SCA Laboratory
Lab1: Experimental setup, Compilation, Execution and Tracing

C. Álvarez, D. Jiménez

Fall 2024

Contents

How to do this laboratory

Part of this document comes from PAR course at FIB. This laboratory aims to provide a brief introduction to the environment and tools that you may use in your SCA practicals, in particular, for those of you that have not ever worked with parallel computing or this environment or those tools. Consequently, this laboratory session documentation is longer than what a neophyte could expect to accomplish in 2 hours. The idea is that those of you already acquainted with some of the concepts explained could skip them while providing a comprehensive documentation of the tools that we are going to use in remaining sessions of the laboratory. In order to get graded you should deliver a **report** answering the questions in this document (you can do it by pairs) in the corresponding "Practicals" section of the Racó.

If you feel that you can skip the contents of this laboratory because you already know the concepts and the work environment, but you still want to get graded (remember that this laboratory is optional) you should:

1. Answer the questions in section 3.3 in your deliverable.
2. Add to your report an image of a trace of one of the OpenMP executions as in question 15.

Part 1

Experimental setup

The objective of this laboratory is to become familiar with the environment that you may use along the semester to do the SCA assignments. From your PC/terminal booted with Linux you will access a multiprocessor server located at the Computer Architecture Department, establishing a connection to it using secure shell: `"ssh -X scaXXYY@boada.ac.upc.edu"`, being `XXYY` the user number assigned to you. Option `-X` is necessary in order to forward the X11 and be able to open remote windows in your local desktop. You can change the password for your account using `"ssh -t scaXXYY@boada.ac.upc.edu passwd"`.

Once you are logged in you will find yourself in your "home" directory in any of the interactive nodes: `boada-6` to `boada-8`, the login nodes for the whole machine where you can execute interactive jobs and from where you can submit execution jobs to the rest of the nodes in the machine. In fact, `boada` is composed of 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 the numerical part of the 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 SCA 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/scaXXYY` (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/scaXXYY`.

All necessary files to do each laboratory assignment will be posted in `/scratch/nas/1/sca0/sessions`. For the session today, copy `lab1.tar.gz` **from** that location **to** your home directory in `boada` and uncompress it at the **root of your home directory** with this command line: `"tar -zxvf lab1.tar.gz"`. In order to set up all environment variables you have to process the `environment.bash` file now available in your home directory with `"source environment.bash"`. **Note:** since you have to do this every time you login in the account or open a new console window, it is strongly recommended that you add this command line in the `.bashrc` file in your home directory, a file that is executed every time a new session is initiated.

In case you need to transfer files from `boada` to your local machine (laptop or desktop in laboratory room), or viceversa, you have to use the secure copy `scp` command. For example if you type the follow-

ing command "scp -O scaXXYY@boada.ac.upc.edu:lab1/serial/pi_seq.c ."[1] in your local machine you will be copying the source file pi_seq.c inside directory lab1/serial of your home directory in boada to the current directory in the local machine with the same name.

1.1 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).

We strongly suggest to use option 1) when you want to ensure that the execution is done in isolation inside a single node of the machine; the execution starts as soon as a node is available. When using option 2) your execution starts immediately but will share resources with other programs and interactive jobs, not ensuring representative timing results. For the introductory purposes of this lab, we will provide some scripts for both options (submit-xxxx.sh and run-xxxx.sh, respectively). The commands that allow you to execute for both options follow:

- 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" 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.

1.2 Node architecture and memory

With the following experiments you will be able to figure out the node architecture and memory of the execution nodes.

Submit script architecture/submit-arch.sh to the queue. This executes the lscpu and lstopo to know:

1. the number of sockets, cores per socket and threads per core in a node of the machine;
2. the amount of main memory in a node of the machine, and each NUMAnode;
3. the cache memory hierarchy (L1, L2 and L3), private or shared to each core/socket.

"--of fig map.fig" option for lstopo is useful to generate a .fig file with the architecture of your node. You can use the xfig command to visualize the output file generated (map.fig) and export to a different format (PDF or JPG, for example) using File → Export[2].

Report Question 1: *Include in the report the image generated by the script and a small table with the main characteristics of the node (e.g. which node have you measured, CPU model name, number of cores, number of threads, memory per node...)*

[1] The -O option is necessary to use the legacy scp protocol.

