Lab1:
Parallel Execution Environment

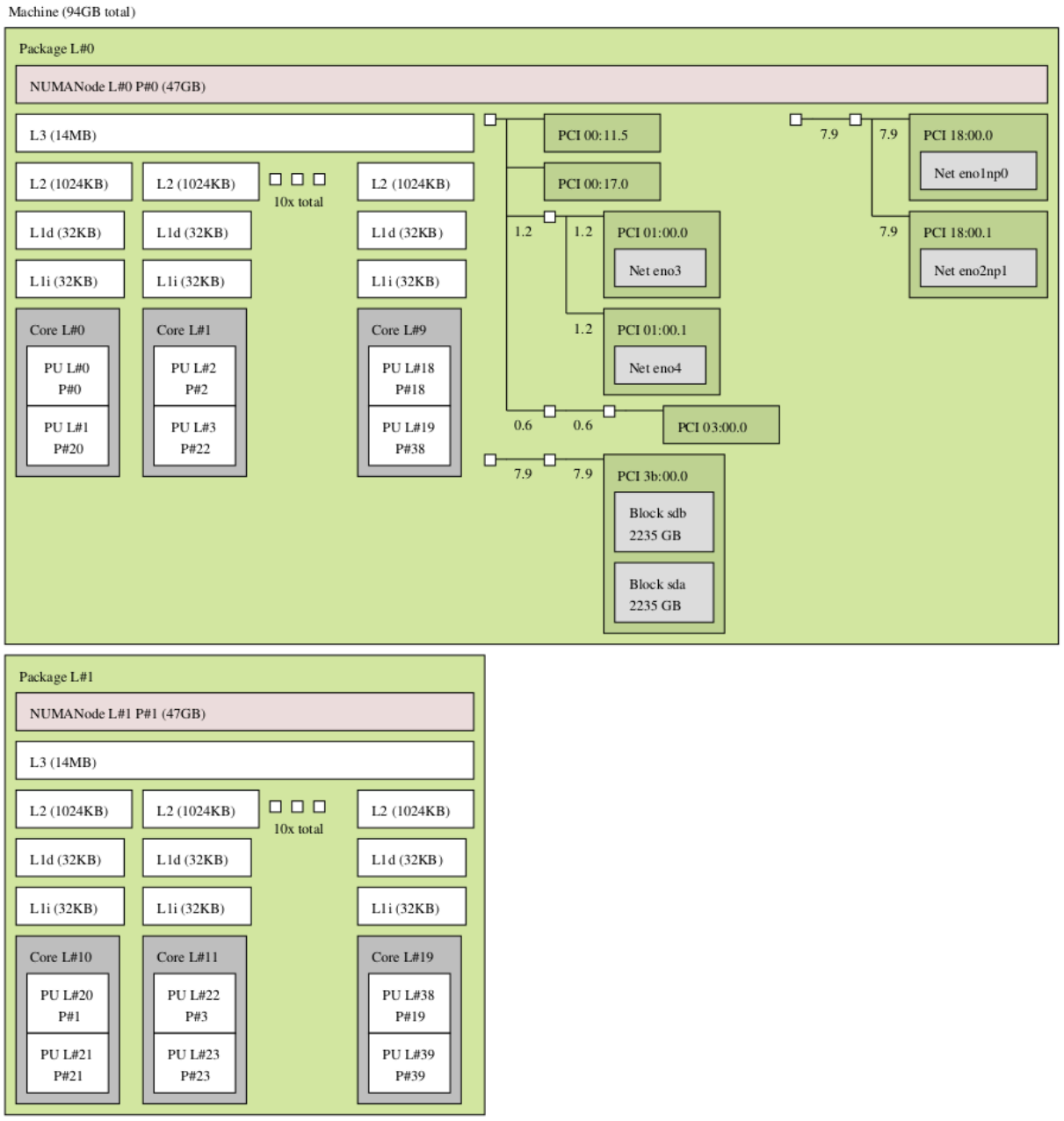
Laia Barcenilla Mañá
Username: hpc1203
NIA: 107694
05-04-2024
2023/2024
Third term
2nd Bioinformatics

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 = 1Mb
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`.

