PAR Laboratory Assignment
Lab 1: Experimental setup and tools

Mario Acosta, Eduard Ayguadé, Rosa M. Badia (Q1),
Josep Ramon Herrero (Q1), Daniel Jiménez-González, Pedro Martínez-Ferrer, Adrian Munera,
Jordi Tubella and Gladys Utrera

Spring 2022-23

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

Index **1**

1 Objectives **3**

2 Experimental setup **4**
2.1 Node architecture and memory . . . 5
2.2 Execution modes: interactive vs queued . . . 6
2.3 Serial compilation and execution . . . 6
2.4 Compilation and execution of *OpenMP* programs . . . 7
2.5 Strong vs. weak scalability . . . 9
2.6 Discussion at laboratory . . . 9

3 Systematically analysing task decompositions with *Tareador* **10**
3.1 *Tareador* API . . . 10
3.2 Brief *Tareador* hands–on . . . 11
3.3 Exploring new task decompositions for `3DFFT` . . . 13
3.4 Discussion at laboratory . . . 14

4 Understanding the execution of *OpenMP* programs **15**
4.1 Discovering `modelfactors`: Overall analysis . . . 15
4.2 Discussion at laboratory: understanding modelfactors metrics . . . 18
4.3 Discovering *Paraver* (Part I): execution trace analysis . . . 18
4.3.1 Timelines: navigation and basic concepts . . . 19
4.3.2 Profiles: summarizing information in the trace . . . 20
4.3.3 Flags in the trace . . . 21
4.4 Discovering *Paraver* (Part II): understanding the parallel execution . . . 21
4.4.1 Implicit and Explicit tasks . . . 21
4.4.2 Example `3DFFT`: Reducing Parallelisation Overheads and Analysis . . . 22
4.4.3 Example `3DFFT`: Improving ϕ and Analysis . . . 23
4.5 Discussion at laboratory . . . 24

A Modelfactors **25**
A.1 What information does modelfactors.py use? . . . 25
A.2 Computing the metrics for OpenMP . . . 25

B *Paraver* **27**
B.1 Implicit tasks . . . 27
B.2 Explicit tasks . . . 27

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

1 Objectives

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

1. Know how to discover the architecture of a node in our cluster environment and understand the 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 Dependence 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 execution traces generated for your parallel implementation in OpenMP.
6. Apply a parallelization/optimization methodology based on the use of the above mentioned tools:
   (a) Define a parallelization strategy and analyze 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* tool 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.

2

Experimental setup

The objective of this laboratory session is to familiarise yourself with the hardware and software environment that you will use during this semester to do all laboratory assignments in PAR. From your local terminal booted with Linux[1] you will access `boada`, a multiprocessor server located at the Computer Architecture Department. To connect to it you will have to establish a connection using the secure shell command: `"ssh -X parXXYY@boada.ac.upc.edu"`, being `XXYY` the user number assigned to you. Option `-X` is necessary in order to forward the X11 commands necessary to open remote windows in your local desktop[2]. Once you have the account credentials, the first thing you should do is to change the password for your account using `"ssh -t parXXYY@boada.ac.upc.edu passwd"`[3].

Once you are logged in you will find yourself in any of the interactive nodes:`boada-6` to `boada-8`, where you can execute interactive jobs and from where you can submit execution jobs to the rest of the nodes in the machine. In fact, `boada` is composed of several nodes (named `boada-1` to `boada-15`), equipped with five different processor generations, as shown in the following table:

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 + 4 x 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

However, in this course you are going to use only nodes `boada-6` to `boada-8` interactively and nodes `boada-11` to `boada-14` through the `execution` queue, as explained in the next subsection. The rest of the nodes have restricted access and PAR users are not allowed to send jobs to their corresponding queues.

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

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

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

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

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

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 `.bashrc` file in your home directory[4], a file that is executed every time a new session is initiated.

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

2.1 Node architecture and memory

The first thing you will do is to investigate the architecture of the available nodes in `boada`. Run `sbatch submit-arch.sh` command in directory `lab1/arch`. This command will enqueue `submit-arch.sh` script. More detail about queue execution below. This script will execute the `lscpu` and `lstopo` commands in order to obtain information about the hardware in one of the nodes of `execution` queue (`boada-11 to 14`).

The execution of these two commands using `submit-arch.sh` script generates three files, where `number` may be 11, 12, 13 or 14: 1) `lscpu-boada-number`, 2) `lstopo-boada-number`, and `map-boada-number.fig`. You can use the `xfig` command to visualise the output file generated (`map-boada-number.fig`) and export to a different format (PDF or JPG, for example) using `File` → `Export` in order to keep this information[5].

Those 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 NUMAnode;
- the cache memory hierarchy (L1, L2 and L3), private or shared to each core/socket.

Based on the lscpu and lstopo results and .fig image, we suggest you to fill in the table 2.1 indicating which are 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 NUMAnode; and the cache memory hierarchy (L1, L2 and L3), private or shared to each core/socket. This information is really important to understand some performance degradation that may happen when increasing the number of threads.

	Any of the nodes among `boada-11 to boada-14`
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)	

Table 2.1: Summary of memory hierarchy and architecture node

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

[5]In the `boada` Linux distribution you can use `xpdf` to open pdf files and `display` to visualise graphics files. You can also use the "`fig2dev -L pdf map.fig map.pdf`" command to convert from `.fig` to `.pdf`; look for alternative output graphic languages by typing "`man fig2dev`".

2.2 Execution modes: interactive vs queued

There are two ways to execute your programs in boada:

1. via a queueing system (in one of the nodes `boada-11` to `boada-14`);
2. interactively (in any of the login nodes `boada-6` to `boada-8`). In this case, the system limits the number of cores to be used in parallel executions to two.

