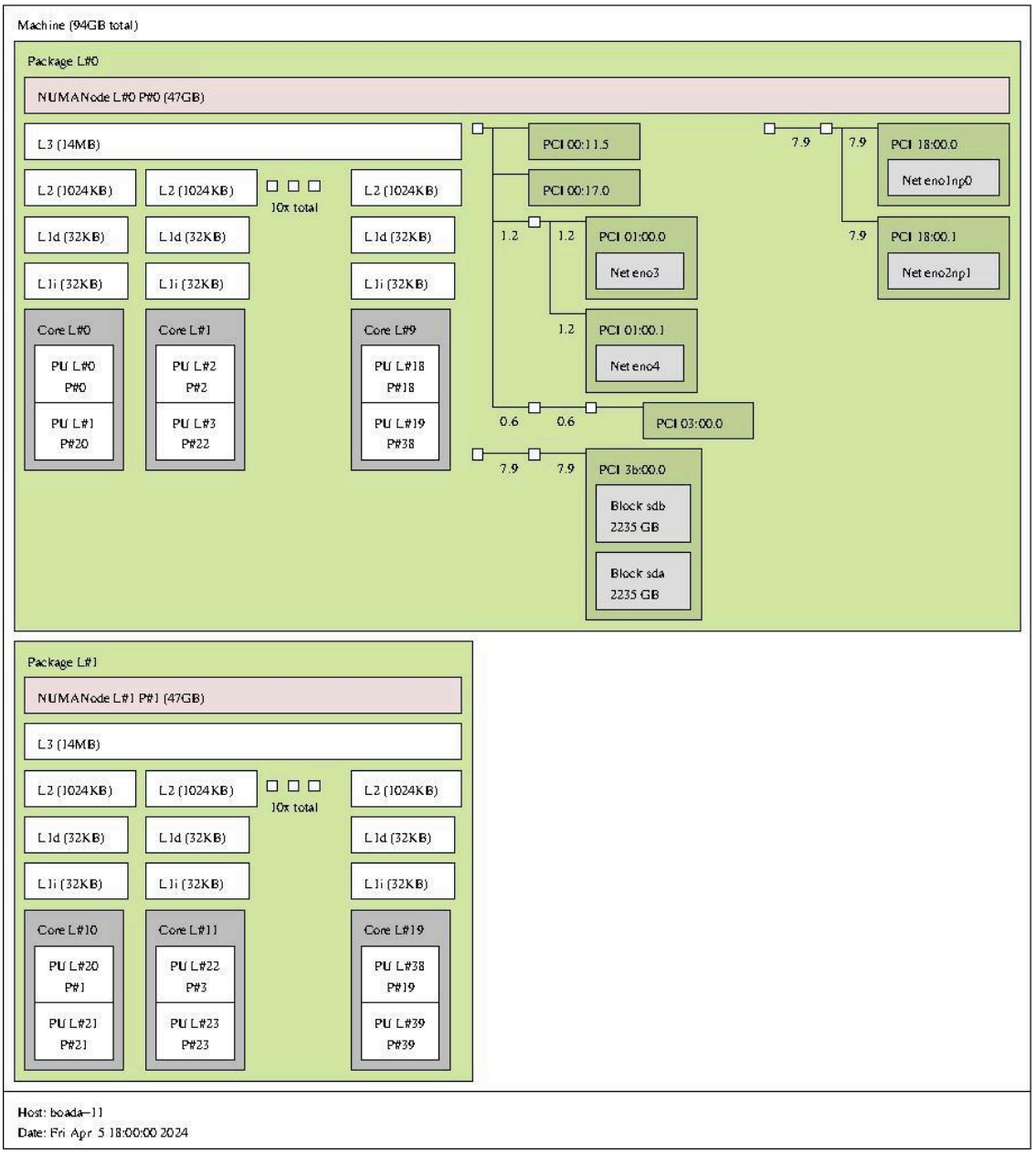
Lab 1:
Understanding Parallelism

Jan Izquierdo Ramos
hpc1211
8/04/2024
Bioinformatics

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

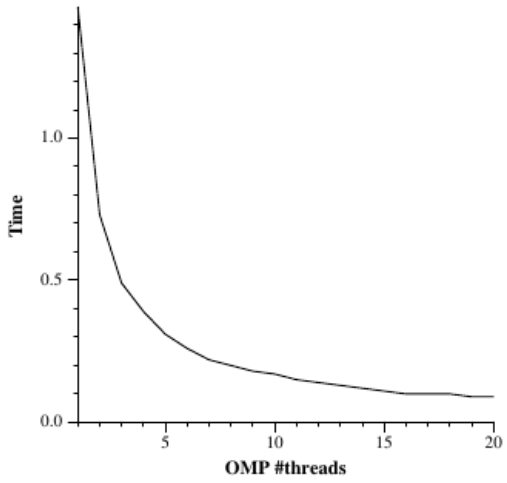
	Boada 11 to 14
Number of sockets per node	2
Number of cores per socket	10
Number of threads per core	2
Maximum core frequency	3200 MHz
L1-I cache size (per-core)	32 KiB
L1-D cache size (per-core)	32 KiB
L2 cache size (per-core)	1024 KiB
Last-level cache size (per-socket)	14 MB
Main memory size (per socket)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