[2] In the boada Linux distribution you can use xpdf to visualize PDF and other graphic formats. You can also use the "fig2dev -L pdf map.fig map.pdf" command to convert from .fig to .pdf; look for alternative output graphic languages by typing "man fig2dev".

Report Question 2: *Explain what is the utility of the Makefile in the directory "architecture"*

Report Question 3: *In which login node are you executing? Which of the characteristics of the execution node reported in question 1 differ from the ones in the login node? Will you be able to obtain the same results in the login node than in the execution queue?*

1.3 Serial compilation and execution

For this laboratory you are going to use two different codes that compute π. For this first part you are going to use the `pi_seq.c` source code, which you can find inside the `lab1/serial` directory. It performs the computation of the number 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 $4/(1 + x2)$. You can see the code in figure 1.1.

```
static long num_steps = 100000;
void main () {
   double x, pi, step, sum = 0.0;
   step = 1.0/(double) num_steps;
   for (long int i=0; i<num_steps; ++i) {
      x = (i+0.5)*step;
      sum += 4.0/(1.0+x*x);
   }
   pi = step * sum;
}
```

Figure 1.1: Serial code of π computation.

For the second part of the laboratory you are going to use the `pi_seq.c` source code, which you can find inside the `lab1/mpi` directory. `pi_seq.c` performs the computation of π using a Monte Carlo computation[3]. Figure 1.2 shows a simplified version of the code you have in `lab1/mpi/pi_seq.c`. Variable `trials` defines the number of loop iterations.

```
float host_monte_carlo(unsigned long long trials) {
        float x, y;
        unsigned long long points_in_circle;
        for(long i = 0; i < trials; i++) {
                x = rand() / (float) RAND_MAX;
                y = rand() / (float) RAND_MAX;
                points_in_circle += (x*x + y*y <= 1.0f);
        }
        return 4.0f * points_in_circle / trials;
}
```

Figure 1.2: Serial code of a Monte Carlos computation that computes π.

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

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

2. Interactively execute the binary generated to compute the `pi` code by doing 1073741824 iterations using the `run-seq.sh` script which returns the user and system CPU time, the elapsed time, and the % of CPU used (using GNU `/usr/bin/time`). In addition, the program itself also reports the

[3] http://mathfaculty.fullerton.edu/mathews/n2003/MonteCarloPiMod.html

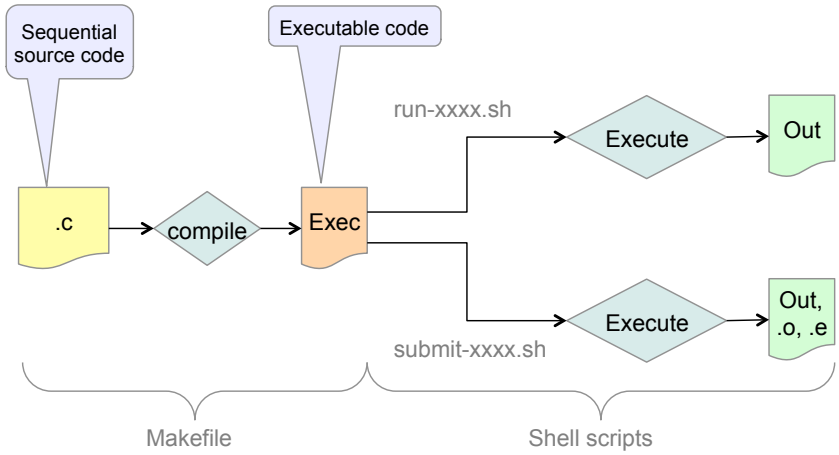

Figure 1.3: Compilation and execution flow for sequential program.

elapsed execution time using `gettimeofday`. Look at the source code and identify the function invocations and data structures required to measure execution time.

3. Submit the execution to the queueing system using the `"sbatch submit-seq.sh"` command. Use `"squeue"` to see that your script is queued and running. Look at `submit-seq.sh` script and the results generated (the standard output and error of the script and the `pi_seq_time.txt` file).