It is mandatory to use option 1 when you want to execute scripts that require several processors in a node, ensuring that your job is executed in isolation (and therefore reporting reliable performance results) and to avoid adding additional load to the interactive node accessed by all users; the execution starts as soon as a node is available. When using option 2 your execution starts immediately but will share resources with other programs and interactive jobs, not ensuring representative timing results. In addition, the login node has a different architecture and memory hierarchy!!!. Usually, scripts for both options (`submit-xxxx.sh` and `run-xxxx.sh`, respectively) will be provided:

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

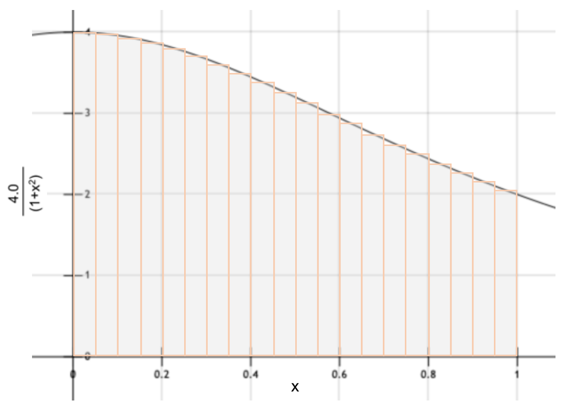
2.3 Serial compilation and execution

Next you will get familiar with the compilation and execution steps for both sequential and parallel applications. You are going to use a very simple code, `pi_seq.c`, which you can find inside the `lab1/pi` directory. `pi_seq.c` performs the computation of the number Pi (π) by computing an approximation of the integral in the interval [0,1] of the equation $1/(1+x^2)$ (the derivative of the arctangent funtion of x). In fact the integral equals $\pi/4$, for this reason the function used is $4/(1+x^2)$, as depicted in Figure 2.1. The computation of the integral is numerically done by computing the area under the function in the range above, which is approximated by the sum of a series of rectangles (N in Figure 2.1), being more accurate as N increases.
Figure 2.2 shows a simplified version of the code you have in `pi_seq.c`. The variable `num_steps` defines the number of rectangles, whose area is computed in each iteration of the `i` loop.

Figure 2.3 shows the compilation and execution flow for a sequential program. You will always compile programs to generate binary executable files through a `Makefile`, with multiple targets that specify the rules to compile each program version; the appropriate `Makefile` will be provided in each assignment. In this course `icc` (the C front–end from the *Intel Compilers* collection) will be used to generate your binary files; you can type `"icc -v"` to know about which specific version of the compiler you are using.