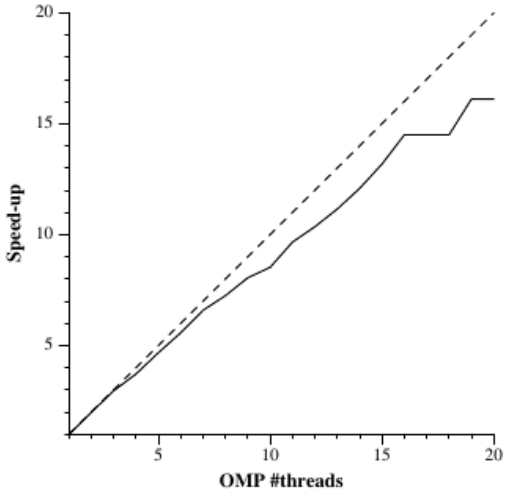
2. **Include in the document the architectural diagram for one of the nodes boada-11 to boada-14**



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). If you did the optional part, show the resulting plot and comment on the reason for this behavior. 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.**

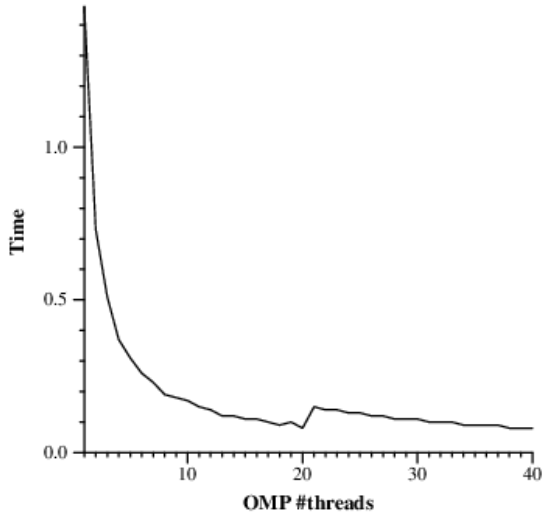
   Strong scalability distributes a problem among different processors, reducing the amount of work for each processor. It is used to reduce the execution time of the problem, we can see that in the plot below, as we increase the number of threads, the execution time goes down.

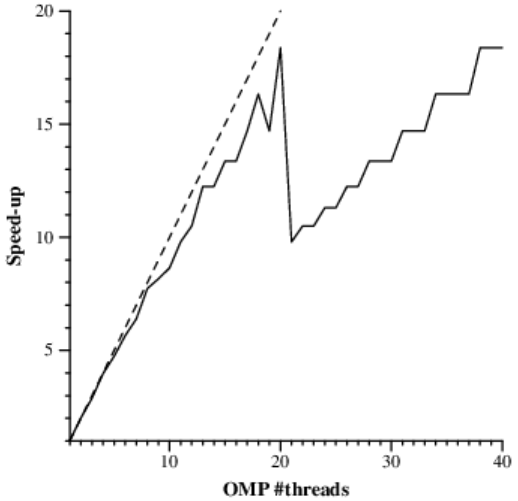




   The first figure plots a relation between the number of threads and the time it takes to execute a program, we can observe that as we increase the number of threads the execution time is reduced.
   The second plot shows a relation between the increase in execution speed and the number of threads, we can observe that the increase in the number of threads is almost proportional to the speed-up.

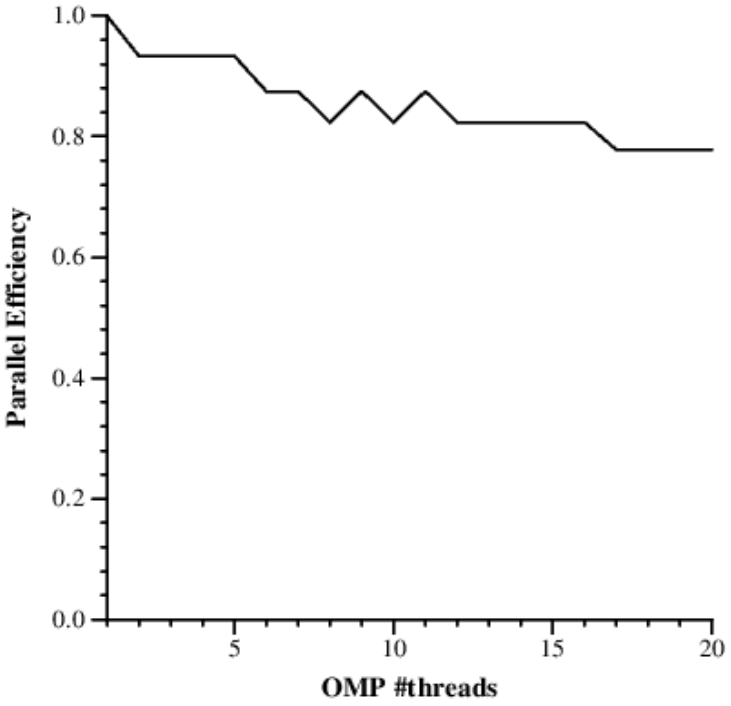
   OPTIONAL:





