PAR Laboratory Assignment
Lab 1: Experimental setup and tools

Ll. Àlvarez, E. Ayguadé, R. M. Badia, J. R. Herrero, J. Morillo and J. Tubella

Fall 2017-18

Contents

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

Session 1

Experimental setup

The objective of this chapter is to become familiar with the environment that will be used during the semester to do the laboratory 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 parXXYY@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 parXXYY@boada.ac.upc.edu passwd"`. Once you are managed to login, you will be placed in your "home" directory in boada, where you will copy all necessary files and do the different laboratory assignments.

All files necessary to do this laboratory assignment are available in `/scratch/nas/1/par0/sessions`. Copy `lab1.tar.gz` **from** that location **to** your home directory in `boada.ac.upc.edu` and uncompress it with this command line: `"tar -zxvf lab1.tar.gz"`. Process the `environment.bash` file with this other command line `"source environment.bash"` in order to set all necessary environment variables. **Note:** since you have to do this every time you login in the account or open a new console window, we recommend that you add this command line in a `.profile` file in your home directory, which is executed when a new session is initiated.

1.1 Node architecture and memory

Execute the `lscpu`, `lstopo` and `"more /proc/meminfo"` commands 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.

Draw the architecture of each node based on the information generated by the tools above. The `"--of fig map.fig"` option for `lstopo` can be very useful for that purpose. Then 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`[1].

1.2 Serial compilation and execution

For this first part of the laboratory assignment you are going to use the `pi_seq.c` source code, which you can find inside the `lab1/pi` directory. `pi_seq.c` performs the computation of the pi number by computing the integral of the equation in Figure 1.1. The equation can be solved by computing the area defined by the function, which at its turn it can be approximated by dividing the area into small rectangles and adding up its area.

[1] In the boada Linux distribution you can use `display` 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"`.

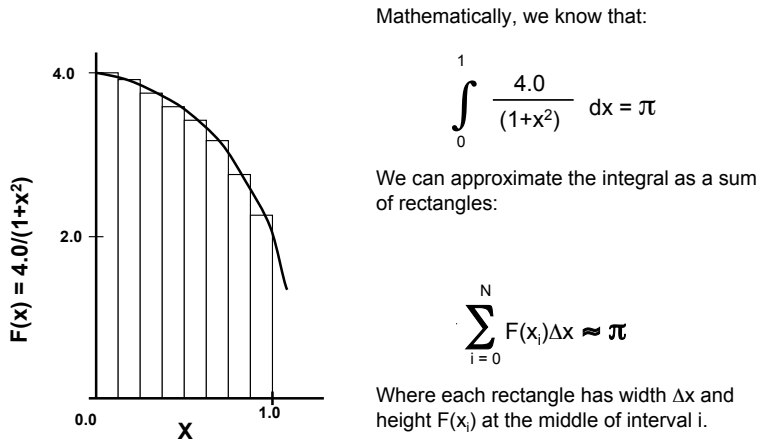

Figure 1.1: Pi computation

Figure 1.2 shows a simplified version of the code you have in `pi_seq.c`. Variable `num_steps` defines the number of rectangles, whose area is computed in each iteration of the `i` loop.

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

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

Figure 1.2: Serial code for Pi

Figure 1.3 shows the compilation and execution flow for a sequential program. We will always compile programs to generate binary executable files through a `Makefile`, with multiple targets that specify the rules to compile each program version.

There are two ways to execute your programs: 1) via a queueing system or 2) interactively. 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. Usually, we will provide scripts for both options (`submit-xxxx.sh` and `run-xxxx.sh`, respectively):

- Queueing a job for execution: `"qsub -l execution submit-xxxx.sh"`. If you do not specify the name of the queue with `"-l execution"` your script will not be run, remaining in the queue forever. In the script you can specify additional options to run the script, configure environment variables and launch the execution of your program. Use `"qstat"` to ask the system about the status of your job submission. You can use `"qdel"` to remove a job from the queueing system.
- Interactive execution: `./run-xxxx.sh`. Additional parameters may be specified after the script name. Jobs interactively executed have a short time limit to be executed.

