HPC Laboratory Assignment

Lab 1: Parallel Execution Environment

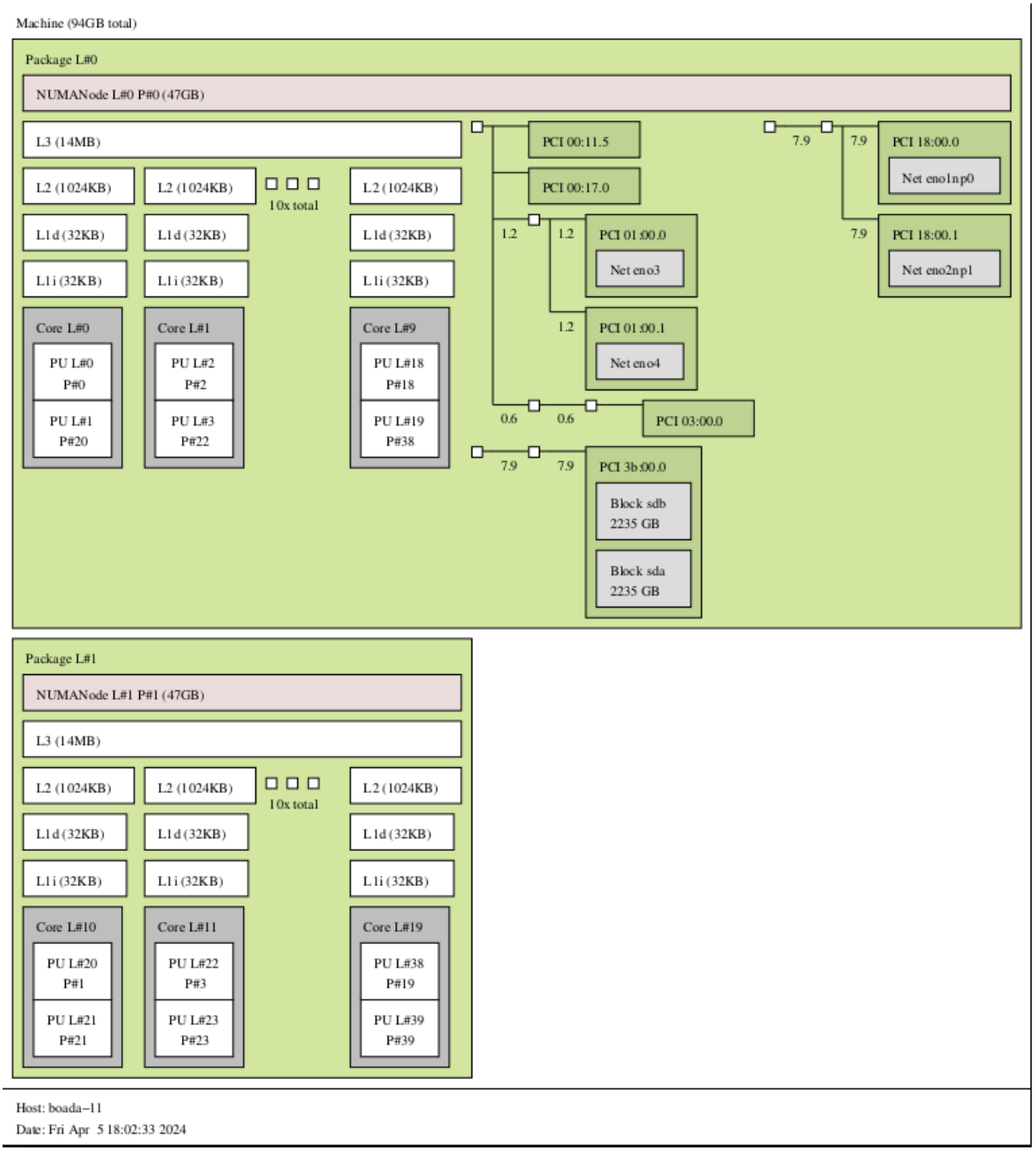
Name: Jan Carreras Boada
NIA: 107765
Username: hpc1205
Date: 5/4/2024
Academic Course: 2n Bioinformatics, 3r Term

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)	640 / 20 = 32 Kb
L1-D cache size (per-core)	640 / 20 = 32 Kb
L2 cache size (per-core)	1024 Kb
Last-level cache size (per-socket)	14 Mb
Main memory (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.

