v4, and v5) as described below:

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

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 j, k;
  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 dependencies that appear in the TDG: analyze the *Edges-in* for one of the transposition tasks and make sure that you understand what is reported by Tareador.

[1]Herein the word "replace" actually means: firstly, remove the *original* task definitions; and secondly, add the *new* task definitions.

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 v2 for `ffts1_planes`. Try to understand what is causing data dependencies.

v4. Starting from v3, replace the definition of the task for the `init_complex_grid` function with fine-grained tasks inside the body function. For this version, 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 v4?

v5. Create a new version to explore an even finer granularity. Take into consideration that, due to the large number of tasks, Tareador may take a while to compute and draw the TDG. Once more, simulate the parallel execution for 1, 2, 4, 8, 16, 32, and 128 (i.e., ∞) processors. According to the new results, is it worth going to this granularity level?

Table 2.1: Parallelism for each version of `3dfft_tar.c`.

Version	T_1	T_∞	Parallelism ($T_1 \div T_\infty$)
seq			
v1			
v2			
v3			
v4			
v5			

2.5 Discussion at the laboratory

These questions are to be discussed during and/or at the end of your laboratory session with your professor:

- What is the worst and best parallel strategy according to Table 2.1?
- Which are the main differences between them?
- In a real-world scenario, is the *best strategy* the one that you would have expected to have the *best performance*?

Chapter 3

Understanding the execution of OpenMP programs

This chapter presents a methodology to analyze and improve the performance of your parallel implementation with OpenMP. It is based upon the following three steps that you can repeat until being satisfied with the performance:

1. Use of Modelfactors in order to analyse the overall performance and scalability, the actual parallel fraction (ϕ) of the task implementation, and its efficiency based on load balancing and overheads.
2. Use of Paraver in order to analyse the traces generated from the parallel execution that will help you to diagnose the performance inefficiencies previously reported by Modelfactors.
3. If necessary, modification of the current parallel implementation based on the previous diagnosis and repetition of the process.

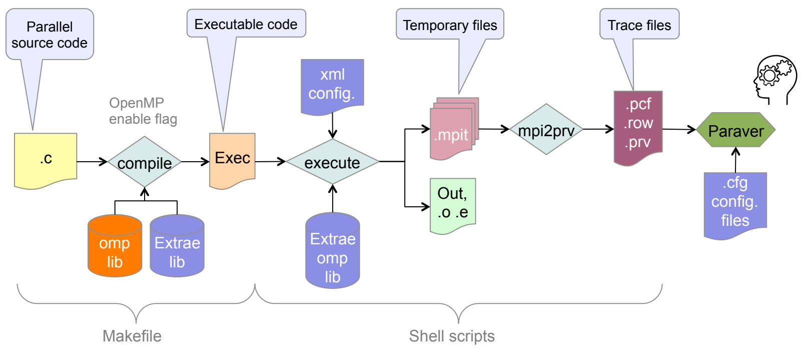


Figure 3.1: Compilation and execution flow for obtaining traces.

The steps described above are based on the generation of traces from the instrumented execution of a parallel program. Figure 3.1 shows the complete compilation and execution flow that needs to be taken in order to generate such traces. The environment is mainly composed of:

Extrae. It transparently instruments the execution of OpenMP binaries, collecting information about the status of each thread and different events related with the execution of the parallel program.[1] After the program execution, a trace composed of three files (with extensions `.row`, `.pcf`, and more importantly `.prv`) is generated containing all the information collected at execution time.

Paraver. It visualises the trace and analyses the execution of the program.

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

When using Modelfactors, the execution of the Extrae instrumented program is repeated for an increasing number of processors, generating traces for each of them. Next, Modelfactors analyses all of them in order to generate the reports.

3.1 Discovering Modelfactors: overall analysis

`mfLite.py` is a Python program that invokes the analysis with *modelfactors*, generating and analysing a collection of execution traces generated by *Extrae* for different number of processors. In order to generate the collection of execution traces we provide you with a script (`submit-strong-extrae.sh`), which also runs `mfLite.py` to generate the strong scaling analysis of your parallel implementation. More information about `mfLite.py` can be found in Annex B.1.

We will firstly focus on the information reported in the generated text file `modelfactors.out`. This output consists of three different tables. Each row starts with a short description of its content. From left to right, the correponding values for each metric are shown for an increasing number of threads/processors.

Another script that executes in the background generates a PDF, `modelfactor-tables.pdf`, gathering the aforementioned three tables and their captions. You should include both the tables and their captions in all your reports.

3.1.1 Modelfactors tables

Table 3.1: Analysis done on Thu Jul 28 07:54:35 AM CEST 2024, par0.