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

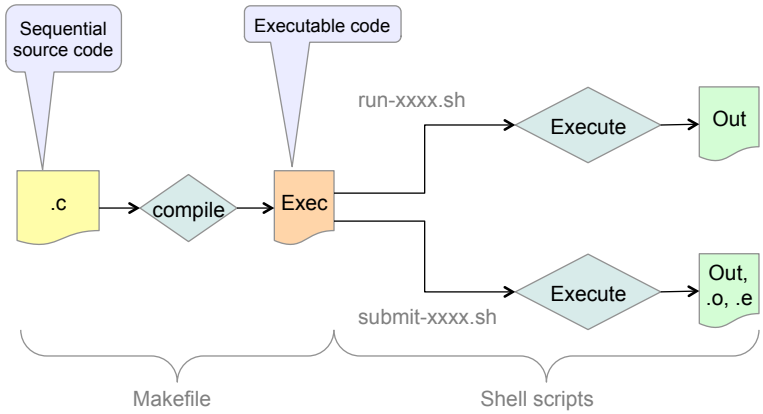

Figure 1.3: Compilation and execution flow for sequential program.

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 number by doing 1.000.000.000 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 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 `"qsub submit-seq.sh"` command. Use `"qstat"` to see that your script is queued; however it is not executed since you have not specified an `execution` queue name. Identify your job-ID number in the `"qstat"` output and use `"qdel job-ID-number"` to remove it from the queue. Submit the execution to the queueing system using the `"qsub -l execution submit-seq.sh"` command and use `"qstat"` to see that your script is 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).

1.3 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 later in this same laboratory assignment, in this section we will see how to compile and execute parallel programs in `OpenMP`. Figure 1.4 shows the compilation and execution flow for an OpenMP program. The main difference with the flow shown in Figure 1.3 is that now the `Makefile` will include the appropriate compilation flag to enable `OpenMP`.

1.3.1 Compiling OpenMP programs

1. In the same `lab1/pi` directory you will find an `OpenMP` version of the code for doing the computation of pi in parallel (`pi_omp.c`). Compile the `OpenMP` code using the appropriate target in the `Makefile`). 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"`).

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`).

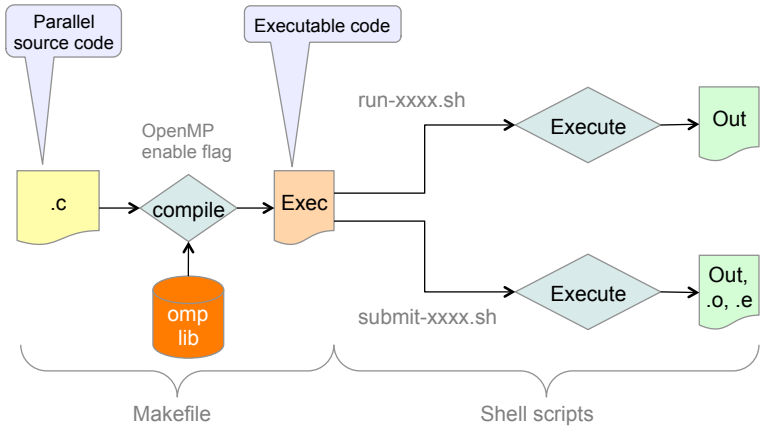

Figure 1.4: Compilation and execution flow for OpenMP.

1.3.2 Executing OpenMP programs

1. Interactively execute the `OpenMP` code with 8 threads (processors) and same number of iterations (1.000.000.000) using the `run-omp.sh` script. What is the `time` command telling you about the user and system CPU time, the elapsed time, and the % of CPU used? Take a look at the script to discover how 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 8 threads in isolation. Do you observe a major difference between the interactive and queued execution?

1.3.3 Strong vs. weak scalability

Finally, in this last part of the chapter 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 12 (`np_NMAX`) threads. The problem size for strong scalability is 1.000.000.000 iterations; for weak scalability, the initial problem size is 100.000.000 which grows proportionally with the number of threads. As a result the script generates a plot (in Postscript format) showing the resulting execution time and speed–up. 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[2] the plots generated and reason about how the speed–up changes with the number of threads in the two scenarios.

[2]In the boada Linux distribution you can use the ghostscript `gs` command to visualize Postscript files or convert the files to PDF using the `ps2pdf` command and use `display` to visualize PDF files.

Session 2

Analysis of task decompositions using Tareador

In this chapter we will introduce *Tareador*, an environment to analyze the potential parallelism that could be obtained when a certain parallelization strategy (task decomposition) is applied to your sequential code. *Tareador* 1) traces the execution of the program based on the specification of potential tasks to be run in parallel, 2) records all static/dynamic data allocations and memory accesses in order to build the task dependence graph, and 3) simulates the parallel execution of the tasks on a certain number of processors in order to estimate the potential speed-up. Figure 2.1 shows the compilation and execution flow for *Tareador*, starting from the taskified source code.