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

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



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 2.1: Pi computation

```
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 2.2: Serial code for Pi computation.

2. Interactively execute the binary file generated to compute the number Pi using the `run-seq.sh` script with the appropriate arguments (executable name and number of iterations 1.000.000.000). The execution returns the user and system CPU time, the elapsed time, and the % of CPU used (using GNU `/usr/bin/time`). In addition, the program itself also reports the elapsed execution time using `gettimeofday`. Look at the source code and identify the function invocations and data structures required to measure execution time. Please also take a look at the `run-seq.sh` script to understand how the binary file is executed.

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

2.4 Compilation and execution of *OpenMP* programs

In this course we are going to use *OpenMP*, the standard for parallel programming using shared-memory, to express parallelism in the C programming language. Although *OpenMP* will be explained in more detail after this first laboratory assignment, in this section we will see how to compile and execute parallel programs in *OpenMP*. Figure 2.4 shows the compilation and execution flow for an OpenMP program. The main difference with the flow shown in Figure 2.3 is that now the `Makefile` will include the appropriate compilation flag to enable *OpenMP*.

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

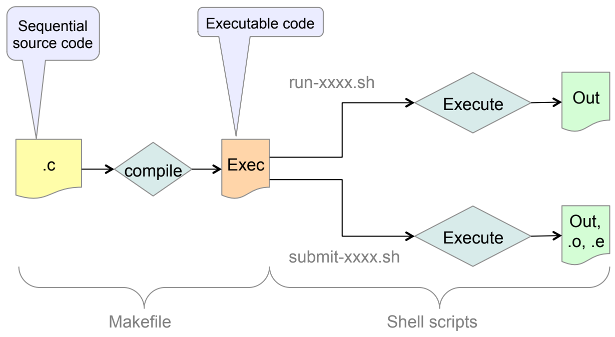


Figure 2.3: Compilation and execution flow for sequential programs.

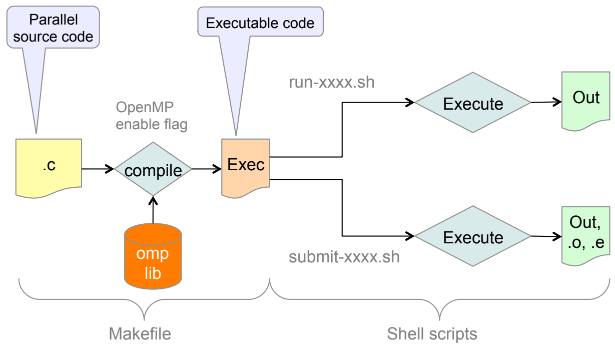


Figure 2.4: Compilation and execution flow for OpenMP.

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

3. Interactively execute the *OpenMP* code with 1, 2, 4, 8, 16 and 20 threads and same number of iterations (1.000.000.000) using the `run-omp.sh` script. What is the `time` command telling you about the user and system CPU time, the elapsed time, and the % of CPU used? Take a look at the script to discover how the number of threads to use in *OpenMP* is specified.

4. Submit the execution of the `submit-omp.sh` script to the execution partition using the `sbatch` command, specifying the *OpenMP* code, the same number of iterations (1.000.000.000) and the number of threads (do it with 1, 2, 4, 8, 16 and 20 threads). Look at the `time-pi_omp-X-boada-Y` files, being `X` the number of threads used and `Y` the node where the execution happened.

We suggest you to fill in table 2.2 to show the user and system CPU time, the elapsed time, and the % of CPU used in the two scenarios (interactive and queued) and with the number of threads enumerated before.

# threads	Interactive: Timing information				Queued: Timing information			
	user	system	elapsed	% of CPU	user	system	elapsed	% of CPU
1								
4								
8								
16								
20								

Table 2.2: Interactive vs Queued Elapsed Execution time

2.5 Strong vs. weak scalability

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

- In *strong* scalability the number of threads is changed with a fixed problem size. In this case parallelism is used to reduce the execution time of the program.
- In *weak* scalability the problem size is proportional to the number of threads. In this case parallelism is used to increase the problem size for which the program is executed.

Two scripts are provided to analyse scalability, `submit-strong-omp.sh` and `submit-weak-omp.sh`, which should be submitted to the queueing system. The scripts execute the parallel code using from 1 (`np_NMIN`) to 20 (`np_NMAX`) threads. The problem size for strong scalability is 1.000.000.000 iterations; for weak scalability, the initial problem size is 100.000.000 which grows proportionally with the number of threads. As a result the script generates a plot (in Postscript format) showing the resulting parallel execution time and speed–up.

1. Submit the execution of the `submit-strong-omp.sh` script (no arguments are required to execute the script). The execution will take some time because several executions are done for each test (in order to get a minimum time), please be patient!. Use the ghostscript `gs` command to visualise the Postscript file generated[6]. Observe how the execution time and speed–up varies with the number of threads in the *strong* scaling scenario.
2. Change the value for `np_NMAX` in the `submit-strong-omp.sh` from 20 to 40 and execute again. Can you explain the behaviour observed in the scalability plot?
3. Submit the execution of the `submit-weak-omp.sh` script (no arguments are required to execute the script). Observe now how the parallel efficiency varies with the number of threads in the *weak* scaling scenario.

2.6 Discussion at laboratory

Set of questions that will be discuss at the during and/or at the end of your lab sessions with your lab professor:

- Do you observe a major difference between the interactive and queued execution (See table 2.2)?
- Let's look at the weak and strong scalability plots generated and reason about the performance you obtained.

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

3

Systematically analysing task decompositions with *Tareador*

This chapter introduces *Tareador*, an environment useful for analysing the potential parallelism that can be obtained when a certain *task decomposition* strategy is applied to your sequential code. With *Tareador* the programmer simply needs to identify which are the tasks in the task decomposition strategy that wants to be evaluated. Then *Tareador* 1) traces the execution of the program based on the specification of potential tasks to be run in parallel, 2) records all static/dynamic data allocations and memory accesses in order to build the task dependence graph, and 3) simulates the parallel execution of the tasks on a certain number of processors in order to estimate the potential speed-up. Figure 3.1 shows the compilation and execution flow for *Tareador*, starting from the taskified source code.

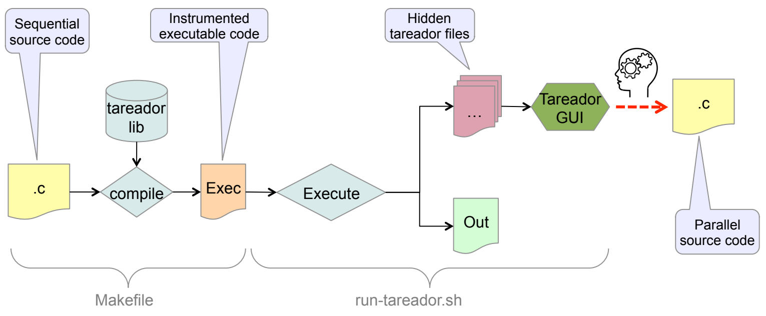


Figure 3.1: Compilation and execution flow for *Tareador*.

3.1 *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 running "`make 3dfft_tar`".

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

3.2 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 3.2). Each node of the graph represents a task: different shapes and colours are used to identify task instances generated from the same task definition and each one is labeled with a task instance number. In addition, each node contains the number of instructions that the task instance has executed, as an indication of the task granularity; the size of the node also reflects in some way this task granularity. Simply zoom in and out to see the names of the tasks (the same that were provided in the source code) and the information reported for each node.

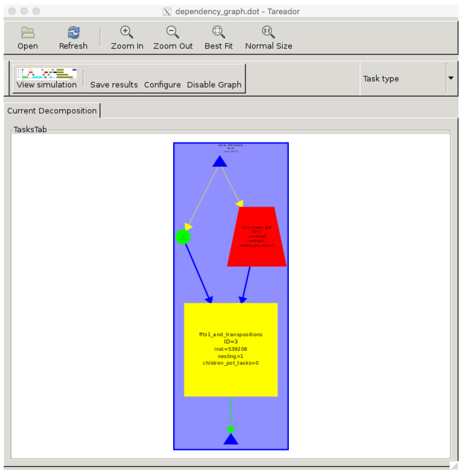
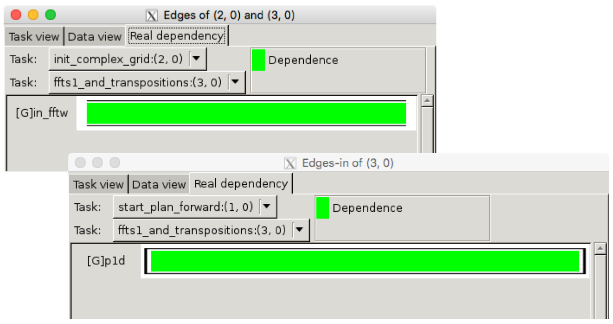


Figure 3.2: Task dependence graph for the initial task decomposition expressed in `3dftt_tar.c`.

