HPC Laboratory Assignment

Lab 1: Parallel Execution Environment

Bioinformatics Degree – Second course, Third Term.

Done by Ilia Lecha Santamaria

User identifier: hpc1217

27/04/2023 – Campus Mar, Barcelona.

Table of Contents

Introduction

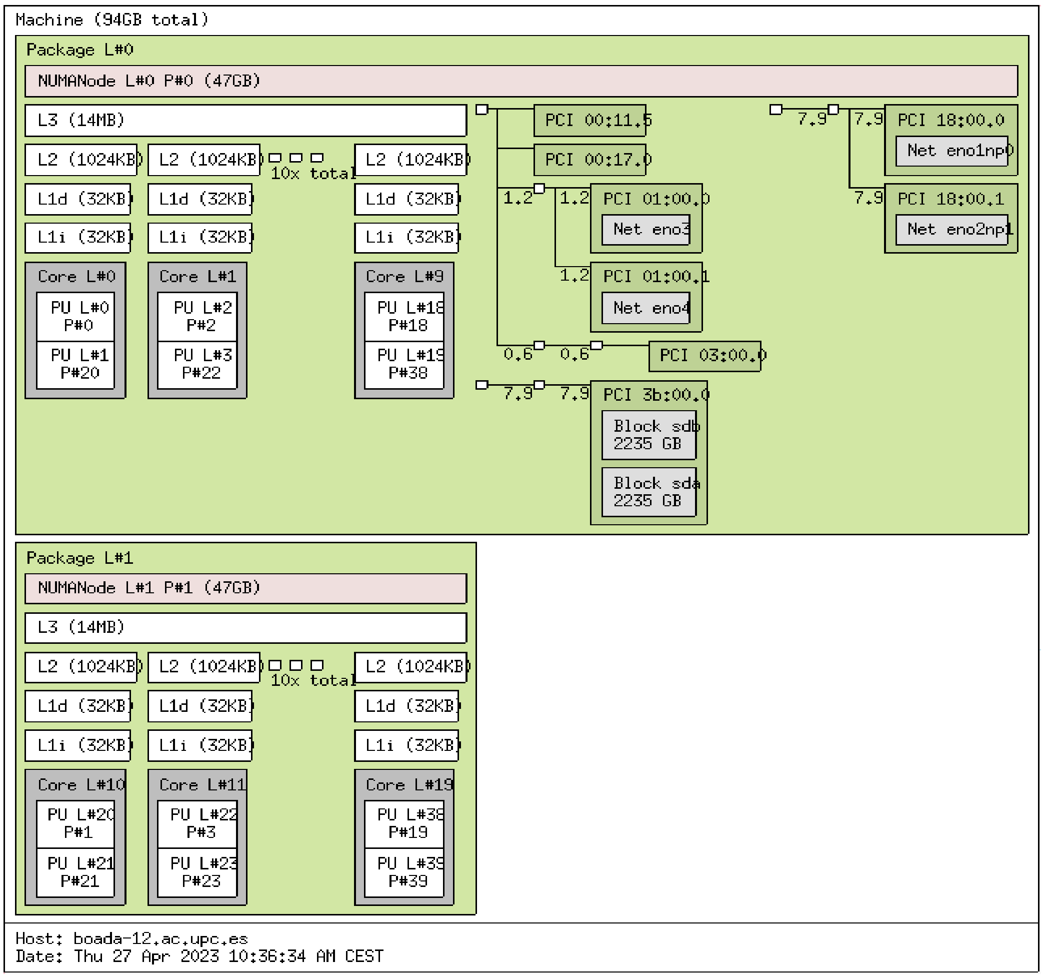
In this first practical I am going to get familiarized with the cluster envoirnment by checking some features of the architecture and memory, executing some examples of serialized and parallized programs for computing Pi approximations. For further understanding an analysis of the execution times and Speedup when comparing Serial and Parallel paradigms.