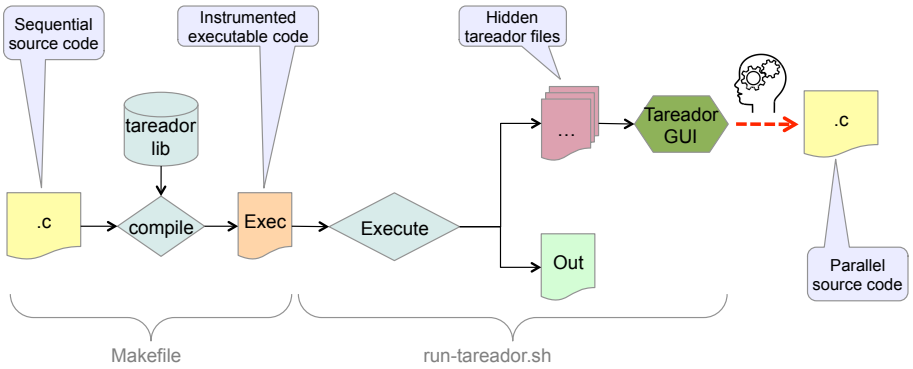

Figure 2.1: Compilation and execution flow for Tareador.

2.1 Using *Tareador*

Tareador offers an API 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.

For the first part of this session you will use a very simple program that performs the computation of the dot product (`result`) of two vectors (`A` and `B`) of size 16. The program first initializes both vectors and then calls function `dot_product` in order to perform the actual computation:

1. Go into the `lab1/dot_product` directory, open the `dot_product.c` source code and identify the calls to the instrumentation functions mentioned above. Open the `Makefile` to understand how the source code is compiled and linked to produce the executable. Generate the executable by doing ("`make dot_product`").

2. Execute the *Tareador* environment by invoking `./run_tareador.sh dot_product`[1]. This will open a new window in which the task dependence graph is visualized (see Figure 2.2, left). Each node of the graph represents a task: different shapes and colors are used to identify task instances generated from the same task definition and each one labeled with a task instance number. In addition, each node contains the number of instructions that the task instance has executed, as an indication of the task granularity; the size of the node also reflects in some way this task granularity. Edges in the graph represent dependencies between task instances; different colors/patterns are used to represent different kind of dependences (blue: data dependences, green/yellow: control dependences).

3. *Tareador* allows you to analyze the data that create the data dependences between nodes in the task graph. With the mouse on an edge (for example the edge going from the red task (`init_B`) to the yellow task (`dot_product`), right click with the mouse and select *Dataview* → *edge*. This will open a window similar to the one shown in Figure 2.2, right/top. In the *Real dependency* tab, you can see the variables that cause that dependence. Also you can right click with the mouse on a task (for example `dot_product`) and select *Dataview* → *Edges-in*. This will open a window similar to the one shown in Figure 2.2, right/middle. 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) by the task selected (for example `dot_product`) and written (i.e. with a store memory access, blue color in the window) by any of the tasks that are source of a dependence with it (in this case, either `init_A` or `init_B` as selected in the chooser). In this tab, the orange color is used to represent data that is written by the source task and read by the destination task, i.e. a data dependence. 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) performed by each task.

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

5. Once you understand the data dependences and the task graph generated, you can simulate the execution of the initial task decomposition, for example with 4 processors, by clicking *View Simulation* in the main *Tareador* window. This will open a *Paraver* window showing the timeline for the simulated execution, similar to the one shown in Figure 2.2, right/bottom. In fact you will have to zoom into the initial part of the trace in order to visualize the same part of the trace; you can do this by clicking the left button in yiur mouse and selecting the zone you want to zoom. Colors are used to represent the different tasks (same colors that are used in the task graph). In the next laboratory session you will learn more details about the usage of *Paraver*; for example, you can activate the visualization of dependencies between tasks by selecting View → Communication Lines when you click the right button of the mouse.

