CPDS Laboratory Assignment 1: Parallel Programming with MPI A Distributed Data Structure

J.R Herrero

Fall 2024

Contents

Chapter 1

Message Passing with MPI

The goal of this assignment is to have you learn about having the data distributed while still collaborating by exchanging segment boundaries; the need for *ghost points*; and some new `MPI` primitives or parameters. For the session today, you need to extract the files from file `a1.tar.gz` from ∼cpds0/sessions by uncompressing it into your home directory in `boada`. From your home directory unpack the files with the following command line: `"tar -zxvf /scratch/nas/1/cpds0/sessions/a1.tar.gz"`.[1] We provide a Makefile so compilation can be done via the `make` command. All the codes will be compiled with `mpicc`. And all but the last one, run with exactly 4 `MPI` processes as: `mpirun -np 4 MPI_EXECUTABLE_NAME_HERE`.

If you run this codes on your own laptop you can directly use `mpirun` as stated above. However, remember that when working in `boada` we need to submit parallel jobs to the queues for batched execution. For this reason in file `submit-mpi.sh` provided for launching jobs to the queing system, the default is set to use 4 MPI processes. This script requires a parameter indicating the executable file.

```
sbatch submit-mpi.sh MPI_EXECUTABLE_NAME_HERE
```

In addition, it accepts an optional second parameter indicating the number of processes. This will only be required for the last code in section 1.2. The output will be written into a file with a name dynamically created as `Output-$PROGRAM-$MPI_PROCESSES.txt`.

We ask you to look for information about the MPI primitives mentioned in this document, and generate different versions of the codes by completing the codes that we provide.

1.1 A distributed data structure

This assignment implements a simple distributed data structure. This structure is a two dimensional regular mesh of points divided into *segments*, or slabs, i.e. sets of consecutive rows, with each segment allocated to a different process. The full data structure is a two dimensional array (matrix):
`double x[maxn][maxn];`

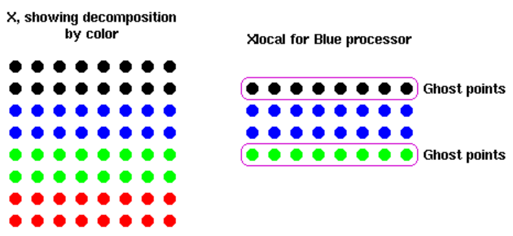


Figure 1.1: Parallel data structure with *ghost points*.

but we want to arrange it so that each process stores only a subset of the data. Figure 1.1 represents with four different colours four segments of a matrix given to four different processors.

[1] You can also work in your own computer if you install an `MPI` distribution such as `OpenMPI` or `MPICH`.

The data structure that each process has locally is therefore a subset of the total:

```
double xlocal[maxn/size +2 ][maxn]; /* +2 rows for ghost points (see below) */
```

where `size` is the size of the group associated with the communicator (i.e., the number of processes). Note that the definition of `xlocal` allocates two additional rows. The reason for this comes from the fact that, for the computation that we're going to perform on this data structure, we'll need the adjacent values in both dimensions. That is, to compute a new value for x[i][j], we will need the point itself together with its four *neighbors*:

```
x[i][j-1] <- Same row, previous column    (left)
x[i][j+1] <- Same row, next column        (right)
x[i-1][j] <- Previous row, same column    (up)
x[i+1][j] <- Next row, same column        (down)
```

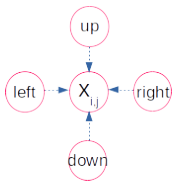


Figure 1.2: Five-point *Stencil*: repetitive pattern of five positions in `x` used for computing each $x_{i,j}$.