Overview of whole program execution metrics			
Number of Processors	1	2	4
Elapsed time (sec)	1.27	0.72	0.41
Speedup	1.00	1.76	3.11
Efficiency	1.00	0.88	0.78

Each table targets a different metric. Starting with the first one (see Table 3.1 obtained from a parallel program executed on up to 4 processors), Modelfactors provides a summary of the *overall* program execution metrics for an increasing number of p threads:

- elapsed parallel execution time T_p, measured in seconds;
- speedup, calculated as $S_p = T_1 \div T_p$;
- efficiency, calculated as $E_p = S_p \div p$.

With this information you can figure out the scalability of your code. If the reported results seem unsatisfactory, you can go to the next table to better understand the reasons.

Table 3.2: Analysis done on Thu Jul 28 07:54:35 AM CEST 2024, par0.

Overview of the efficiency metrics in parallel fraction, $\phi = 99.94$			
Global efficiency	**99.97%**	**87.95%**	**77.95%**
Parallelisation strategy efficiency	**99.97%**	**99.50%**	**95.91%**
Load balancing	100.00%	99.60%	96.60%
In execution efficiency	99.97%	99.91%	99.28%
Scalability for computation tasks	**100.00%**	**88.38%**	**81.27%**
IPC scalability	100.00%	89.43%	85.12%
Instruction scalability	100.00%	100.00%	100.00%
Frequency scalability	100.00%	98.83%	95.48%

The second table of Modelfactors (e.g., Table 3.2) provides a deep dive into the efficiency metric, but only for the parallel fraction, denoted by ϕ, of the program, that is, the part of the program that it *currently* parallelised. This is a good metric to understand whether the inefficiencies come from a large serial part in your program that has not yet been parallelised or cannot be further optimised. Note that there are two metrics, namely the "parallelization strategy efficiency" and the "scalability for computation tasks", whose product form the "global efficiency" of the program.

The parallelisation strategy efficiency is calculated as the product of the "load balancing" and the "in execution efficiency". The former reflects the ratio between the *average* useful code executed by the threads and the *maximum* useful code executed by any of them, while the latter shows the *lack of* overheads due to work generation and synchronisation in the critical path of the execution. The higher and closer to 100% these two values are, the better. With these two metrics it can be often inferred what to do next with regard to the analysis and optimisation of the code.

The scalability for computation tasks is related to how the processors are actually executing the computation tasks, in terms of the total number of useful instructions executed, number of useful instructions executed per cycle (IPC) of operation, and the frequency (number of cycles useful of operation per second). As with the previous metric, it is calculated as the product of these three variables. The value of these metrics for p threads is relative to 1 thread and, once more, the closer to 100% these values are, the better. Note that lower values are to be expected for specific cases in which there are plenty of synchronization overheads.

Table 3.3: Analysis done on Thu Jul 28 07:54:35 AM CEST 2024, par0.

Statistics about explicit tasks in parallel fraction			
Number of explicit tasks executed (total)	16.0	32.0	64.0
LB (number of explicit tasks executed)	1.0	1.0	1.0
LB (time executing explicit tasks)	1.0	1.0	0.98
Time per explicit task (average us)	79070.36	44729.83	24320.53
Overhead per explicit task (synch %)	0.0	0.47	4.23
Overhead per explicit task (sched %)	0.02	0.02	0.02
Number of taskwait/taskgroup (total)	8.0	8.0	8.0

Finally, the third Modelfactors table (see Table 3.3) completes the report. If the code contains *explicit* tasks, then details about them will be provided in this table. Otherwise, information about *implicit* tasks will be reported. These metrics can be very helpful to detect issues related to overheads caused by synchronizations or an excessive number of tasks created. LB stands for load balancing.

You can find more details on how to calculate the metrics of Tables 3.2–3.3 in Annex B.2.

Example 3DFFT: Obtaining parallelisation metrics using Modelfactors

Overall Analysis

A programmer has provided an OpenMP implementation for the FFT code similar to version 3 that you explored in the previous laboratory session. We ask you to carry out the following steps:

1. Go into the `lab1/3dfft` directory.
2. Open the `3dfft_omp.c` file in that directory and look for the lines containing OpenMP pragmas:
   (a) With `#pragma omp parallel` (line 49) 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 `parallel` construct in a replicated way: *implicit* task.
   (b) With `#pragma omp single` (line 50) we are indicating 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. Note that the thread that entered the `single` region will encounter `#pragma omp taskloop` (line 53), indicating that:
      i. 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.
      ii. When the thread that entered the `single` region finishes with the generation of all the tasks in the `taskloop`, it will also participate in the execution of the explicit

tasks that it created.

iii. When all the explicit tasks are executed, the `parallel` region will be finished returning to a serial execution until a new parallel region is found, if any in the code. In your program, three parallel regions are initially defined.[2]

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

   ```
   make 3dfft_omp
   ```

4. Submit the execution of the `submit-strong-extrae.sh` script indicating the name of the binary program `3dfft_omp`:

   ```
   sbatch submit-strong-extrae 3dfft_omp
   ```

   This may take up to several minutes as the script traces the execution with 1, 4, 8, 12, 16, and 20 threads and also performs the analysis with Modelfactors.