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

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

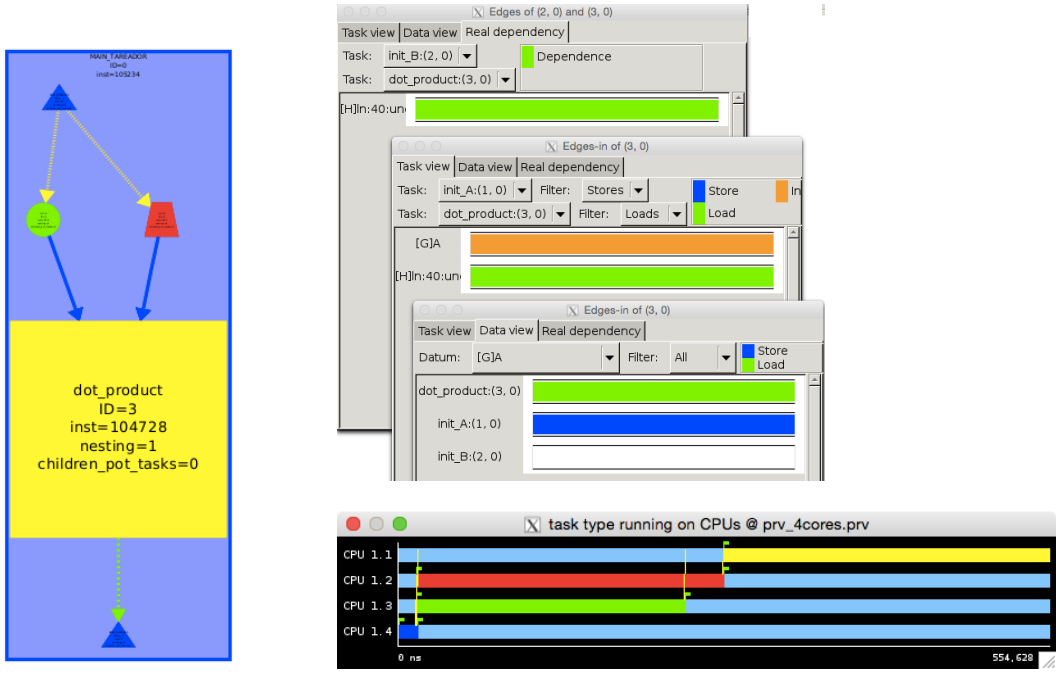

Figure 2.2: Left: Initial task dependence graph for `dot_product`. Right/Top: Visualization of data involved in task dependencies. Right/Bottom: *Paraver* visualization of the simulated execution with 4 processors, after zooming into the initial part of the trace.

Next you will refine the initial task decomposition in order to exploit additional parallelism inside the computation of the `dot_product` task:

5. Edit the source code `dot_product.c` and use `tareador_start_task` and `tareador_end_task` to identify as a potential task each iteration of the loop inside the `dot_product` function. Compile the source code and execute *Tareador* in order to visualize the task dependence graph and simulate the execution with 4 processors. Do you observe any improvement in the parallelism that is obtained?

6. With the *Dataview* option in *Tareador* identify the variables that are sequentializing the execution of the new tasks and locate the instructions in the source code where these dependences are created. Once you know which variables are causing these dependences, you can temporarily filter their analysis by 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 can remove certain dependences from the analysis, assuming that you will use the appropriate mechanisms in the parallel code to actually remove them. Insert these calls into the source code in order to disable the variables that you have identified. Compile and run *Tareador* again. Are you increasing the parallelism? Perform the simulation with different number of processors and observe how the execution time changes. How will you handle the dependences caused by the accesses to these variables?

2.2 Exploring task decompositions

Go into the `lab1/3dfft` directory, open the `3dfft_seq.c` source code and identify the calls to the *Tareador* API, understanding the tasks that are initially defined.

1. Generate the executable and instrument it with the "`./run_tareador`" script. Analyze the task dependence graph and the dependences that are visualized, using the *Dataview* option in *Tareador*.

2. Next you will be refining the potential tasks with the objective of discovering more parallelism in `3dfft_seq.c`. You will incrementally generate four new task decompositions (named v1, v2, v3 and v4) as described in the following bullets. For the original and the four new decompositions compute T_1, T_∞ and the potential parallelism from the task dependence graph generated by *Tareador*, assuming that each instruction takes one time unit to execute.

   (a) Version v1: REPLACE[2] the task named `ffts1_and_transpositions` with a sequence of finer grained tasks, one for each function invocation inside it.