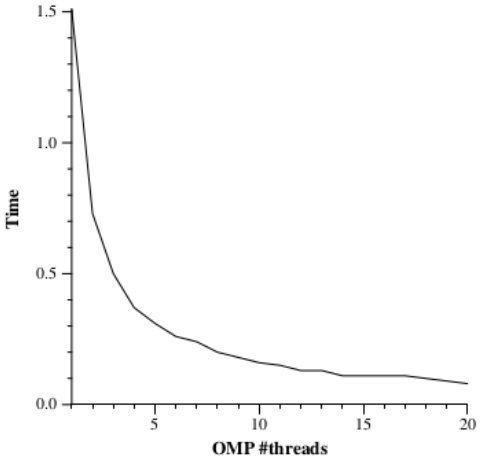
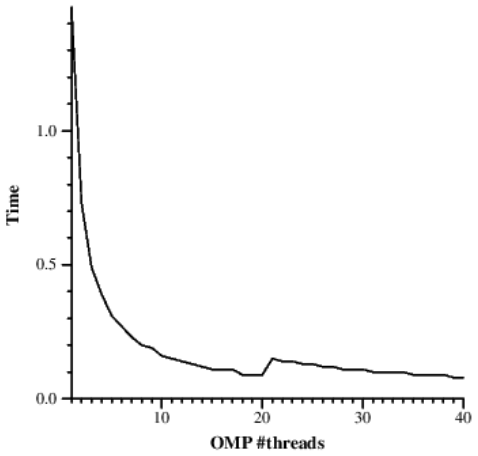
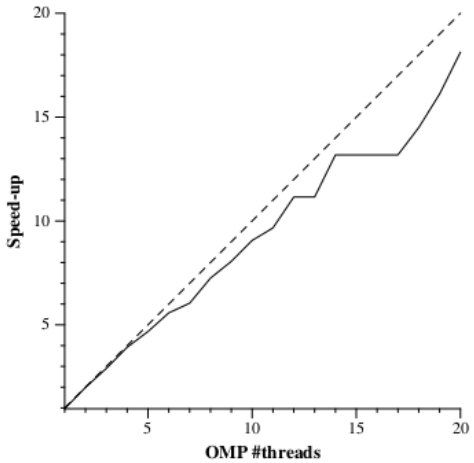
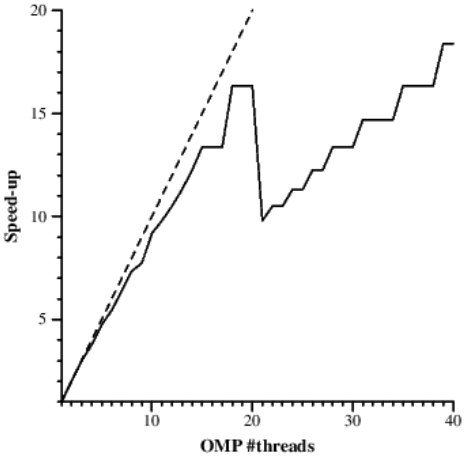


TIMING SEQUENTIAL AND PARALLEL EXECUTIONS

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). If you did the optional part, show the resulting plot and comment 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 SCALING

Threads = 20	Threads = 40
	
	

Threads = 20

On the first graph, we can see the correlation between the number of threads and the execution time. It is shown that as the number of threads increases, the execution time decreases. Moreover, we can also note that with a low number of threads, the program's execution time is high. Although the reduction in execution time occurs as we increase the number of threads, with each additional thread, the time decreases slightly less. Therefore, we can state that this graph is nonlinear (with a horizontal asymptote).

This happens due to the fact that we are using strong scalability, where with the same size of the problem we increase the number of threads. This results in a reduction in time each time we add a thread because the work is divided into more parts, reducing the computational cost of each one.

On the second graph, we can identify a diagonal line (indicated with dashes) that shows a linear relationship between speed-up and the number of threads. Furthermore, there is a continuous line representing the speed-up of our program while increasing the number of threads.

Initially, as we increase the number of threads, the speed increases in an almost linear way. However, between threads 14 and 17, there is no further increase in the program's speed-up. After increasing the threads beyond 17, we can also see an increase in the program's speed.