Node architecture and memory

1. Complete the following table with the relevant architectural characteristics of the different node types available in boada:

	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	3200MHz
L1-I cache size (per-core)	640 KiB (20 instances) (32KiB)
L1-D cache size (per-core)	640 KiB (20 instances) (32KiB)
L2 cache size (per-core)	20 MiB (20 instances) (1 MiB)
Last-level cache size (per-socket)	27.5 MiB (2 instances) (13.75 MiB)
Main memory size (per socket)	16GiB
Main memory size (per node)	47GiB

2. Include in the document the architectural diagram for one of the nodes boada-11 to boada-14.



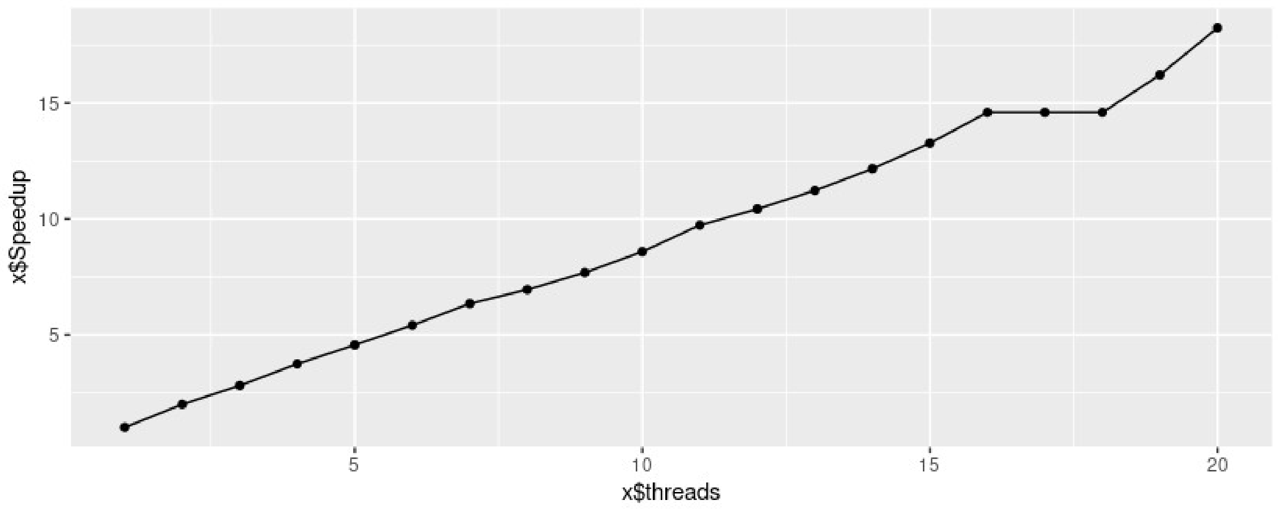
Timing sequential and parallel executions

For each of the following sections show the plots or tables required. But, in addition, please provide some reasoning about the results obtained. For instance, comment whether the scalability is good or not, and why.

3. Plot the execution time and speed–up that is obtained when varying the number of threads (strong scalability) by submitting the jobs to the execution queue (section 1.4.3).

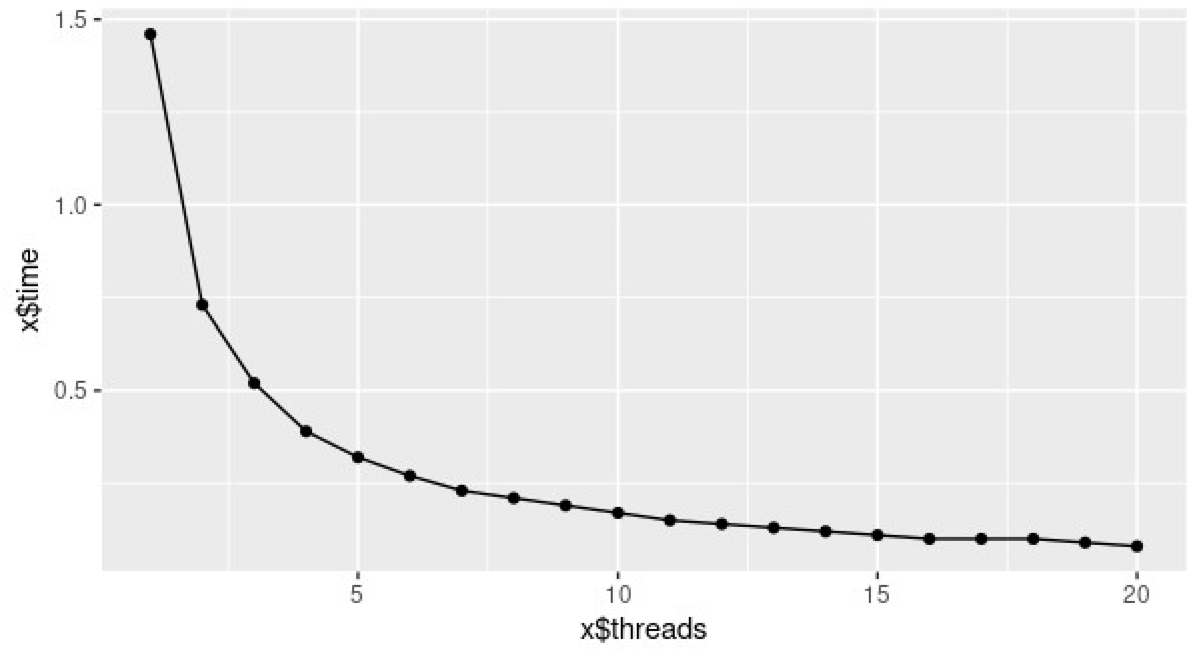
Plot of relation between number of threads and the speedup:

As can be seen, as the number of threads increase, the speedup increases as well.



Plot of relation between number of threads and the elapsed time:

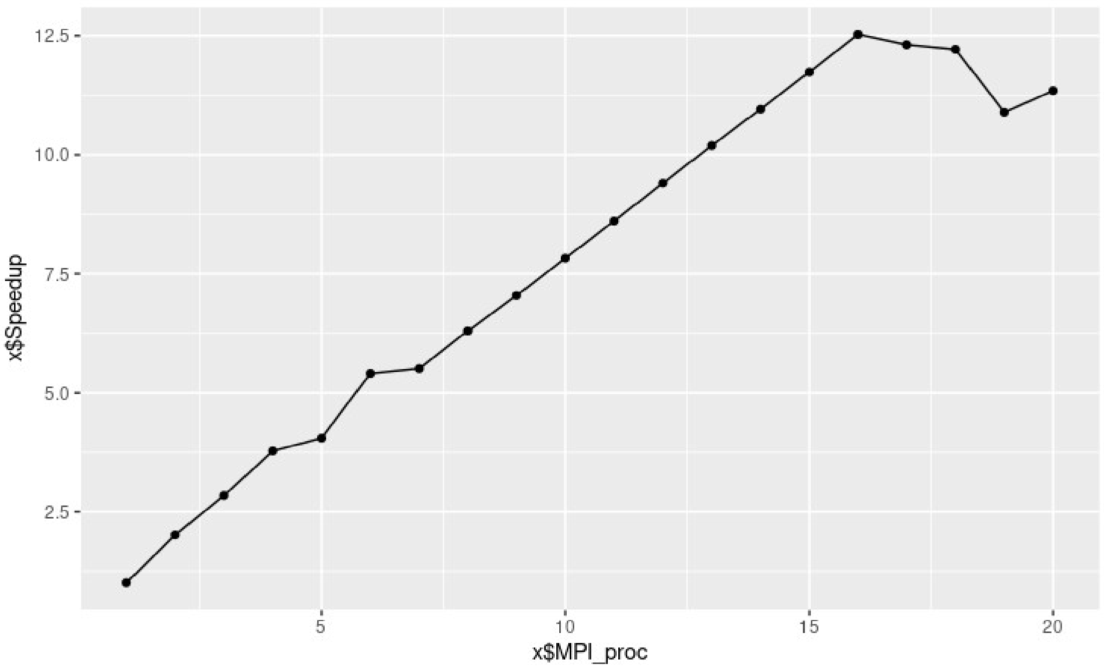
As can be seen as the number of threads increase, the time decreases really quickly.



4. Plot the execution time and speed–up that is obtained when varying the number of MPI processes from 1 to 20 (strong scalability) by submitting the jobs to the execution queue (section 1.5.2).