Report Question 4: *Report the time it has taken the code in three different executions in the interactive node and the execution node. Maybe you will observe that the time in the interactive node fluctuates depending on the node load.*

Report Question 5: *What do the generated ".eXXXXXX" and ".oXXXXXX" files contain?*

Part 2

Compilation and execution of OpenMP programs

In this course, one of the parallel programming models we suggest you to use is `OpenMP`, the standard for parallel programming using shared-memory, to express parallelism in the C programming language. We have provided you with an `OpenMP` mini-tutorial and will make you to do some simple exercises in next laboratories. In this part we will see how to compile and execute parallel programs in `OpenMP`. Remember that we will not explain `OpenMP` in detail since this is not an objective of this course. However, you can do CPDS course or ask SCA professors for specific details of the programming model.

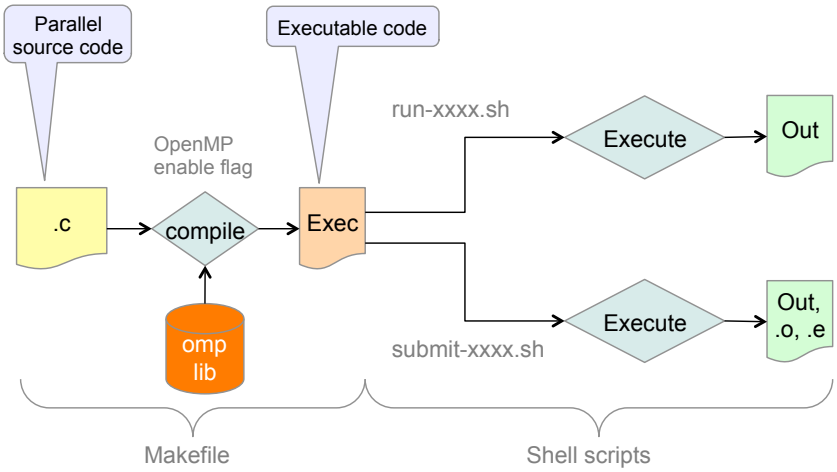

Figure 2.1: Compilation and execution flow for OpenMP.

2.1 Compiling OpenMP programs

1. In the `lab1/openmp` directory you will find an `OpenMP` version of the code to compute `pi` in parallel (`pi_omp.c`). Compile the `OpenMP` code using the appropriate target in the `Makefile`.

   Report Question 6: *What is the compiler telling you? Is the compiler issuing a warning or an error message? Is the compiler generating an executable file?*

2. Figure out what is the option you have to add to the compilation line in order to be able to execute the `pi_omp.c` in parallel (using `"man gcc"`).

   Report Question 7: *What have you had to change?*

3. Generate again the `OpenMP` executable of the `pi_omp.c` source code after adding the necessary compilation flag in the `Makefile`. Double check to be sure that the compiler has compiled `pi_omp.c` again with the new compilation flag provided (i.e. `"'pi_omp' is up to date"` is not returned by `make`).

2.2 Executing OpenMP programs

1. Interactively execute the `OpenMP` code with 20 threads (processors) and same number of iterations ($1073741824 = 1024^3$) using the `run-omp.sh` script.

   Report Question 8: *What is the `time` command telling you about the user and system CPU time, the elapsed time, and the % of CPU used? Take a look at the script to discover how do we specify the number of threads to use in `OpenMP`.*

2. Use `submit-omp.sh` script to queue the execution of the `OpenMP` code and measure the CPU time, elapsed time and % of CPU when executing the `OpenMP` program when using 20 threads in isolation.

   Report Question 9: *Do you observe a major difference between the interactive and queued execution? Note that the answer to this question may depend greatly on you doing the experiment while being alone in the machine or at the same time as other students.*

2.3 Strong vs. weak scalability

Finally, in this last part of the part you are going to explore the scalability of the `pi_omp.c` code when varying the number of threads used to execute the parallel code. To evaluate the scalability the ratio between the sequential and the parallel execution times will be computed. Two different scenarios will be considered: *strong* and *weak* scalability.

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

We provide you with two scripts, `submit-strong-omp.sh` and `submit-weak-omp.sh`, which should be submitted to the queueing system. The scripts execute the parallel code using from 1 (`np_NMIN`) to 20 (`np_NMAX`) threads. The problem size for strong scalability is 1000000000 iterations; for weak scalability, the initial problem size is 100000000 which grows proportionally with the number of threads. As a result the script generates a plot (in Postscript format) showing the resulting execution time and speed–up (You can `gs` file.ps to visualize the figure). The execution will take some time because several executions are done for each test (in order to get a minimum time), please be patient!. Visualize the plots generated and reason about how the speed–up changes with the number of threads in the two scenarios. Change the number of threads to 40 and try to repeat the experiments[1].