In the scalability plot, we can observe that the performance improvement does not scale proportionally to the number of threads. From 1 to 20 threads we can observe a similar improvement than the one we saw before, this because we are using available, existing cores. However, when we increase the number of threads past 20 a decrease in the performance appears in the graph, this happens because we are attempting to use more cores than the ones that exist in the machine, from this we can conclude that the amount of physical cores limits performance.

Weak scalability is used when working on a problem that is too big for 1 node to handle, then you will use parallelism to distribute the workload evenly across different nodes and threads, allowing for the execution of problems that would otherwise be too large to run on the same machine.

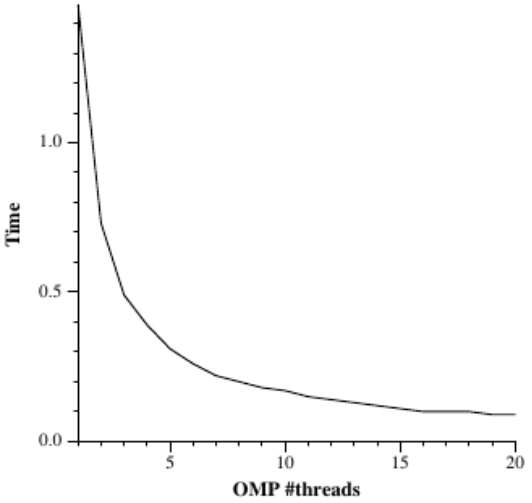


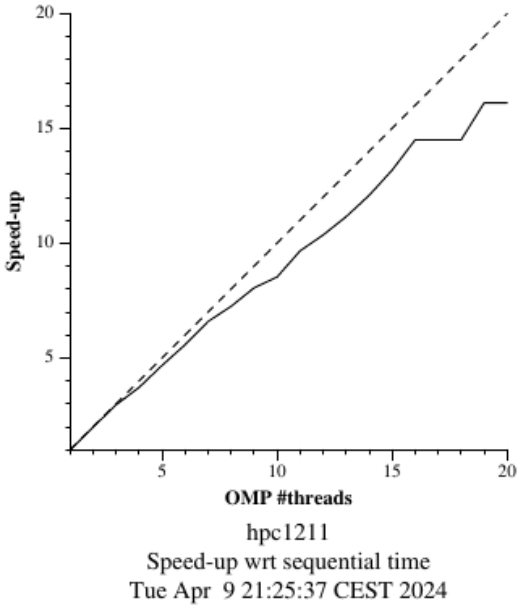
This plot shows that as we increase the number of threads the parallel efficiency decreases, the threshold that marks a too-low efficiency is 0.75, this decrease may happen because as the number of threads increases so does the communication overhang.

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 9. 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**

Threads per process	Elapsed time	Speedup
1	1.45	1
2	0.77	1.88
3	0.49	2.96
4	0.37	3.92
5	0.31	4.68
6	0.27	5.37
7	0.22	6.59
8	0.21	6.9
9	0.18	8.06
10	0.17	8.53
11	0.15	9.67
12	0.14	10.36
13	0.13	11.15
14	0.12	12.08
15	0.11	13.18
16	0.11	13.18
17	0.10	14.5
18	0.10	14.5
19	0.09	16.11
20	0.09	16.11

Threads per process	Time
1	9.844613
2	4.934280
3	3.330215
4	2.627359
5	2.121938
6	1.793750
7	1.61325
8	1.433365
9	1.328820
10	1.257226
11	1.154498
12	1.056119
13	0.975491
14	0.907182
15	0.860530
16	0.823399
17	0.756730
18	0.907627
19	0.896188
20	0.900581





The original sequential time was of 9.843565 (taken from elapsed.txt, generated in the execution of submit-strong-mpi.sh)
The time with 20 MPI processes was of 0.900581 (also taken from elapsed.txt)
The time with 2 MPI processes each using 20 OpenMP threads was of 0.319 seconds, this time can be seen in file “submit-mpi2-omp.sh.o[...]”