Let's consider the last two accesses, i.e. the ones in the dimension in which we have partitioned the data (by rows). These accesses to the previous row (up) and next row (down) could be a problem if they were not stored in `xlocal`. But, due to our partitioning those rows are assigned and therefore stored and computed on the adjacent processes. To handle this difficulty we define *ghost points* that will keep a copy of the values of these adjacent rows which belong to adjacent processors. The *ghost points* will be stored in rows `0` and `maxn/size+1` of the local matrix: `xlocal[0]` will store the row received from the previous processor; `xlocal[maxn/size+1]` will store the row received from the next processor. Therefore, `rows` 1 to `maxn/size` will be the rows used to hold the segment owned by a process, i.e. the local values.

We provide some codes which you will need to understand and complete. The codes divide the array x into equal-sized strips and copy the adjacent edges to the neighboring processes. The processes are given a logical topology in 1D known as *linear array*[2]: a process with rank equal to `k` will regard process `k-1` as its predecessor in this topology; and process `k+1` as its successor. We assume that the domain is not periodic, that is, the first process (rank=0) only sends to and receives data from the next one (rank=1), and the last process (rank=size-1) only sends to and receives data from the previous one (rank=size-2).[3]

In order to test the communication performed in the routines, we have each process initialize all values within its segment with its rank and the ghost points with `-1`. After the communications take place, the code checks that both the segment and the ghost points have the proper values: the segment should continue having the same values (the process rank) while the ghost points should have been overwritten with the rank of the corresponding neighbor.

```
Initial values for xlocal in Process 1      Location              Final values for xlocal in Process 1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1         xlocal[0]              0  0  0  0  0  0  0  0  0  0  0  0
 1  1  1  1  1  1  1  1  1  1  1  1         xlocal[1]              1  1  1  1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1                                1  1  1  1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1         xlocal[maxn/size]      1  1  1  1  1  1  1  1  1  1  1  1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1         xlocal[maxn/size+1]    2  2  2  2  2  2  2  2  2  2  2  2
```

We assume that x has size `maxn` by `maxn`, and that maxn is evenly divided by the number of processes. For simplicity, we also assume fixed matrix dimensions and number of processes: we consider a value of maxn equal to 12 and use 4 processes. For this section, if the codes are correct, their execution will output a simple message: `No errors`.

The codes are almost complete, but have some missing parameters in some calls to MPI primitives. If you try to compile them by executing `make` you will get lots of errors. **We ask you** to complete the codes contained in the files we provide. Each of those statements are identified with a label S_i. In the deliverable, you will have to indicate the identifier of the statement followed by how you have filled it

[2]See slide 10 on Interconnection Networks within the slides on Distributed Memory Systems.

[3]If we decided to exchange messages between processes 0 and size-1 then we would be creating a logical ring, also known as 1D torus. See slide 10 on Interconnection Networks within the slides on Distributed Memory Systems.

in. Note that the three codes in this section do the same but using different communication primitives. Thus, learn from each of them and use your understanding to complete the missing parts in others. Also, the comments included are there to guide you. And there are plenty of comments! Observe how some new `MPI` primitives work and look for additional information about them if you find it necessary.

1. Complete file `par_data_struct.c`. This code uses `MPI_Send` and `MPI_Recv`.

   ```
   > make par_data_struct
   > sbatch submit-mpi.sh par_data_struct
   ```

   Once the execution has been completed please check the output file `Output-par_data_struct-4.txt` which should contain the message `No errors`.

2. Similarly, complete file `par_data_struct_nonblocking.c` which uses non-blocking point-to-point routines instead of the blocking routines: `MPI_Send` and `MPI_Recv` routines have been replaced by `MPI_Isend` and `MPI_Irecv`. Additionally, see how we use `MPI_Waitall` to test for completion of several nonblocking operations.

3. Finally, complete file `par_data_struct_sendreceive.c`, where we use `MPI_SendRecv`: we have replaced each pair of `MPI_Send` and `MPI_Recv` calls in the initial solution with a call to `MPI_Sendrecv`. The first call to `MPI_Sendrecv` sends the last row in the local segment to the next process; and receives data from the previous process and stores it in its initial row (*ghost area*). The second call should send the first row in the local segment to the previous process; and receive a row from the next process and store it in the final row (*ghost area*).