2. Edges in the graph represent dependencies between task instances; different colours/patterns are used to represent different kind of dependences (blue: data dependences, green/yellow: control dependences). *Tareador* allows you to analyse the variables whose access provokes each data dependence between a pair of nodes: with the mouse on an edge (for example the edge going from the red task (`init_complex_grid`) to the yellow task (`ffts1_and_transpositions`), right click with the mouse and select *Dataview* → *edge*. This will open a window similar to the one shown in Figure 3.3. 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).

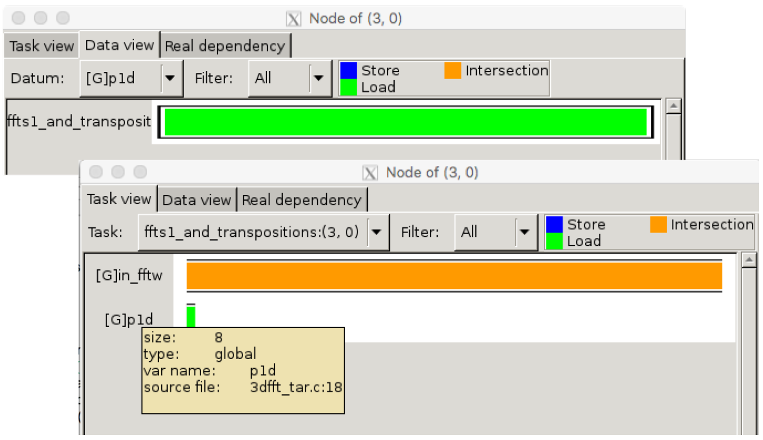


Task: init_complex_grid:(2, 0)
Task: ffts1_and_transpositions:(3, 0)
[G]in_fftw

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

Figure 3.3: 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 3.4. 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.



Datum: [G]p1d
Filter: All
ffts1_and_transposit

Task: ffts1_and_transpositions:(3, 0)
Filter: All
[G]in_fftw
[G]p1d
size: 8
type: global
var name: p1d
source file: 3dfft_tar.c:18

Figure 3.4: 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).

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 3.5. Each horizontal line shows the task(s) executed by each processor (`CPU1.x`, with `x={1..4}`). Colours are used to

represent the different tasks (same colours that are used in the task graph). The number on the lower-right corner of the window indicates the simulated execution time (in time units, assuming each instruction takes 1 time unit) for the parallel execution. In the next laboratory session you will go deep into the use of this tool, but for example you can zoom into the initial part of the timeline in order to visualise the same part of the trace that is shown in that figure; you can do 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.

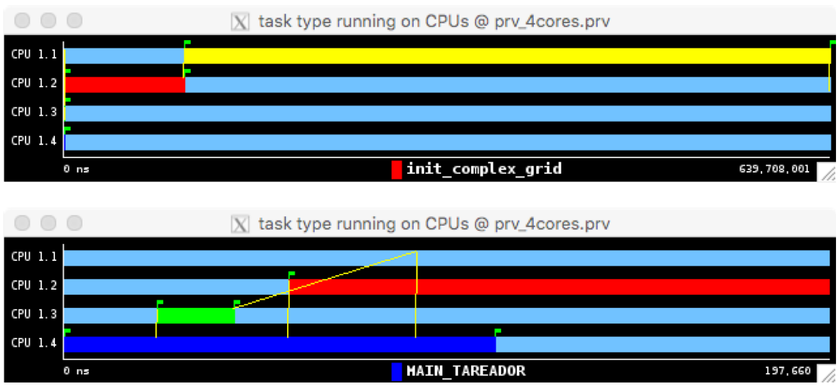


Figure 3.5: *Paraver* visualisation of the simulated execution with 4 processors, full view and after zooming into the initial part of the trace.

6. You can also save the timeline for the simulated parallel execution by clicking *Save* $\rightarrow$ *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.

3.3 Exploring new task decompositions for 3DFFT

Once you are familiar with the basic features in *Tareador*, and motivated by the reduced parallelism obtained in the initial task decomposition (named v0 from now on), you will proceed refining the initial tasks with the objective of discovering more parallelism. You will incrementally generate five new finer–grained task decompositions (named v1, v2, v3, v4 and v5) as described in the following bullets. For each task decomposition compute T_1, T_∞ and the potential parallelism ($T_1 \div T_\infty$) from the task dependence graph generated by *Tareador*, assuming that each instruction takes one time unit to execute. You can obtain T_∞ by simulating the execution of the graph with a sufficiently large number of processors. We advise you to save and compare all the task dependence graphs obtained and fill in table 3.3.

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

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

Version	T_1	T_∞	Parallelism
seq			
v1			
v2			
v3			
v4			
v5			

Table 3.1: Summary of the parallelism performance of each of the versions

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, 32 and 128 (as ∞) 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?

3.4 Discussion at laboratory

Set of questions that will be discuss at the during and/or at the end of your lab sessions with your lab professor:

- What is the worst and best parallel strategy (See table 3.3)?
- Which are the main differences between them?
- In the real life, is the best strategy the one you will expect to have the best performance?

4

Understanding the execution of *OpenMP* programs

This chapter presents a methodology to analyze and improve the performance of your parallel implementation in *OpenMP*. This methodology is based on the following three steps that you can repeat while you are not satisfied with the performance of your parallel implementation:

1. Using `modelfactors` (explained below), analyze the overall performance and scalability, the actual parallel fraction in your parallel implementation (ϕ) and parallelisation strategy efficiency based on load balancing and overheads.
2. Using *Paraver*, analyze traces generated from the parallel execution in order to better diagnose the performance inefficiencies reported by `modelfactors`.
3. Modify, if necessary, your parallel implementation based on the previous diagnose and repeat the process.

As you will be able to note, all the process is based on the generation of traces from the instrumented execution of the parallel program. Figure 4.1 shows the complete compilation and execution flow that needs to be taken in order to do this tracing process. The environment is mainly composed of *Extrae* and *Paraver*. *Extrae* 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]. 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. When using `modelfactors`, the execution of the *Extrae* instrumented program is repeated for different number of processors, generating traces for ech of them; then all these traces are analyzed in order to generate the reports that are provided by the tool.

4.1 Discovering `modelfactors`: Overall analysis

`modelfactors.py` is a Python program to analyse 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`). It also runs `modelfactors.py` to generate the strong scaling analysis of your parallel implementation. We will indicate in the next section how to execute it with the required arguments. By now, we will just explain you the information reported in the text file `modelfactors.out` generated. The output consists of three tables, each with a number of rows. To the left of each row a description of the contents of that row appears. From left to right, the correponding values for each metric are shown for the execution with different number of threads/processors. We have also created a script to generate a pdf document with all the tables and captions (`modelfactor-tables.pdf`).

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

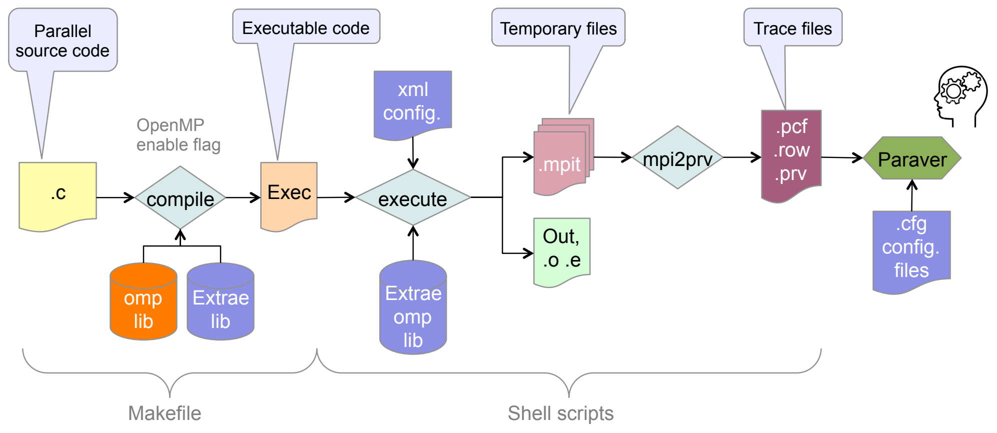


Figure 4.1: Compilation and execution flow for tracing.

You should include those tables, including their captions, in your reports. The three tables group different metrics:

- In a first table the tool provides a summary of program execution metrics for the entire program and for the different number of threads p analysed: elapsed execution time (in seconds, let's name it T_p), Speed–up (i.e. $S_p = T_1 \div T_p$) and Efficiency (i.e. $E_p = S_p \div p$). With this information you can observe the scalability of your code. If the reported results seem unsatisfactory and you want to understand the reasons, you can go to the next table in the summary. Table 4.1 presents an analysis example for a parallel program using up to 4 processors.

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

Table 4.1: Analysis done on Thu Jul 28 07:54:35 AM CEST 2022, par0

