Sergi Ocaña Alamo

NIA: 107746

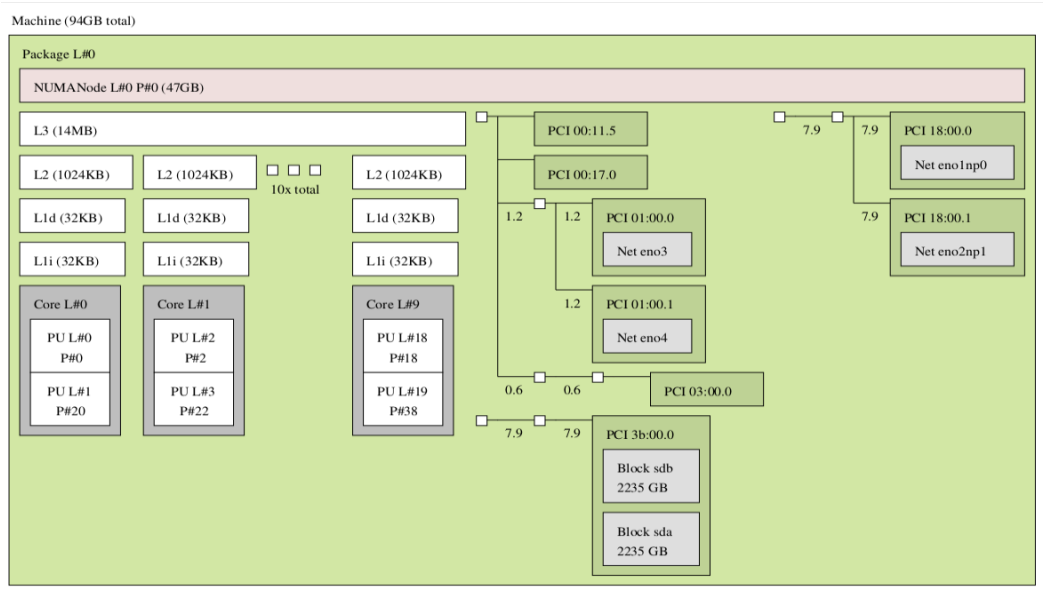
LAB 1

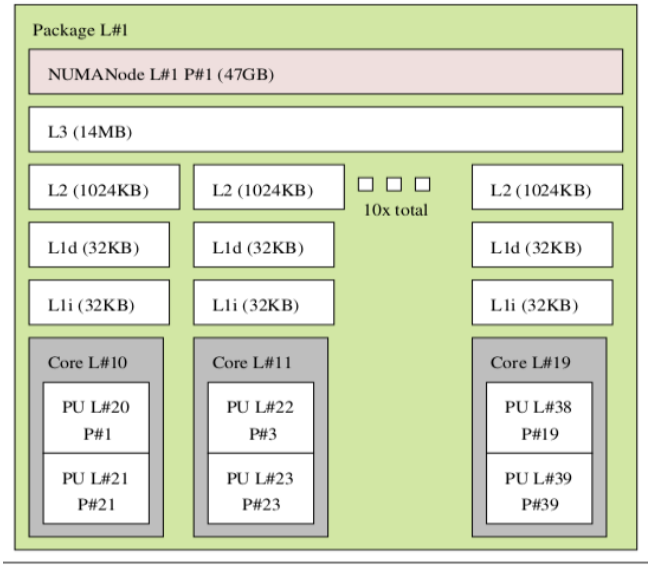
Parallel execution environment

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

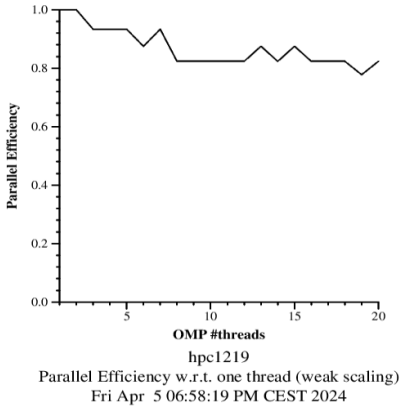
	Boada-11 to boada-14
Number of sockets per node	2
Number of cores per socket	10
Number of threads per core	2
Maximum core frequency	3200 Hz
L1-I cache size (per-core)	32 KB
L1-D cache size (per-core)	32 KB
L2 cache size (per-core)	1024 KB
Last-level cache size (per-socket)	14 MB
Main memory size (per socket)	47 GB
Main memory size (per node)	94 GB