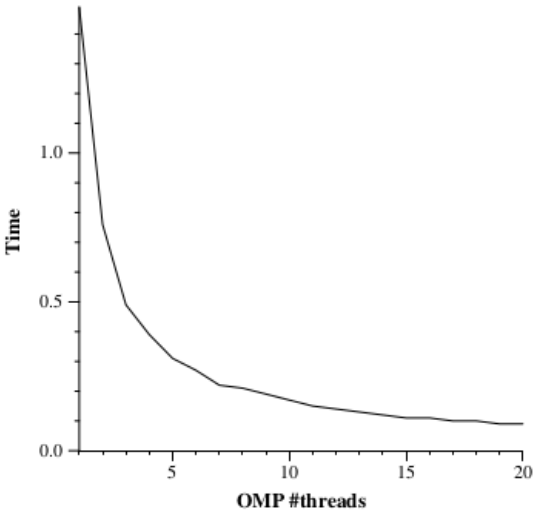


Timing sequential and parallel executions

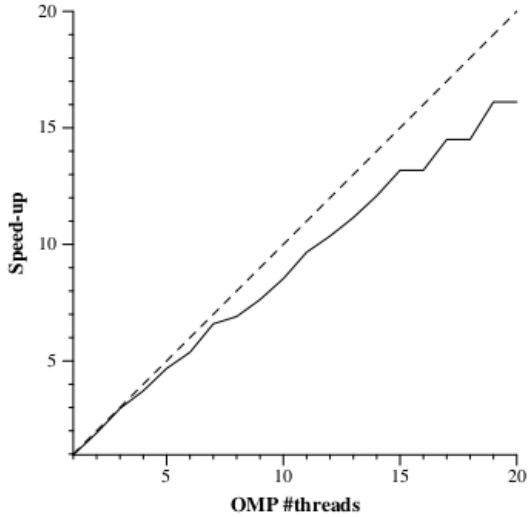
For each of the following sections show the plots or tables required. But, in addition, please provide some reasoning about the results obtained. For instance, comment whether the scalability is good or not, and why.

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). 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.