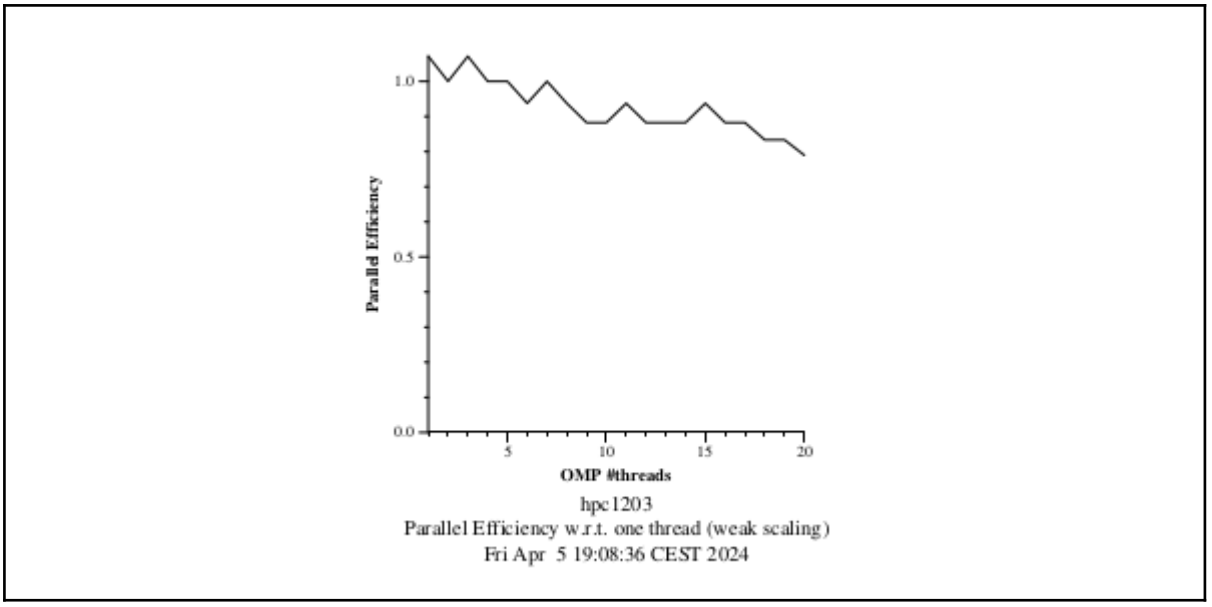
Threads = 40

At the beginning of both graphs, we can observe the same pattern as explained earlier. However, when reaching 20 threads, we notice that the time starts increasing. Subsequently, when more threads are added, the time begins decreasing again, following the same trend as before.

In the second graph, we also see this fact where reaching 20 threads leads to a significant decrease in the program's speed. However, once we add more threads, the speed-up starts increasing again.

This significant decrease observed at 20 threads happens due to Hyper-Threading. This occurs because we are running a program with 40 threads while only having 20 available cores. Consequently, the additional threads can’t be executed simultaneously on individual cores, leading to an increase in execution time.

WEAK SCALING



Weak scalability implies that as both the problem size and the number of threads increase, the program's execution time remains constant, which allows handling larger problems with the same efficiency.

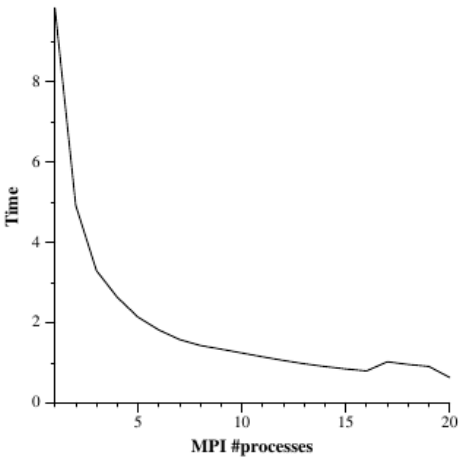
The plot above, which is an example of low scalability, shows the parallel efficiency as the number of threads increases. It shows its highest value at the beginning when the number of threads is low. However, as we continue to increase the number of threads, the efficiency slightly decreases. Nevertheless, it is noticeable that the efficiency value never falls below 0.75, which is considered good.

