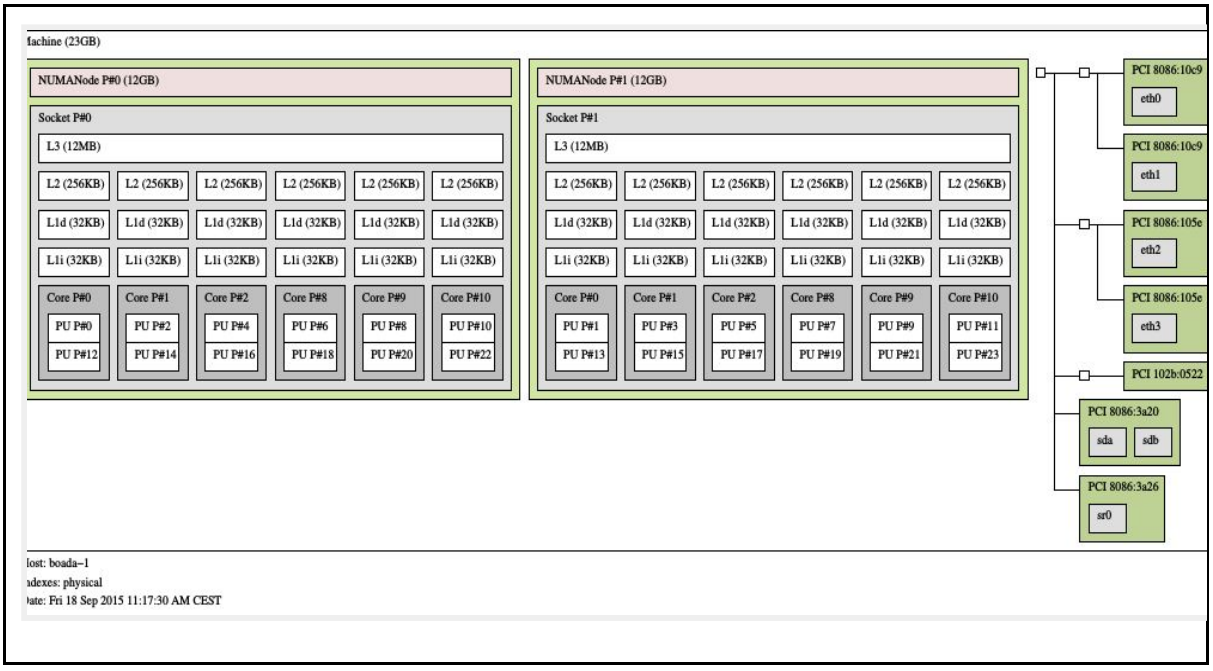
1st Deliverable

par1103
Gallego, Victor
Iqbal, Omair
2015-2016
March 2016

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).



From the picture above we can say that our PC has 2 sockets, 6 cores for each socket and each core has 2 threads.
The cache hierarchy for each node socket is:

- L3 shared 12MB
- L2 private 256KB
- L1 data private 32KB
- L1 instructions private 32KB

The amount of main memory is 23 GB in the machine and 12GB in each NUMA node memory.
All this information is provided by the following commands: **#lstopo, #lscpu and #more /proc/meminfo**

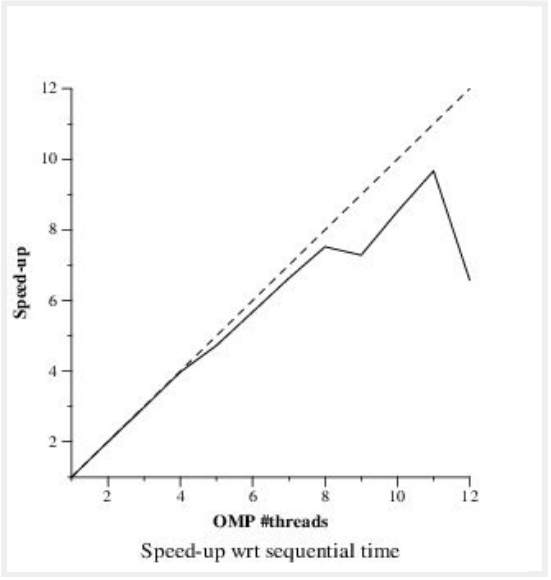
Timing sequential and parallel executions

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.