STRONG 20 THREADS

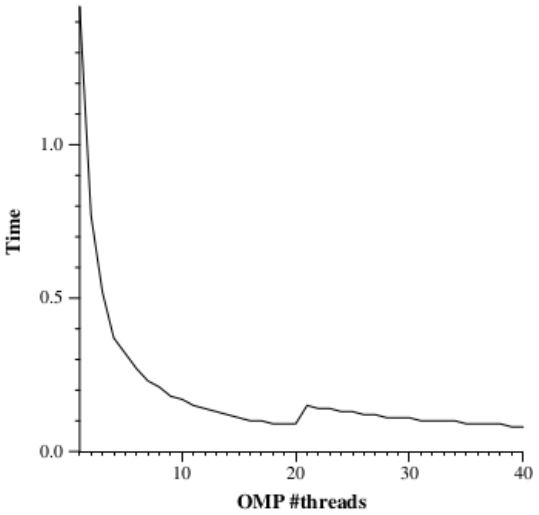


hpc1205
Min elapsed execution time
Tue Apr 9 18:12:40 CEST 2024

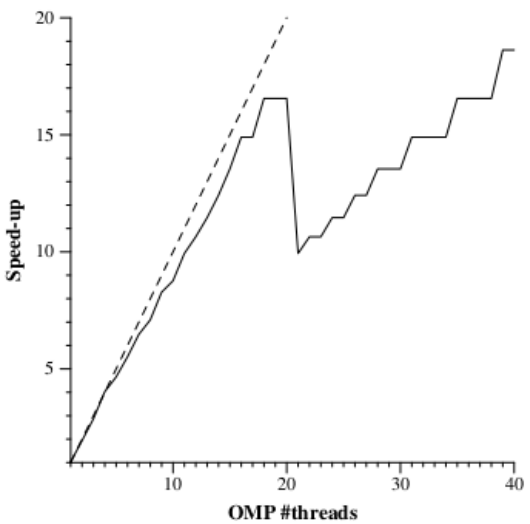


hpc1205
Speed-up wrt sequential time
Tue Apr 9 18:12:40 CEST 2024

STRONG 40 THREADS



hpc1205
Min elapsed execution time
Tue Apr 9 18:17:15 CEST 2024



hpc1205
Speed-up wrt sequential time
Tue Apr 9 18:17:15 CEST 2024

STRONG SCALING 20 THREADS

The first graph shows how the number of threads and execution time relate to one another, emphasizing that a higher thread count results in a faster execution time. It is seen that when there are fewer threads in the program, the execution time increases significantly. The decrease in execution time, however, gets lower as the number of threads increases, suggesting a non-linear relationship.

This phenomenon is attributed to employing strong scalability, which involves keeping the problem size constant while increasing the number of threads. This strategy effectively reduces execution time by distributing the workload more broadly, thereby lowering the computational demand on individual parts.

In the second graphic, a dashed diagonal line indicates a linear relationship between the number of threads and the program speed-up, while a continuous line shows the actual speed-up attained by adding threads one at a time. At first, when more threads are introduced, the program's speed increases in a nearly linear manner. However, there appears to be a ceiling of 14–17 threads, after which more threads are added and program speed increases once more.

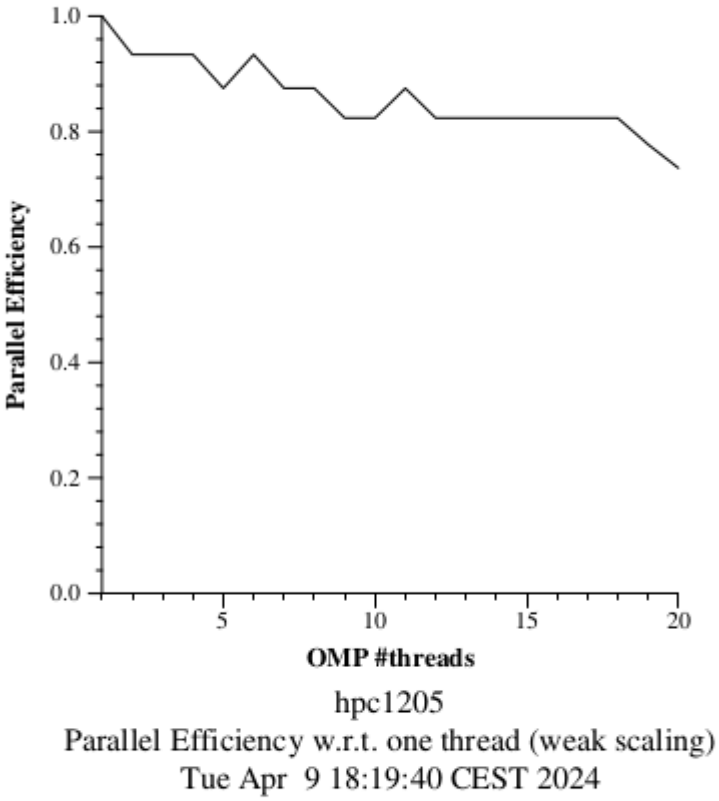
STRONG SCALING 40 THREADS

We note that the trend that was previously stated is still present at the beginning of both graphs. However, there's a noticeable change as the number of threads hits 20, and the execution time starts to increase. After this, the execution time starts to decrease again, and adding more threads causes the original pattern to resume.

This exact pattern is similarly seen in the second graph, where the program's speed noticeably decreases as it reaches 20 threads. Past this point, the speed-up resumes growing faster with the addition of more threads.

