Lab1 deliverable

Daniel Sattler and Cristina Raluca Vijulie par1207

October 13, 2017

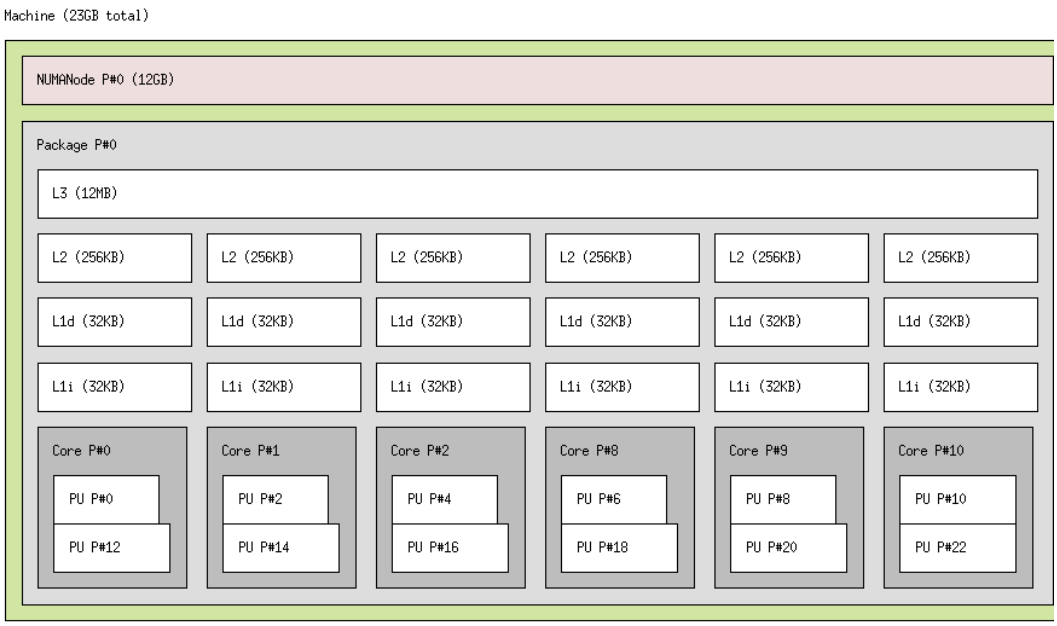
Node architecture and memory

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).

(a) sockets: 2

(b) cores/socket: 6

(c) threads: 2



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.

To measure the time we need to get a timestamp with the time of the day when the execution starts and subtract that value to the time of day when the program ends. To do so we use the following:

- The library:

```
#include <sys/time.h>
```

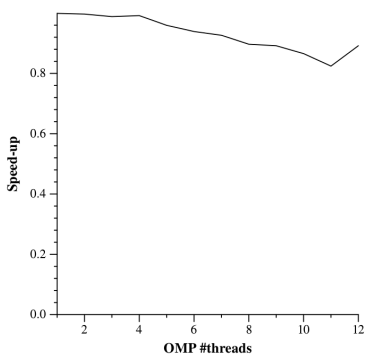
- The function:

```
int  gettimeofday(struct timeval *, void *);
```

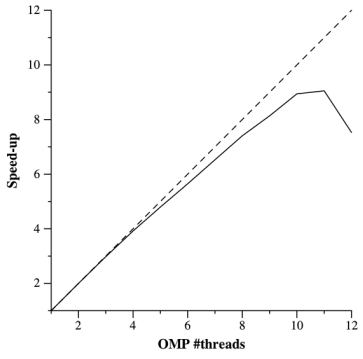
- The data structure **timeval**, which includes the following fields:

```
time_t          tv_sec      seconds
suseconds_t     tv_usec     microseconds
```

3. Plot the speedup obtained when varying the number of threads (strong scalability) and problemsize (weak scalability) for pi omp.c. Reason about how the scalability of the program.



Speed-up wrt one thread, weak scaling



Speed-up wrt sequential time

When we are in weak scalability, the speedup does not change so much, it means the scalability is good and we can keep increasing the problem size while increasing the threads and the execution time will not be much greater. In the case of strong scalability, we can see that more than 11 threads start to decrease the speed-up, so it does not have a great strong scalability, with 11 as the optimal number of threads.