- In a second table the tool provides a deep dive into the Efficiency metric, but only for the *parallel fraction* (ϕ) of your program, that is, the part of the program that can be (or it is currently) parallelised. This is a good metric to realise if the inefficiencies come from a large serial part in your program that has not yet been optimised or simply can not be optimised. In this table, *IPC* stands for Instructions Per Cycle. For the parallel fraction of the code the tool reports two metrics that contribute to its efficiency, being the *Global efficiency* the product of these two metrics:
  - *Parallelisation strategy efficiency*, which is related to two main aspects and calculated as their product: *Load balancing* and *In execution efficiency* where the latter reflects the lack of overheads due to work generation and synchronisation in the critical path of the execution. The higher and close to 100% these two values are, the better. With these two metrics it should be enough to know what to do next in your analysis and optimisation. Load balancing reflects the ratio between the average useful code executed by the threads and the maximum useful code executed by any of them.
  - *Scalability for computation tasks*, which is related to how the processors in your system are actually executing the computation tasks, in terms of total number of instructions useful executed, number of instructions useful executed per cycle of operation, and frequency (number of cycles useful of operation per second) of the processor. As with the previous metric, it is calculated as their product. The value of these metrics for p threads are relative to 1 thread and again, the closer to 100% the better. But they can be lower for specific cases due to,

for example, plenty of synchronization overheads. You can find more details about how the metrics of this table are computed in the Annex.

Table 4.2 presents an analysis example for a parallel program using up to 4 processors.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.94%			
Number of processors	1	2	4
Global efficiency	99.97%	87.95%	77.95%
Parallelization 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%

Table 4.2: Analysis done on Thu Jul 28 07:54:35 AM CEST 2022, par0

- A third table completes the report. If the code contains explicit tasks then details about them will be provided. Otherwise, information about the implicit tasks will be reported. This table can be very useful to detect overhead problems that may be due to overheads of synchronizations or number of tasks created. Table 4.3 presents an analysis example for a parallel program using up to 4 processors. *LB* stands for Load Balancing.

Statistics about explicit tasks in parallel fraction			
Number of processors	1	2	4
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

Table 4.3: Analysis done on Thu Jul 28 07:54:35 AM CEST 2022, par0

You can find more details about the output metrics of modelfactors in the Annex.

Example `3DFFT`: Obtaining parallelisation metrics using `modelfactors`

Overall Analysis

To demo the use of these tools, a programmer has provided an *OpenMP* implementation for the *3DFFT* code which implements a very similar task decomposition to version *v3* that you explored in the previous laboratory session.

1. Go into the `lab1/3dfft` directory. Open the `3dfft_omp.c` file in that directory and look for the lines containing *OpenMP* pragmas. We will study *OpenMP* in the next laboratory assignment, but for now just a bit of explanation of what you have here in the program. With '`#pragma omp parallel`' (for example in line 48) 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 (what we will call the *implicit task*). With '`#pragma omp single`' (line 49) we are telling that only one of the threads in the parallel region will continue with the execution of the body of the `single` construct; the other threads will remain idle waiting

at the end of the `single` construct for additional tasks to be executed. And finally, the thread that entered into `single` will encounter '`#pragma omp taskloop`' (line 52), indicating that the for loop that follows will be divided in a number of so called *explicit tasks*, each one to be executed by any of the idle threads. When the thread that entered into the `single` finishes with the generation of all tasks in the `taskloop`, it will also participate in the execution of the *explicit tasks* that it generated. When all *explicit tasks* are executed, the `parallel` region will be finished returning to serial execution until a new parallel region is found, if any in the code. In our program, three parallel regions are initially defined[2].

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

3. Submit the execution of the `submit-strong-extrae.sh` script indicating the name of the binary program `3dfft_omp`. This will take some time (minutes) since the script is tracing the execution with 1, 4, 8, 12, 16 and 20 threads and performs the analysis with `modelfactors`; so monitor the status of the job in the queue (type "`watch squeue`") to check when it is finished ("control-C to stop watching the status of the queue"). Once finished check that the directory `3dftt_omp-strong-extrae` exists, and inside it the `modelfactors-tables.pdf` file that you can open with `xpdf`.

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

4.2 Discussion at laboratory

Set of questions that will be discuss at the during and/or at the end of your lab sessions with your lab professor. Based on the output metrics of modelfactors, it is time to think about the following questions:

- What is the information that you can see at the modelfactor tables? How can we use it?
- For our case:
  - Is the scalability appropriate?
  - Is the overhead due to synchronization neglegible?
  - Is this overhead affecting the execution time per explicit task?
  - Which is the parallel fraction (ϕ) for this version of the program?
  - Is the efficiency for the parallel regions appropriate?
  - Which is the factor that is negatively influencing the most?

4.3 Discovering *Paraver* (Part I): execution trace analysis

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*.

We suggest you to keep the information of ϕ value of the `modelfactors` tables and capture the *Paraver* timeline(s) window(s) that allow to make the appropriate comparison of the three versions that we are going to analyze of `3DFFT` (initial, reducing overhead, and improving ϕ) using the timeline window and profile of thread states. For instance it would be good you can fill in the table below:

You should be able to fill in the table of ϕ value of the intial version. Next sections will help you to fill in the rest of data step by step.

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

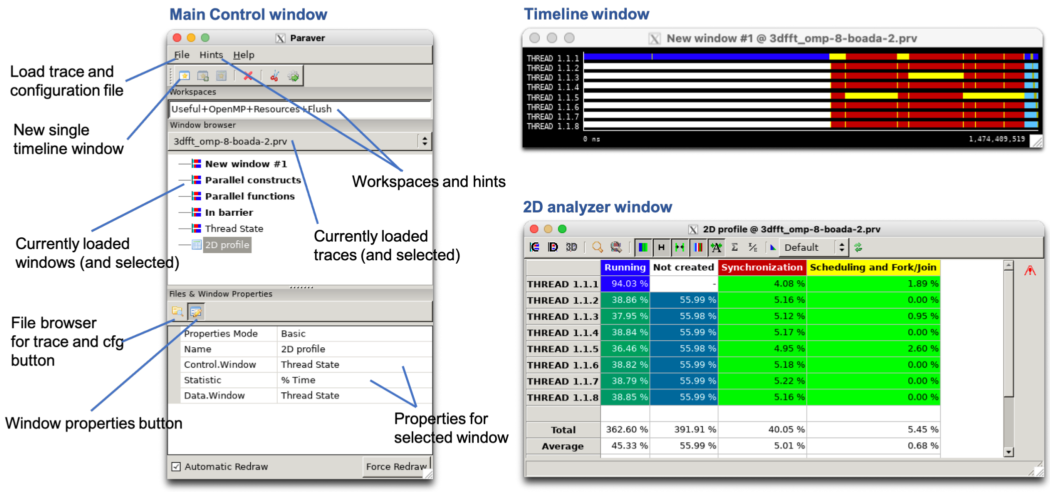
Version	ϕ	ideal S_{12}	T_1	T_{12}	real S_{12}
initial version in `3dfft_omp.c`					
new version with reduced parallelisation overheads					
final version with improved ϕ					

4.3.1 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 4.2 (left).