5. Monitor the status of your submitted job in the queue:

   ```
   watch squeue
   ```

   to check whether it has finished (press `Ctrl-C` keys to stop the command).

6. After the completion of the job, check that the directory `3dftt_omp-strong-extrae` exists, and inside it, the `modelfactor-tables.pdf` or its equivalent text version `modelfactors.out`.

7. Open the PDF via the following command:

   ```
   xpdf modelfactor-tables.pdf
   ```

Before continuing, make sure to rename the `3dfft_omp-strong-extrae` directory before running a new Modelfactors analysis in order to avoid data deletion (since the same directory name will be reused). Furthermore, beware of the limited disk quota assigned to your account and consider deleting old traces once they are not longer necessary to prepare your laboratory deliverables.

3.1.2 Discussion at the laboratory

Based on the metrics reported by Modelfactors, it is time to think about the following questions:

- What is the information that you can see in the tables generated by Modelfactors? How can you use it?
- Is the scalability appropriate?
- Is the overhead due to synchronisation negligible?
- Does this overhead affect the execution time per explicit task?
- Which is the parallel fraction, i.e. value of ϕ, for this version of the program?
- Is the efficiency of the parallel regions appropriate?
- Which factor is negatively influencing the most?

3.2 Discovering Paraver: execution trace analysis

Table 3.4: Results of the Paraver execution analysis on three versions of the FFT benchmark.

Version	ϕ	ideal S_{12}	T_1	T_{12}	real S_{12}
Initial version in `3dfft_omp.c`					
New version with reduced parallelisation overheads					
Final version with improved ϕ					

[2]There is a fourth parallel region that will be activated later.

Let's dive into the parallel execution of the application by visualizing one of the execution traces generated by *modelfactors*. In this guided tour you will learn the basic features of *Paraver*, the graphical browser of the traces generated by *Extrae*. For your informantion, in Atenea (*Paraver* package section) we have prepared three packages (for Linux, Windows and Mac) of *Paraver* software so that you can locally install *Paraver* binaries on your computer to analyze the traces generated. This is usually faster than remotely open the GUI of *Paraver* when you are outside the UPC campus.

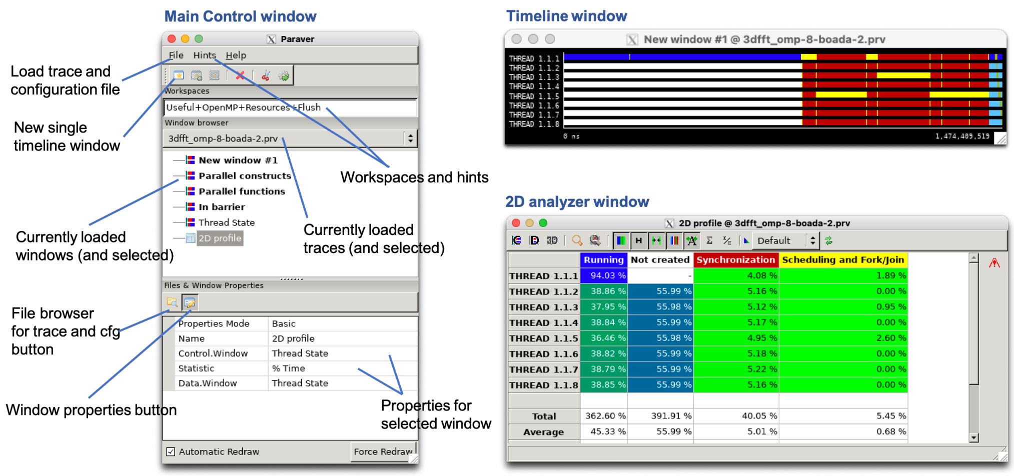
We suggest you to keep the information of parallel fraction value (found in the second table of Modelfactors) and capture the Paraver timeline(s) window(s) in order to make appropriate comparisons between the three proposed versions of the FFT benchmark:

- initial,
- reduced overhead,
- improved parallel fraction,

using the timeline window and profile of thread states. We will ask you to fill in Table 3.4.