   (b) 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");
    }
}
```

   (c) 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`.

   (d) Version v4: starting from v3, propose which should be the next task(s) to decompose with fine-grained tasks?. Modify the source code to instrument this task decomposition.

For version v4, simulate the parallel execution for 2, 4, 8, 16 and 32. For the original (`seq`) decomposition, simulate its execution time with just 1 processor; the time reported in the trace will be used to compute the speed-up obtained by v4 when using different numbers of processors, as requested in the first deliverable.

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

Session 3

Tracing the execution of programs

The objective of this chapter 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 complete compilation and execution flow for tracing is shown in Figure 3.1. 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 status of each thread and different events related with the execution of the parallel program[1]. The `Extrae` library is appropriately set in the `Makefile` and 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 visualize the trace and analyze the execution of the program.

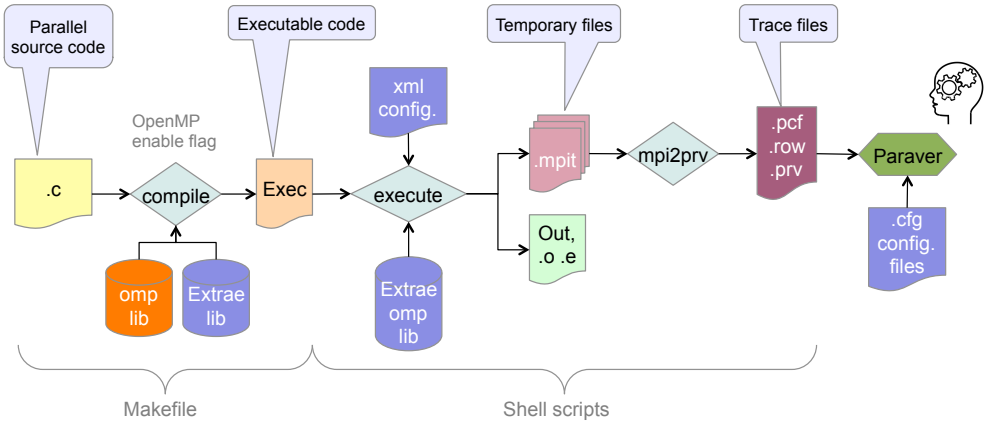

Figure 3.1: Compilation and execution flow for tracing.

3.1 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/pi` 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):

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

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

3.2 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. Compile those two programs with `Makefile`.

3. 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, number of iterations and number of threads to use are specified inside the script file. The script invokes your binary, which will use the `Extrae` library (by using the `LD_PRELOAD` mechanism) to emit evens at runtime; the script also invokes `mpi2prv` to generate the final trace (`.prv`, `.pcf` and `.row`) and removes all intermediate files.

4. Submit for execution both `submit-seq-i.sh` and `submit-omp-i.sh` scripts. The execution of the scripts will generate a trace file whose file name includes the name of the executable, the size of the problem[2] 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.

3.3 Paraver hands on

In this laboratory session your professor will do a hands-on to show the main features of `Paraver`, a graphical browser of the traces generated with `Extrae`, and the set of configuration files to be used to visualize and analyze the execution of your program. A guide for this hands-on can be found in the *Intro2ParaverPAR.pdf* document available through Atenea.

Configuration files are available in your home directory inside the `cfgs` directory. Some of them are used to visualize timelines showing different aspects of the parallel execution, as briefly described in the following table.

[2] Use 100.000.000 iterations for both the sequential and parallel execution.

User events	Timeline showing ...
APP_userevents	type 1000 events manually introduced by programmer

OpenMP events	Timeline showing ...
OMP_parallel_functions	the parallel function each thread is executing
OMP_parallel_functions_duration	the duration for the parallel functions
OMP_worksharings	when threads are executing worksharing regions (for or single)
OMP_worksharings_duration	the duration of worksharing regions
OMP_in_barrier	when threads are in a barrier synchronization
OMP_in_lock	when threads are in/out/entering/exiting critical sections
OMP_in_schedforkjoin	when threads are scheduling work, forking or joining
OMP_in_ordered	when threads are executing ordered sections inside loops

Other configuration files compute statistics (profiles) about the parallel execution, as briefly described in the following table.

User events	Profile showing ...
APP_userevents_profile	the duration for type 1000 events

OpenMP events	Profile showing ...
OMP_state_profile	the time spent in different OpenMP states (useful, scheduling/fork/join, synchronization, ...)
OMP_critical_profile	the total time, percentage of time, number of instances or average duration spent in the different phases of a critical section
OMP_critical_duration_histogram	histogram of the duration of the different phases of the critical section
OMP_ordered_profile	the total time, percentage of time, number of instances or average duration spent in the different phases of an ordered section

Later in the course we will use other configuration files to show timelines related with the creation and execution of tasks and their synchronization constructs.

OpenMP events	Timeline showing ...
OMP_tasks/task_instantiation	when tasks are created
OMP_tasks/taskloop	when tasks in a taskloop are created
OMP_tasks/task_execution	when tasks (individual or from taskloop) are executed
OMP_tasks/useful_task_execution	useful time during task execution
OMP_tasks/taskid_instantiation	task identifier at instatiation time