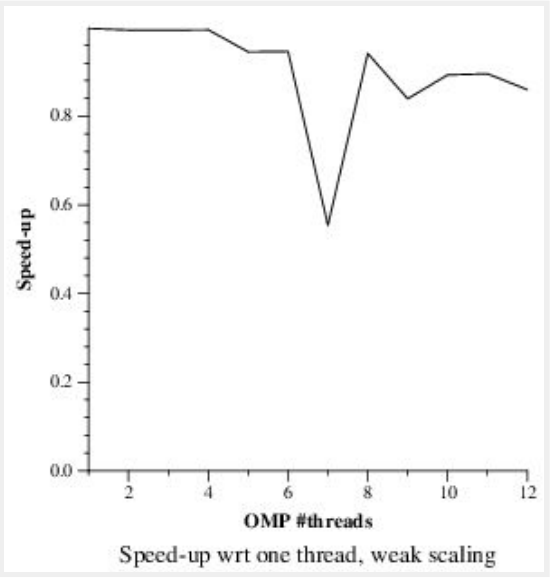
We need to add the function getusec_(), provided by the library header #include <sys/times.h>, between the pair of points in which we want to know the elapsed execution time.
This library has a data structure called " timeval " which is responsible for giving the result of the difference between the two function calls.

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

Strong scalability



Weak scalability



In the both graphs we can see that we do not get a significant speed-up by increasing the number of threads from a certain point, the problem arises when the complexity of cores increases.

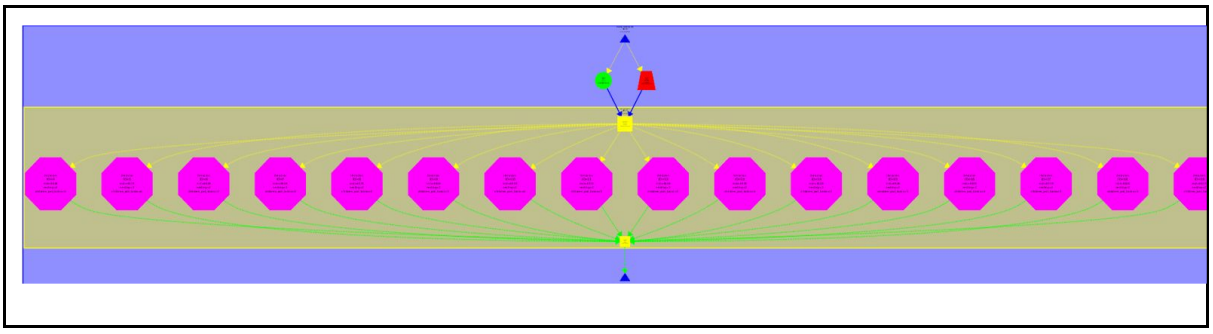
Visualizing the task graph and data dependences

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 depende(s).

```
tareador_ON ();
int i;
tareador_start_task("init_A");
for (i=0; i< size; i++) A[i]=i;
tareador_end_task("init_A");
tareador_start_task("init_B");
for (i=0; i< size; i++) B[i]=2*i;
tareador_end_task("init_B");
gettimeofday(&start,NULL);
```

```
tareador_start_task("dot_product");
dot_product (size, A, B, &result);
tareador_end_task("dot_product");
tareador_OFF ();
```

