PAR Laboratory

13/09/2014 - Q1
FIRST DELIVERABLE

group: par1107
Gabriel Carrillo
Younes Zeriahi

Node architecture and memory:

1.- Draw and briefly describe the architecture of the computer in which you are doing this lab session:

To get the numbers of sockets, the cores per socket and the threads per core we use the “lscpu” command:

```
----------------------------------------------------------------------------------------------------
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Stepping:              2
CPU MHz:               1596.000
BogoMIPS:              4799.90
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23
----------------------------------------------------------------------------------------------------
```

As we can see, every node of the machine has 2 sockets, each of one has 6 cores, and each core has 2 threads. That means that every node has:

2 sockets
12 cores
24 threads

The command we used to get the cache hierarchy size and sharing is the “lstopo” command:

```
Machine (24GB)
  NUMANode L#0 (P#0 12GB) + Socket L#0 + L3 L#0 (12MB)
    L2 L#0 (256KB) + L1 L#0 (32KB) + Core L#0
      PU L#0 (P#0)
      PU L#1 (P#12)
    L2 L#1 (256KB) + L1 L#1 (32KB) + Core L#1
      PU L#2 (P#2)
      PU L#3 (P#14)
    L2 L#2 (256KB) + L1 L#2 (32KB) + Core L#2
      PU L#4 (P#4)
      PU L#5 (P#16)
    L2 L#3 (256KB) + L1 L#3 (32KB) + Core L#3
      PU L#6 (P#6)
      PU L#7 (P#18)
    L2 L#4 (256KB) + L1 L#4 (32KB) + Core L#4
      PU L#8 (P#8)
      PU L#9 (P#20)
    L2 L#5 (256KB) + L1 L#5 (32KB) + Core L#5
      PU L#10 (P#10)
      PU L#11 (P#22)
  NUMANode L#1 (P#1 12GB) + Socket L#1 + L3 L#1 (12MB)
    L2 L#6 (256KB) + L1 L#6 (32KB) + Core L#6
      PU L#12 (P#1)
    L2 L#7 (256KB) + L1 L#7 (32KB) + Core L#7
      PU L#14 (P#3)
      PU L#15 (P#15)
    L2 L#8 (256KB) + L1 L#8 (32KB) + Core L#8
      PU L#16 (P#5)
      PU L#17 (P#17)
    L2 L#9 (256KB) + L1 L#9 (32KB) + Core L#9
      PU L#18 (P#7)
      PU L#19 (P#19)
    L2 L#10 (256KB) + L1 L#10 (32KB) + Core L#10
      PU L#20 (P#9)
      PU L#21 (P#21)
    L2 L#11 (256KB) + L1 L#11 (32KB) + Core L#11
      PU L#22 (P#11)
      PU L#23 (P#23)
  HostBridge L#0
    PCIBridge
      PCI 8086:10c9
        Net L#0 "eth0"
      PCI 8086:10c9
        Net L#1 "eth2"
    PCIBridge
      PCI 8086:105e
        Net L#2 "eth1"
      PCI 8086:105e
        Net L#3 "eth3"
    PCIBridge
      PCI 102b:0522
    PCI 8086:3a20
      Block L#4 "sda"
      Block L#5 "sdb"
    PCI 8086:3a26
      Block L#6 "sr0"
PU L#13 (P#13)
```

As we can see, the computer has 24GB, 12 for each node. Each node has 6 levels of cache, each one with 32KB.

For knowing the amount of main memory that the computer has, we use the “more /proc/meminfo” command:

```
------------------------------------------------------------------------------------------------------------------------
MemTotal:       24628820 kB
MemFree:        3260100 kB
Buffers:        180668 kB
Cached:         20480016 kB
SwapCached:     0 kB
Active:         10553492 kB
Inactive:       10159580 kB
Active(anon):   7688 kB
Inactive(anon):45080 kB
Active(file):   10545804 kB
Inactive(file): 10114500 kB
Unevictable:    128 kB
Mlocked:        128 kB
SwapTotal:      25165820 kB
SwapFree:       25165820 kB
Dirty:          212 kB
Writeback:      0 kB
AnonPages:      52520 kB
Mapped:         17088 kB
Shmem:          380 kB
Slab:           319252 kB
Sreclaimable:   253752 kB
Sunreclaim:     65500 kB
KernelStack:    2752 kB
PageTables:     3844 kB
NFS_Unstable:0 kB
Bounce:         0 kB
WritebackTmp:           0 kB
CommitLimit:            37480228 kB
Committed_AS:           310832 kB
VmallocTotal:           34359738367 kB
VmallocUsed:            339156 kB
VmallocChunk:           34346619196 kB
HardwareCorrupted:      0 kB
AnonHugePages:          0 kB
HugePages_Total:        0
HugePages_Free:         0
HugePages_Rsvd:         0
HugePages_Surp:         0
Hugepagesize:           2048 kB
DirectMap4k:            113796 kB
DirectMap2M:            7163904 kB
DirectMap1G:            17825792 kB
------------------------------------------------------------------------------------------------------------------------
```

As we can see, the computer has a total of 24628820 kB of main memory, 3260100 kB of which are free for us to use.

Timing sequential and parallel executions:

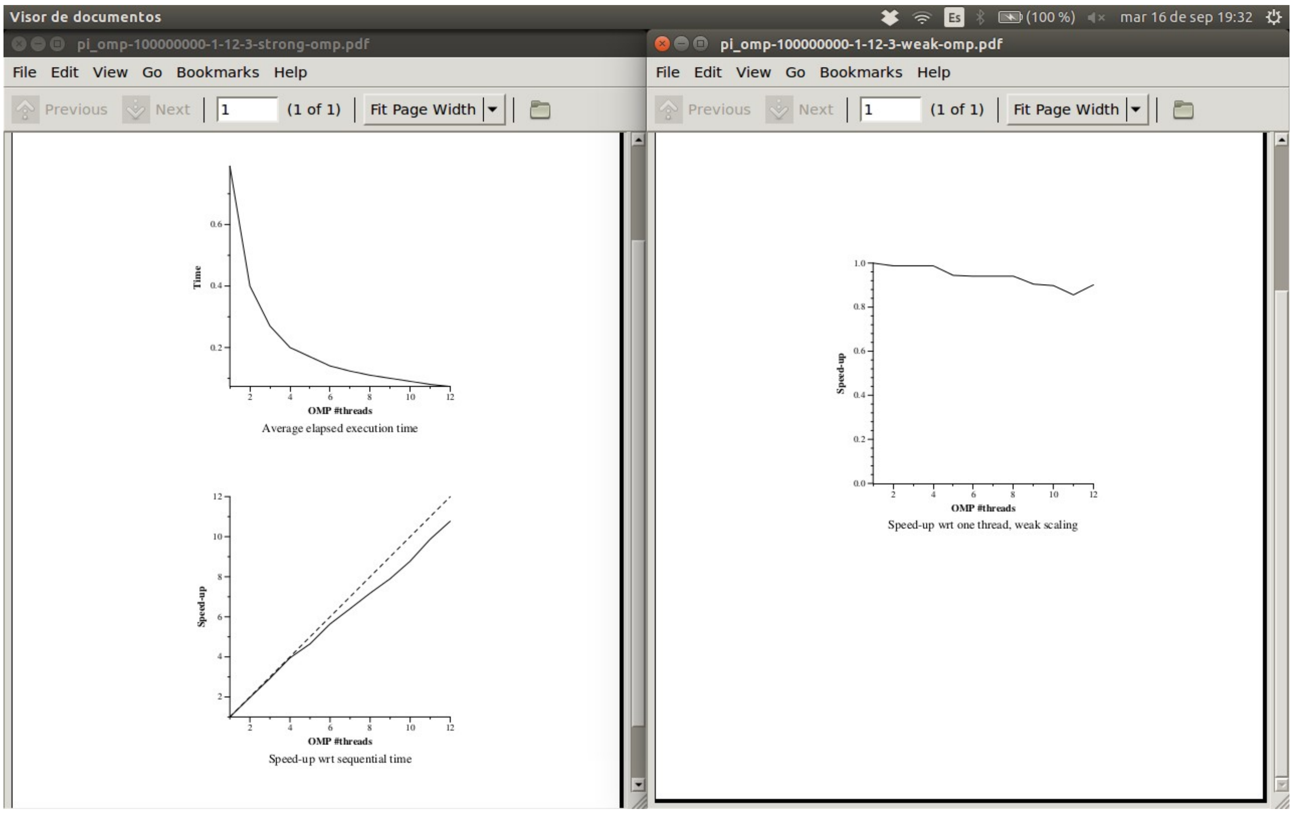
2.- Indicate the library header where the structure *struct timeval* is declared and which are its fields:

