Practica-1.pdf

Bioinformatica

Computacion de alto rendimiento

2º Grado en Bioinformática

Escuela Superior de Comercio Internacional
Universidad Pompeu Fabra

HPC Laboratory Assignment

Lab 1. Parallel Execution Environment

Name: Lluís Giménez Gabarró

NIA: 106397

Username: hpc1206

Date: 28/04/2023

Academic Course: 2n Bioinformatics

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	3200.0000 MHz
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

coches.net

En este momento hay alguien teniendo relaciones sexuales dentro de un coche.

Y no eres tú.
Pero podrías serlo.

coches.net

No pierdas ni un minuto

Computacion de alto rendimiento

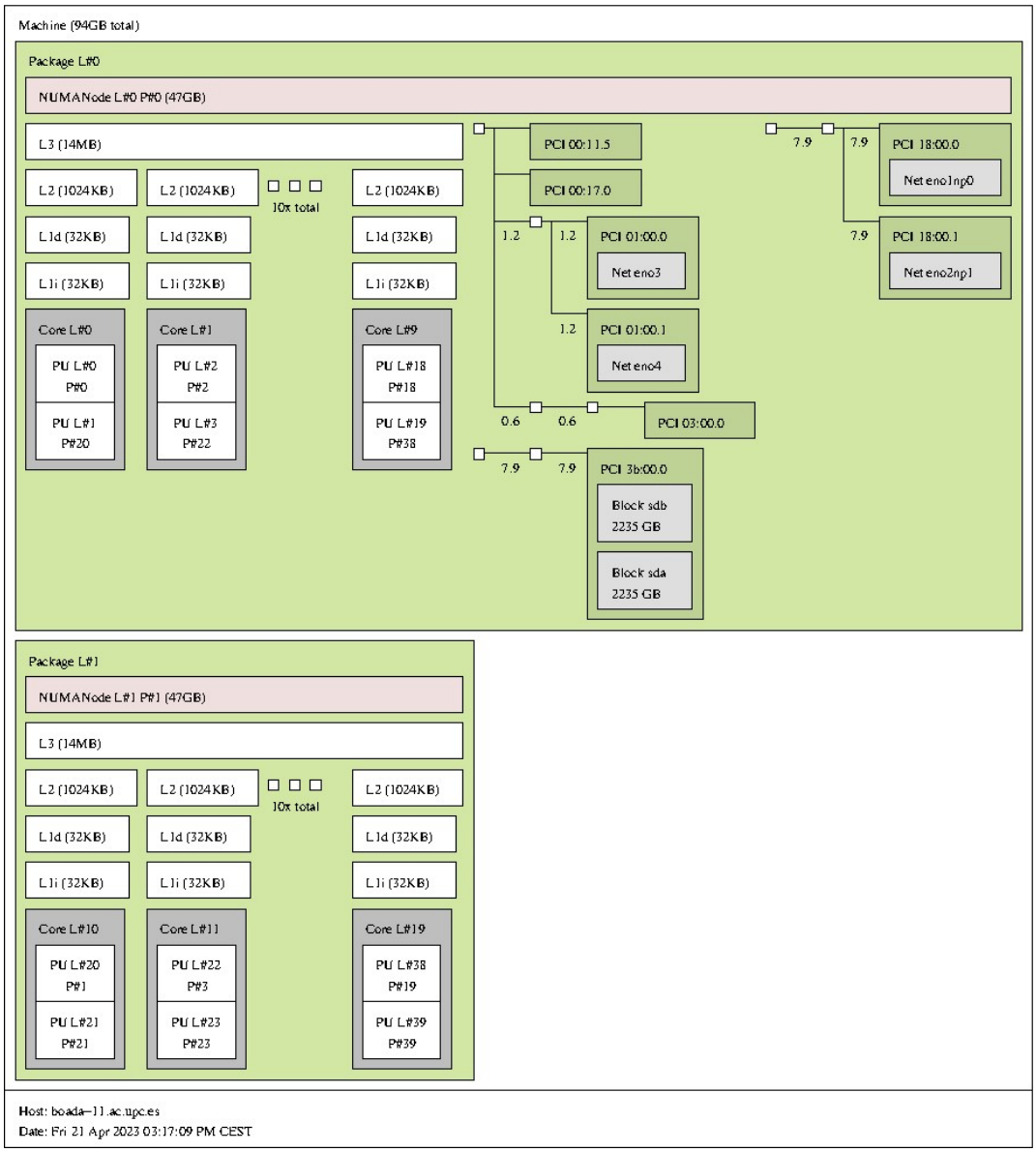
Banco de apuntes de la

WUOLAH

Comparte estos flyers en tu clase y consigue más dinero y recompensas

1. Imprime esta hoja
2. Recorta por la mitad
3. Coloca en un lugar visible para que tus compis puedan escanar y acceder a apuntes
4. Llévate dinero por cada descarga de los documentos descargados a través de tu QR