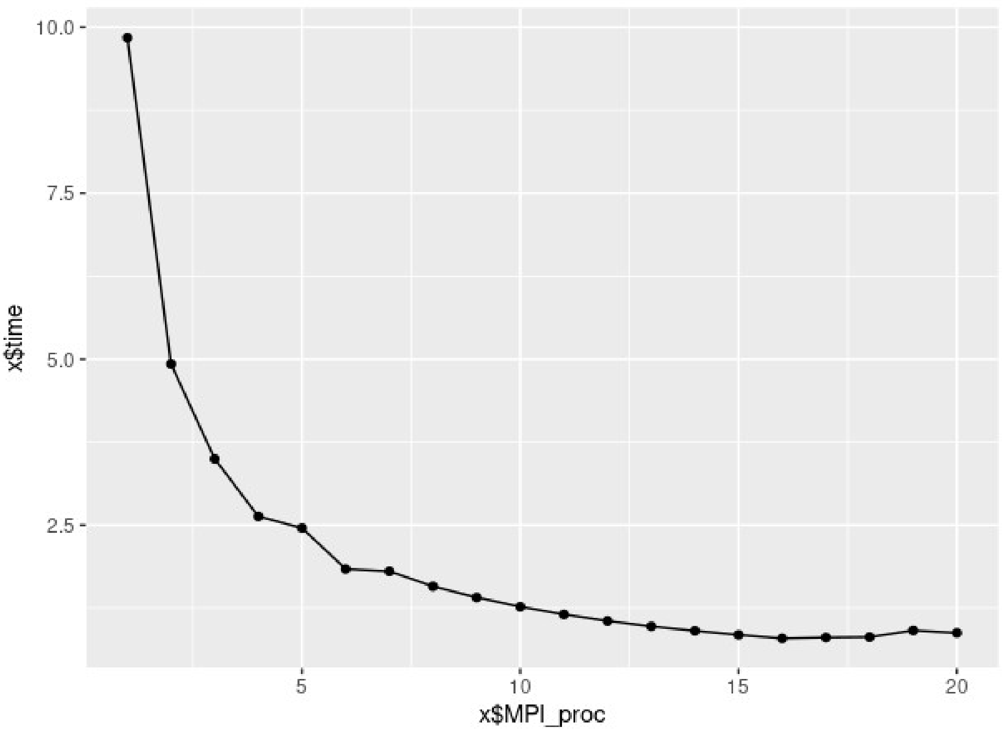
Plot of relation between number of MPI processes and the speedup:

As the number of processes increase, the speedup increases as well.



Plot of relation between number of MPI processes and the time:

As the number of processes increase, the time dicreases.



In addition, show in a table the elapsed execution time when executed with 2 MPI processes when varying the number of threads from 1 to 20.

	Processes	Threads	Time
0	2	1	4.921567
1	2	2	2.466497
2	2	3	1.751128
3	2	4	1.316400
4	2	5	1.086223
5	2	6	0.905747
6	2	7	0.776483
7	2	8	0.679629
8	2	9	0.604338
9	2	10	0.544537
10	2	11	0.571621
11	2	12	0.523772
12	2	13	0.485484
13	2	14	0.450675
14	2	15	0.422935
15	2	16	0.396436
16	2	17	0.748262
17	2	18	0.353468
18	2	19	0.335879
19	2	20	0.319017

Which was the original sequential time?; the time with 20 MPI processes?; and the time with 2 MPI processes each using 20 OpenMP threads?

Original sequential time: 0:02.69 seconds. (Obtained from run-seq.sh pi_seq 1000000000)

Time with 20 MPI processes: Approximately 1 second (Extracted from the graph)

Time with 2 MPI Proc using 20 threads: 0.319017 seconds.

MPI

Explain the deadlock problem and how it can be fixed:

Deadlock is a situation that can occur in a computer system where two or more processes are unable to proceed because they are stuck waiting for each other to release resources. Essentially, a deadlock is a state of affairs where each process is holding a resource that another process needs in order to complete, resulting in a deadlock.

There are several methods for resolving deadlocks, including:

1. For example, one way to prevent deadlocks is to use a resource ordering protocol, which ensures that resources are allocated in a specific order to prevent circular waits.

2. Avoidance: This approach involves dynamically checking whether a resource allocation request will lead to a deadlock, and only granting the request if it will not. One way to detect a deadlock is by constructing a wait-for graph, which shows the dependencies between processes and resources. When a deadlock is detected, recovery can be achieved by either killing one or more processes or by releasing resources held by one or more processes.