OMP_tasks/taskid_execution	task identifier at execution time
OMP_tasks/taskwait	when taskwait synchronization constructs are executed
OMP_tasks/in_taskgroup	taskgroup regions
OMP_tasks/taskgroup	when taskgroup synchronization is executed

Finally, other configuration files related with *Tareador* that you already used in the previous laboratory session.

3.4 Visualizing and analyzing the trace

Using `Paraver` and the appropriate configuration files, answer the following questions:

1. Open the trace generated for `pi_seq.c` and visualize the entry and exit to the serial and potential parallel regions in this code. Compute the *parallel fraction* ϕ from this timeline.
2. Open the trace generated for `pi_omp.c` and try to understand how the parallel execution evolves through the different execution states (fork/join, scheduling, useful work, different sorts of synchronization, ...). Then use the appropriate configuration file to display a table with the % of time spent in the different OpenMP states when using 8 threads.

Deliverable

After the last session for this laboratory assignment, and before starting the next one, you will have to deliver a **report** in `PDF` format (other formats will not be accepted) containing the answers to the following questions. Your professor will open the assignment at the Raco website and set the appropriate delivery dates for the delivery. Only one file has to be submitted per group through the Raco website.

Important: In the front cover of the document, please clearly state the name of all components of the group, the identifier of the group (username `parXXYY`), title of the assignment, date, academic course/semester, ... and any other information you consider necessary.

As part of the document, you can include any code fragment, figure or plot you need to support your explanations. In case you need to transfer files from boada to your local machine (laptop or desktop in laboratory room), or viceversa, you can use the secure copy `scp` command. For example `"scp parXXYY@boada.ac.upc.edu:lab1/foobar.txt local/directory/."` executed in your local machine to copy file `foo.txt` inside directory `lab1` in your home directory of boada to directory `local/directory/` with the same name.

Node architecture and memory

1. Draw and briefly describe the architecture of the computer in which you are doing this lab session (number of sockets, cores per socket, threads per core, cache hierarchy size and sharing, and amount of main memory).

Timing sequential and parallel executions

2. Describe what do you need to add to your program to measure the elapsed execution time between a pair of points in the program, clearly indicating the library header file that needs to be included, the library functions that need to be invoked, the data structure and its fields.

3. Plot the speed–up obtained when varying the number of threads (strong scalability) and problem size (weak scalability) for `pi_omp.c`. Reason about how the scalability of the program.

Visualizing the task graph and data dependences

4. Include the source code for function `dot_product` in which you show the *Tareador* instrumentation that has been added to study the potential parallelism in the code. This instrumentation has to appropriately define tasks and filter the analysis of variable(s) that cause the dependence(s).

5. Capture the task dependence graph for that task decomposition and the execution timelines (for 8 processors) that allow you to understand the potential parallelism attainable. Briefly comment the relevant information that is reported by the tools.

Analysis of task decompositions

6. Complete the following table for the initial and different versions generated for `3dfft_seq.c`, briefly commenting the evolution of the metrics with the different versions.

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

7. With the results from the parallel simulation with 2, 4, 8, 16 and 32 processors, draw the execution time and speedup plots for version `v4` with respect to the sequential execution (that you can estimate from the simulation of the initial task decomposition that we provided in `3dfft_seq.c`, using just 1 processor). Briefly comment the scalability behaviour shown on these two plots.

Tracing sequential and parallel executions

8. From the instrumented version of `pi_seq.c`, and using the appropriate `Paraver` configuration file, obtain the value of the *parallel fraction* ϕ for this program when executed with 100.000.000 iterations, showing the steps you followed to obtain it. Clearly indicate which `Paraver` configuration file(s) did you use.

9. From the instrumented version of `pi_omp.c`, and using the appropriate `Paraver` configuration file, show a profile of the % of time spent in the different OpenMP states when using 8 threads and for 100.000.000 iterations. Clearly indicate which `Paraver` configuration file(s) did you use and your own conclusions from that profile.