Roberto Meroni – cpds1102

Introduction to the Parallel Execution Environment

deliverable for assignment 0

CPDS – PARALLELISM for MIRI (HPC specialization)

28/09/2023

NODE ARCHITECTURE AND MEMORY

1. ARCHITECTURE CHARACTERISTICS OF MULTIPROCESSOR SERVER **BOADA**, NODES 11 TO 14

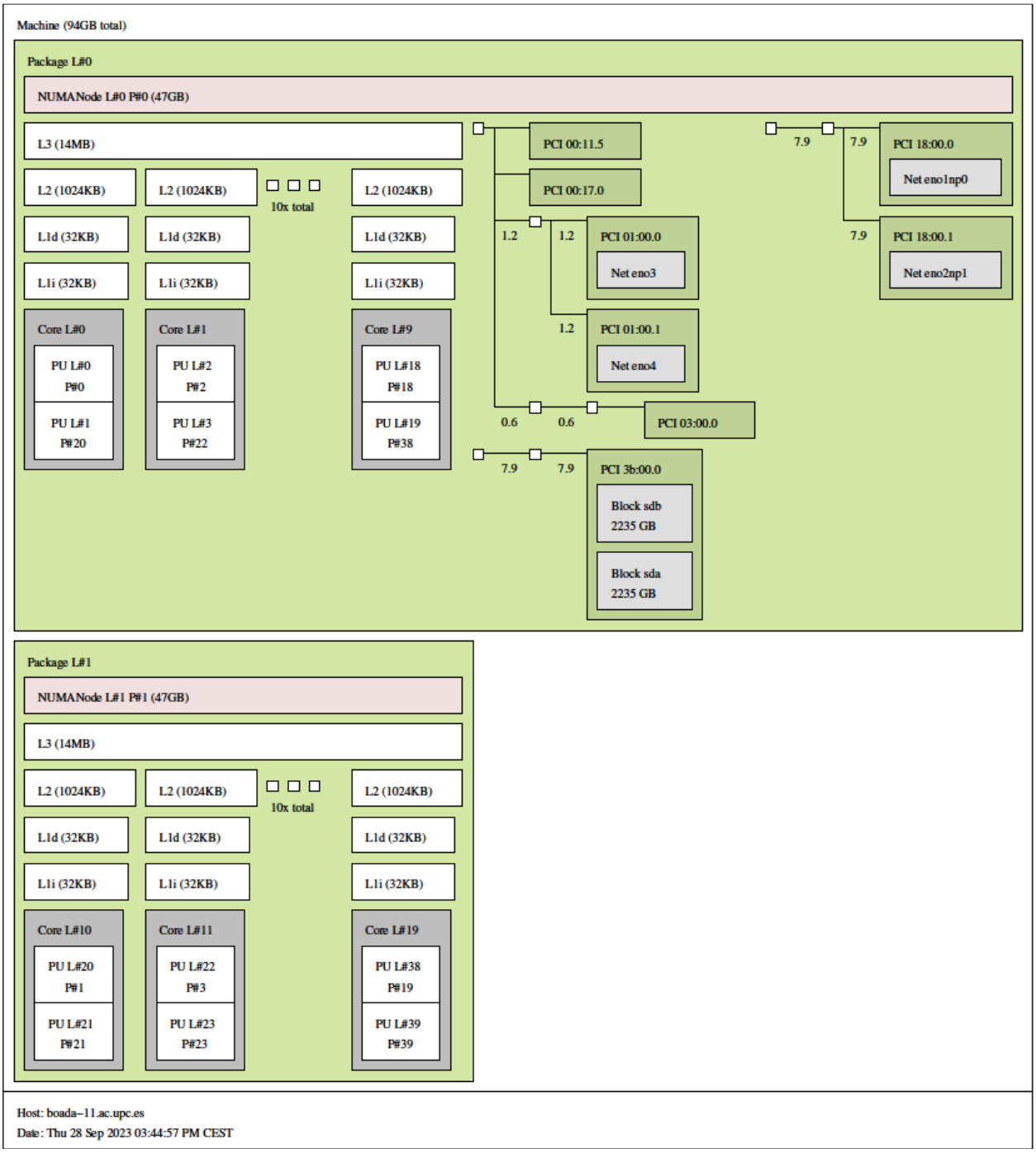
	boada-11 to boada-14
Number of sockets per node	2
Number of cores per socket	10
Number of threads per core	2
Maximum core frequency	3200 MHz
L1-I cache size (per-core)	32 KiB
L1-D cache size (per-core)	32 KiB
L2 cache size (per-core)	1 MiB
Last-level cache size (per-socket)	13.75 MiB
Main memory size (per socket)	47 GB
Main memory size (per node)	94 GB