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

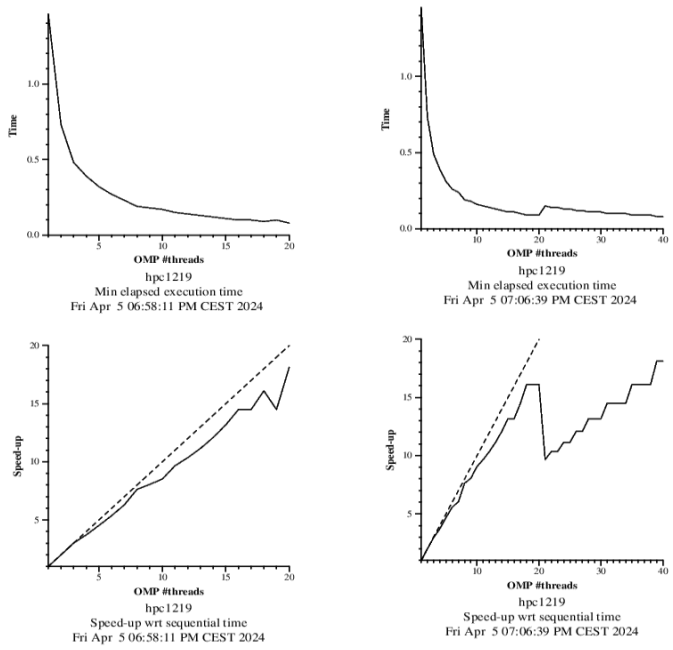




3. Plot the execution time and speed–up that is obtained when varying the number of threads (strong scalability) by submitting the jobs to the execution queue (section 1.4.3). It you did the optional part, show the resulting plot and comment the reason for this behaviour. Show the parallel efficiency obtained when running the weak scaling test. Explain what strong and weak scalability refer to, exemplifying your explanation with the plots that you present.



As we can see in the weak scalability increasing the threads it does not affect the efficiency of it. From the following graph we observed that starting form 1 which is the maximum, we want to have a better time by increasing the number of threads but the lowest we go is around 0.8 which is still a good result . Weak scaling deals with how execution time behaves with an increasing number of processors, while proportionally increasing the problem size per processor to maintain a constant workload per processor, because the problem magnitude increases as the number of threads increases. In order to run the application with a higher problem size in the same execution time, parallelism is used. This suggests that the task decomposition's granularity is maintained constant, so each thread completes the same amount of work

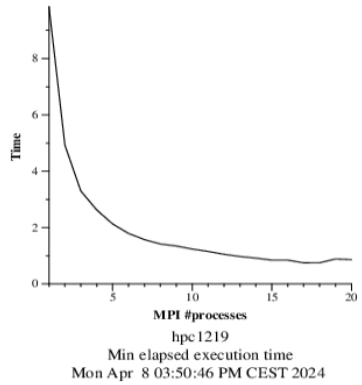


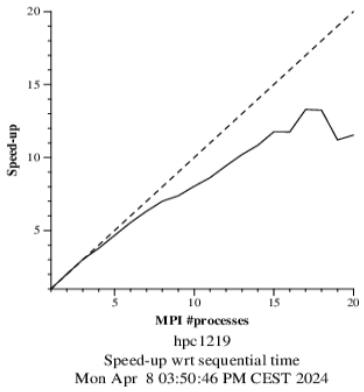
Strong scalability is used when you have a program to be executed and you divided in different parts to used parallelism and execute them to reduce the time. As we can see in the figure from the left the time is reduced a lot. The amount of work to be performed per processor (the granularity of the task decomposition), gets finer as the number of processors increases.