The reason for the significant decrease at 20 threads is Hyper-Threading. This is what happens when you try to run a 40-thread program on a system with just 20 cores. The additional threads are consequently unable to operate concurrently on different cores, which causes a temporal increase in execution time.

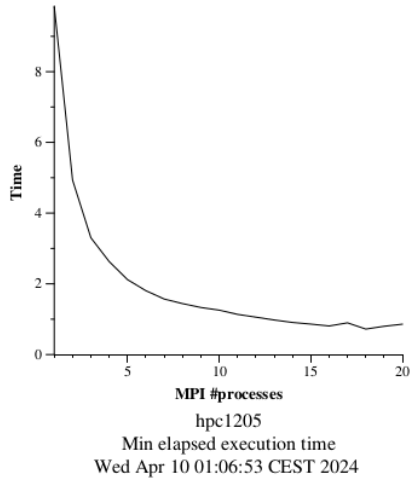
WEAK SCALING 20 THREADS

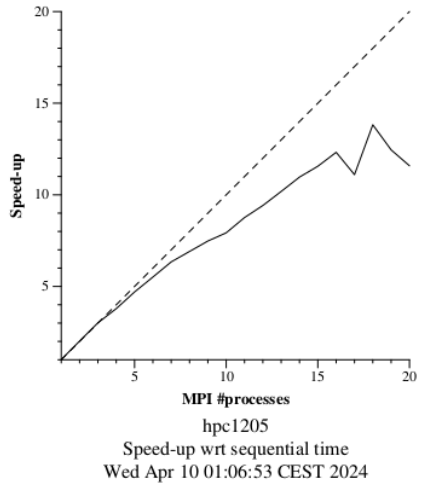


Weak scalability implies that the program's execution time should ideally stay constant as the size of the problem and the number of threads increase, allowing for the efficient handling of larger problems.

The graphic above illustrates the parallel efficiency as the number of threads increases and is an example of low scalability. When there are few threads at the beginning, it displays its maximum value. However, efficiency a bit decreases when we increase the number of threads further. However, it is evident that the efficiency number, which is regarded as good, never drops below 0.75+

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	9.846212
2	4.933874
3	3.301285
4	2.627426
5	2.117771
6	1.804936
7	1.569489
8	1.439911
9	1.330524
10	1.255249
11	1.135925
12	1.056522
13	0.976941
14	0.907176
15	0.860731
16	0.807600
17	0.896385
18	0.719904
19	0.799028
20	0.858843

We can see in our table that the original sequential problem takes 9.846212 time units to solve, while the time it takes with 20 MPI processes is 0.858843 time units. The table above displays these outcomes.
This results comes from the *submit-strong-mpi.sh.o181963* file (also in *elapsed.txt* file)

submit-strong-mpi.sh.o181963

```
#MPI processes Elapsed min
1    9.846212
2    4.933874
3    3.301285
4    2.627426
5    2.117771
6    1.804936
7    1.569489
8    1.439911
9    1.330524
10    1.255249
11    1.135925
12    1.056522
13    0.976941
14    0.907176
15    0.860731
16    0.807600
17    0.896385
18    0.719904
19    0.799028
20    0.858843
```

elapsed.txt file

```
hpc1205@boada-6:~/lab1/MPI/pi$ less elapsed.txt
1    9.846212
2    4.933874
3    3.301285
4    2.627426
5    2.117771
6    1.804936
7    1.569489
8    1.439911
9    1.330524
10    1.255249
11    1.135925
12    1.056522
13    0.976941
14    0.907176
15    0.860731
16    0.807600
17    0.896385
18    0.719904
19    0.799028
20    0.858843
```

The figure below illustrates how long the program takes to run with two MPI processes each using twenty OpenMP threads: 0.318879 time units. (comes from the result of the execution of sbatch submit-mpi2-omp.sh)
We can see from the data that the execution time reduces as the number of processes increases. As a result, the sequential program using just one thread will execute far more slowly than the one using twenty threads.

```
Launching 2 MPI processes. Number of threads per process: 20
Number pi after 1073741824 iterations = 3.141507148742676
0.318879
```

According to the table, the first plot indicates that the execution time lowers as the number of MPI processes increases. Strong scaling is seen here, wherein we

increase the number of threads for a given problem size, which reduces the execution time by dividing the work among multiple processes, each of which needs to do fewer computations.