The library header is the <sys/time.h>, and it's fields are:

```
        time_t       tv_sec;      /* seconds */
        suseconds_t  tv_usec;     /* microseconds */
```

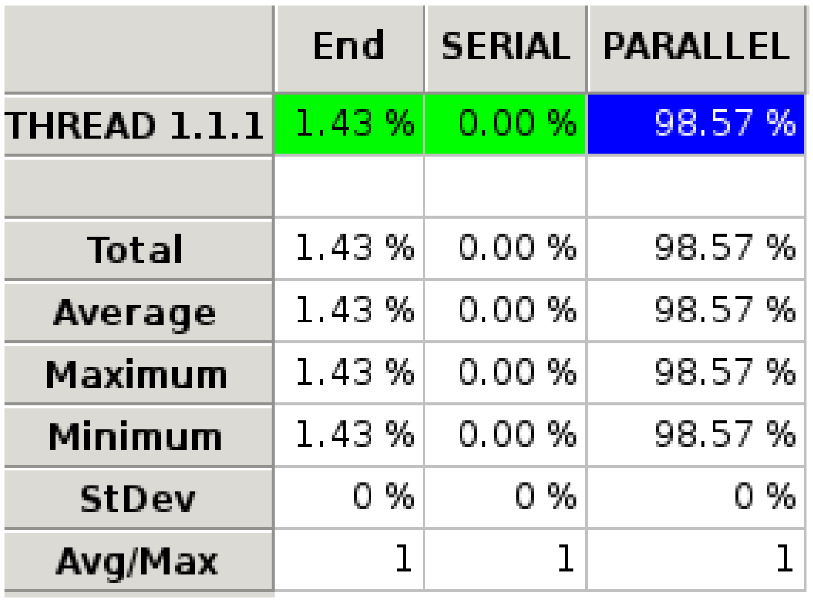
3.- Plot the execution time when varying the number of threads (strong scalability) and problem size (weak scalability) for pi_omp.c. Reason about the scalability of the program.

The scalability is correct because when we execute the program with strong scalability, we can see in the first graph that the curve decreases approaching zero while increasing the number of threads, and the speed-up tends to the 45 degrees line, getting wider with the increase of the number of threads.
On the other hand, when we execute the program with weak scalability, we can see that the speed-up remains almost constant with the increase of the number of threads.


Visor de documentos
(100 %)
mar 16 de sep 19:32
pi_omp-100000000-1-12-3-strong-omp.pdf
pi_omp-100000000-1-12-3-weak-omp.pdf
File Edit View Go Bookmarks Help
Previous
Next
1
(1 of 1)
Fit Page Width
Time
0.6
0.4
0.2
2
4
6
8
10
12
OMP #threads
Average elapsed execution time
Speed-up
12
10
8
6
4
2
OMP #threads
Speed-up wrt sequential time
1.0
0.8
0.6
0.4
0.2
0.0
Speed-up
OMP #threads
Speed-up wrt one thread, weak scaling

<u>Tracing sequential and parallel executions:</u>

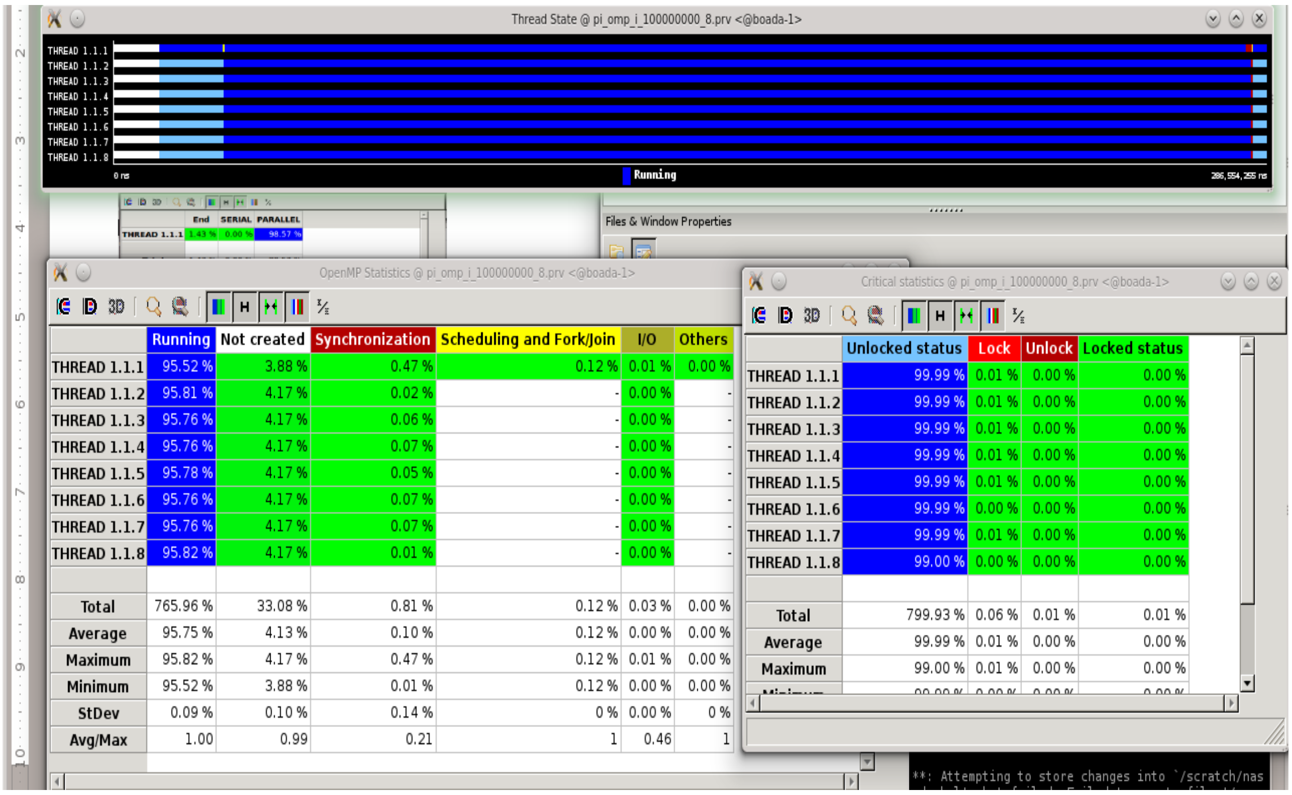
4.- From the instrumented version of pi_seq.c, and using the appropriate *Paraver* configuration file, obtain the value of the *parallel fraction ø* for this program when executed with 100.000.000 iterations:



	End	SERIAL	PARALLEL
THREAD 1.1.1	1.43 %	0.00 %	98.57 %
Total	1.43 %	0.00 %	98.57 %
Average	1.43 %	0.00 %	98.57 %
Maximum	1.43 %	0.00 %	98.57 %
Minimum	1.43 %	0.00 %	98.57 %
StDev	0 %	0 %	0 %
Avg/Max	1	1	1

As we can see in the picture above, 98'57% of the instrumented version of pi_seq is done in parallel when we execute it with 100.000.000 iterations.

5.- From the instrumented version of pi_omp.c, and using the appropiate *Paraver* configuration file, show a profile of the % of time spent in the different OpenMP states ONLY during the execution of the parallel region (not considering the sequential part before and after) when using 8 threads and for 100.000.000 iterations.



	End	SERIAL	PARALLEL
THREAD 1.1.1	1.43 %	0.00 %	98.57 %

OpenMP Statistics @ pi_omp_i_100000000_8.prv <@boada-1>

	Running	Not created	Synchronization	Scheduling and Fork/Join	I/O	Others
THREAD 1.1.1	95.52 %	3.88 %	0.47 %	0.12 %	0.01 %	0.00 %
THREAD 1.1.2	95.81 %	4.17 %	0.02 %	-	0.00 %	-
THREAD 1.1.3	95.76 %	4.17 %	0.06 %	-	0.00 %	-
THREAD 1.1.4	95.76 %	4.17 %	0.07 %	-	0.00 %	-
THREAD 1.1.5	95.78 %	4.17 %	0.05 %	-	0.00 %	-
THREAD 1.1.6	95.76 %	4.17 %	0.07 %	-	0.00 %	-
THREAD 1.1.7	95.76 %	4.17 %	0.07 %	-	0.00 %	-
THREAD 1.1.8	95.82 %	4.17 %	0.01 %	-	0.00 %	-
Total	765.96 %	33.08 %	0.81 %	0.12 %	0.03 %	0.00 %
Average	95.75 %	4.13 %	0.10 %	0.12 %	0.00 %	0.00 %
Maximum	95.82 %	4.17 %	0.47 %	0.12 %	0.01 %	0.00 %
Minimum	95.52 %	3.88 %	0.01 %	0.12 %	0.00 %	0.00 %
StDev	0.09 %	0.10 %	0.14 %	0 %	0.00 %	0 %
Avg/Max	1.00	0.99	0.21	1	0.46	1

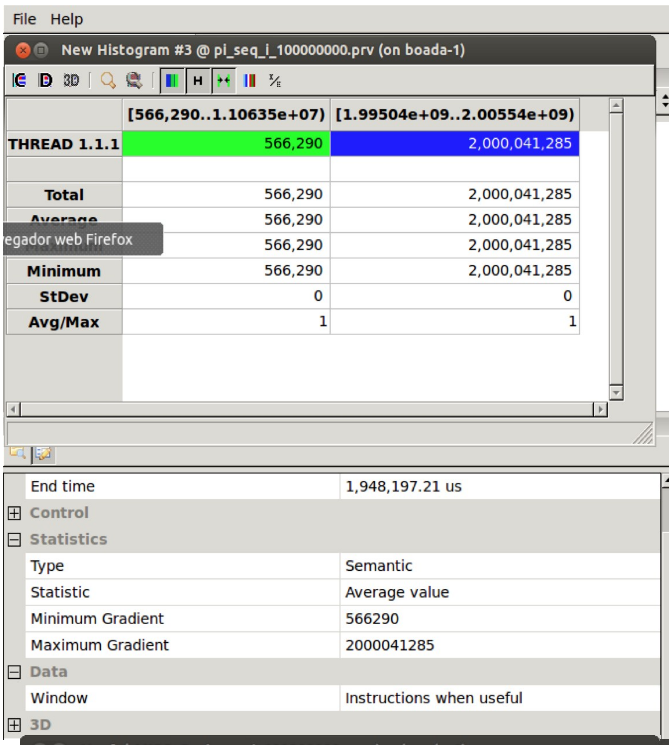
Critical statistics @ pi_omp_i_100000000_8.prv <@boada-1>

	Unlocked status	Lock	Unlock	Locked status
THREAD 1.1.1	99.99 %	0.01 %	0.00 %	0.00 %
THREAD 1.1.2	99.99 %	0.01 %	0.00 %	0.00 %
THREAD 1.1.3	99.99 %	0.01 %	0.00 %	0.00 %
THREAD 1.1.4	99.99 %	0.01 %	0.00 %	0.00 %
THREAD 1.1.5	99.99 %	0.01 %	0.00 %	0.00 %
THREAD 1.1.6	99.99 %	0.00 %	0.00 %	0.00 %
THREAD 1.1.7	99.99 %	0.01 %	0.00 %	0.00 %
THREAD 1.1.8	99.00 %	0.00 %	0.00 %	0.00 %
Total	799.93 %	0.06 %	0.01 %	0.01 %
Average	99.99 %	0.01 %	0.00 %	0.00 %
Maximum	99.00 %	0.01 %	0.00 %	0.00 %

```
**: Attempting to store changes into `/scratch/nas
```

As we can see in the picture above, the average time of the processors during the parallel region using 8 processors and 100.000.000 iterations is 95'75%.
A 4'13% of the time is spent on the synchronization and the remaining 0'12% is invested in scheduling and Forking/Joining the tasks.