   This code introduces `MPI_PROC_NULL`. It can be used instead of a rank wherever a source or a destination argument is required in a call. A communication with process `MPI_PROC_NULL` has no effect. A send to `MPI_PROC_NULL` succeeds and returns as soon as possible. A receive from `MPI_PROC_NULL` succeeds and returns as soon as possible with no modifications to the receive buffer. When a receive with `source=MPI_PROC_NULL` is executed then the status object returns `source=MPI_PROC_NULL`, `tag=MPI_ANY_TAG` and `count=0`.

1.2 A simple Jacobi iterative method

In this section, you will put together some of the previous examples to implement a simple *Jacobi iterative method* for approximating the solution to a linear system of equations. In this example, we solve the *Laplace equation* in two dimensions with finite differences. Do not worry if you never heard of the Jacobi iterative method and/or the Laplace equation. Any numerical analysis text, as well as plenty of materials on Internet[4], will show that iterating

```
while (not converged) {
  for (i=1; ... )
   for (j ... )
    xnew[i][j] = (x[i+1][j] + x[i-1][j] + x[i][j+1] + x[i][j-1])/4;
  for (i=1; ... )
   for (j ... )
    x[i][j] = xnew[i][j];
  }
```

will compute an approximation for the solution of Laplace's equation. There is one last detail; this replacement of xnew with the average of the values around it is applied only to the interior elements of the matrix; the boundary values are left fixed. In practice, this means that if the mesh is `n` by `n`, then the values

```
x[0][j]      <- First row
x[n-1][j]    <- Last row
x[i][0]      <- First column
x[i][n-1]    <- Last column
```

[4]This problem appears also in one of examples in the MPI Tutorial in lab1. You can find it under `lab1/MPI/Tutorial/laplace`.

are left unchanged. When we have these values read but not written during the computation of our loops we say we have a *halo*. Of course, these comments above refer to the complete mesh stored in `x`; you'll have to figure out what to do with for the decomposed data structures (`xlocal`). Because the values are replaced by averaging around them, these techniques are called *relaxation methods*.

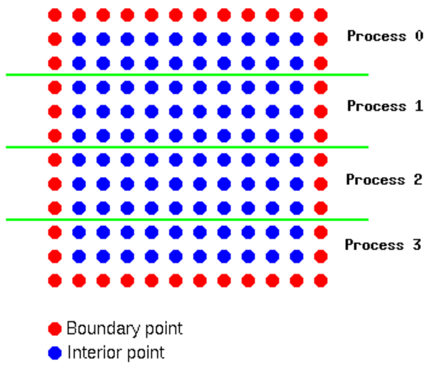
We wish to compute this approximation in parallel. We have written a program to apply this approximation. In order to decide when the calculations have reached the end (*convergence testing*), we compute

```
diffnorm = 0;
for (i ... )
  for (j ... )
    diffnorm += (xnew[i][j] - x[i][j]) * (xnew[i][j] - x[i][j]);
diffnorm = sqrt(diffnorm);
```

We'll need to use `MPI_Allreduce` for this. (Why not use `MPI_Reduce`?) Have process zero write out the value of diffnorm and the iteration count at each iteration. When diffnorm is less that 1.0e-2, consider the iteration converged. Also, if we reach 100 iterations, we want to exit the loop.

For simplicity, the code considers a 12 x 12 mesh processed using 4 processes. The initial data used in the calculation have the boundary values from the previous exercise; they are -1 on the top and bottom rows since they are part of a *halo* (values read but not written during the computation of our loops), and the rank of the process in the inner positions.