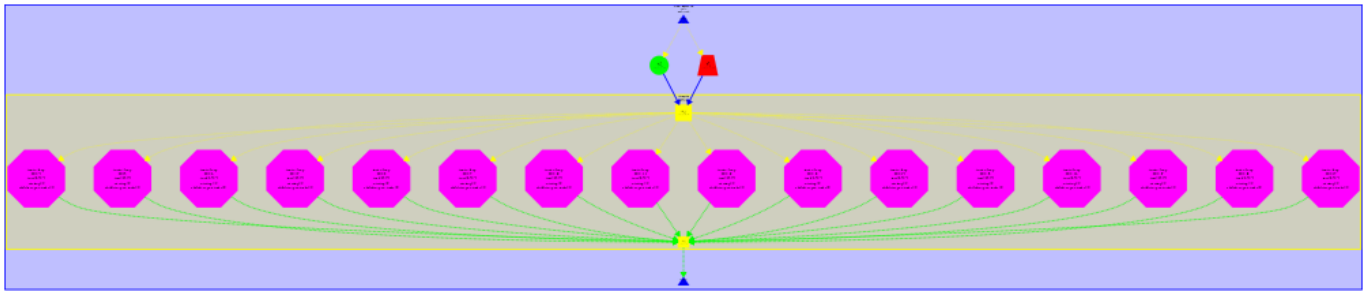
Visualizing the task graph and data dependences

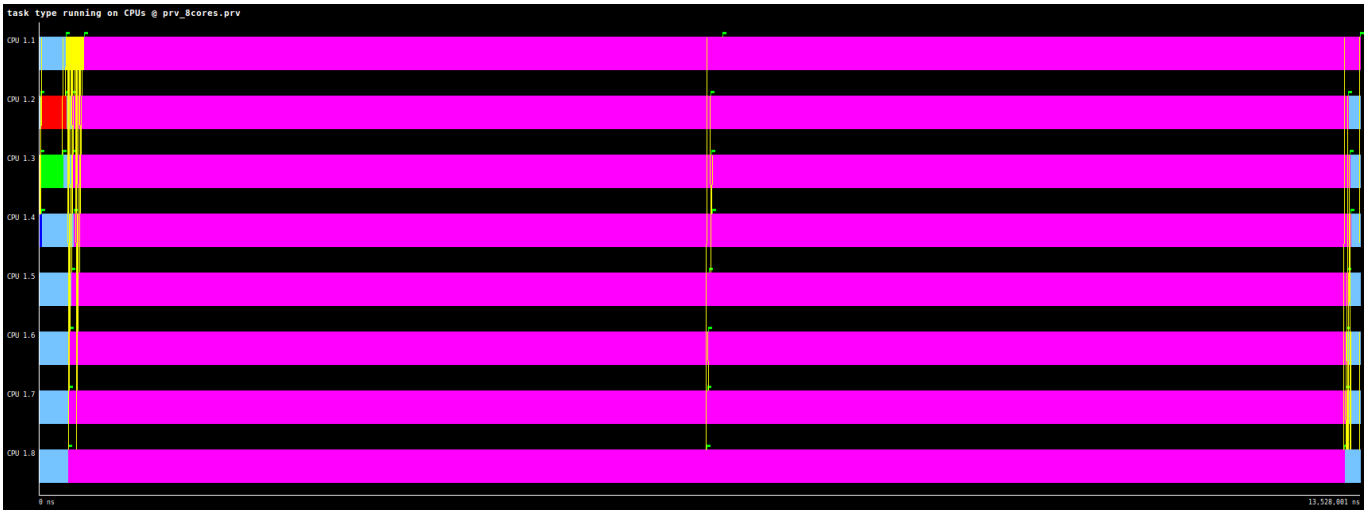
4. Include the source code for function dot product in which you show the Tareador instrumentation that has been added to study the potential parallelism in the code. This

instrumentation has to appropriately define tasks and filter the analysis of variable(s) that cause the dependence(s).

The source code has been added to the archive and is named dot_product.c

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	593772	593758	1.000
v1	593772	593758	1.000
v2	594054	315537	1.883
v3	594054	108701	5.465
v4	594054	59694	9.952

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.

