PAR - lab1

Rosa M Badia

rosa.maria.badia@upc.edu

Sessió 1

Login and set-up

- Lab instructions: from Atenea, after the theory documents
- Login:
  - ssh -X par41YY@boada.ac.upc.edu
  - Change password:
    - ssh -t parXXYY@boada.ac.upc.edu passwd
- Lab material posted in /scratch/nas/1/par0/sessions
  - cp /scratch/nas/1/par0/sessions/lab1.tar.gz .
- Unpack
  - tar -zxvf lab1.tar.gz
- ALWAYS
  - source ~/environment.bash
  - Add this line to your .bashrc
    - Source $HOME/environment.bash
- Copying files
  - scp parXXYY@boada.ac.upc.edu:./lab1/pi/pi seq.c .

Node architecture

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

- Run in directory lab1/arch:
  - sbatch submit-arch.sh
- This will run the lscpu and lstopo commands
- This will generate (where number can be 11, 12, 13 or 14):
  - lscpu-boada-number
  - lstopo-boada-number
  - map-boada-number.fig.
- Use the xfig command to visualize the output file generated map-boada-number.fig
  - Using File->export save a PDF or JPG version
- For the deliverable:
  - Fill the table in deliverable about the node architecure

Executions

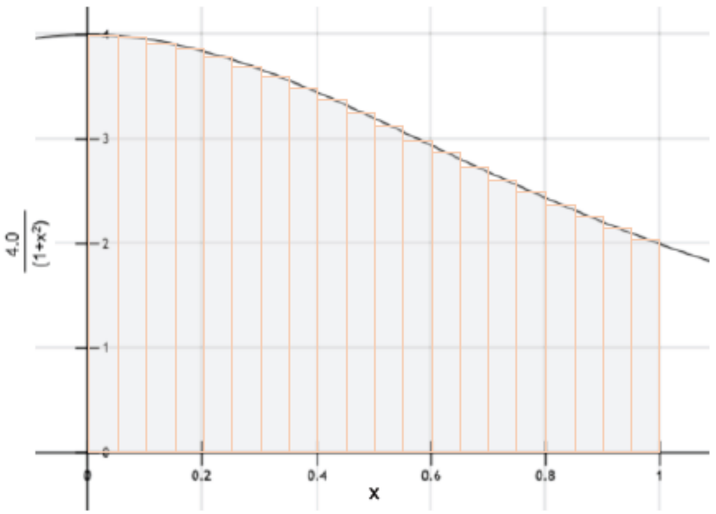
- Queueing jobs (nodes boada-11 to boada-14):
  - sbatch [-p partition] ./submit-xxxx.sh
    - (if partition is not specified will run in the “execution” partition)
  - Exclusive access to the node
  - MANDATORY for performance scripts
    and long running jobs
  - squeue – to check the queue
  - scancel – to cancel a job
- Interactive runs:
  - ./run-xxxx.sh
  - Share resources with other users