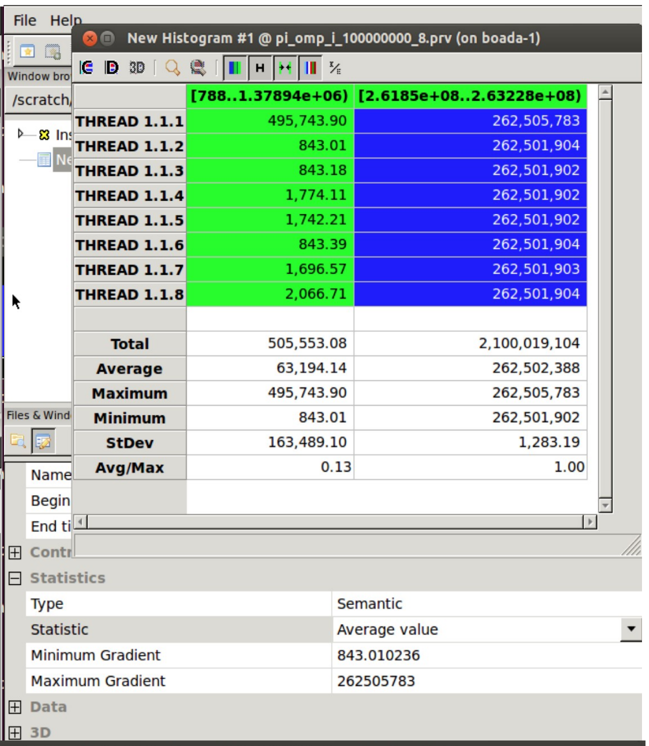
6.- From the instrumented versions of pi_seq.c and pi_omp.c (executed with the same number of iterations), and using the appropriate *Paraver* configuration files, obtain the avarage number of instructions executed and MIPS per thread during the execution of the parallel region (when using 8 threads). Are the numbers obtained coherent?



New Histogram #3 @ pi_seq_i_100000000.prv (on boada-1)

	[566,290..1.10635e+07)	[1.99504e+09..2.00554e+09)
THREAD 1.1.1	566,290	2,000,041,285
Total	566,290	2,000,041,285
Average	566,290	2,000,041,285
Maximum	566,290	2,000,041,285
Minimum	566,290	2,000,041,285
StDev	0	0
Avg/Max	1	1

End time	1,948,197.21 us
Control	
Statistics	
Type	Semantic
Statistic	Average value
Minimum Gradient	566290
Maximum Gradient	2000041285
Data	
Window	Instructions when useful
3D	

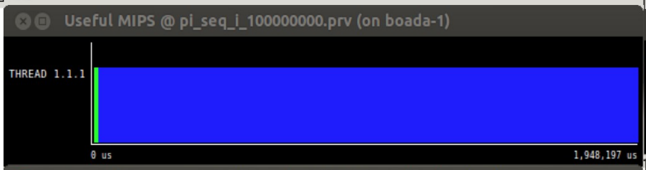


New Histogram #1 @ pi_omp_i_100000000_8.prv (on boada-1)

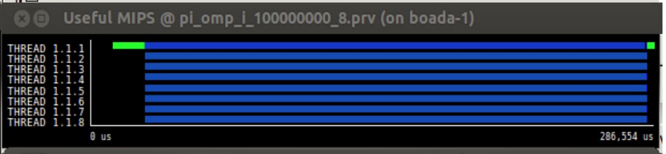
	[788..1.37894e+06)	[2.6185e+08..2.63228e+08)
THREAD 1.1.1	495,743.90	262,505,783
THREAD 1.1.2	843.01	262,501,904
THREAD 1.1.3	843.18	262,501,902
THREAD 1.1.4	1,774.11	262,501,902
THREAD 1.1.5	1,742.21	262,501,902
THREAD 1.1.6	843.39	262,501,904
THREAD 1.1.7	1,696.57	262,501,903
THREAD 1.1.8	2,066.71	262,501,904
Total	505,553.08	2,100,019,104
Average	63,194.14	262,502,388
Maximum	495,743.90	262,505,783
Minimum	843.01	262,501,902
StDev	163,489.10	1,283.19
Avg/Max	0.13	1.00

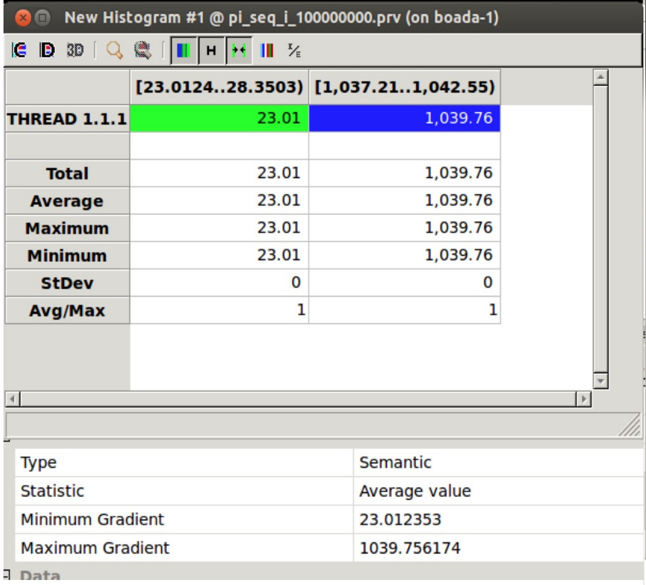
Statistics	
Type	Semantic
Statistic	Average value
Minimum Gradient	843.010236
Maximum Gradient	262505783
Data	
3D	

Useful MIPS @ pi_seq_i_100000000.prv (on boada-1)



Useful MIPS @ pi_omp_i_100000000_8.prv (on boada-1)

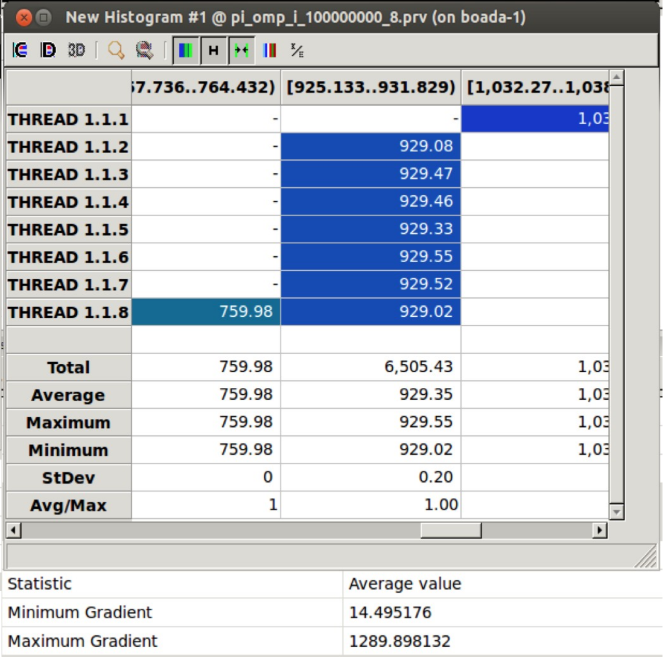




New Histogram #1 @ pi_seq_i_100000000.prv (on boada-1)

	[23.0124..28.3503)	[1,037.21..1,042.55)
THREAD 1.1.1	23.01	1,039.76
Total	23.01	1,039.76
Average	23.01	1,039.76
Maximum	23.01	1,039.76
Minimum	23.01	1,039.76
StDev	0	0
Avg/Max	1	1

Type	Semantic
Statistic	Average value
Minimum Gradient	23.012353
Maximum Gradient	1039.756174
Data	



New Histogram #1 @ pi_omp_i_100000000_8.prv (on boada-1)