2. **Load trace:** From the main menu, select *"File → Load Trace"*, and navigate through the directory structure until you find the trace file (`.prv`) generated from the instrumented execution of the `3dfft_omp` binary during the execution of the `modelfactor` tool (they should be inside directory `3dfft_omp-strong-extrae`), load for example the one for 12 processors. Alternatively, traces can be located through the browser at the bottom of the *Main Window*: double clicking on a `.prv` file will load it. For the purposes of this guided tour, traces mainly contain two types of records: *states* and *flags*. These two kind of records are used by *Extrae* to inject information about the parallel execution in the trace.

3. Once the file is loaded, click on the *New single timeline window* box (top left icon in *Main Window*). A new window, similar to the one shown in Figure 4.2 (top-right), appears showing a timeline with the *state* (encoded in colour) of each thread (vertical axis) in the parallel program. The horizontal axis represents time, advancing from left to right.



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 4.2: Paraver Main Window, Timeline and 2DAnalyzer windows.

- Colours: While moving the mouse over the window, a textual description of the meaning of each colour is shown (at the bottom of the same window): light blue (*idle*), dark blue (*running*), red (*synchronisation*), white (*not created*), yellow (*scheduling and fork-join*), ... It is important to be aware that the meaning of each color is specific to each window. Through this hands-on you will see different timeline windows each of them displaying a different information with its own colouring table. Click with the right button of mouse at any point inside the window and select *Info Panel* and then the *Colors* tab to see the colouring table for the window.

- Textual information: Double click with the left button of your mouse on any point in the red or yellow areas in the timeline. This action will activate the *What/Where* tab of the Info Panel, showing in textual form information at the selected point in the trace (thread and time where you have clicked, thread state at this point and how long the time interval with that state is). With *"Right Button → Info Panel"* in the trace window you can decide to show or hide this panel.

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

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

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

   - Zoom: Click with the left button of the mouse to select the starting time of the zoomed view, drag the mouse over the area of interest, and release the mouse to select the end time of the zoomed view.
   - *Undo Zoom* and *Redo Zoom* commands are available on the right button menu. You can do and undo several levels of zooming.
   - *Fit time scale* can be used to return to the initial view of the complete execution.

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

4.3.2 Profiles: summarizing information in the trace

5. Finally, mentioning that *Paraver* also offers *Hints* (also named *workspaces*) to ease the analysis of *Extrae* traces. The hints are grouped in different categories and available from the main menu in the *Main Window.* Open hint `OpenMP/thread_state_profile`: as you see, instead of opening a timeline this pops up a table, as the one shown in Figure 4.2 (bottom-right), with one row per thread and one column per thread state (*Running*, *Synchronization*, *Scheduling and Fork/Join*, ...). Each cell value shows the percentage of time spent by a thread in a specific state. Observe that all threads, except the the first one, spent some time in *Not created* and that the percentage spent in *Synchronization* is significant compared to *Running.* The later means that the efficiency of thread utilisation is low for this version of the code (don't worry, you will solve this in the next section). To see a different statistic change the *Statistic* selector in the *Window Properties* panel of the *Paraver Main Window.* Other interesting metrics at this time may be:

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

4.3.3 Flags in the trace

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

Again, 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 parallelisation details using *Paraver*

Detailed Analysis

1. For the trace you have loaded in previous section, open *Paraver* hint `User_funtions` which identifies when the different functions in the program are executed for the previous *Paraver* trace you loaded. Each function is encoded with a different colour.

2. Aligning windows: In *Paraver* every timeline window represents a single metric or view for all selected threads and time span. It is possible to align two timelines by making them display the exact same threads and time span. To practise this, just take 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*; then on the target window, right-click and select *"Paste → Default"* (or separately *"Paste → Size"* and *"Paste → Time"*). Both windows will then have the same size and represent different views (metrics) for the same part of the trace. If you put one above the other there is a one to one correspondence between points in vertical.

3. You can also synchronise several windows by adding them to the same group. For example select *Synchronise → 1* after a Right-click with your mouse on the initial state timeline window showing the parallel activity. Repeat this with the hint `user functions` timeline window. Once synchronised they will continue aligned after zooming, undoing or redoing zoom. With the two windows synchronised it is easy, for example, to correlate the duration of tasks with the implicit task they belong to, allowing us to observe if parallel regions take the same time and observe if there is some work unbalance among threads in any of them. You can always un-synchronise a window by un-selecting again *Synchronise* after a Right-click with your mouse on the timeline window. You can create new groups of synchronised windows at your convenience. Sychronize the two timeline windows opened: User functions and the one you open in previous section.

There are two key factors that influence the overall scalability and final performance. Looking at the two timelines windows we can see these two factors: 1) there is one function that is not parallelised and 2) there is a thread state that is predominant along the timeline window. At this moment, the second one seems to more important because there is a strong scalability problem (slowdown from 12 threads) due to the number of tasks and synchronizations in the program.

4.4 Discovering *Paraver* (Part II): understanding the parallel execution

4.4.1 Implicit and Explicit tasks

To better understand from where this inefficiency comes from, let's further analyse with *Paraver* the tasks that are created in our program. A couple of *Paraver* hints will be helpful for that:

1. Open Hint `OpenMP/implicit_tasks_duration`. It shows the duration of the implicit tasks executed by threads in the parallel regions. A gradient coloured timeline is used to show the semantics of the window, painting each pixel in the window with a colour, from light green (low value for the duration) to dark blue (high value for the duration). By moving the mouse arrow on top of the bursts *Paraver* will tell you their duration. Of course, clicking in one of them you can also have this information; but the gradient coloured window gives you a more global picture of parallel region durations in the same parallel region and across parallel regions.