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.

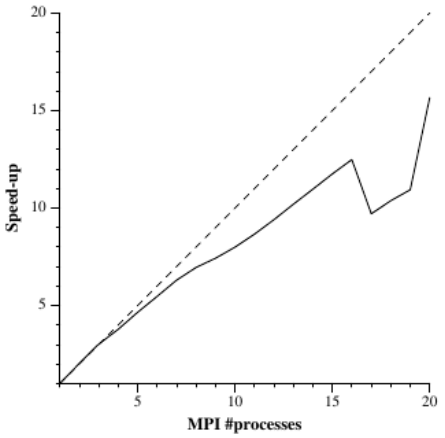
   Which was the original sequential time? the time with 20 MPI processes? and the time with 2 MPI processes each using 20 OpenMP threads?

   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.

N. Threads	Time
1	9.842546
2	4.926666
3	3.296919
4	2.633553
5	2.138363
6	1.819481
7	1.579102
8	1.431249
9	1.339976
10	1.244560
11	1.148940
12	1.058280
13	0.975837
14	0.907194
15	0.846825
16	0.796991
17	1.025329
18	0.959060
19	0.909804
20	0.635456



hpc1203
Min elapsed execution time
Tue Apr 9 12:53:33 CEST 2024



hpc1203
Speed-up wrt sequential time
Tue Apr 9 12:53:33 CEST 2024

The time that takes the original sequential problem is 9.842546 time units, and the time with 20 MPI processes is 0.635456 time units. These results are shown in the table above (comes from the elapsed.txt file)

```
Launching 2 MPI processes. Number of threads per process: 20
Number pi after 1073741824 iterations = 3.141507148742676
0.318775
```

The time that takes the program with 2 MPI processes each using 20 OpenMP threads the time is 0.318775 time units, as the picture above shows (comes from the output of the execution of sbatch submit-mpi2-omp.sh)

On the table, we can observe that as we vary the number of processes, the execution time decreases. Therefore, the program with a single thread (sequential) will take much longer than the one with 20 threads.

The first plot shows that as we increase the number of MPI processes, the execution time decreases, just like it's shown in the table. This demonstrates strong scaling, where with the same problem size, we increase the number of threads, causing the time to reduce as the work is divided into more processes, each having fewer computations to perform.

In the second plot, we can observe the dashed line representing a perfect linear relationship between the speed-up of the program and the number of threads. Moreover, there is a continuous line showing the actual speed-up that our program takes. It's important to mention that this speed-up doesn't always increase consistently. For example, around thread 17, there is a decrease in the program's speed-up because there is no corresponding increase in the execution time.

UNDERSTANDING MPI CODES

- Explain the deadlock problem and how it can be fixed

The deadlock problem occurs when tasks are waiting for events that haven't been initialized, leading to a block since the execution is waiting for something that will never happen.

The problem comes from the MPI_Ssend. MPI_Ssend is a synchronous and blocking send operation, which means that it waits until the message has been received.

To solve this issue, we can use MPI_Isend, which is asynchronous and doesn't wait after sending, unlike MPI_Send. This modification helps prevent blocking and avoids situations where multiple processes are waiting to receive a message, thus preventing deadlock.

- Show the code excerpt related to the gather and scatter collective operations. Also, show the output of their execution, commenting on it.

MPI_Gather()

The MPI_Gather is an operation that collects data from all processors within an MPI communicator and gathers them into a single process, usually the root.

In the code below, it collects data from all processes in MPI_COMM_WORLD and stores them in the y list on the root process. Each process sends a character from the x list to the root.

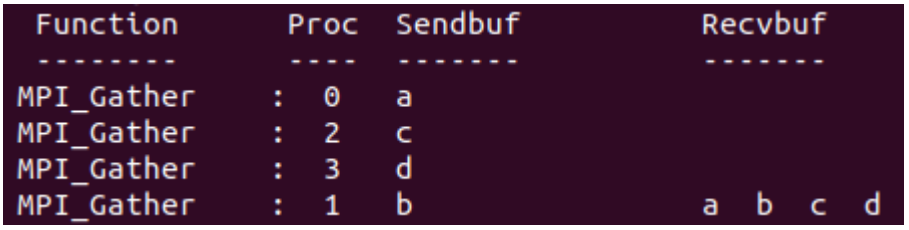
Code:

```
  /*-----------------------------------*/
 /* 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);
```

First, the code initializes the first element of the x array and iterates over the y list, filling it with a blank character in each iteration. Then it executes the MPI_Gather function, which

collects all the information into the root process. In this step, all the elements from the x array will be sent to the y array of the root process. Afterward, a loop is run to print the elements in x (data sent by each process), followed by another loop that iterates over y (data received by the root).