Report Question 10: *Copy the plots generated in the report. Are there any major differences if you use up to 40 threads? What happens when you use more than 20 threads?*

[1] Be careful, you may need to change the scripts to get the correct plots/numbers when moving to 40 threads.

Part 3

Compilation and execution of MPI programs

In this course we suggest you to use `MPI` to express parallelism in the C programming language in more than one node of the machine. We have provided you with a very simple mini-tutorial of `MPI` and will make you do some exercises using `MPI` in next laboratories. In this section we will show you how to compile and execute parallel programs in `MPI`.

3.1 Compiling MPI programs

1. In the `lab1/mpi` directory you will find an `MPI` version of the code for doing the computation of pi in parallel (`pi_mpi.c`). `pi_mpi.c` is the `MPI` version of `pi_seq.c`. `MPI` is the most used library for parallel programming using message passing. In order to compile it you should use `mpicc` compiler (which includes the appropriate include and library paths for `MPI`); if you try to compile it with gcc you will get compilation errors. Use the target of the Makefile to compile it.

   You will find two different targets:

   - `pi_mpi`: uses point-to-point communications.
   - `pi_mpi_collectives`: uses collective communications.

   Open the `pi_mpi.c` in order to understand the code and macro directives. You can do the experiments in this section with any of the target alternatives.

3.2 Executing `MPI` programs

1. Interactively execute the `MPI` executable that you have obtained with 3 processes using `run-mpi.sh` and 10000 trials (within the script, -np is used to specify the number of processes) to check that your MPI executable is correct. Make `man mpirun.openmpi` for more information.

   You should see something like that when running `pi` using 10000 trials:

   ```
   Hello world from process 1 of 3 at hostname boada-x
   Hello world from process 2 of 3 at hostname boada-x
   Number pi after 10000 iterations = 3.158799886703491
   Hello world from process 0 of 3 at hostname boada-x
   0.000180
   ```

   Where all the `pi` codes have been run in the interactive node `boada-x`.

Report Question 11: *Does the MPI version of the code scale up to the maximum available threads? Is it faster or slower than the previous OpenMP version? What happens when you use more than the available threads?*

2. In order to be able to use the execution nodes you need to use the execution queue. Submit the `submit-mpi.sh` script to queue execution using `sbatch submit-mpi.sh` command line. Once it has finished, look at the `submit-mpi.sh.oXXXX`.

   You should see something like that when running `pi` using 1073741824 trials:

```
Number pi after 1073741824 iterations = 3.141551494598389
Hello world from process 0 of 3 at hostname boada-3
6.154783
Hello world from process 1 of 3 at hostname boada-2
Hello world from process 2 of 3 at hostname boada-4
```

   As it can be observed, now the execution has used the three nodes. Now you can open the `pi_mpi.c` and deactivate the code of the "Hello..." printing, removing the definition of `_DEBUG_` at the beginning of the `pi_mpi.c` code.

3. Now you can do some scale analysis with different number of nodes (changin `--nodes=` and `--ntasks=` from 1 to 4 in the `submit-mpi.sh` script)[1].

   Report Question 12: *Does the MPI code scale with the different number of nodes?*

4. Try now increasing the number of tasks (`--ntasks=`) in the submit script. This increases the number of processes computed in each node (in a similar way to what we did in the interactive node).

   Report Question 13: *Does the code scale well with the number of processes executed when using different nodes? Up to which number of processes?*

3.3 Combining `OpenMP` and `MPI`

Now you should introduce `OpenMP` code in the `MPI` code so that you can perform `OpenMP` + `MPI` hybrid execution. This will give you the opportunity to exploit the parallelism in more than one node. Note that by default the `OpenMP` execution uses the maximum number of threads in the node so in order to do a scalability analisys you will need to specify the amount of threads that you want to use. You can follow the method used in `submit-mpi-omp.sh` script available in the `mpi` directory.

1. Copy the `pi_mpi.c` file into a new `pi_mpi_omp.c` file. Modify it with the necessary code to execute codes inside each node using `OpenMP`. You can inspire yourself in the `OpenMP` code already provided to you in the `openmp` directory.

2. Compile the code with `make pi_mpi_omp`. The `Makefile` file already has the proper target defined. Although you can modify the `Makefile` file it is not necessary to do so, it should work as is.