5. Capture the task dependence graph and execution execution timeline (for 8 processors) for that task decomposition.



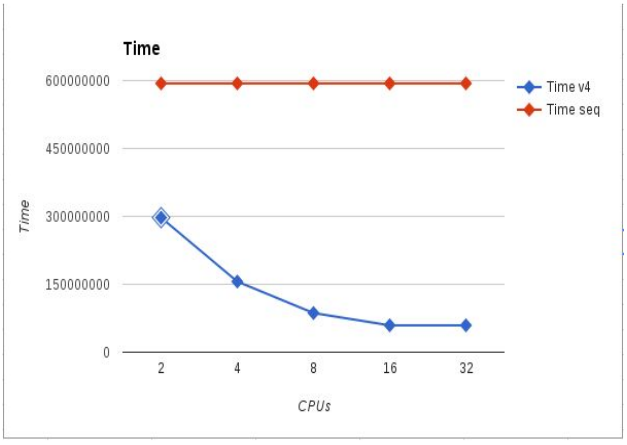
Analysis of task decompositions

6. Complete the following table for the initial and different versions generated for 3dfft_seq.c

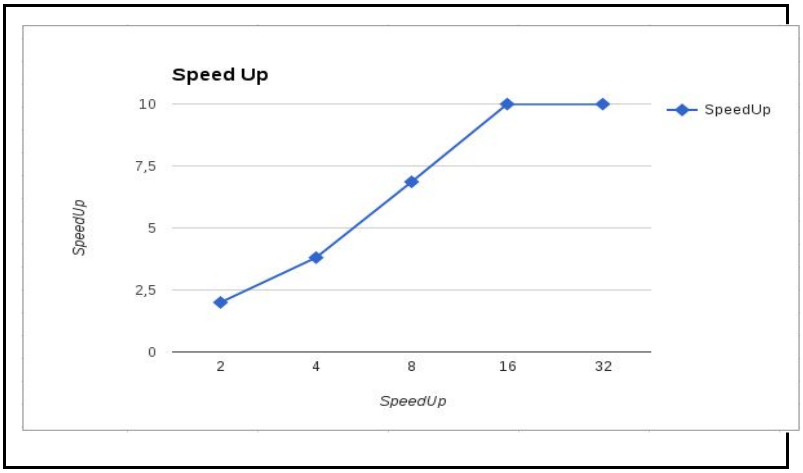
Version	T1	Tinf	Parallelism
seq	593.772.001 ns	593.772.001 ns	1
v1	593.772.001 ns	593.772.001 ns	1
v2	593.772.001 ns	315.437.001 ns	1.88237905
v3	593.772.001 ns	108.937.001 ns	5.45059985
v4	593.772.001 ns	59.415.001	9.99363782

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).

CPUs	Time v4	Time seq
2	297.308.001	593.772.001
4	156.088.001	593.772.001
8	86.561.001	593.772.001
16	59.415.001	593.772.001
32	59.415.001	593.772.001



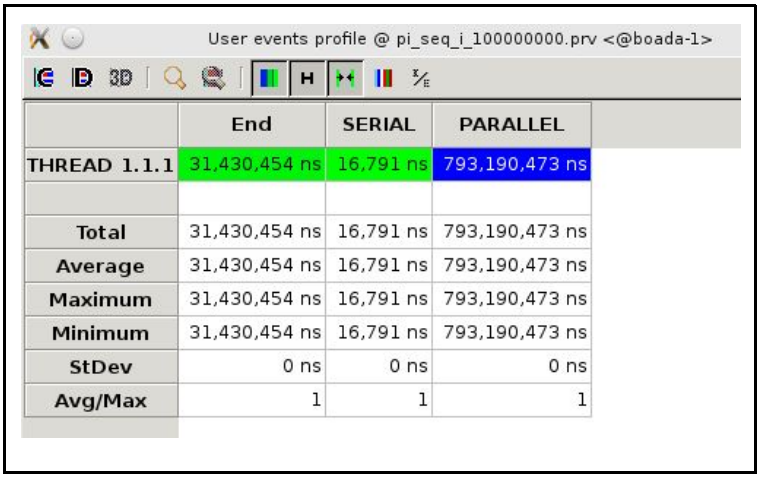
CPUs	SpeedUp
2	1,99716119
4	3,804084857
8	6,859578726
16	9,993637819
32	9,993637819



Tracing sequential and parallel executions

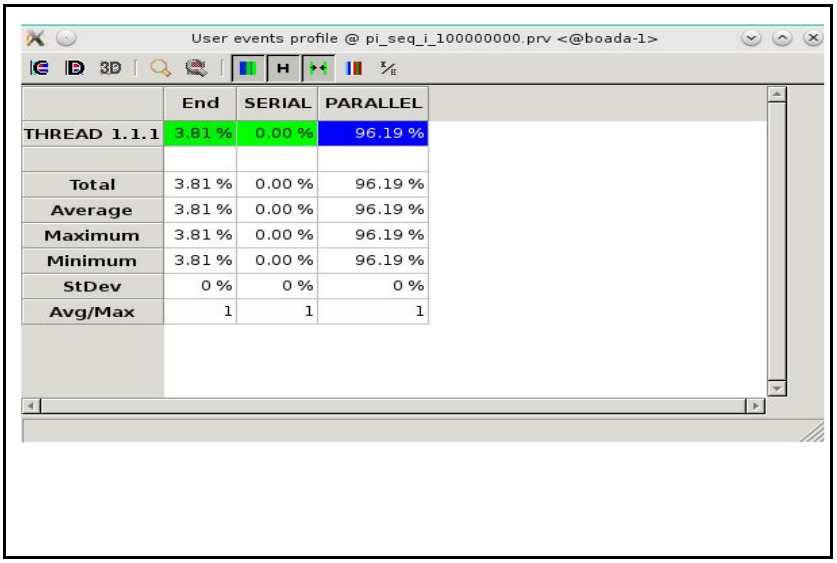
8. From the instrumented version of pi_seq.c, and using the appropriate Paraver configuration file, obtain the value of the parallel fraction iO for this program when executed with 100.000.000 iterations.