Output:



Comments:

- Function: represents the function in MPI used at this moment (in this case MPI_Gather()).
- Proc: represents the number of the process that is running the operation.
- Sendbuf: shows the content of the array that is being sent from each process.
- Recvbuf: shows the content of the array that is being received by the root process.

In the output shown above, we can see that each process is sending one character to the root. Then, we can also see that the data received are all the characters sent by each process ordered by rank.

MPI_Scatter()

The MPI_Scatter() function is used to distribute data from a root process to all the processes in a communicator, MPI_COMM_WORLD. Each process receives a portion of the data in a balanced way.

In the code below, the elements from the x array are sent to each process in the communicator (MPI_COMM_WORLD). Each process receives an element in the y array.

Code:

```
  /*-----------------------------------*/
 /* 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");

 MPI_Barrier(MPI_COMM_WORLD);
```

Iterate over x and y. On each iteration: assign a value to the x array and a space character in the y list. When the loop ends, call the MPI_Scatter function, which will distribute the data from the root to all the other processes. Each process prints the rank of its process and an MPI_Scatter value. Then we start a loop where all the elements in the x array are printed out (data sent from the root). Finally, we have a last loop which prints the contents of the y array (data received by the processes).

Output:

```
 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
```

Comments:
In the output shown above, we can see that the data from the array x is divided into different chunks and sent to each process. For example, process 1 has received the elements 'e', 'f', 'g', and 'h' in the array y; all of them come from the root process and are stored in x. The 'f' is the last character that is shown as part of the program's output.

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.

This code computes the summation up to a given number, which can be changed by modifying the N parameter.

First, MPI is initialized. Then, we obtain the rank (the identifying number of the current process) and the size (the number of processes). Afterward, we initialize the variables N, the number up to which we want to sum, local_sum, which will contain the sum of each process independently of the other processes, and total_sum, which will accumulate the local sums once all processes have finished.

We iterate over the numbers up to N, adding the loop variable i in each iteration. The loop variable i increases by adding the rank and 1. In each iteration, we add i to the local_sum.

The MPI_Send and MPI_Reciv are used to establish communication between processes. MPI will send in each process that has an ID distinct from 0. The process with rank 0 will receive all the messages. After receiving, the root process will add all the local sums to the total sum variable.

Finally, if the process has rank 0 (i.e., it is the process with ID 0), it will print out the result. The execution ends after this.

We can compile the program using `$mpicc -o sum sum.c`
And run it using `$mpirun -np 4 sum`

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

int main(int argc, char** argv) {
    int rank, size;     // define the rank and the size as integers
    MPI_Init(&argc, &argv);     // initialize the MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);       // get the rank
    MPI_Comm_size(MPI_COMM_WORLD, &size);       // get the size

    int N = 10; // number up to which we want to sum.
    int local_sum = 0; // local sum (for each process)
    int total_sum = 0; // global sum (sum for all the processes)

    // each process computes a part of the sum
    for (int i = rank + 1; i <= N; i += size) {
        local_sum += i;
    }

    // each process sends its local_sum to the root process (rank 0)
    if (rank != 0) {
        MPI_Send(&local_sum, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    } else {
        total_sum = local_sum; // for root process, total_sum is initialized with its local_sum
        for (int p = 1; p < size; p++) {
            MPI_Recv(&local_sum, 1, MPI_INT, p, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            total_sum += local_sum; // add the local_sum received from other processes
        }
    }

    // the process with rank = 0 will print out the result
    if (rank == 0) {
        printf("The sum of the first %d numbers is: %d\n", N, total_sum);
    }

    MPI_Finalize();     // finalize the execution
    return 0;
}
```