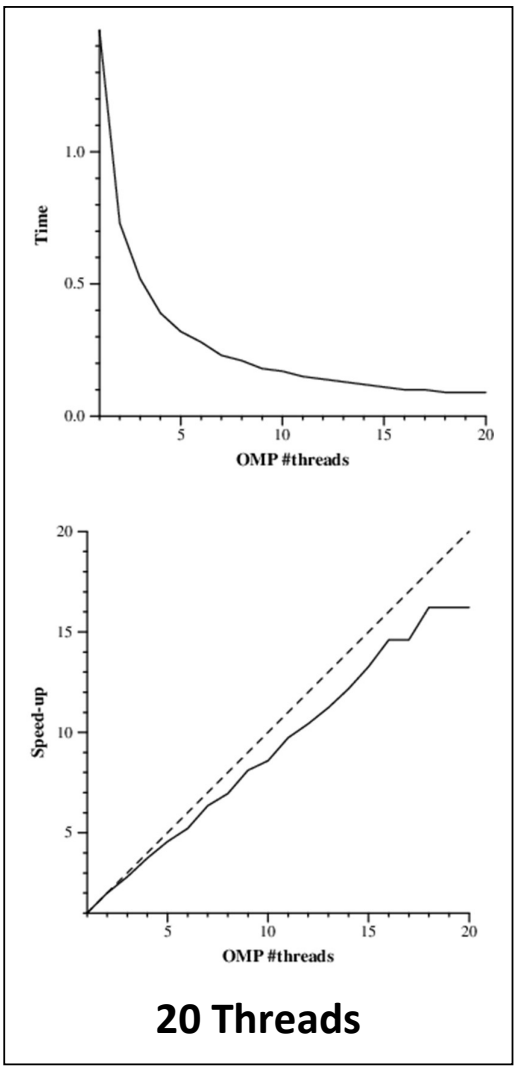
2. **Include in the document the architectural diagram for one of the nodes boada-11 to boada-14.**

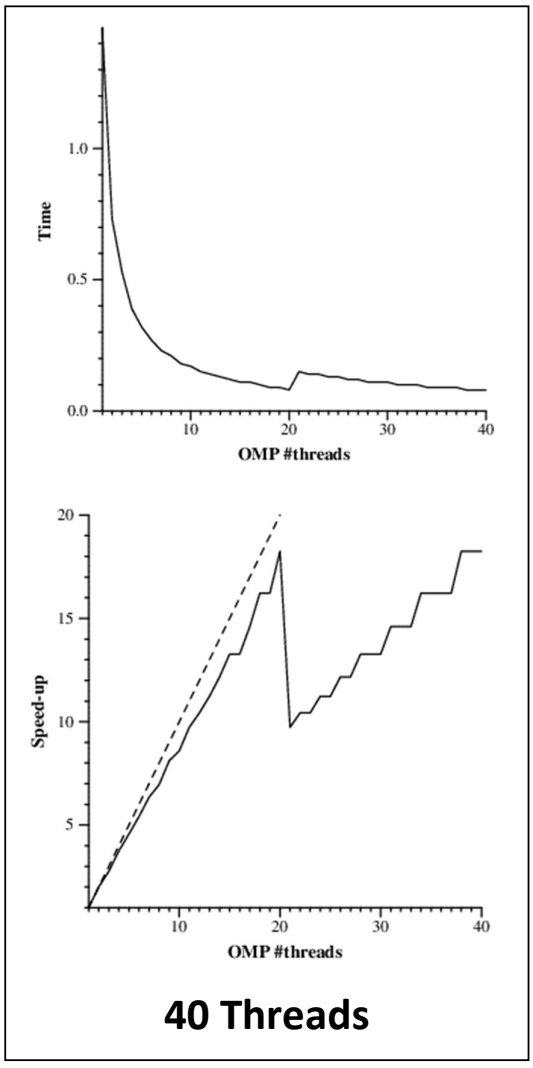


Timing sequential and parallel executions

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). 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.**





Explanation:

- **Using 20 threads:** As we can see in the first plot, as we increase the number of threads to make the computations, we are decreasing the execution time.
  This is a clear example of **strong scaling**, where we are increasing the number of threads with constant problem size. Thus, we are reducing the execution time because the work is distributed to more threads and, consequently, they have to perform less computations.
  For this reason, scalability is good when using 20 threads.