3.2.1 Paraver timelines: navigation and basic concepts

We ask you to follow the next steps:



Properties Mode	Basic
Name	2D profile
Control.Window	Thread State
Statistic	% Time
Data.Window	Thread State

	Running	Not created	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	94.03 %	-	4.08 %	1.89 %
THREAD 1.1.2	38.86 %	55.99 %	5.16 %	0.00 %
THREAD 1.1.3	37.95 %	55.98 %	5.12 %	0.95 %
THREAD 1.1.4	38.84 %	55.99 %	5.17 %	0.00 %
THREAD 1.1.5	36.46 %	55.98 %	4.95 %	2.60 %
THREAD 1.1.6	38.82 %	55.99 %	5.18 %	0.00 %
THREAD 1.1.7	38.79 %	55.99 %	5.22 %	0.00 %
THREAD 1.1.8	38.85 %	55.99 %	5.16 %	0.00 %
Total	362.60 %	391.91 %	40.05 %	5.45 %
Average	45.33 %	55.99 %	5.01 %	0.68 %

Figure 3.2: Paraver's main control window, timeline, and 2D analyzer windows.

1. Launch Paraver by typing `wxparaver` on the command line, which will open the so-called "main control window", shown in Fig. 3.2 (left).
2. Load the trace corresponding to 12 processors from the main menu, by selecting `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 during the execution of the Modelfactors tool (they should be inside the directory `3dfft_omp-strong-extrae`).
3. Once the file is loaded, click on the box `New single timeline window` (top left icon in the main window).
4. A new window, similar to the one shown in Fig. 3.2 (top-right), appears showing a timeline where the horizontal axis represents time (advancing from left to right) and with the state (encoded in colour) of each thread (vertical axis) in the parallel program.
5. 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`),
- . . .,
- note that the meaning of each color is specific to each window.

6. Right-click at any point inside the window and select `Info Panel` and then the `Colors` tab to see the colouring table for the window.
7. 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).
8. With `Right Button` → `Info Panel` in the trace window you can decide to show or hide this panel.

It is interesting to see how the OpenMP parallel execution model is visualised:

- Firstly, a *master* thread executes 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).

Zooming in/out timelines

In order to go deep in the understanding of how a particular program is being executed in parallel, you need to learn how to zoom in the trace. Before that, it is worth mentioning that a *single* pixel of the window does represent a time *interval* where the thread may change its state multiple times. Since Paraver can only use a single colour to paint each pixel, it has to choose one to summarise those states. Therefore, do not get fooled by the colours shown and do not extract premature conclusions such as:

> *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 seem to be running useful code.*

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:

1. 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.
2. `Undo Zoom` and `Redo Zoom` commands are available on the right button menu. You can do and undo several levels of zooming.
3. `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:

1. Zoom into the beginning of the first parallel region (yellow part that is setting up the threads and giving them work to do).

2. 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 indicating when tasks are being generated.
3. Play a little bit more zooming in and out to observe how threads transition between states.

3.2.2 Paraver profiles: summarising information in the trace

Paraver also offers hints to ease the analysis of the traces generated by Extrae. These hints are grouped in different categories and are available from the main menu. One of them is the hint `OpenMP/thread_state_profile`, which pops up a table similar to the one shown in Fig. 3.2 (bottom-right), with one row per thread and one column per thread state (`Running`, `Synchronization`, `Scheduling and Fork/Join`, ...).

Within the thread state profile, each cell value shows the percentage of time spent by a thread in a specific state. Observe that all threads, excluding the first one, spent some time in `Not created` state and that the percentage spent in `Synchronization` is significant compared to `Running`. The later means that the thread utilisation is low for this initial version of the code.

To see a different statistic in the histogram change the `Statistic` selector in the `Window properties` panel placed in the main window of Paraver. You can have a look at the following metrics:

Time. It shows the total time spent on each state, per thread.

# instances. It counts the number of times each state occurs.

Average duration. It shows the average time a thread is in each state.

3.2.3 Paraver flags

Flags provide information about the parallel execution and have two properties: type and value. The type is used to encode the kind of event while the value gives a specific value to the event:

1. Right-click with your mouse on the window, and select the `View` → `Event Flags` checkbox. Flags signal the entry and exit points of different OpenMP events (e.g., start/end of parallel region, function executed by *explicit* tasks, ...).
2. Click on one of the flags and enable the `Event` checkbox on the `What/where` tab of the `Info Panel`.
3. Sometimes you need to click several times to select the pixel where the flag is painted (or click the `Prev./Next` checkbox to show the information around the selected pixel).

Flags are also useful to differentiate different bursts in what may look like a simple burst in the timeline. Once more, for the purpose of this guided tour, we will be using Paraver hints to extract and process the information in these flags.