2. Open Hint `OpenMP_tasking/explicit_tasks_duration`. It visualises the gradient coloured timeline showing the duration of the tasks executed by the different threads. You will be able to observe the task granularities generated in the different `taskloop` constructs and observe, for example, if the explicit tasks generated from the same `taskloop` have the same duration or not.

But also the following two that will produce histograms with the duration of the implicit and explicit tasks:

3. Open *Paraver* hint `OpenMP/Histogram of Implicit task duration`. A window pops up, showing with a colour gradient the distribution of the different implicit task durations. For each of the threads you can see the histogram of different implicit task executed.

4. Open *Paraver* hint `OpenMP tasking/Histogram of explicit execution task duration`. A window pops up, showing with a colour gradient the distribution of the different explicit task durations. For each of the threads you can see the histogram of different explicit task executed.

There are other hints that you can see in the Annex.

Once you have seen the histograms and timeline windows of the explicit task durations, it seems to be clear that fine granularity may not be the best grain parallel strategy.

4.4.2 Example `3DFFT`: Reducing Parallelisation Overheads and Analysis

Optimization

We suggest you to explore the possibility of increasing the granularity of tasks by simply commenting the innermost `taskloop` in each parallel region and uncommenting the outer one. This should reduce parallelisation overheads and as a consequence have some impact in the overall performance obtained.

Overall Analysis

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?

It seems that we have reduced the overhead problem. On the other hand, why do you think you can not achieve better speedup for 12 or more threads? Hint: Compute the maximum speedup that you achieve with the current parallel fraction that you are parallelising (see `modelfactors` results). This value is limiting the maximum speedup that we can achieve for the machine we have.

Detailed Analysis

We think it is necessary to perform some more analysis to understand the reason of the overall improvement and the value of ϕ value.

1. Load trace (`3dfft_omp-strong-extrae/3dfft_omp-12-boada-11-cutter.prv`).

2. Once the file is loaded, click on the *New single timeline window* box (top left icon in *Main Window*). A new window, similar to the one shown in Figure 4.2 (top-right), appears showing a timeline with the *state* (encoded in colour) of each thread (vertical axis) in the parallel program. The horizontal axis represents time, advancing from left to right. Also open hint User functions.

3. Load again the trace generated for 12 threads of the execution of the intial version of `3DFFT`. It should be in the `modelfactors` directory of the overall analysis of the intial version.

4. Make sure that both traces, for optimizated and initial versions, are with the same temporal scale.

5. Capture the *Paraver* timeline(s) window(s) that allow to make the comparison of the two versions and see the impact on the overall execution time. Remember to explain your conclusions briefly.

6. Obtain the histogram with the durations of the different **explicit and implicit tasks** (hint OpenMP/Histogram of implicit task duration) for the new version. You can also open the timeline window using the corresponing hint explained above.

7. For the execution trace of the optimised version, also open *Paraver* hint `User_funtions` which identifies when the different functions in the program are executed for the previous *Paraver* trace you loaded. Each function is encoded with a different colour.

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

There is a function (`init_complex_grid`) that has not been parallelized yet and its execution time is limiting the overall performance.

4.4.3 Example `3DFFT`: Improving ϕ and Analysis

Optimization

We propose you a parallel implementation strategy optimization based on your analysis with `modelfactors` and *Paraver* analysis. You have seen that maximum speedup is limited by the serial execution of function `init_complex_grid`. Open the `3dfft_omp.c` file and uncomment the pragmas that will allow the parallel execution of the code inside function `init_complex_grid`. At this point only uncomment the outer loop `taskloop` together with `parallel` and `single`.

Overall Analysis

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

Detailed Analysis

We would also like to do a final comparison of the three versions looking a the their timelines executions.

1. Load trace (`3dfft_omp-strong-extrae/3dfft_omp-12-boada-11-cutter.prv`).

2. Once the file is loaded, click on the *New single timeline window* box (top left icon in *Main Window*). A new window, similar to the one shown in Figure 4.2 (top-right), appears showing a timeline with the *state* (encoded in colour) of each thread (vertical axis) in the parallel program. The horizontal axis represents time, advancing from left to right.

3. At this point we recommend that you visualise the traces for the three versions simultaneously, for example visualising at the same time the initial timeline window for both of them. In order to see the net effect of this optimization (reduction of execution time), click *Copy* in the window for the first version and *Paste* $\rightarrow$ *Time* in the second window.

4.5 Discussion at laboratory

Set of questions that will be discuss at the during and/or at the end of your lab sessions with your lab professor:

- Do you think using a very fine granurality is always the best parallel strategy? why? Look again at Table 4.3 of `modelfactors` you generated at subsection 4.1 and timelines and histograms in section 4.3.
- Was the synchronization overhead constant or function of the number of threads?
- How did you solve the overhead problem of the original fine grain granularity parallel strategy?
- Once we solved the overhead problem, which was the performance limitation of the parallel strategy you observed at section 4.4.2?
- Compare the ϕ of the tree versions you have done: initial, reducing overhead and improving ϕ under the point of view of strong scalability.

Appendix A

Modelfactors

A.1 What information does modelfactors.py use?

`Modelfactors` extracts information from Extrae traces: Execution time (runtime) and `in useful state` which is used to compute different duration metrics: `useful_avg`, `useful_max` and `useful_tot`.

On the other hand, two basic hw counters (`PAPI_TOT_CYC` and `PAPI_TOT_INS`) are used to extract two raw metrics:

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

The tool has been extended to isolate the metrics only inside parallel regions, what is called the parallel fraction (ϕ).

A.2 Computing the metrics for OpenMP

In this section, we explain in detail second and third tables of `modelfactors` output.