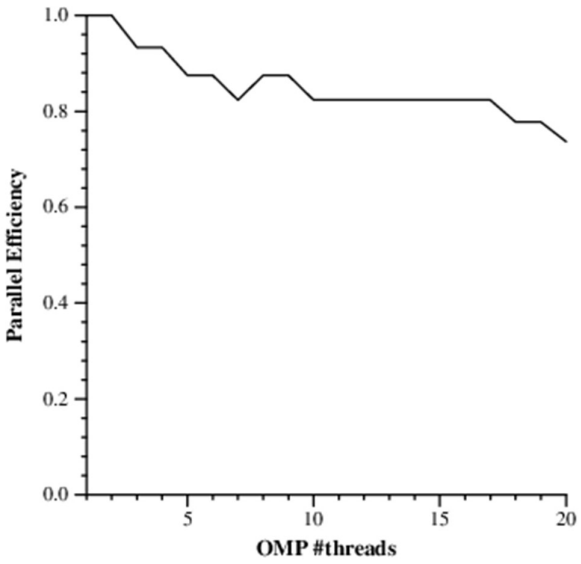
  As we have seen in theory classes, the amount of work to be executed per each processor (the granularity of the task decomposition) is finer and finer when the number of processors is increased.
  Note that, in the plot, we are not observing a straight diagonal but a curve that has an horizontal asymptote. So, the relationship between the number of threads used and the execution time is not linear.

  Regarding the second plot, we can see a diagonal dashed line that represents the ideal speed-up as we increase the number of threads (linear relationship).
  In a continuous line, we see the real speed-up, which does increase linearly at the beginning (more or less) but at the end it does stop increasing, which indicates that using 17 or more threads does not improve the performance (in fact, it decreases the speed up).

- **Using 40 threads:** The same idea as when using 20 threads. But in this case, we can see that when using more than 20 threads, there is an increase of execution time. Moreover, we can also see that the speed-up is significantly decreased when using more than 20 threads.

  This is due to Hyper-Threading → We are using 2 threads to run on each core. The reason is that the system can take instructions quickly from 40 threads, but there are only 20 cores available.

  For this reason, scalability is NOT good when using more than 20 threads.



Explanation:

In this plot we can see the parallel efficiency when using different number of threads. As we have seen in theoretical lectures, there are 3 possible behaviours for the efficiency:

- Linear speedup (100% efficiency), which is the ideal case, and it is difficult to achieve → When the speedup = P
- The usual case is to obtain sublinear speedup → Efficiency < 1
- Super linear (efficiency > 1) speedup is possible due to registers and caches.

In this case, efficiency decreases as we increase the number of threads (sublinear speedup). This could be due to many reasons, for example:

- As the number of processors increases, the amount of communication required between them also increase. Thus, the efficiency decreases a little bit due to this extra communication operations.

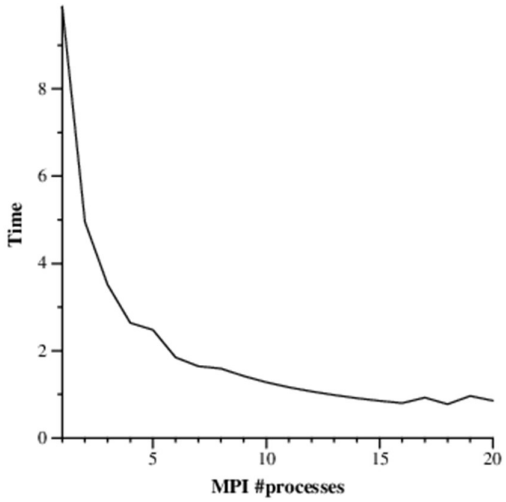
Note that the value of the efficiency does not drop below 0.75, which is quite good.