	7.736..764.432)	[925.133..931.829)	[1,032.27..1,03
THREAD 1.1.1	-	-	1,03
THREAD 1.1.2	-	929.08	
THREAD 1.1.3	-	929.47	
THREAD 1.1.4	-	929.46	
THREAD 1.1.5	-	929.33	
THREAD 1.1.6	-	929.55	
THREAD 1.1.7	-	929.52	
THREAD 1.1.8	759.98	929.02	
Total	759.98	6,505.43	1,03
Average	759.98	929.35	1,03
Maximum	759.98	929.55	1,03
Minimum	759.98	929.02	1,03
StDev	0	0.20	
Avg/Max	1	1.00	

Statistic	Average value
Minimum Gradient	14.495176
Maximum Gradient	1289.898132

As we can see in the pictures above, we have executed both programs with 100.000.000 iterations and shown:

- the average number of instructions executed during the execution of the parallel region when using 8 threads:
  - seq: 200.000.041'285
  - omp: 262.502.388
- the average MIPS per thread during the execution of the parallel region when using 8 threads:
  - seq: 1.039'76
  - omp: 929'35

As we can see, the numbers are not coherent, because on one hand, the number of instructions executed in the parallel region in seq are less than in omp, but on the other hand, the MIPS in the parallel region in seq are more than in omp. Given that the MIPS are the number of instructions divided by the total time of the execution, the relation between the instructions and the MIPS should be the same.

<u>**Visualizing the task graph and data dependences:**</u>

7.- 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).

In the main, we do a simple Tareador instrumentation with a sequence of fin grained tasks.

Code sample of the main function in “dot_product.c” :

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

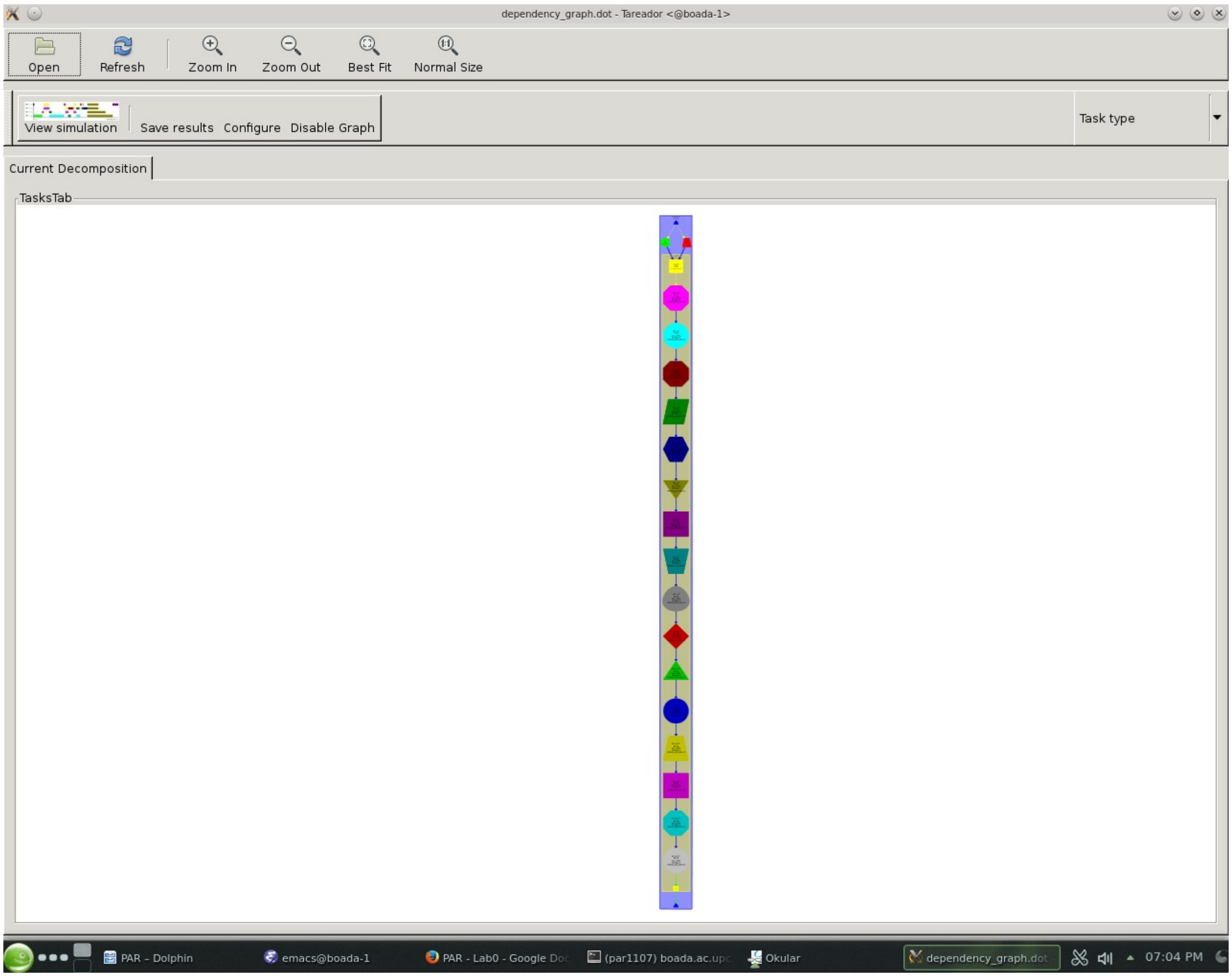
Then, in the code of the dot_product function, we added a Tareador instrumentation with a sequence of finer grained tasks : one task per loop.
We added the following instrumentation (colored in blue) in order to study the potential parallelism in the code.

Code sample of the dot_product function in “dot_product.c” :

```
void dot_product (long N, double A[N], double B[N], double *acc){
  double prod;
  *acc=0.0;
  int i;
  for (i=0; i<N; i++) {
     char task_name[10] = "dot_p_";
     char tmp[10];
     sprintf(tmp,"%d",i);
     strcat(task_name,tmp);
     tareador_start_task(task_name);
     prod = my_func(A[i], B[i]);
     *acc += prod;
     tareador_end_task(task_name);
  }
}
```

We observe that it doesn’t parallelize more the code, because of a data dependency between each task of the loop.