Serial and parallel time for pi_seq



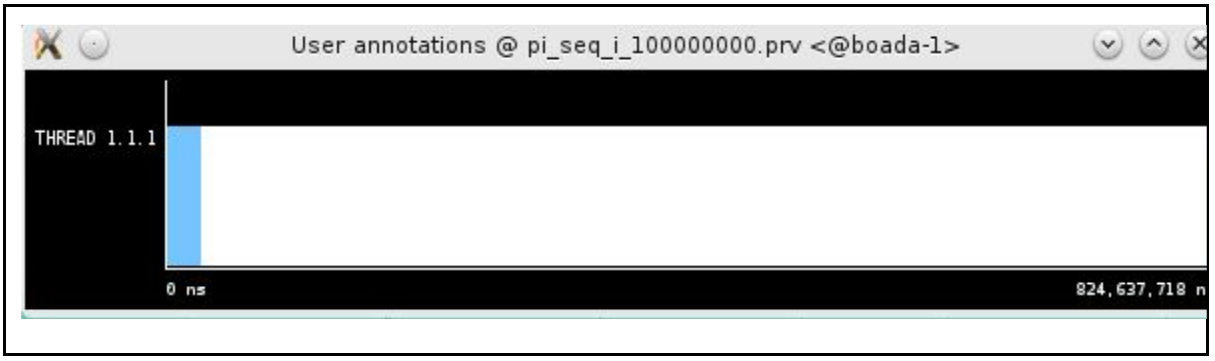
	End	SERIAL	PARALLEL
THREAD 1.1.1	31,430,454 ns	16,791 ns	793,190,473 ns
Total	31,430,454 ns	16,791 ns	793,190,473 ns
Average	31,430,454 ns	16,791 ns	793,190,473 ns
Maximum	31,430,454 ns	16,791 ns	793,190,473 ns
Minimum	31,430,454 ns	16,791 ns	793,190,473 ns
StDev	0 ns	0 ns	0 ns
Avg/Max	1	1	1

Percentage of time of serial and parallel for pi_seq



	End	SERIAL	PARALLEL
THREAD 1.1.1	3.81 %	0.00 %	96.19 %
Total	3.81 %	0.00 %	96.19 %
Average	3.81 %	0.00 %	96.19 %
Maximum	3.81 %	0.00 %	96.19 %
Minimum	3.81 %	0.00 %	96.19 %
StDev	0 %	0 %	0 %
Avg/Max	1	1	1