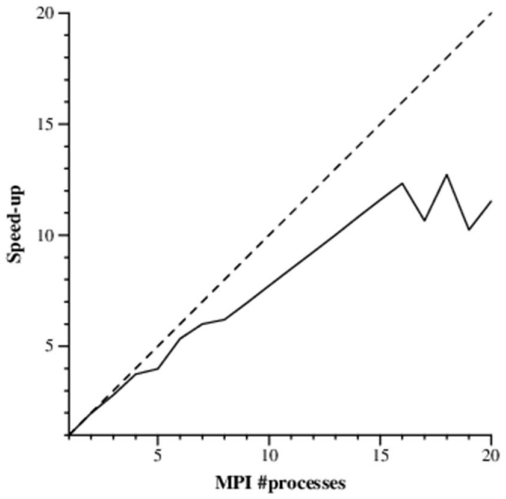
This is an example of weak scaling, since we increase the number of threads with the problem size. So, the purpose is to use parallelism to execute the application with a larger problem size in the same execution time. This implies that the granularity of the task decomposition is kept constant, which means that all threads execute the same amount of work.

Overall, this is a good weak scaling.

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.

Nº Threads	Execution time
1	4.940482
2	2.538572
3	1.759074
4	1.320217
5	1.093109
6	0.909455
7	0.779719
8	0.682544
9	0.607399
10	0.547322
11	0.575272
12	0.528167
13	0.488507
14	0.453894
15	0.425992
16	0.399542
17	0.376713
18	0.356022
19	0.337875
20	0.321546





The original sequential time is 9.883156 time units.

The time with 20 MPI processes is 0.856009 time units.

The time with 2 MPI processes, each using 20 Open MP threads is 0.321546 time units.

Explanation:

- The first plot shows that as we increase the number of MPI processes to make the computations, we are decreasing the execution time.
  As mentioned before, this is a clear example of **strong scaling**, where we are increasing the number of processes with constant problem size. Thus, we are reducing the execution time because the work is distributed to more processes and, consequently, they have to perform less computations.

- The second plot we can see a diagonal dashed line that represents the ideal speed-up as we increase the number of MPI processes (linear relationship).
  In a continuous line, we see the real speed-up, which does not increase linearly (it is below the dashed line). Meaning that we are losing efficiency as we increase the number of processes.
  Moreover, when adding more than 16 or 18 processes produce a loss of speedup since it does not improve the execution time.

- In the table, we can see that, when using 2 MPI processes and increasing the number of threads, the execution time decreases. This is a clear example of good strong scaling. In fact, using this approach we obtain the best execution time.

Understanding MPI codes

- **Explain the deadlock problem and how it can be fixed**

In the tutorial, we have a code that illustrates a potential "deadlock" if we do not take care of our communications. First of all, we initialize some variables that we will use in the MPI_Send and MPI_Recv operations.

Then we have a conditional (if & else):

- We enter the first conditional if we are in the masternode (rank == 0)
- We enter the second conditional if we are NOT in the masternode (rank != 0)

Then, each process will print 2 messages:

- Once the message has been sent, it will print: The message has been sent.
- Once the message has been received, it will print: The message has been received.

We execute the program, and we obtain the first message but then the program gets stuck and we do not obtain the second message → Deadlock

The problem here is that both processes are making a call to MPI_Send (both of them are trying to call each other). Since MPI_Send is blocking, the calls will only return when the other processes make the reception. But the reception will never happen because the other process is also trying to send to send the message.

Two things can be done here to solve this:

- Go to the If and ELSE IF and say: If I am process 0 I force send and if I am process 1 I force receive.
  So, by using a different order for sending and receiving in process 0 and process 1: Only one of them should send first while the other should receive first.

- Use Non-blocking communication:
  - MPI_Isend (add the identifier parameter at the end). So, we will not wait until the message has been sent (we will NOT block) and we will be able to continue.
    We will need to add an MPI_Wait after the last print.
  - We could also use MPI_Ssend, which will initiate a synchronous send operation that blocks the program until the message has been received by the destination process. This ensures that the message has been completely transmitted before the program continues executing.
    The use of MPI_Ssend can reduce the overall performance of the program, as it may increase communication overhead.

- **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.**

Code excerpt related to MPI_Gather:

We first assign a character to the first element of the array "x":

- In our case, since alphabet = 'a' and Iam = 2 (for example), x[0] = 'c'.
  Since the ASCII value of 'a' is 97 + the integer 2 equals 99, which corresponds to the ASCII value of 'c'.

```
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");
```

Then we enter a for loop that is going to iterate from 0 to the number of processors we are using minus 1. Since we are using P = 4, it will iterate 4 times. The elements 0 to 3 of the array 'y' will be initialized with a white space.

Then we make the operation MPI_Gather, which collects the data from all the processes into the root process, which is process = 1. In this case, each process will send the information from the array "x" to the array 'y' of the root process.

Then, each process will print ''MPI_Gather:" and its process number (Iam).