Computing pi

```
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



Mathematically, we know that:

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

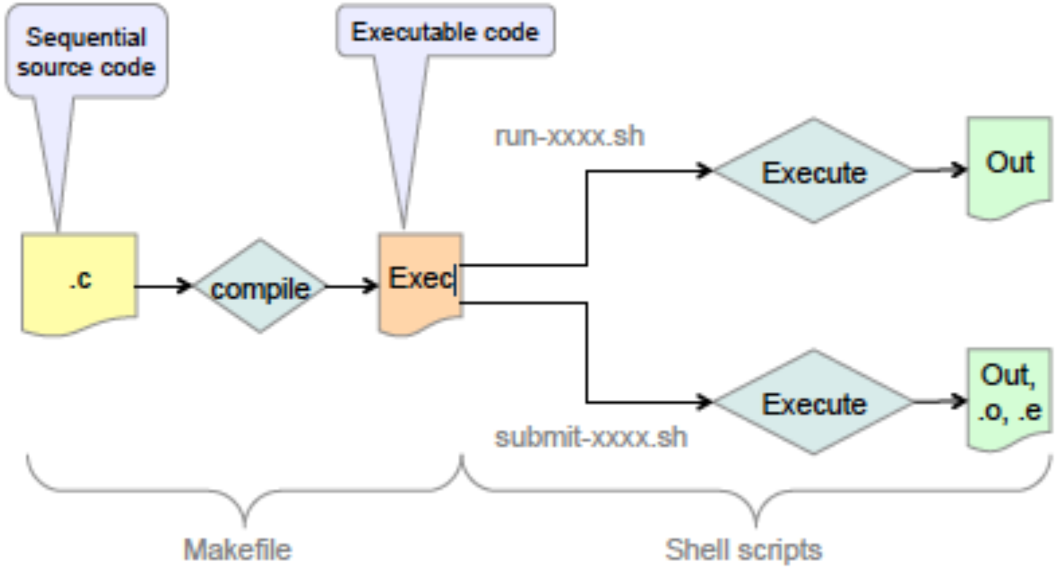
We can approximate the integral as a sum of rectangles:

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

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

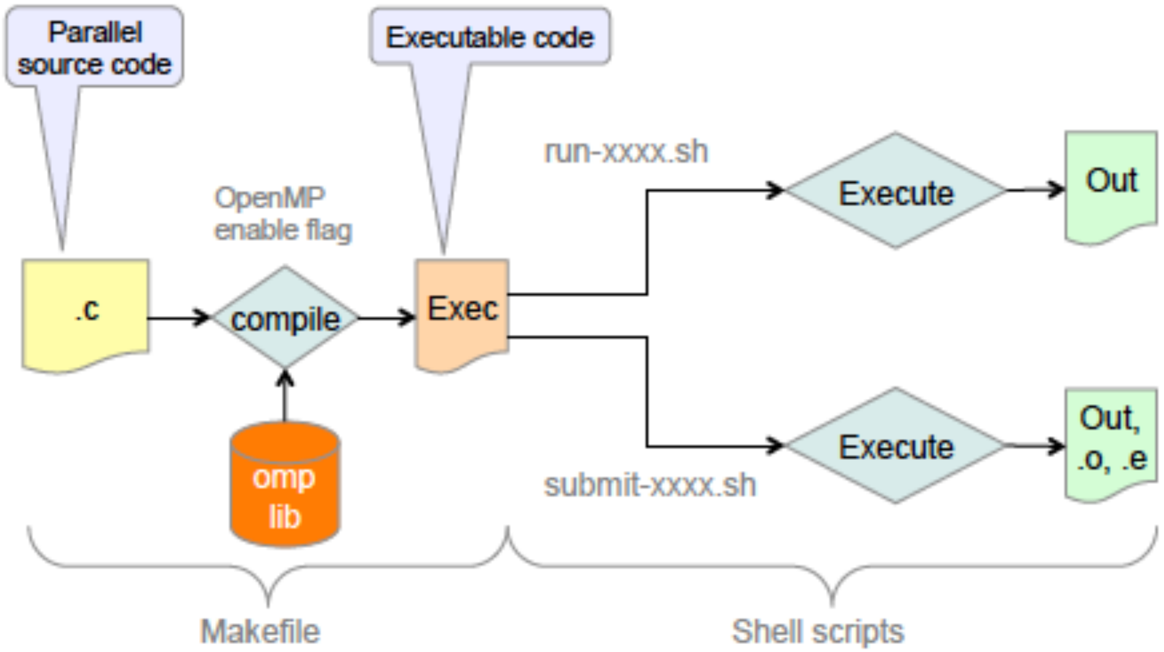
Compilation of Pi

- icc compiler
- Type: icc –v
- Open Makefile and identify the target to compile, how compiler is invoked
- Execute: make
- Execute: ./run-seq.sh
  - ./run_seq.sh pi_seq 1000000000
- Submit to queues
  - sbatch –p execution submit-seq.sh pi_seq 1000000000



Compilation of OpenMP

- Have a look at pi_omp.c
- Identify target in Makefile
- Run interactively with variable number of threads (1, 2, 4, 8, 16 and 20)
  - ./run-omp.sh
- Execute in queues with submit-omp.sh
  - Look at the time-pi omp-X-boada-Y



Strong versus weak scalability

- Strong scalability: same problem size