Example `3DFFT`: Obtaining further parallelisation details with Paraver

Detailed Analysis

For the trace you have loaded in the previous section:

1. Open the Paraver hint `User_funtions` which identifies when the different functions in the program are executed. Note that each function is represented with a different colour.
2. It is possible to align two different timelines by making them display the exact same threads and time span. To practise this, just take the initial state timeline window showing the parallel activity and the last window with the user functions. Right-click inside one of the two windows (the *source* or *reference* window) and select `Copy`;
3. Right-click on the target window and select `Paste` → `Default` (or, separately, `Paste` → `Size` and `Paste` → `Time`).

4. 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.
5. You can also synchronise different windows by adding them to the same `group`. For example, select `Synchronise` $\rightarrow$ `1` after a right-click with your mouse on the initial state timeline window showing the parallel activity.
6. Repeat this with the hint `User_functions` timeline window.
7. Once synchronised, they will continue aligned after zooming, undoing or redoing zoom. With the two windows synchronised it is easy to correlate the duration of tasks with the implicit task they belong to, allowing you to observe if parallel regions take the same time and if there is some work imbalance among threads in any of them.
8. You can always unsynchronise a window by unselecting `Synchronise` after a right-click with your mouse on the timeline window. Note that you can create new groups of synchronised windows at your convenience.
9. Open the hint `Useful/Instantaneous parallelism` to see the number of threads executing tasks in parallel. In order to figure out the existing parallelism at any moment of the execution trace, you can double-click with your mouse and the information panel will show you the instantaneous parallelism at that moment (a value of 1 means no parallelism at all).

There are two key factors that influence the overall scalability and final performance. Looking at the two timelines windows we can see that:

- there is one function that is not parallelised;
- there is a predominant thread state along the entire timeline window.

The second factor seems to have a greater influence because there is a strong scalability problem, that is, a slowdown from twelve or more threads, due to the number of tasks and synchronisations in the program.

3.2.4 Paraver hints for implicit and explicit tasks

To better understand from where this inefficiency comes from, we can use the following two Paraver hints to analyse the tasks that are created in our program:

1. Open the hint `OpenMP/implicit_tasks_duration` to show the duration of the implicit tasks executed by threads in the parallel regions.
2. A gradient coloured timeline shows the semantics of the window, painting each pixel in the window with a colour, from light green (low duration) to dark blue (high duration).
3. By moving the mouse arrow on top of the bursts Paraver will tell you their duration.
4. By clicking in one of the burst you can also get this information, but the gradient coloured window gives you a more global picture of the duration within the same parallel region and across different ones.
5. Now open the hint `OpenMP_tasking/explicit_tasks_duration` to visualise the gradient coloured timeline showing the duration of the tasks executed by the different threads.
6. You should be able to observe the task granularities generated in the different `taskloop` constructs and see, for example, if the explicit tasks generated from the same `taskloop` have the same duration or not.

We can also use the following two hints that produce histograms with the duration of the implicit and explicit tasks:

1. Open the Paraver hint `OpenMP/Histogram of Implicit task duration`.
2. A window pops up, showing with a colour gradient the distribution of the different implicit tasks duration. You can see the histogram of different implicit task executed for each thread.

3. Open the Paraver hint `OpenMP tasking/Histogram of explicit execution task duration`.
4. Another window pops up, showing with a colour gradient the distribution of the different explicit tasks duration. You can see the histogram of different explicit task executed for each thread.

There are other hints for both implicit and explicit tasks that you can see in Annexes C–D.

3.2.5 Paraver configuration files for implicit and explicit tasks

To facilitate the use of Paraver, you can employ the following configuration files that *automate* the process of opening several hints and synchronise different timelines:

1. Close and reopen Paraver and load the trace corresonding to 12 threads.
2. Click on `File` → `Load configuration...`
3. Go to the very top of the directory list, look for your lab directory, and click on `lab1/3dfft/`
4. Load one of the two configuration files avalaible (`*.cfg`) corresponding to either implicit or explicit tasks.
5. Verify that all the timelines are already synchronised.
6. Make sure that the timelines do cover the entire execution time by clicking on `Fit time scale`.

Note that a single configuration file can be used to automatically load several hints and synchronise the corresponding timelines and profiles. In this particular case, the following Paraver windows were loaded:

Thread states. You will see different states: (a) running, (b) not created, (c) schedule and fork/join, (d) synchronization, and (e) idle.

Parallel constructs. The parallel region limits are shown on thread 0.

Worksharing constructs. The single region is only shown on the thread executing the single region.

Implicit/Explicit task function creation (line and file). The thread executing the single region creates all the tasks in the taskloop. If you move the mouse over any of these regions, you will see the line and file where the taskloop (task) is created.