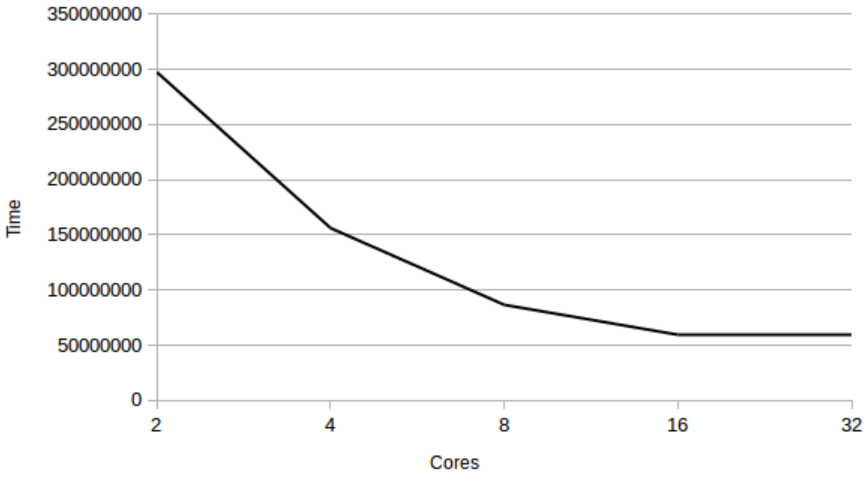


Figure 1: Execution time in ns

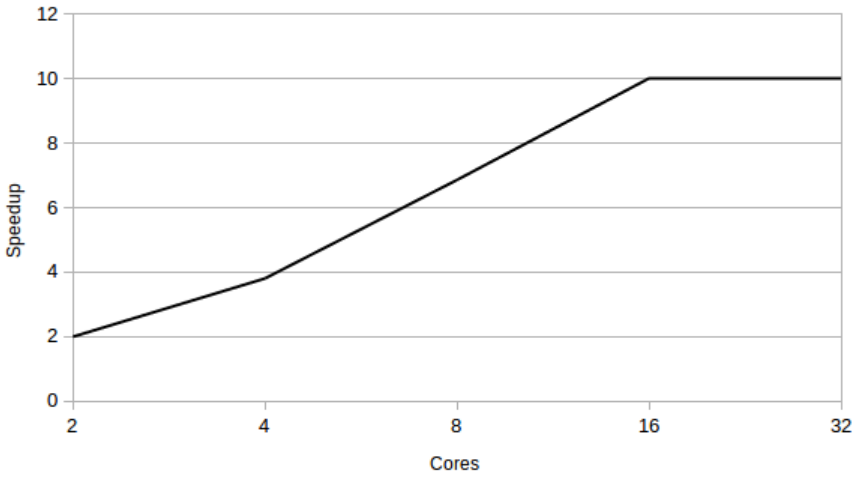
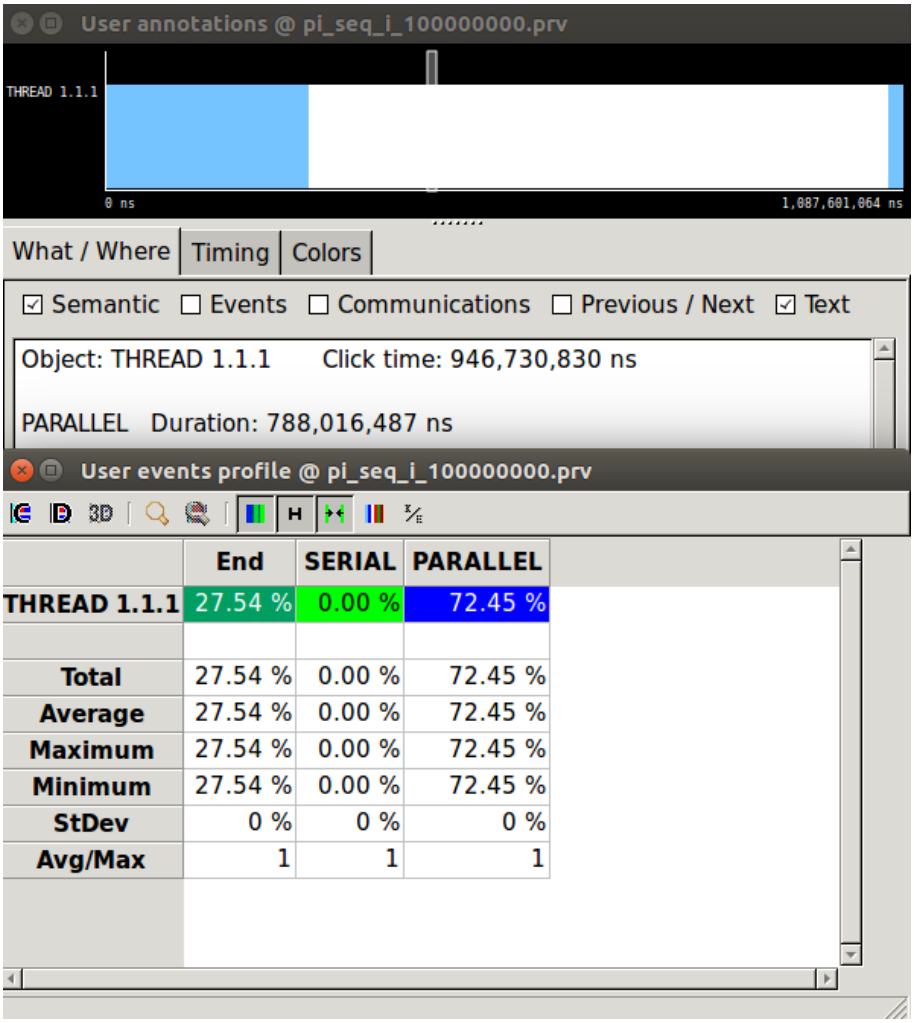