Then we enter another 2 for loops:

- The first one will print the contents of array 'x', which corresponds to the data sent by each process.
- The second one will print the contents of array 'y', which corresponds to the data received by the root process thanks to the MPI_Gather primitive.

The output that corresponds to MPI_Gather operation:

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

As we can see, the output corresponds with the explanation provided.

Process 0, 2 and 3 have some information (characters a, c and b respectively) that is going to be gathered by the root process (process 1).

Code excerpt related to MPI_Scatter:

We first enter a for loop that is going to initialize the arrays ‘x’ and ‘y’:

- Array x: Each element is assigned a value based in which is the value of the current process (Iam) and the value of the number of the process.
- Array y: Each element will be initialized with a white space.

```
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");
```

Then we make the operation MPI_Scatter, which will distribute the data from the root process to all the other processes. It’s like a dealer (root) giving the cards (elements of array x) to all the players (the other processes).

Then, each process will print ‘’MPI_Scatter:” and its process number (Iam).

Then we enter another 2 for loops:

- The first one will print the contents of array ‘x’.
  As we can see, the contents of array ‘x’ of the **root** will be sent to the other processes.
- The second one will print the contents of array ‘y’, which corresponds to the data received by each process thanks to the MPI_Scatter primitive.
  Note that the data received corresponds to the elements of array ‘x’ of the root process.

The output that corresponds to MPI_Scatter operation:

Function	**Proc**	**Sendbuf**	**Recvbuf**
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

Again, the output corresponds with the explanation provided.

Process 0, 2 and 3 receive information (characters e, g, h respectively) from the array ‘x’ of the root process.

Write your own MPI code

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.**

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

int main(int argc, char *argv[]){
    int Rank, size, fact, lower, upper, i;
    double local_result = 1.0;
    double total;

    /* INITIALIZE MPI */
    MPI_Init(&argc, &argv);

   /* OBTAIN THE RANK AND THE NUMBER OF PROCESSORS */
    MPI_Comm_rank(MPI_COMM_WORLD, &Rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /*IF RANK IS 0, OBTAIN INPUT NUMBER AND SEND IT TO ALL PROCESSORS USING MPI_Send */
    if(Rank==0){
        printf("Enter a number:");
        scanf("%d", &fact);
        for(i=1;i<size;i++){ MPI_Send(&fact, 1, MPI_INT, i, 0, MPI_COMM_WORLD); } }

    /*IF RANK IS NOT 0, RECEIVE THE NUMBER USING MPI_Recv */
    else{  MPI_Recv(&fact, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); }

   /*DEFINE THE UPPER BOUNDARY TO WHICH EACH PROCESSOR IS GOING STOP COMPUTING*/
    if(Rank==0){ lower=1; }
    else{ lower = Rank * (fact/size) + 1; }

   /*DEFINE LOWER BOUNDARY TO WHICH EACH PROCESSOR IS GOING TO START COMPUTING*/
    if(Rank==(size-1)){ upper = fact; }
    else{ upper = (Rank + 1) * (fact /size); }

   /*MAKE THE COMPUTATIONS INSIDE THE DEFINED BOUNDARIES*/
    for(i=lower;i<=upper;i++){
        local_result = local_result * (double)i;
    }

   /*COMBINE ALL THE LOCAL RESULTS BY MULTIPLYING THEM TOGETHER*/
    MPI_Reduce(&local_result, &total, 1, MPI_DOUBLE, MPI_PROD, 0, MPI_COMM_WORLD);

    if(Rank==0){
        printf("The factorial of %d is %lf, and was calculated using %d processes\n", fact, total, size);
    }

    MPI_Finalize();
    return 0;
}
```

I have created a code that computes the factorial of a given number using MPI.

First of all, we just initialize the MPI and obtain the rank and number of processors we are using.

Then, the process with rank equal 0 is going to request for the number and then it is going to send it to everyone else using point to point communication.

Note that it would be more efficient to use MPI_Bcast to send this information to all the other processes instead of going one by one. But since we need to use both point to point and collective primitives, I implemented it this way.

Then, the other processes receive the information that has been sent from process with rank equal 0.

Then it calculates the upper and lower boundaries of the factorial computation for each process based on its rank and the given number. Then it computes the factorial inside those boundaries defined and stores the result in a local variable.

Finally, using MPI_Reduce we combine all the local results int a global result by multiplying all the local results together. This global result is going to be printed by the process with rank 0.

Compile: mpicc -o factorial factorial.c

Execute: mpirun -np {NUMBER OF PROCESSES} factorial