Implicit/Explicit task function execution (line and file). Any thread can execute a task created in the taskloop. If you move the mouse over such a region, you will see the first line and file where each task starts.

Implicit/Explicit tasks executed duration. If you move the mouse over any of these regions, you will see the execution time (granularity) of each task.

From the previous hints and configurations, and more specifically the histograms and timeline windows of the explicit task duration, it can be inferred that the *fine* granularity employed in this parallel implementation may not be adequate.

3.2.6 Final thoughts on Paraver

The information gathered in this section constitutes a *very short* introduction to Paraver and you will get to know better this visualisation aid along the next laboratory sessions.

Annexes C–D summarise the most important Paraver hints to be used with implicit and explicit tasks, respectively. We strongly recommend you to try them out *now* by using the trace corresponding to the initial version of the FFT problem executed with twelve threads.

3.3 Example `3DFFT`: Parallel Improvements and Analysis

3.3.1 Reducing the parallelisation overheads

Code Optimization

We ask you to coarsen the current task granularity of the FFT problem. This should reduce the parallelisation overheads and, as a consequence, have some positive impact in the overall performance. In order to achieve this:

1. Comment the innermost `taskloop` inside each parallel region.
2. Uncomment the outermost `taskloop` inside each parallel region.

Overall Analysis with *Modelfactor*

Now, it is good to see the impact of the new optimization in the overall performance.

1. Recompile and submit for execution the `submit-strong-extrae.sh` script, obtaining the new values for all the previous metrics using *modelfactors*.
2. Compare the *modelfactors* results to the previous ones. Can you see any improvement in the speedup? Is there still slowdown for 12 or more threads?

Now that the overhead problem seems to have been solved, why do you think you cannot achieve better speedup for twelve or more threads? Try to compute the maximum speedup using the *current* parallel fraction (from the second table generated by Modelfactors). Is this value limiting the maximum speedup that we can achieve?

Detailed analysis with *Paraver*

The analysis described below will help you to understand the overall improvement of the code:

1. Load the *newly* generated trace in Paraver (`3dfft_omp-12-boada-??-cutter.prv`), where ?? is the number of the node your program has been executed.
2. Click on the `New single timeline window` box (top left icon in the main window).
3. Load the trace corresponding to the *initial* version generated also with 12 threads. This trace should be located in another directory that you previously backup!
4. Make sure that both traces, i.e. initial and reduced overhead versions, use the same temporal scale.
5. Capture the Paraver timeline(s) window(s). The screenshot must show the comparison of the two versions, as well as the impact on the overall execution time. Remember to explain your conclusions briefly.
6. Obtain the histogram with the duration of different implicit and explicit tasks. You can either use the hints or the configuration files.
7. Open the hint `User_funtions` for the reduced overhead version in order to identify where and when the different functions in the program are being executed.
8. Remember to rename/backup the `3dfft_omp-strong-extrae` directory containing all the traces to avoid loosing data on future Modelfactors analysis.

3.3.2 Discussion at the laboratory

At this point, you should be able to answer each of the following questions:

- Do you think that using a very fine granurality is *always* the best parallel strategy? Why? Have another look at the third table generated by Modelfactors for the two versions.
- Was the synchronisation overhead constant or a function of the number of threads?
- How did you solve the overhead problem of the initial, fine-grained granularity strategy?

3.3.3 Improving ϕ

Code Optimization

Note that the function `init_complex_grid` has not yet been parallelised and its execution time is limiting the overall performance of the FFT program. We propose you another optimisation based on your previous analysis with both Modelfactors and Paraver:

1. Open the `3dfft_omp.c` file.
2. Uncomment the pragmas that will allow the parallel execution of the code inside function `init_complex_grid`.
3. At this point, only uncomment the outer loop `taskloop` together with `parallel` and `single`.

Overall Analysis with *Modelfactor*

Recompile and submit for execution the `submit-strong-extrae.sh` script, obtaining the new values for all the previous metrics using *modelfactors*.

Detailed analysis with *Paraver*

The final steps described below allow for a final comparison between the three parallel versions of the code:

1. Load the newly generated trace in Paraver (`3dfft_omp-12-boada-??-cutter.prv`), where ?? is the number of the node your program has been run.
2. Click on the `New single timeline window` box (top left icon in the main window).
3. Perform the same steps for the two previous traces (initial and reduced overhead versions) executed with 12 threads.
4. Capture the three traces simultaneously, for example, by placing the corresponding timelines on top of each other.
5. In order to see the net effect of the optimisations (i.e., the reduction of the execution time), click `Copy` in the timeline window of the initial version and `Paste` $\rightarrow$ `Time` in the timeline windows of the newer versions.
6. You can also open the hint `Useful/Instantaneous parallelism` to compare the behaviour of the three parallel implementations.