- Weak scalability: problem size proportional to number of threads
- Use two scripts IN THE QUEUES!
  - submit-strong-omp.sh
  - submit-weak-omp.sh
- Both scripts generate a chart (postscript)
  - Visualize with gs
- Run again from 20 to 40 threads

Deliverable

- Check what is asked when following the sessions
  - “For the deliverable” sentences
- First session:
  - Fill the table with summary of node architecture
  - Draw a table showing the user and system CPU time, elapsed time and % of CPU used in the two scenarios (interactive and queued)
  - Include scalability plots generated by the strong/weak scaling scripts

Lab1 deliverable due date

- Wednesday September 27th, 15:59 - Atenea
- Deliver a single pdf file, one delivery per group
- Do not forget to add names of group participants in first page

Sessió 2

- Goal: learn to use tareador
- Extract possible parallelism of a code

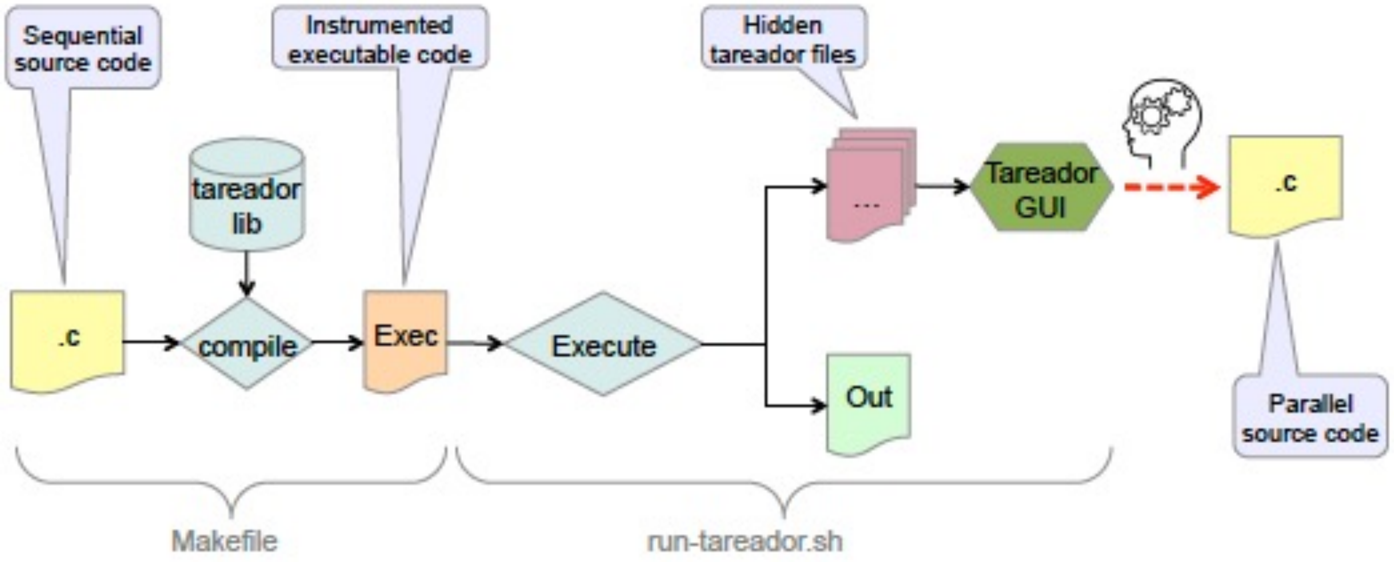


Figure 2.1: Compilation and execution flow for *Tareador*.

Tareador API

- Enabling tareador:

```
tareador_ON();
...
tareador_OFF();
```

- Indicate possible regions that can be a parallel task:

```
tareador_start_task("NameOfTask");

/* Code region to be a potential task */

tareador_end_task("NameOfTask");
```

First example

- FFT: Check the code
  - lab1/3dfft/3dfft_tar.c
- Compile the code
  - make 3ddft_tar
- Execute the binary:
  - ./run-tareador.sh 3dfft_tar

- Follow tareador hands-on

Additional calls to the API

- Disable object from analysis

```
tareador_disable_object(&name_var)

// ... code region with memory accesses to variable name_var

tareador_enable_object(&name_var)
```

Exploring new task decompositions for 3DFFT

- Follow the guide to perform the different v1 : v5 versions
- REPLACE the old task definitions with new ones

- Deliverable:
  - **Task dependence graphs**
  - Table with T1 , T_∞ and parallelism
  - T1 = time with 1 processor
  - T_∞ = time that can be obtained with infinite resources = time of the critical path
  - Parallelism = T1 / T_∞
  - Commented scalability plots for V4 and V5
  - Relevant parts of the code (with regard v5 and v4)

Sessio 3

- Methodology to analyse and improve performance of parallel OpenMP applications
  1. Use modelfactors to analyse overall performance and scalability, find parallel fraction (ϕ) and parallelisation efficiency
  2. Using Paraver, analyse traces to diagnose performance inefficiencies
  3. Modify code and repeat

Session 3

- Generation and analysis of Paraver tracefiles

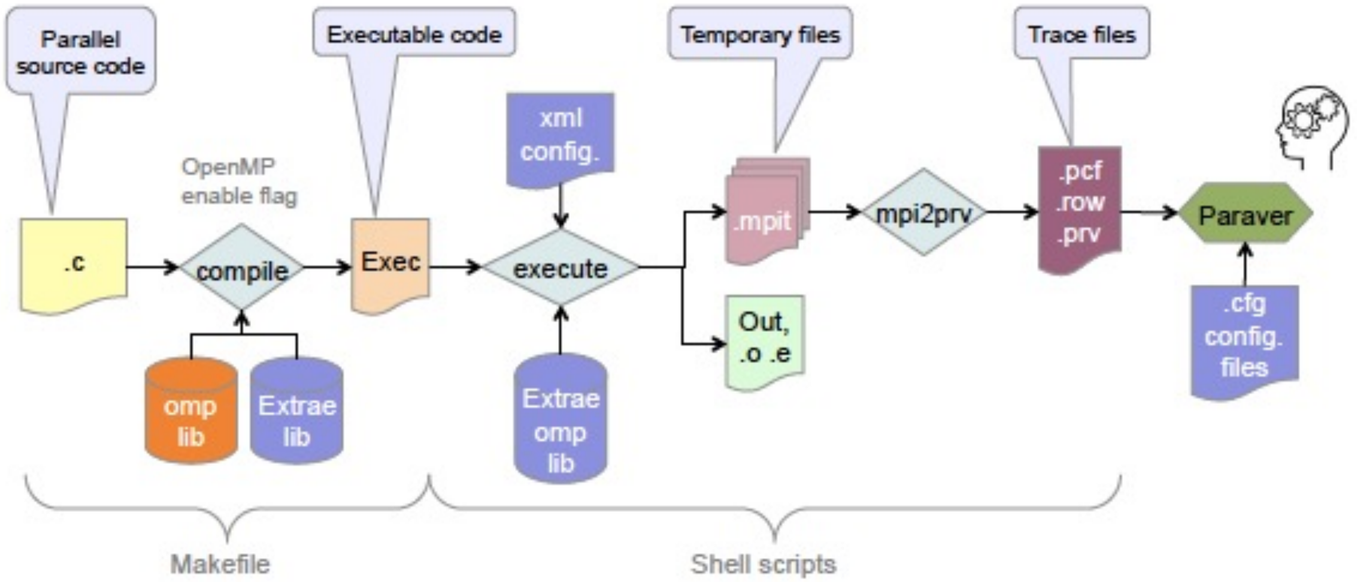


Figure 3.1: Compilation and execution flow for tracing.

- Generate traces in queues
- Modelfactors generates traces automatically for different number of processors

modelfactors

- Python program that analyse a set of traces and generates a set of metrics
  - Use the following script: submit-strong-extrae.sh
- Output of modelfactors:
  - modelfactors.out -> three different tables
  - modelfactors.pdf → pdf document with all the tables
  - INCLUDE THE TABLES + CAPTIONS IN YOUR REPORTS

Table 1: summary

- Overview of the program execution
  - Speedup = T1 / Tp
  - Efficiency = Sp/p

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

Table 2: efficiency for parallel function ϕ

- Only for the part of the code that has been parallelised

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

$\times$

$\times$

$\times$

The higher the better!

Table 2

- Parallelization efficiency: how well the code has been parallelised
  - In execution efficiency: low value means that there are overheads due to work generation and synchronisation in the critical path
  - Load balancing: reflects if some threads are not doing real work
- Scalability for computation tasks: how well the processors are executing the tasks
  - IPC scalability: is <100, parallel version has worst IPC than sequential version
  - Instruction scalability: is <100, parallel version of code is executing more instructions than sequential version
  - Frequency: lower frequency in parallel version vs sequential version

Table 3

- Information about explicit tasks

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

Preliminar analysis

- lab1/3dfft
- Check 3dfft_omp.c file and the parallel regions that are created
  - parallel, single, taskloop
- Compile and run:
  - make 3dfft_omp
  - sbatch -p execution submit-strong-extrae.sh 3dfft_omp
- squeue
  - Shows queue status
  - watch squeue
- Folder 3dfft_omp-strong-extrae
- xpdf modelfactor-tables.pdf
- Parallel fraction = Tpar / (Tseq + Tpar)