Task dependance graph :



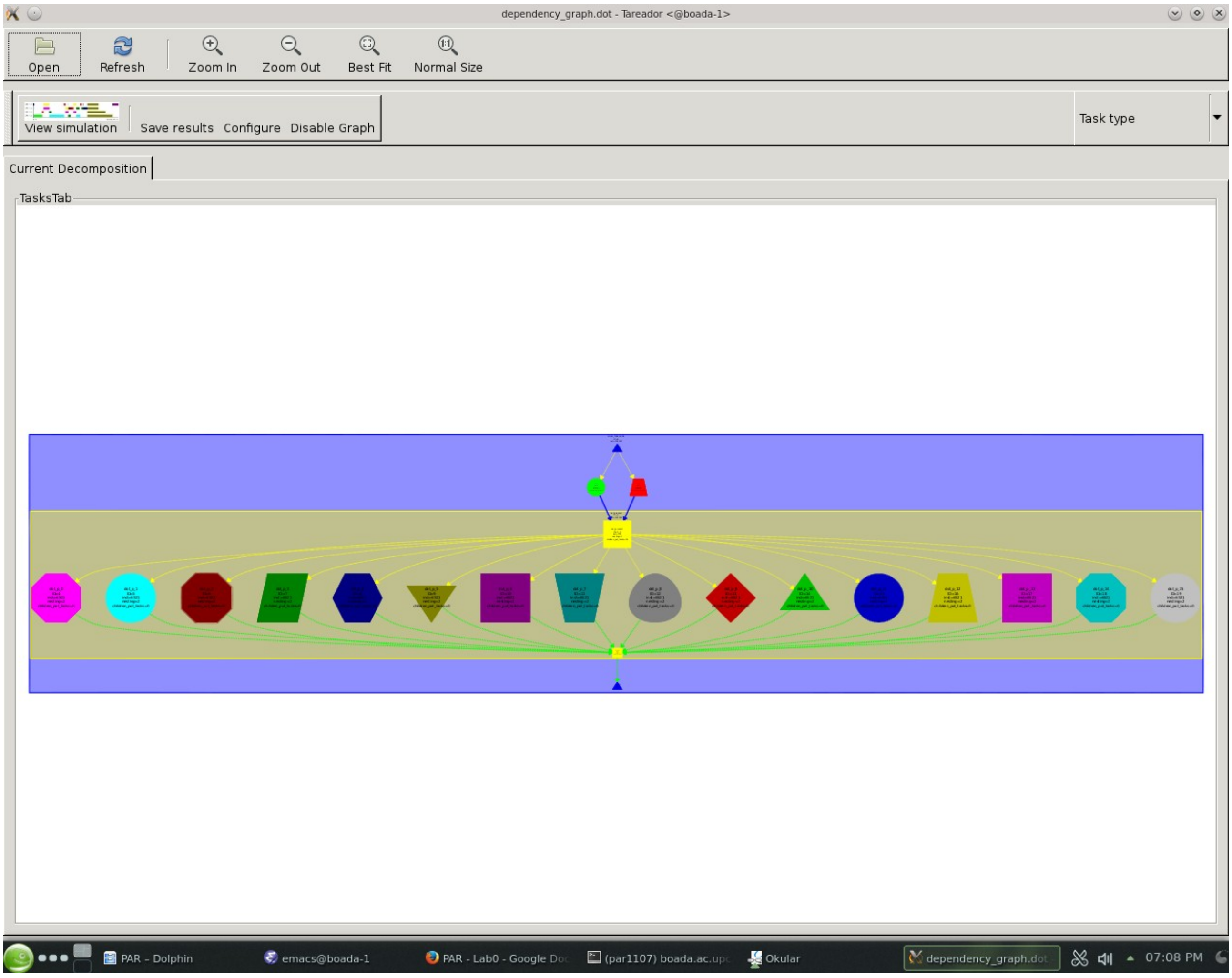
Finally, we wanted to see which variable(s) cause(s) the dependence(s).
We realized that it was the variable “acc” which causes that, because when we add these two lines (colored in red in the following code sample), the task dependence graph is no longer the same.

Code sample of an extract of the dot_product function in “dot_product.c” :

```
tareador_start_task(task_name);
prod = my_func(A[i], B[i]);
tareador_disable_object(&acc);
*acc += prod;
tareador_enable_object(&acc);
tareador_end_task(task_name);
```

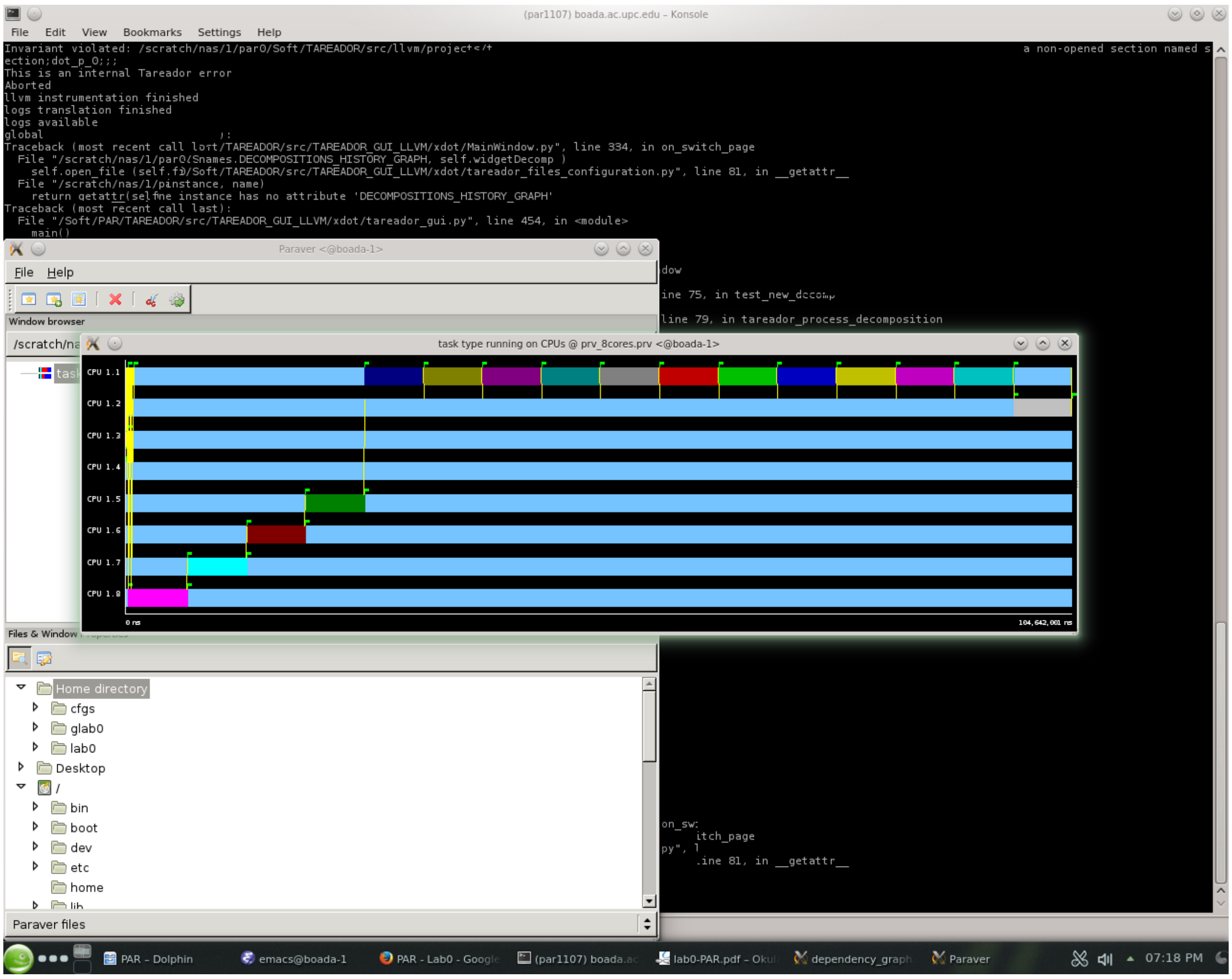
Just by disabling this variable for the Tareador instrumentation, the tasks can now be executed in parallel (but the potential result wouldn’t be correct).