3.3.4 Discussion at the laboratory

At this point, you can answer the following questions:

- Once the overhead problem was solved, what was limiting the performance of the parallel strategy?
- Compare the parallel fraction of the three versions: initial, reduced overhead, and improved parallel fraction.

Appendix A

Accessing the boada cluster

You can access boada from your laptop, booted with Linux, macOS or Windows, if a secure shell client is installed. However, to be able to access the boada server from a computer outside the UPC you will need to use a UPC VPN connection.

On Windows you need 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 use *MobaXterm* which integrates both secure shell and X11:

    https://mobaxterm.mobatek.net/download.html

On macOS you need *XQuartz* and the option `-Y`:

```
ssh -Y par????@boada.ac.upc.edu
```

where `????` refers to your user number.

Appendix B

Modelfactors

B.1 Modelfactors script `mfLite.py`: extracting information

The script `mfLite.py` extracts information from Extrae traces:

Execution time . Also known as runtime and

In useful state . It is used to compute different duration metrics such as `useful_avg`, `useful_max`, and `useful_tot`.

On the other hand, two basic hardware counters, i.e. `PAPI_TOT_CYC` and `PAPI_TOT_INS`, are used to extract two raw metrics:

- `useful_ins`: number of instructions executed while in useful state and
- `useful_cycles`: number of execution cycles while in useful state.

This tool evaluates the metrics only *inside* parallel regions, what is called the parallel fraction (ϕ).

B.2 Computing the Modelfactors metrics for OpenMP

The metrics reported in the second and third tables of Modelfactors are computed only for the parallel fraction (ϕ) of the program, according to Amdahl's law. The parallel fraction may contain multiple parallel regions.

With p being the number of processors, the efficiency of the parallel fraction of a program is given by:

$$eff(p) = parallel\_eff(p) \times comp\_scal(p),$$

where

- $parallel\_eff(p)$ is the "parallelisation strategy efficiency" from the second table of Modelfactors and exposes the inefficiencies in the task decomposition strategy;
- $comp\_scal(p)$ is the "scalability for computation tasks" from the same table and characterises the inefficiencies associated with computation tasks.

The variable $parallel\_eff(p)$ is proportional to two factors:

- LB_eff, the "load balancing" from the second table of Modelfactors;
- in_exec_eff, the "in execution efficiency" from the same table;

where the first factor exposes the load balancing

$$LB\_eff = \frac{\text{avg}(useful_i)}{\max(useful_i)},$$

while the second factor highlights when threads spend time executing useful code

$$in\_exec\_eff = \frac{\max(useful_i)}{t},$$

being t the total execution time for the parallel fraction and $useful_i$ the time that thread i executes useful code. Finally, $parallel\_eff(p)$ can be computed as follows:

$$parallel\_eff(p) = \frac{\Sigma_{i=0}^{p-1} useful_i}{p \times t} = \frac{\frac{\Sigma_{i=0}^{p-1}(useful_i)}{p}}{\max(useful_i)} \frac{\max(useful_i)}{t}.$$

The scalability for computation tasks can also be expressed as $useful(1) \div useful(p)$, where $useful(p)$ is the total time all threads execute useful code in the parallel fraction, i.e. $\Sigma_{i=0}^{p-1} useful_i$. In particular, $useful(p)$ is equal to $inst(p) \times CPI(p) \div freq(p)$ where:

- $inst(p)$ is the total number of instructions executed in useful code;
- $CPI(p) = cycles(p) \div inst(p)$ is the average number of cycles needed to execute one instruction, being *cycles* the total number of cycles spent in useful code;
- $freq(p) = cycles(p) \div useful(p)$ is the average frequency of the processors while executing useful code.

In order words, we can compute the scalability for computation tasks as:

$$comp\_scal(p) = \frac{IPC(p)}{IPC(1)} \frac{inst(1)}{inst(p)} \frac{freq(p)}{freq(1)},$$

which corresponds to $IPC\_scal(p) \times inst\_scal(p) \times freq\_scal(p)$ and is precisely the product of the three rows ("IPC scalability", "instruction scalability" and "frequency scalability") of the second table of Modelfactors. Take into consideration that:

- an instruction scalability smaller than unity means that the parallel program needs more instructions than the sequential code;
- an *IPC* smaller than unity means that the *IPC* for p threads is worse than its sequential counterpart;
- a frequency scalability smaller than unity means that, when using p threads, the frequency decreases.

The third table of Modelfactors focuses on the information associated with explicit tasks. In particular, it exposes different factors that affect the overall load balance and overheads due to explicit tasks. The factors "LB (number of explicit tasks executed)" and "LB (time executing explicit tasks)" show the load imbalance and execution time related to the number of explicit tasks, respectively.