Figure 2: Speedup v4 vs. seq

As we can see in the plots, the scalability is bad. The maximum number of cores we can use at the same time are 16, more than that do not help the execution time.

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 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.



	End	SERIAL	PARALLEL
THREAD 1.1.1	27.54 %	0.00 %	72.45 %
Total	27.54 %	0.00 %	72.45 %
Average	27.54 %	0.00 %	72.45 %
Maximum	27.54 %	0.00 %	72.45 %
Minimum	27.54 %	0.00 %	72.45 %
StDev	0 %	0 %	0 %
Avg/Max	1	1	1

Figure 3: Parallel fraction of the code

The parallel fraction of this program is the parallel duration * 100 / the total duration. The value of the fraction is: 58.4% This was obtained using the APP_userevents configuration.

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.

OpenMP Statistics @ pi_omp_i_100000000_8.prv

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	400,116,985 ns	-	95,696,104 ns	7,046,914 ns	35,325 ns	2,340 ns
THREAD 1.1.2	196,674,779 ns	286,782,307 ns	27,562 ns	-	9,640 ns	-
THREAD 1.1.3	127,916,627 ns	286,825,420 ns	68,842,150 ns	-	6,485 ns	-
THREAD 1.1.4	131,808,203 ns	286,825,448 ns	64,937,458 ns	-	5,935 ns	-
THREAD 1.1.5	171,757,636 ns	286,740,269 ns	25,091,412 ns	-	5,600 ns	-
THREAD 1.1.6	127,851,843 ns	286,779,100 ns	68,944,797 ns	-	5,648 ns	-
THREAD 1.1.7	128,374,151 ns	286,779,702 ns	68,427,482 ns	-	5,683 ns	-
THREAD 1.1.8	149,729,056 ns	286,749,842 ns	47,089,383 ns	-	5,565 ns	-
Total	1,434,229,280 ns	2,007,482,088 ns	439,056,348 ns	7,046,914 ns	79,881 ns	2,340 ns
Average	179,278,660 ns	286,783,155.43 ns	54,882,043.50 ns	7,046,914 ns	9,985.12 ns	2,340 ns
Maximum	400,116,985 ns	286,825,448 ns	95,696,104 ns	7,046,914 ns	35,325 ns	2,340 ns
Minimum	127,851,843 ns	286,740,269 ns	27,562 ns	7,046,914 ns	5,565 ns	2,340 ns
StDev	86,695,894.44 ns	30,596.63 ns	28,062,975.68 ns	0 ns	9,662.87 ns	0 ns
Avg/Max	0.45	1.00	0.57	1	0.28	1

Figure 4: Time spent in the different states of OMP

This was obtained using the OMP_state_profile configuration.