3. Test your code with different number of nodes (up to 4) and threads per node (up to 40).

   Report Question 14: *Make a graph with the results. Can you use effectively all the resources in the system? What speed-up are you able to obtain? What can you say about this approach compared with the previous ones used (only OpenMP or only MPI)?*

[1] Hint: you can set these values directly as parameters in the `sbatch` command.

Part 4

Tracing the execution of Sequential and OpenMP programs

The objective of this part is to present you the environment that will be used to gather information about the execution of a parallel application in `OpenMP` and visualize it. The environment is mainly composed of `Extrae` and `Paraver`. `Extrae` provides an API (application programming interface) to manually define in the source code points where to emit events. Additionally, `Extrae` transparently instruments the execution of `OpenMP` collecting information about the different states in the execution of a parallel program and the values of the hardware counters available in the architecture, reporting information about the processor and memory accesses. 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 visualize the trace and analyze the execution of the program.

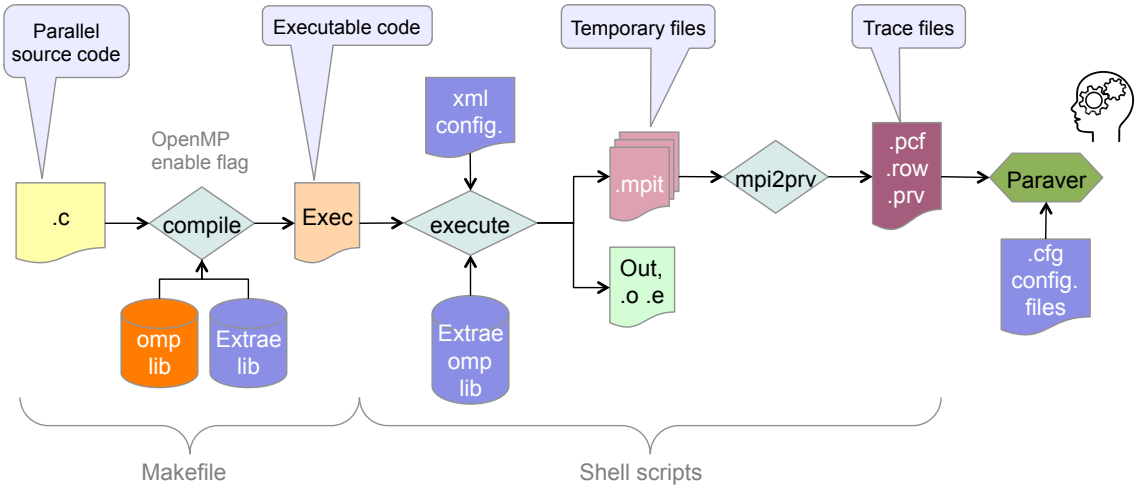

Figure 4.1: Compilation and execution flow for tracing.

4.1 Instrumentation API

Function `Extrae_event(int type, int value)` is used to emit an event in a certain point of the source code. Each event has an associated `type` and a `value`. For example we could use `type` to classify different kind of events in the program and `value` to differentiate different occurrences of the same `type`. In the instrumented codes that we provide you inside the `lab1/tracingomp` directory this function is invoked to trace the entry and exit to different parts of the program (serial and parallel regions). In particular,

a constant value for the `type` argument is used (`PROGRAM` with value 1000) and different values for the `value` argument are used to trace the entry to different program regions, as shown below (defined in the source code):

```
// Extrae Constants
#define  PROGRAM    1000
#define  END        0
#define  SERIAL     1
#define  PARALLEL   2
...
Extrae_event (PROGRAM, PARALLEL);
....
Extrae_event (PROGRAM, END);
...
```

4.2 Trace generation

Next you are going to generate the trace that will be later visualized with `Paraver`.

1. Open any of the source codes provided (`pi_seq.c` or `pi_omp.c`) to observe how the previous instrumentation API is used to identify code regions in the code.

2. Open the `Makefile` and identify the targets to compile the instrumented versions of the both codes. Observe that we specify the location of the `Extrae` include file (`IINCL=-I$(EXTRAE_HOME)/include`) and library (`-ILIBS=-L$(EXTRAE_HOME)/lib -lomptrace`). Make sure that the appropriate flag for compilation of `OpenMP` is applied to them.