There should be a good correlation between the load balance information (execution time due to useful code) shown in the second table and the load balance information (explicit tasks) in the third table. On the other hand, the percentage of overhead due to the synchronisation and scheduling can be a good indicator of scalability problems of a given parallel strategy, usually related to the number of tasks and their granularity.

Appendix C

Visualising implicit tasks in Paraver

C.1 Hint `OpenMP/parallel construct`

It 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): two of these nine regions are *fake* parallel regions intentionally introduced to delimit the start and the end of the main program.

C.2 Hint `OpenMP/implicit tasks in parallel constructs`

It identifies the functions that encapsulate the code regions that each thread executes when a parallel region is found and 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 can check the line number in the original source code to see which one of the three parallel regions identified in the code is executed. Can we know how much time it takes for a thread to execute these implicit tasks?

C.3 Hint `OpenMP/implicit tasks duration`

It shows the duration of the implicit tasks executed by threads inside 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). Moving the mouse arrow on top of the bursts Paraver shows their duration interval. Clicking in one of the burst also reveals this information but the gradient coloured window gives a more *global* picture of the parallel region duration in the same parallel region and across different parallel regions.

C.4 Hint `#pragma omp single`

For each parallel region, it shows the thread that remains active as it enters the `single` region. For the remaining threads, there is a very short green bar at the beginning to check if the `single` region has already been taken by someone.

C.5 More hints

There are other hints in the Paraver menu gathered in Tables C.1–C.2 associated with timelines and histograms, respectively.

Table C.1: Paraver hints for implicit tasks that generate timelines.

Paraver hint	Timeline content
Parallel constructs	When a `parallel` construct is executed
Implicit tasks	The implicit task each thread executes in a `parallel` region
Implicit tasks duration	The duration of the implicit task executed in a parallel region
Worksharing constructs	When threads are in worksharings (in this course, only `single`)
In barrier syncrhonisation	When threads are in a `barrier` synchronisation
In critical syncrhonisation	When threads are in/out/entering/exiting `critical` synchronisations

Table C.2: Paraver hints for implicit tasks that generate histograms.

Paraver hint	Histogram content
Thread state profile	Thread states (useful, scheduling, synchronisation, ...)
In critical syncrhonisation profile	Different phases a thread goes on when synchronising in a `critical` construct
Implicit tasks profile	Implicit tasks
Histogram Implicit tasks duration	Durations of implicit tasks

Appendix D

Visualising explicit tasks in Paraver

Remember that these *explicit* tasks are created via the `task` or `taskloop` constructs present in a parallel program.

D.1 Hint `OpenMP tasking/explicit tasks task created and executed`

It visualises explicit tasks, and more specifically, when they are *created* and when they are *executed*.

Synchronise the two windows that just opened and do a bit of zooming in order to see the tasks in the different parallel regions (flags are here useful to delimit each tasks generated by the `taskloop` construct). Observe that each burst provides information about the line number in the source code where the tasks are defined. By opening the source code, you can verify that each task is associated to a particular `taskloop` construct.

D.2 Hint `OpenMP taskloop/in taskloop construction`

It visualises the `taskloop` constructs where the tasks belong to.

Synchronise the new window with the previous two windows in order to see when the thread executing the `taskloop` construct waits for the termination of all the tasks that have been generated. Note that, while this threads awaits, it also contributes with the execution of the generated tasks.

D.3 Hint `OpenMP tasking/explicit tasks duration`

It visualises the gradient coloured timeline showing the duration of the tasks executed by the different threads.

You should be able to identify the task granularities generated in the different `taskloop` constructs and observe, for instance, whether the explicit tasks generated from the same `taskloop` have all the same duration or not.

D.4 More hints

There are other hints in the Paraver menu gathered in Tables D.1–D.2 associated with timelines and histograms, respectively.

Table D.1: Paraver hints for explicit tasks that generate timelines.

Paraver hint	Timeline content
Tasking/Explicit tasks function created and executed	When explicit tasks are created and executed (file name and line number)
Tasking/Explicit tasks executed duration	Duration of explicit task functions executed
Taskloop/In taskloop constructs	When the `taskloop` construct is executed
Tasking/In taskwait constructs	When a `taskwait` construct is executed
Tasking/In taskgroup constructs	When a task is starting or waiting in a `taskgroup`

Table D.2: Paraver hints for explicit tasks that generate histograms.

Paraver hint	Histogram content
Tasking/Profile of explicit tasks creation and execution	Task creations and executions
Tasking/Histogram of explicit tasks execution duration	Durations of the explicit tasks executed