New task dependence graph :



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

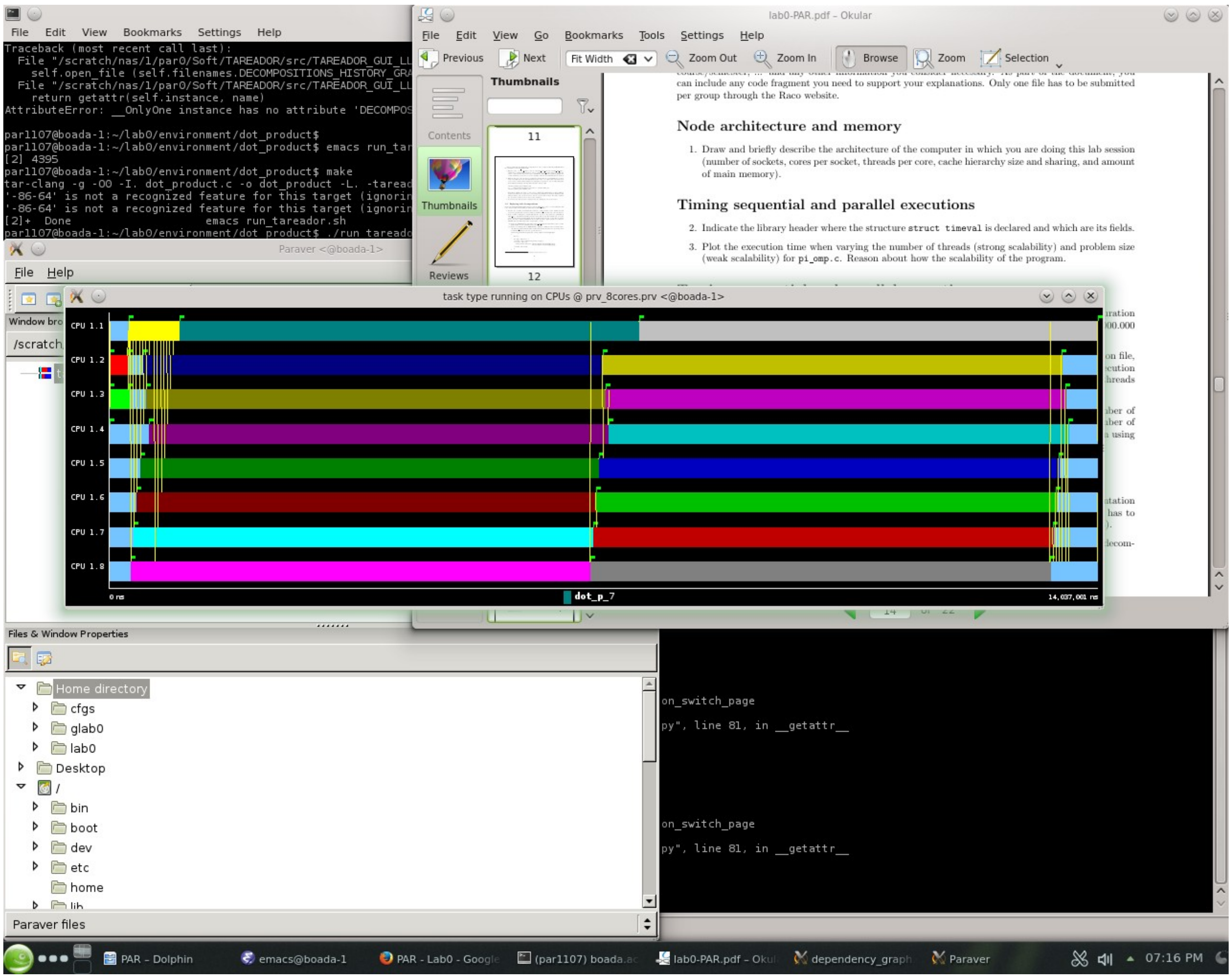
Here is the execution timeline for that task decomposition:



```
Invariant violated: /scratch/nas/1/par0/Soft/TAREADOR/src/llvm/project</+                                    a non-opened section named s
ection;dot_p_0;;;
This is an internal Tareador error
Aborted
llvm instrumentation finished
logs translation finished
logs available
global                          ;:
Traceback (most recent call lort/TAREADOR/src/TAREADOR_GUI_LLVM/xdot/MainWindow.py", line 334, in on_switch_page
  File "/scratch/nas/1/par0/Snames.DECOMPOSITIONS_HISTORY_GRAPH, self.widgetDecomp )
    self.open_file (self.fi)/Soft/TAREADOR/src/TAREADOR_GUI_LLVM/xdot/tareador_files_configuration.py", line 81, in __getattr__
  File "/scratch/nas/1/pinstance, name)
    return getattr(selfne instance has no attribute 'DECOMPOSITIONS_HISTORY_GRAPH'
Traceback (most recent call last):
  File "/Soft/PAR/TAREADOR/src/TAREADOR_GUI_LLVM/xdot/tareador_gui.py", line 454, in <module>
    main()
dow
ine 75, in test_new_decomp
line 79, in tareador_process_decomposition
on_sw:
     itch_page
py", l
     .ine 81, in __getattr__
```

And when we add the two lines to disable the object "acc", we can see the difference (as we saw it on the task dependence graph) with the tasks executing in parallel.

Here is the execution timeline for the modified task decomposition :



```
Traceback (most recent call last):
  File "/scratch/nas/1/par0/Soft/TAREADOR/src/TAREADOR_GUI_LL
    self.open_file (self.filenames.DECOMPOSITIONS_HISTORY_GRA
  File "/scratch/nas/1/par0/Soft/TAREADOR/src/TAREADOR_GUI_LL
    return getattr(self.instance, name)
AttributeError: __OnlyOne instance has no attribute 'DECOMPOS

par1107@boada-1:~/lab0/environment/dot_product$
par1107@boada-1:~/lab0/environment/dot_product$ emacs run_tar
[2] 4395
par1107@boada-1:~/lab0/environment/dot_product$ make
tar-clang -g -O0 -I. dot_product.c -o dot_product -L. -taread
'-86-64' is not a recognized feature for this target (ignorin
'-86-64' is not a recognized feature for this target (ignorin
[2]+  Done                    emacs run_tareador.sh
par1107@boada-1:~/lab0/environment/dot_product$ ./run_tareado
```

can include any code fragment you need to support your explanations. Only one file has to be submitted per group through the Raco website.

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. Indicate the library header where the structure `struct timeval` is declared and which are its fields.

3. Plot the execution time 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.

```
on_switch_page
py", line 81, in __getattr__


on_switch_page
py", line 81, in __getattr__
```

Analysis of task decompositions:

9.- Complete the following table for the initial and different versions generated for 3dfft seq.c.