The optimum speed-up as we increase the number of threads is represented by the diagonal dashed line. The speed-up is represented by a continuous line, which more or less, increase linearly at first but stops increasing at the end. This suggests that utilising more than 17 threads has no effect on performance instead, it actually reduces speed up.

The difference between the figures from the left and the right is the number of threads which passes from 20 to 40 but the time does not reduce and the speed up also remains at the end the same this is because we only have 20 threads so when we want to use 40, we simply don't have them so it just continues with the 20 that we were working with.

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). 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 . Which was the original sequential time? the time with 20 MPI processes? and the time with 2 MPI processes each using 20 OpenMP threads? You can retrieve such information for 1 and 20 MPI processes from file elapsed.txt; and for 2 MPI processes each with 20 threads within the output file created after the execution of sbatch submit-mpi2-omp.sh.





1	9.841627
2	4.927373
3	3.298825
4	2.626597
5	2.128850
6	1.792070
7	1.572814
8	1.418509
9	1.346853
10	1.237852
11	1.152386
12	1.053204
13	0.974174
14	0.916617
15	0.844961
16	0.845499
17	0.747619
18	0.750925
19	0.886855
20	0.861821

The original time was about 9.84

For the 20 MPI processes we see our time has gone to 0.86.

```
Launching 2 MPI processes. Number of threads per process: 15
Number pi after 1073741824 iterations = 3.141513824462891
0.422741
Launching 2 MPI processes. Number of threads per process: 16
Number pi after 1073741824 iterations = 3.141489505767822
0.398050
Launching 2 MPI processes. Number of threads per process: 17
Number pi after 1073741824 iterations = 3.141508579254150
0.373839
Launching 2 MPI processes. Number of threads per process: 18
Number pi after 1073741824 iterations = 3.141491413116455
0.353418
Launching 2 MPI processes. Number of threads per process: 19
Number pi after 1073741824 iterations = 3.141519784927368
0.335848
Launching 2 MPI processes. Number of threads per process: 20
Number pi after 1073741824 iterations = 3.141507148742676
0.318808
hpc1219@boada-6:~/lab1/MPI/pi$
```

Launching 2 MPI processes. Number of threads per process: 20

Number pi after 1073741824 iterations = 3.141507148742676

Time: 0.318808

Understanding MPI

A) We start by initialising a few variables that are needed for the MPI_Send and MPI_Recv functions.

Next, we have an if-and-else condition:

- If we are in the master node, we put the first conditional.
- If we are not in the master node (rank!= 0), we go on to the second conditional.

Next, two messages will be printed by each process:

- Following transmission, the message will print: The communication has been dispatched.
- Following receipt of the message, it will print: We've gotten the message.

When we run the programme, the first message appears but then the programme becomes stuck, and we fail to receive the subsequent message so we have Deadlock.

The issue is that MPI_Send is being called by both processes (they are both attempting to call MPI_Send). Calls will only return when the other processes make the reception because MPI_Send is blocking. However, because another process is attempting to deliver the message as well, the reception won't occur.

B) Show the code excerpt related to the MPI Gather and MPI Scatter collective operations. Also, show the output of their execution in program collectives (only the output of these two operations, not the whole output of the program), explaining it briefly.

```
  /*----------------------------------*/
 /* MPI_Gather()                     */
/*----------------------------------*/

x[0] = alphabet+Iam;
for (i=0; i<p; i++) {
      y[i] = ' ';
}
MPI_Gather(x,1,MPI_CHAR,          /* send buf,count,type */
           y,1,MPI_CHAR,          /* recv buf,count,type */
           root,                  /* root (data origin)  */
           MPI_COMM_WORLD);       /* comm                */

printf(" MPI_Gather     :  %d ", Iam);
for (i=0; i<p; i++) {
  printf("  %c",x[i]);
}
printf("     ");
for (i=0; i<p; i++) {
  printf("  %c",y[i]);
}
printf("\n");

MPI_Barrier(MPI_COMM_WORLD);
```

We create the MPI_Gather operation, which gathers data from each process and stores it in process = 1, the root process. In this instance, data from the array "x" will be sent by each process to the root process's array "y."