Short paraver hands-on

- Basics
  - Zoom (and undo zoom and redo zoom)
  - Fit time scale
  - Select a square section in the trace
  - Undo zoom
- Events
  - Green flags indicate beginning and end of certain events -> parallel region, ...
  - Type and value
  - Switch ”text” option to see actual type and value
- Colors
  - Representation issue: only one color can be represented per pixel
  - Zoom to get details

Paraver hints

- Hints or workspaces
  - Main menu of the Main window
  - Open OpenMP/thread_state_profile
    - Observe %time in each state
    - Change Statistic, i.e. to Time
- Flags
  - Type and value
  - View -> Event flags
  - Click on a event and see in the What/where window its type and value

Detailed analysis

- Open User_functions -> User_functions
  - Each function is encoded with a different colour
- Aligning windows
  - Copy
  - Paste->time
  - Paste->size
- Synchronizing windows
  - Right button of the mouse -> synchronize
- Easy to see, for example, if different parallel regions take the same time

Detailed analysis

- Open Hint->OpenMP->implicit tasks duration.
  - Show implicit tasks duration as a gradient
- Open Hint-> OpenMP tasking->explicit tasks duration.
  - Shows duration of explicit tasks executed by the threads
  - Different granularity (tasks generated by taskloops)
  - Do all tasks in a taskloop have the same duration?
- Open Hint -> OpenMP->Histogram of Implicit task duration
  - Shows a distribution of the different implicit task durations
- Open Hint -> OpenMP tasking-> Histogram of explicit execution task duration
  - distribution of the different explicit task durations
- Fine versus coarse granularity parallelism

Optimization

- Move folder 3dfft_omp-strong-extrae to another name
  - Be carefull with disk quota...
- Change taskloop granularity
  - In each function, comment internal taskloop and uncomment external
- Compile and run again
  - make 3dfft_omp
  - sbatch -p execution submit-strong-extrae.sh 3dfft_omp
- New version of modelfactors tables
  - Compare with previous? Has changed?
  - What happens now with 12 cores or more?

Detailed analysis

- Open again tracefile with 12 processors
  - Open New single timeline window
  - Open User Functions
- Compare with original tracefile (before optimization)
  - Compare duration
  - Obtain the histogram with the durations of the different explicit and implicit tasks
- Do you observe any major difference on the duration of the implicit and explicit tasks for the initial and optimized versions?

Further optimization

- Parallelize the function init_complex_grid
- Run again the model factors script
- Do the analysis again

Deliverable

- Check what is said in the document ("*For the deliverable*")
- Deliver pdf on Atenea
  - Do not forget to add the names in the first page
  - Do not add figures/captures that you are not going to comment
  - Maximum extension 3000 words
- Deadline:
  - Tuesday 27th at 15.59

Deliverable

- Session 1
  - Fill the table with summary of node architecture
  - Draw a table showing the user and system CPU time, elapsed time and % of CPU used in the two scenarios (interactive and queued)
  - Include scalability plots generated by the strong/weak scaling scripts
- Session 2
  - Analysis of the task decompositions with 3DFFT
- Session 3
  - Comment on the evolution of the three versions
  - Table, paraver captures (with **comments**) and strong scalability plots (with comments)
  - ϕ = Tpar / (Tseq +Tpar)