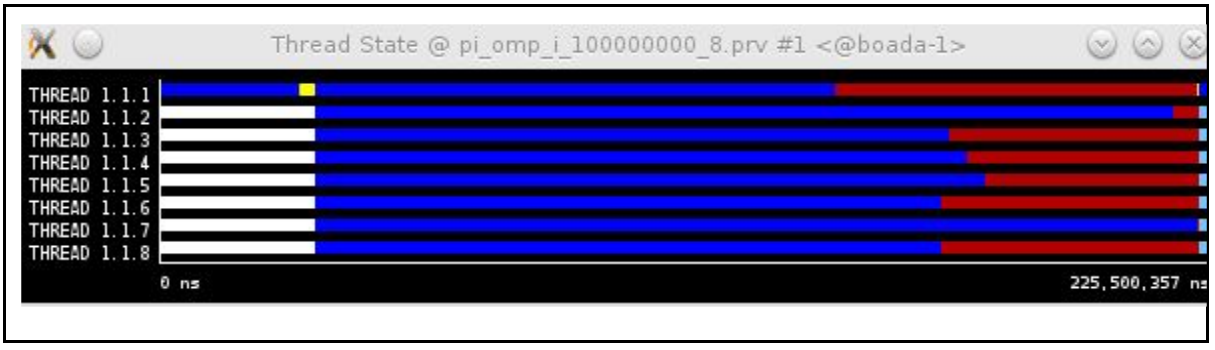
Execution timeline for pi_seq



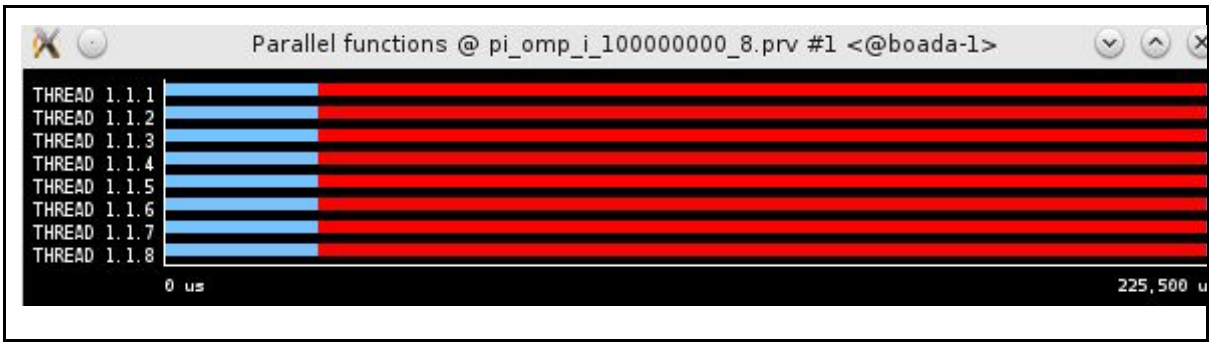
As we can see in the previous graphs the biggest part of the code can be parallelized that corresponds to a 96,19%.

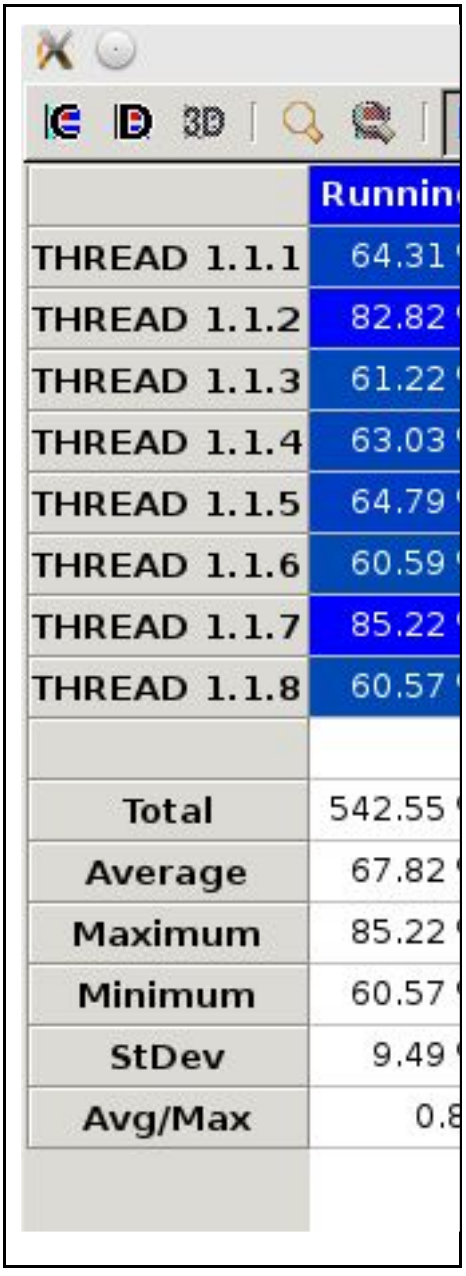
9. From the instrumented version pi_omp.c and using the appropriate Paraver configuration file, show a profile of the % of time spent in the different OpenMp states when using 8 threads and for 1000.000.000 iterations.

Execution timeline for the different states.



Execution timeline for the estates that can be parallelized.





	Runnin
THREAD 1.1.1	64.31
THREAD 1.1.2	82.82
THREAD 1.1.3	61.22
THREAD 1.1.4	63.03
THREAD 1.1.5	64.79
THREAD 1.1.6	60.59
THREAD 1.1.7	85.22
THREAD 1.1.8	60.57
Total	542.55
Average	67.82
Maximum	85.22
Minimum	60.57
StDev	9.49
Avg/Max	0.8