The data have been deducted using the system monitoring utilities *lscpu* and *lstopo* on the node 11.

This refers to the node 11, but can be assumed valid for all the nodes in the same partition (as said in the Assignment, *all the nodes within a particular partition are identical*). Here the table of the nodes structure:

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

2. ARCHITECTURAL DIAGRAM OF BOADA-11



Again, this can be assumed valid for all the nodes from 11 to 14.

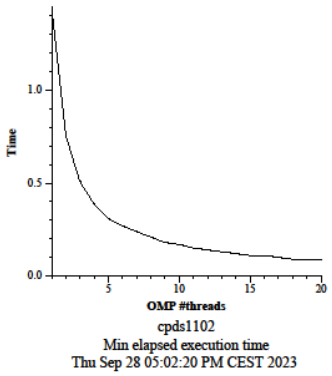
TIMING SEQUENTIAL AND PARALLEL EXECUTIONS

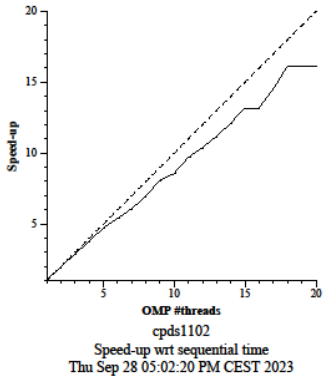
3. COMPUTATION OF PI FROM INTEGRAL

Strong Scalability

In strong scalability we exploit parallelism to obtain a significant reduction in the execution time of a program.

To test the strong scalability of the pi_omp problem, it has been solved varying the number of threads from 1 to 20.

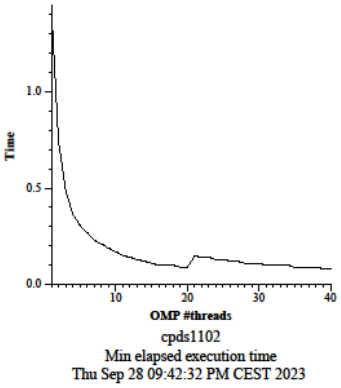


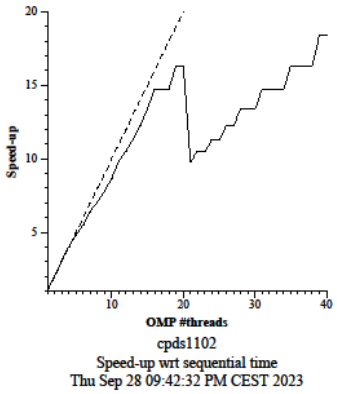


By increasing the number of threads, the elapsed time decreases significantly.

The speedup graph is close to the ideal speedup, meaning the problem has a very good scalability. In fact, the program approximates the PI number by calculating an integral of a specific function. The value of the integral is equal to the area defined by the function, and this can be approximated by dividing the area into small rectangles and summing them up. The areas of the rectangles have no dependency between each other, hence every area can be calculated in a different thread and at the same time (parallel execution). The sum of all the areas can be optimized in parallel too (by summing the areas two by two).

When going from running 20 threads to 40 threads we initially see a decrease in performance. We eventually get some improvement in elapsed time, but not as good as we could expect. This is because the total number of the cores is 20, so running more threads than cores can lead to oversubscription and introduce memory and synchronization overheads.