3. Compile those two programs with `Makefile`.

4. Open the `submit-seq-i.sh` and `submit-omp-i.sh` scripts to see how the sequential and parallel binaries are executed and traced. Notice that the name of the binary and number of threads to use are specified as a parameter while the number or iterations is specified inside the script file. The script invokes your binary, which will use the `Extrae` library (by using the `LD_PRELOAD` mechanism) to emit events at runtime; the script also invokes `mpi2prv` to generate the final trace (`.prv`, `.pcf` and `.row`) and removes all intermediate files.

5. Submit for execution both `submit-seq-i.sh` and `submit-omp-i.sh` scripts. **BE CAREFULL! Do not submit them at the same time because the trace generation shares TRACE.mpits file.** The execution of the scripts will generate a trace file whose file name includes the name of the executable and the number of threads used for the execution, in addition to the standard output and error output files that you can use to check if the execution and tracing has been correct.

4.3 Short `Paraver` hands–on

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

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 select the trace generated from the instrumented execution of the previous parallel code. Alternatively, traces can be located through the browser at the bottom of the *Main Window*: double clicking on a `.prv` file will load

it. For the purposes of this guided tour, traces mainly contain two types of records: *states* and *flags*. These two kind of records are used by *Extrae* to inject information in the trace.

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

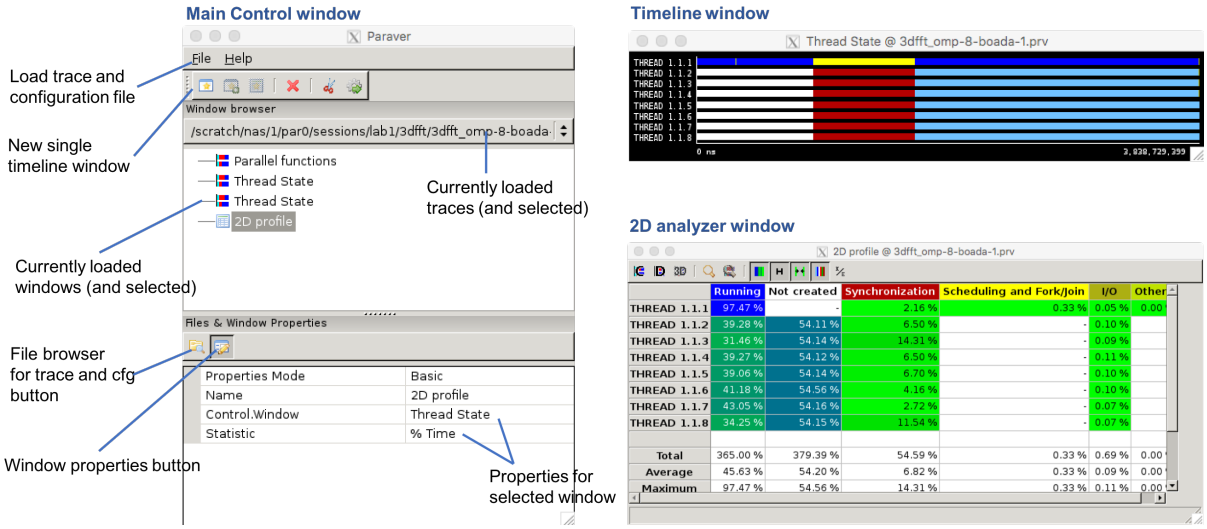

Figure 4.2: Paraver Main Window, Timeline and 2D Analyzer windows.

- 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.
- Textual information: Double click with the left button in your mouse on any point in the window. It will list in textual form the actual value at the point selected and how long the time interval with that colour is. The text display will be in the *What/Where* tab of the Info Panel. *"Right Button → Info Panel"*, can be used to hide the lower info panel.

Spend some time to understand how the parallel execution model in `OpenMP` programs is visualised: a master thread executing the sequential part of the program (with all the other threads not yet created) until the parallel region is reached; at that moment threads are created and start executing and synchronising. Once the parallel region is finished, only the master continues its execution with all other threads remaining idle; but to see all this you need to learn how to zoom in the trace.

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

Now that you know how to zoom in and out, take your time to understand how the fork–join model in `OpenMP` works: zoom into the beginning of the parallel region (yellow part that is setting up the threads and giving them work to do), the red part between dark blue bursts representing the implicit barrier at the end of each `for` work–sharing and parallel region termination (with implicit barrier in red and yellow to indicate the closing of the region), ...