Then, "MPI_Gather:" and the process number for each process will be printed.

Next, we add two more for loops:

- The first one prints the data sent by each process, according to the contents of array 'x'.

- The other one uses the MPI_Gather primitive to print the contents of array 'y,' which corresponds to the data received by the root process.

Function	Process	Send buf	Recv buf
MPI_Gather	0	a	
MPI_Gather	2	c	
MPI_Gather	3	b	
MPI_Gather	1	d	a b c d

```
  /*----------------------------------*/
 /* MPI_Scatter()                    */
/*----------------------------------*/

for (i=0; i<p; i++) {
  x[i] = alphabet+i+Iam*p;
  y[i] = ' ';
}
MPI_Scatter(x,1,MPI_CHAR,      /* send buf,count,type */
            y,1,MPI_CHAR,      /* recv buf,count,type */
            root,              /* root (data origin)  */
            MPI_COMM_WORLD);   /* comm                */

printf(" MPI_Scatter   :  %d ", Iam);
for (i=0; i<p; i++) {
  printf("  %c",x[i]);
}
printf("     ");
for (i=0; i<p; i++) {
  printf("  %c",y[i]);
}
printf("\n");
```

To disseminate the data from the root process to every other process, we create the MPI_Scatter operation. It is analogous to a dealer (root) dealing all of the players (other processes) the cards (items of array x).

Subsequently, each process will output "MPI_Scatter:" along with its process number.

Next, we add two more for loops:

- The first will output what's in the array "x."

As we can see, the other processes will get the contents of the root's array "x."

- The MPI_Scatter primitive will enable the second one to display the contents of array 'y,' which corresponds to the data received by each process.

We need to keep in mind that the members of array "x" of the root process match to the data that was received.

Function	Process	Send buf	Recv buf
MPI_Scatter	1	e f g h	f
MPI_Scatter	3	m n o p	h
MPI_Scatter	0	a b c d	e
MPI_Scatter	2	I j k l	g

5. Create an MPI program of your own which includes calls to several MPI primitives. Here's your opportunity to be creative! The code should include some message exchanges with both 1) point to point primitives and 2) at least one collective communication primitive. Test it with several processes. In your report, you should show the code and explain briefly your implementation and its functionality.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        char message[] = "Hello from rank 0";
        int message_size = strlen(message) + 1; // Include null terminator
        printf("Rank 0 emitting the message: %s\n", message);
        MPI_Bcast(&message_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(message, message_size, MPI_CHAR, 0, MPI_COMM_WORLD);
    } else if (rank <= 3) { // Only ranks 3 and higher receive the message
        int message_size;
        MPI_Bcast(&message_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
        char *message = (char *)malloc(message_size * sizeof(char));
        MPI_Bcast(message, message_size, MPI_CHAR, 0, MPI_COMM_WORLD);
        printf("Rank %d received message: %s\n", rank, message);
        free(message);
    }

    MPI_Finalize();
    return 0;
}
```

Includes: The code includes header files for standard input/output operations (stdio.h), memory allocation (stdlib.h), string usage (string.h), and MPI functions (mpi.h).

Main Function: The main function is the entry point of the program. It takes command-line arguments argc and argv[].

MPI Initialization: The code initializes MPI by calling MPI_Init(), passing pointers to argc and argv[] as arguments.

Rank and Size Determination: It retrieves the rank and size (total number of processes) of the MPI communicator.

Rank 0 Message Broadcast: If the rank is 0, the process creates a message ("Hello from rank 0";). It then broadcasts the message size using MPI_Bcast() and the message itself using another MPI_Bcast() call.

Message Reception by Other Ranks: For ranks other than 0 and not greater than 4, the code receives the broadcasted message size using MPI_Bcast() and allocates memory for the message. It then receives the broadcasted message using MPI_Bcast() and prints the received message.

Confirmation message: All the others ranks that received the message return a new message confirming the transmission.

MPI Finalization: Finally, the code finalizes MPI execution by calling MPI_Finalize().