Deadlock problem:

```c
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
#define MSGLEN 2048
  int ITAG_A = 100;
  int ITAG_B = 200;
  int irank, i, idest, isrc, istag, iretag;
  float rmsg1[MSGLEN];
  float rmsg2[MSGLEN];
  MPI_Status recv_status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &irank);

  for (i = 0; i < MSGLEN; i++)
    {
      rmsg1[i] = 100;
      rmsg2[i] = -100;
    }
  if ( irank == 0 )
    {
      idest  = 1;
      isrc   = 1;
      istag  = ITAG_A;
      iretag = ITAG_B;
    }
  else if ( irank == 1 )
    {
      idest  = 0;
      isrc   = 0;
      istag  = ITAG_B;
      iretag = ITAG_A;
    }

  printf("Task %d has sent the message\n", irank);
  MPI_Ssend(&rmsg1, MSGLEN, MPI_FLOAT, idest, istag, MPI_COMM_WORLD);
  MPI_Recv(&rmsg2, MSGLEN, MPI_FLOAT, isrc, iretag, MPI_COMM_WORLD, &recv_status);
  printf("Task %d has received the message\n", irank);
  MPI_Finalize();
}
```

When we execute the program we see a couple of sent messages but no received ones, this is because the task gets stuck and we fail to receive messages.

A process sends a message and it blocks until the message is received by the destination process(calls only return when the other process makes a reception), but the destination process cannot receive the message because it sends a message first, repeating the situation for the next process, this causes a deadlock.
To fix it we can use non-blocking send and receive operations (MPI_Isend and MPI_Irecv) together with a MPI_Wait to ensure that the receiving process has been completed.

Gather and scatter

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);
```

MPI_gather collects data from all the processes and stores it in process 1. IN this case the data from array x in each process will be sent to array y (in process 1). Then the loops print the data in array x (the data sent by each process) and the contents of the y arrays (the data received by process 1)

```
 Function      Proc  Sendbuf          Recvbuf
 --------      ----  -------          -------
MPI_Gather    :  0  MPI_Gather    :  2   c
 a
MPI_Gather    :  3   d
MPI_Gather    :  1   b                a  b  c  d
MPI_Gatherv   :  0  MPI_Gatherv   :  2   d  e  f
 a
```

Function	Proc	Sendbuf	Rcvbuf
MPI_Gather	0	a	
MPI_Gather	2	c	
MPI_Gather	3	d	
MPI_Gather	1	b	abcd

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");

MPI_Barrier(MPI_COMM_WORLD);
```

MPI_Scatter emits data from process 1 (our root process) to all other processes. In this case it sends the array x to all other processes. The following for loops are used to print the x array (data sent by process 1) and to print the contents in array y (data received by each process)

```
MPI_Scatter    :  1    e  f  g  h        f
MPI_Scatter    :  2    i  j  k  l        g
MPI_Scatter    :  3    m  n  o  p        h
MPI_Scatter    :  0    a  b  c  d        e
```

Function	Proc	Sendbuf	Rcvbuf
MPI_Scatter	1	e f g h	f
MPI_Scatter	2	i j k l	g
MPI_Scatter	3	m n o p	h
MPI_Scatter	0	a b c d	e

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 <mpi.h>


int rank;
int nproc;

int main(int argc, char* argv[] ) {

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &nproc);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        int size=nproc-1;

        int send_data = 1;

        if (rank==0){
                printf("There should be %d process executions\n", nproc);

                MPI_Send(&send_data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
                printf("Process %d, sent data: %d\n",rank, send_data);
                } else{
                        int rcv_data;
                        MPI_Recv(&rcv_data, 1, MPI_INT, rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

                        send_data=rcv_data+1;
                        printf("Process: %d, recieved data: %d\n", rank, rcv_data);

                        if (rank<size){
                                MPI_Send(&send_data, 1, MPI_INT, rank+1, 0, MPI_COMM_WORLD);
                                printf("Process: %d, sent data: %d\n", rank, send_data);
                        }
                }
        printf("%d out of %d processes have been executed so far\n", send_data, nproc);

        int max_send_data;
        MPI_Reduce(&send_data, &max_send_data, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

        if (rank == 0) {
                printf("\n%d processes have been executed\n", max_send_data);
        }

        MPI_Finalize();
        return 0;
}
```

This code starts by defining size as the total number of processes -1, this is for use later, the it defines send_data as 1, and adding 1 to it every time it's sent and received by another process, this way we can get the total number of processes the program uses, we can also use it to track the progress of the execution. In the point to point primitives (MPI_send) we choose where we want to send send_data, for convenience we send it to the next process in order (0 goes to 1, 1 to 2, etc.), this does not sort the process execution, it only determines the communication pattern between processes. At the end of the code we use a collective communication primitive to obtain the largest number that has been defined as send_data, which is the total number of processes executed.