`Modelfactors` are computed only for the parallel fraction (ϕ) of the program, according to the definition used when we apply Amdahl. The parallel fraction may contain multiple parallel regions. Assume P the number of processors. For the parallel fraction of a program, its efficiency ($\mathit{Efficiency}(P)$, *Global efficiency* in the second table) is defined as the product of two factors: $\mathit{Efficiency}(P) = \mathit{Parallel\_Eff}(P) \times \mathit{Comp\_Scal}(P)$

- $\mathit{Parallel\_Eff}(P)$ (*Parallelisation strategy efficiency* in the second table) exposes the inefficiencies in the task decomposition strategy
- $\mathit{Comp\_Scal}(P)$ (*Scalability for computation tasks* in the second table) characterizes the inefficiencies when executing the computation tasks on the parallel architecture.

Regarding to $\mathit{Parallel\_Eff}(P)$, this is proportional to two factors $\mathit{LB\_Eff}$ (*Load balancing* in the second table) and $\mathit{InExecution\_Eff}$ (*In execution efficiency* in the second table). First factor exposes load balancing ($\mathit{LB\_Eff} = \frac{average(useful_i)}{\max(useful_i)}$) and second factor exposes when threads spend time executing useful code ($\mathit{InExecution\_Eff} = \frac{\max(useful_i)}{T}$). Being P the number of processors, T the total execution time for the parallel fraction, and $useful_i$ is the time thread i executing useful code, $\mathit{Parallel\_Eff}(P)$ can be computed as follows:

$$\mathit{Parallel\_Eff}(P) = \frac{\Sigma_{i=0}^{P-1} useful_i}{P \times T} = \frac{\frac{\Sigma_{i=0}^{P-1}(useful_i)}{P}}{\max(useful_i)} \times \frac{\max(useful_i)}{T}$$

For the case of $\mathit{Comp\_Scal}(P)$ (*Scalability for computation tasks* in the second table), this is equal to $\frac{useful(1)}{useful(P)}$ and exposes the efficiency of processors executing useful code. $useful(P)$ is defined as the total time all threads, P, are executing 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*$(P)/$*freq*(P) and :

- *inst*(P) is the total number of instructions executed during useful code
- *CPI*$(P) = \frac{cycles(P)}{inst(P)}$ is the average number of cycles needed to execute one instruction, being cycles the toal number of cycles spent during useful code.
- *freq*$(P) = \frac{cycles(P)}{useful(P)}$ is the average frequency processors are running while executing useful code.

That is, *Comp_Scal*(P) can be computed as $\frac{IPC(P)}{IPC(1)} \times \frac{Inst(1)}{Inst(P)} \times \frac{freq(P)}{freq(1)}$ that corresponds with *IPC_Scal*$(P) \times$ *inst_Scal*$(P) \times$ *freq_Scal*(P); the three factors *IPC scalability*, *Instruction scalability* and *Frequency scalability* in the second table for *Scalability for omputation tasks.*

An instruction scalability smaller than 1 means that the parallel program needs more instructions than the sequential code. An *IPC* smaller than 1 means that the *IPC* for P threads is worse than for 1 thread. A frequency scalability smaller than 1 means that using P threads the frequency decreases.

Regarding to the third table, this shows the detail for explicit tasks. In particular, it exposes different factors that affect the overall load balance and overheads due to explicit tasks. In particular, *LB (number of explicit tasks executed)* and *LB (time executing explicit tasks)* factors in the third table show the load umbalace related to the number and execution time 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 load balance information (explicit tasks) in the third table. On the other hand, the % of overhead due to synchronization and scheduling can be a good indicator of scalability problems of our parallel strategy, usually related with the number of tasks and their granularity.

Appendix B

Paraver

B.1 Implicit tasks

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

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

3. Hint `OpenMP/implicit tasks duration` shows the duration of the implicit tasks executed by threads in the parallel regions. A gradient coloured timeline is used to show the semantics of the window, painting each pixel in the window with a colour, from light green (low value for the duration) to dark blue (high value for the duration). By moving the mouse arrow on top of the bursts *Paraver* will tell you their duration. Of course, clicking in one of them you can also have this information; but the gradient coloured window gives you a more global picture of parallel region durations in the same parallel region and across parallel regions.

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

There are other hints in the *Paraver* menu. The upper part in Table B.1 lists these and some other *Paraver* hints that are available for doing this kind of analysis. We will use them later in this tour or in other laboratory sessions.

B.2 Explicit tasks

Implicit tasks encapsulating the code region executed in parallel constructs are not the only kind of tasks we will have in our *OpenMP* parallel programs. Other tasks can be created during the execution

***Paraver* hint**	**Timeline showing ...**
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 syncrhonization	when threads are in a `barrier` synchronization
In critical syncrhonization	when threads are in/out/entering/exiting `critical` synchronisations

***Paraver* hint**	**Profile showing ...**
Thread state profile	a profile of the thread states (useful, scheduling, synchronization, ...)
In critical syncrhonization profile	a profile of the different phases a thread goes on when synchronising in a `critical` construct
Implicit tasks profile	a profile of the implicit tasks
Histogram Implicit tasks duration	histograms with the durations of implicit tasks

Table B.1: First set of *Paraver* hints – upper part: timeline views; lower part: statistical summaries in the form of profiles (histograms).

of parallel regions by any thread to give work to another thread to execute. These tasks will be named *explicit tasks* and they will be created when the `task` or `taskloop` (this is the case for the `3dfft_omp.c` program we are looking at) constructs are used in our program. Table B.2 adds some new *Paraver* hints to the ones you just used in the previous subsection, all of them at the `OpenMP tasking and taskgroup` menu entries. Let's play with some of them, others for later.

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

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

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

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* $\rightarrow$ *Time* on the table. The analysis will be repeated just for the selected time interval. Select *Fit time scale* to compute the profile for the whole trace.

***Paraver* hint**	**Timeline showing ...**
tasking/explicit tasks function created and executed	when *explicit tasks* are created and executed (file and line inside file)
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`

***Paraver* hint**	**Profile showing ...**
tasking/profile of explicit tasks creation and execution	a profile of task creations and executions
tasking/Histogram of explicit tasks execution duration	histograms with durations of the *explicit tasks* executed

Table B.2: Second set of *Paraver* hints for tasking analysis in *Paraver*– upper part: timeline views; lower part: statistical summaries in the form of profiles (histograms).