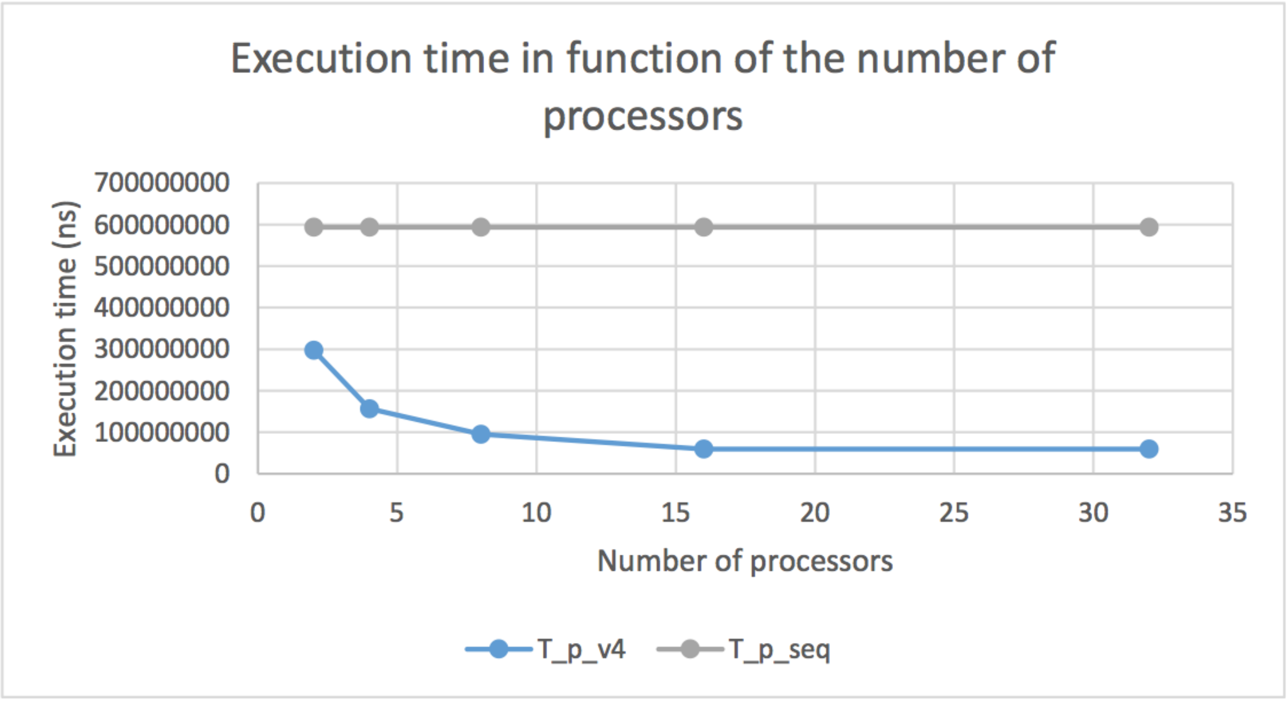
Version	T1 (ns)	T_infinity (ns)	Parallelism
seq	593,762,001	593,711,001	1,0000859
v1	593,762,001	593,711,001	1,0000859
v2	594,128,001	315,221,001	1,88479828
v3	594,616,001	108,413,001	5,48472965
v4	594,738,001	59,511,001	9,9937489

The version v4 was made by adding a few lines of instrumentation code to the previous version v3. In fact, we did the same decomposition in a sequence of fin grained tasks of the "init_complex_grid" function than we did for all the others functions used.

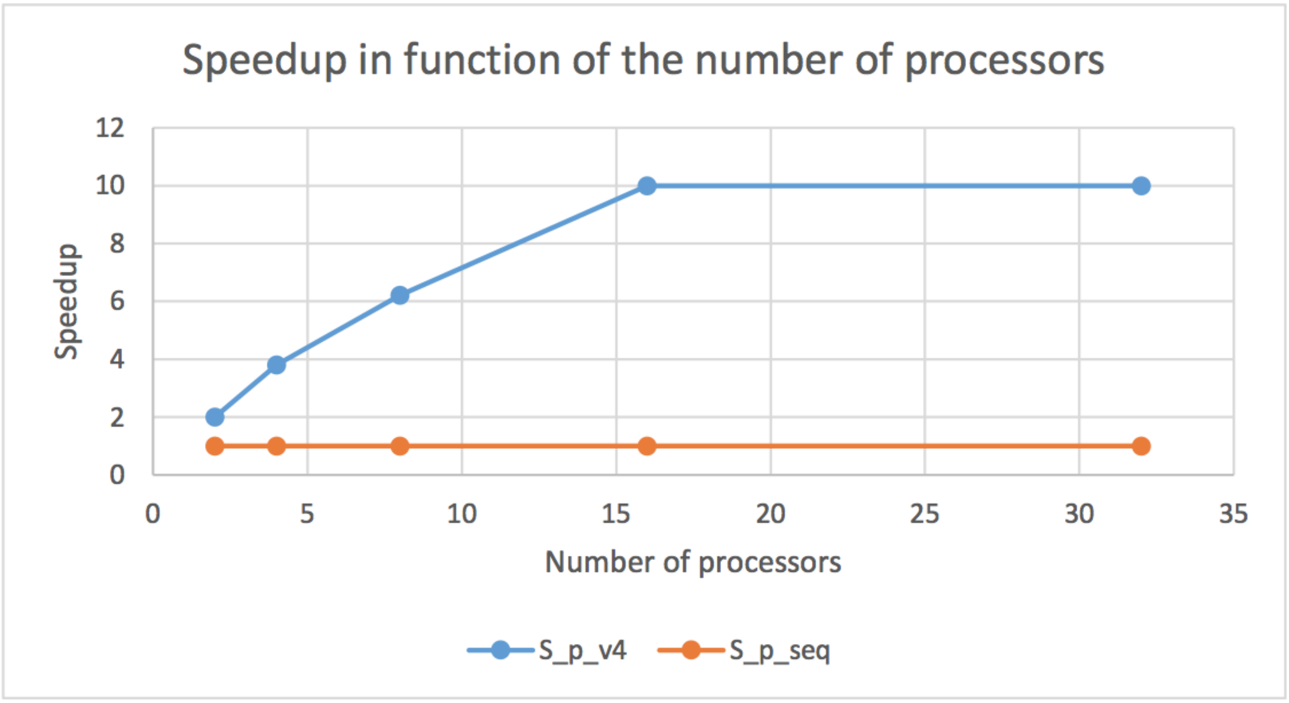
10.- With the results from the parallel simulation with 2, 4, 8, 16 and 32 processors, draw the execution time and speedup plots for version v4 with respect to the sequential execution (that you can estimate from the simulation of the initial task decomposition that we provided in 3dfft seq.c, using just 1 processor).

Here is the results we obtain executing the parallel simulation for version v4 with respect to the sequential execution.

The first graphic represents the evolution of the execution time in function of the number of processors (2, 4, 8, 16 and 32) for both v4 (blue curve) and seq (grey curve) versions.



The second graphic represents the evolution of the speedup in function of the number of processors (2, 4, 8, 16 and 32) for both v4 (blue curve) and seq (orange curve) versions.



We can conclude that the results are coherent with the theory we studied during the lectures. Thus, we can say that the version v4 that we implement parallelize with efficiency the given code.