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

Table 4.1: First set of configuration files to support analysis in `Paraver`– upper part: timeline views; lower part: statistical summaries.

4. *Flags* are the other elements in the trace that provide information about the parallel execution. Right-click with your mouse on the window, and select the *"View → Event Flags"* checkbox. In this trace flags appear to signal the entry and exit points of different OpenMP activities (e.g. parallel region and work–sharing constructs). Click on one of the flags and enable the *Event* tick box on the *What/where* tab. Flags are also useful to differentiate different bursts in what may look like a simple burst. Selecting the visualisation of a subset of flags (type and value) and giving them a specific semantic interpretation is possible through the *Main Window* (selecting the second icon in the *Files & Windows* properties panel); however this is not covered in this guided tour.

5. Configuration files are the simplest way to do the analysis of a trace, and specifically of the *Flags*. Next you will use some of them available in different sub-directories inside the `cfgs` directory in your `lab1` directory.

   - For example, configuration file `OMP_parallel_constructs.cfg` (in `cfgs/OpenMP`) can be used to identify when `parallel` constructs are executed. To load this configuration file, from the main menu, select *"File → Load Configuration"*. For this window, red means when the master thread enters into a parallel region.
   - Configuration file `OMP_implicit_tasks.cfg` (also in `cfgs/OpenMP`) can be used to identify when threads in a team execute the implicit task associated to the parallel constructs shown with the previous configuration file. In this window, different colors are used here to visualise different parallel functions. The textual information shows the line number in the source file associated to the `parallel` construct.
   - Open the `OMP_in_barrier.cfg` configuration file to visualise the synchronisation activity (in implicit barriers at the end of parallel regions in this case) in the parallel execution.
   - Open the `OMP_in_schedforkjoin.cfg` configuration file to visualise when the master thread is forking and joining the team of threads in each `parallel` construct.

   The upper part in Table 4.1 lists the configuration files that are available in your home directory inside the `cfgs` directory for doing this kind of analysis. Also you want to test the `User` configuration files to see the user events, specially in the sequential execution.

6. Aligning and synchronising windows: In *Paraver* every timeline window represents a single metric or view for all selected threads and time span. It is possible to align two timelines by making them display the exact same threads and time span. To practise this, just take two windows already open, for example the initial state timeline window and the one opened with `OMP_parallel_functions.cfg`. Right-click inside one of the two windows (the source or reference window) and select *Copy*; then on the target window, right-click and select *"Paste → Default"* (or separately *"Paste → Size"* and *"Paste → Time"*). Both windows will then have the same size and represent different views (metrics) for the same part of the trace. If you put one above the other there is a one to one correspondence between points in vertical. You can also synchronise several windows, by selecting

Synchronise after a Right-click with your mouse on the timeline window; repeat the process for all the windows you want to synchronise. Once synchronised they will continue aligned after zooming, undoing or redoing zoom. With the two windows synchronised you can for example see if the computation bursts in each of the parallel regions take the same time and observe if there is some work unbalance among threads in the different parallel regions.

Report Question 15: *Include in your report a trace image. Remember to include the details of the execution that the trace is showing.*

4.3.2 Profiles

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

1. Load configuration file `OMP_state_profile.cfg`. A table pops up, as the one shown in Figure 4.2 (bottom-right), with one row per thread and one column per OpenMP state (*Running*, *Synchronization*, *Scheduling and Fork/Join*, ...). Each cell value shows the absolute time spent by a thread in a specific state. To see a different statistic change the *Statistic* selector in the *Main Window*. Interesting options at this time may be:

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

2. All the above statistics are computed based on a single timeline window, which is called the *Control Window* and which can be popped up by clicking on the control window icon in the top left corner of the window. In this example, you will see that it is the initial timeline that was opened at the beginning. The values of the control window determine to which column is a given statistic accumulated/accounted. Any of the opened timelines can be selected in *Control Window* selector in the *Main Window*, for example the one associated to `OMP_implicit_tasks.cfg` or to `OMP_in_barrier.cfg`. With that you can answer questions like "how many times a certain parallel construct is executed?", "how much time or which percentage of time each thread has been spent waiting in barriers?", ...

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

Report Question 16: *Include in your report an image of the 2D Analyzer window. Remember to include the details of the execution that the table is summarising.*