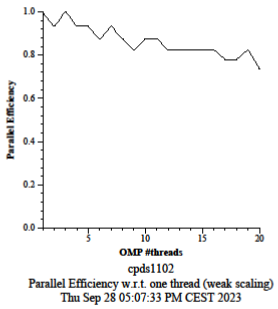


Weak Scalability

In weak scalability we exploit parallelism to increase the size of a problem while maintaining a reasonable execution time.

To test the weak scalability of the pi_omp problem, it has been solved by varying the number of threads from 1 to 20, and the problem size from 100.000.000 to 2.000.000.000 (problem size increases by 100.000.000 for every thread added).

#threads	Problem size	Elapsed min
1	100000000	0.14
2	200000000	0.15
3	300000000	0.14
4	400000000	0.15
5	500000000	0.15
6	600000000	0.16
7	700000000	0.15
8	800000000	0.16
9	900000000	0.17
10	1000000000	0.16
11	1100000000	0.16
12	1200000000	0.17
13	1300000000	0.17
14	1400000000	0.17
15	1500000000	0.17
16	1600000000	0.17
17	1700000000	0.18
18	1800000000	0.18
19	1900000000	0.17
20	2000000000	0.19



By increasing the problem size proportionally to the number of threads, the elapsed time doesn't have a significant penalty. The graph is quite close to the ideal graph (Efficiency = 1 for every thread number).

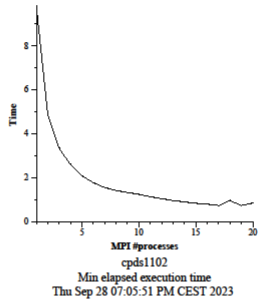
As expected from the same motivations we reasoned for Strong Scalability, the problem has a good Weak Scalability too.

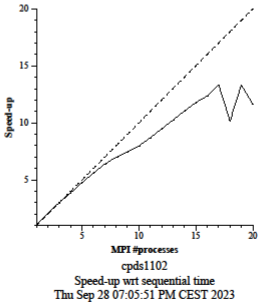
4. Computation of Pi with Monte Carlo

Each simulation of the Monte Carlo algorithm can be executed in parallel (and aggregated in the end), so I expect this program to be highly scalable.

To test the strong scalability of this problem, it has been solved by varying the number of MPI processes from 1 to 20.

MPI





#MPI processes	Elapsed min
1	9.840415
2	4.928346
3	3.374737
4	2.625806
5	2.112349
6	1.796535
7	1.569485
8	1.431782
9	1.338228
10	1.253929
11	1.150687
12	1.055054
13	0.974535
14	0.905401
15	0.846673
16	0.807239
17	0.747876
18	0.981453
19	0.749742
20	0.857390

By increasing the number of MPI processes, the elapsed time decreases significantly.

The speedup graph is close to ideal, the algorithm has excellent scalability.

MPI + OPENMP

Now we test again the Strong Scalability of this program, but this time combining MPI and OpenMP.

We are using MPI to communicate between 2 processes, and OpenMP for parallelization within the processes. For each process the number of threads varies from 1 to 20.

Threads per Process	Elapsed Time (seconds)
1	4.929648
2	2.466945
3	1.643427
4	1.265255
5	1.052350
6	0.875975
7	0.754195
8	0.668129
9	0.604041
10	0.543646
11	0.571194
12	0.523543
13	0.483787
14	0.450727
15	0.422725
16	0.396878
17	0.374220
18	0.353226
19	0.335751
20	0.319395

Elapsed times	
1 MPI process	9.840415
20 MPI processes	0.857390
2 MPI processes each using 20 OpenMP threads	0.319395

As expected, parallelizing the program with 20 MPI processes instead of 1 MPI process has a huge speedup.

More interesting, by combining MPI and OpenMP we have obtained a remarkable improvement in the elapsed time, rather than using only MPI. This hybrid parallelization system applied to the Boada architecture allows to leverage both shared memory (OpenMP) and distributed memory (MPI).