The second plot shows a perfect linear relationship between the number of threads and program speed, represented by the dotted line. In addition, a continuous line illustrates the real speed boost that our program achieves. It is important to emphasize that this acceleration is not always constant. For instance, at approximately thread 17, the program's speed-up diminishes due to the absence of a corresponding rise in execution time.

Understanding MPI codes

• Explain the deadlock problem and how it can be fixed;

The deadlock problem occurs in computer systems when two or more processes block each other while waiting for the other to release a resource, resulting in an indefinite waiting state because neither process can advance.

In other words, imagine that we have two programs; P1 and P2. But to execute P1 we need the output of P2, and to execute P2 we need the output of P1. We are in a constant state of waiting as both programs are waiting for the result of the other

To solve the deadlock problem we can use MPI_Isend. Unlike MPI_Ssend, which waits for another process that needs to complete its work before continuing, MPI_Isend can continue executing code immediately after starting the send, without having to wait for the message to be received by the recipient process.

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

MPI_Gather()

The MPI_Gather operation consolidates data from all processors in an MPI communicator, accumulating it in a singular process, typically designated as the root.

The code below takes information from each MPI_COMM_WORLD process and saves it in the root process's y list. A character is sent to the root by every process from the x list.

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

The code starts by initializing the first entry in the x array and then loops over the y list, appending a blank character to each iteration. Subsequently, the MPI_Gather function is executed, collecting all the data and transferring it to the root process. All of the elements from the x array will be transferred to the root process's y array in this stage. The elements of x (the data sent by each process) are then printed using a loop, while y (the data received by the root) is iterated over using a second loop.

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

We can see that the output contains four columns: function, proc, sendbuf and recvbuf.
First of all, the function column indicates the MPI function currently in use which in our case will be MPI_Gather(). Second column, called Proc denotes the process number executing the operation.
Sendbub displays the contents of the array that every process sends, and the last one, Recvbuf that shows the array's contents that the root process is receiving.

MPI_Scatter()

Data from a root process can be distributed to all of the processes in a communicator using the MPI_Scatter() function.

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

Output:

```
 Function      Proc    Sendbuf       Recvbuf
 --------       ----    -------       -------

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
```

The MPI_Scatter operation was carried out in four distinct processes (0 1 2 and 3, as the output shows. As can be seen from the groupings of four characters in the "Sendbuf" column, each process sends a distinct segment of an array. Segments 'e f g h', 'i j k l', and 'm n o p' are sent by processes 1, 2, and 3, respectively, whereas process 0 sends segments 'a b c d'..

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.

Comments are shown in the code

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

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv); // We initialize the MPI environment

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size); // Get the number of processes

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Get the current process

    char broadcast_message[100] = "Helloo :)";
    // Broadcast: The root process sends the message to all processes
    MPI_Bcast(broadcast_message, 100, MPI_CHAR, 0, MPI_COMM_WORLD);

    // Display the message received from the broadcast indicating the process
    printf("Process %d received broadcast message: %s\n", world_rank, broadcast_message);

    // We include the process ID
    char response_message[120];
    sprintf(response_message, "Process %d processed the message.", world_rank);

    // If we are not the root process ( root = world_rank = 0), send the modified message back to the root
    if (world_rank != 0) {
        MPI_Send(response_message, strlen(response_message) + 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    } else {
        // The root process receives all messages
        for (int i = 1; i < world_size; i++) {
            char recv_buf[120];
            MPI_Recv(recv_buf, 120, MPI_CHAR, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            printf("Root process received: %s\n", recv_buf);
        }
    }

    MPI_Finalize();
    return 0;
}
```