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 0  0  0  0  0  0  0  0  0  0  0  0      Process 0
 0  0  0  0  0  0  0  0  0  0  0  0
 1  1  1  1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1      Process 1
 1  1  1  1  1  1  1  1  1  1  1  1
 2  2  2  2  2  2  2  2  2  2  2  2
 2  2  2  2  2  2  2  2  2  2  2  2      Process 2
 2  2  2  2  2  2  2  2  2  2  2  2
 3  3  3  3  3  3  3  3  3  3  3  3
 3  3  3  3  3  3  3  3  3  3  3  3      Process 3
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```



Note that this is a very poor way to solve this numerical problem, and this method is being used only because it is very simple.

Complete the files below by using your knowledge of the MPI primitives. Each of the code has some errors that you will have to fix. Such statements with errors are identified with a label S_i. In the deliverable, you will have to indicate the identifier of the statement followed by how you have filled it in.

1. In file `Jacobi.c` we use the parallel data strucure and the communication scheme seen in the programs studied in section 1.1 within a Jacobi like method. Also, once a parallel program computes a solution, it it often necessary to write the solution out to disk or display it on the user's terminal. This often means collecting the data onto a single process that handles performing the output (since there are few parallel file systems around). Edit the Jacobi example and modify it so that the computed solution is collected onto process 0, which then writes the solution out (to standard output). You may assume that process zero can store the entire solution. Also, assume that each process contributes exactly the same amount of data. For this, use `MPI_Gather`. Complete file `Jacobi.c` using your knowledge of the MPI primitives.

   Note: The expected output for the input problem of dimension 12 run with 4 processes is available in file `Jacobi-out.txt` so that you can easy check the correctness of your program. You just have to send the standard output from the program, printed with the printf statements, to a file and then compare the two files with Unix programs such as `diff` or `cmp`[5]. For instance:

[5]Both programs finish silently if no differences are found.

```
> make Jacobi
  ... which whill output: mpicc Jacobi.c  -O3 -march=native   -o Jacobi  -lm
> sbatch submit-mpi.sh Jacobi
  ... You should wait for the end of the execution of the job (squeue, ls, ...)
> diff Jacobi-out.txt Output-Jacobi-4.txt

(*) On your laptop you may simply run
     mpirun -np 4 Jacobi > Output-Jacobi-4.txt
```

2. Similarly, complete file `Jacobi_nb.c`. This code has to behave similarly to the previous one. However, we want to use *non-blocking* primitives in the communications. This can be useful to overlap communications and computations. Note that in this code some of the non-blocking calls are followed immediately by a call to `MPI_Wait` so they are not really useful (`MPI_Iallreduce` and `MPI_Igather`). They are just provided for you to see that other calls also have the non-blocking version which works similarly to the ones you have seen so far. However, we compute the last iteration of the original outer loop separately to allow for the communication of the corresponding boundary to be completed while the process is computing the values within its segment other that its last row. Fix the errors in the statements that require doing so. Remember to check that the execution was correct by comparing the ouput files: `diff Jacobi-out.txt Output-Jacobi_nb-4.txt`.

3. Finally, complete file `Jacobi_vr.c`. In most cases, it is impossible to divide a parallel data structure so that each process has exactly the same amount of data. It may not even be desirable, if the amount of work to be done varies. Modify the code so that each process can have a different number of rows of the distributed mesh. We want to balance the workload so that all processors get approximately the same amount of work. For this, we will use the following code[6]:

```
int getRowCount(int rowsTotal, int mpiRank, int mpiSize) {
    /* Adjust slack of rows in case rowsTotal is not exactly divisible */
    return (rowsTotal / mpiSize) + (rowsTotal % mpiSize > mpiRank);
}
```

Calling this routine with the proper parameters, each process will get the number of rows that it has to compute.

```
nrows = getRowCount( TotalNumberOfRows, ProcessRank, NumberOfProcesses);
```

For gathering different amounts of data we use `MPI_Gatherv`. We ask you to fix the errors that appear in the code. The expected output for the input problem of dimension 13 run with 5 processes is available in file `Jacobi_vr_p5r13-out.txt` so that you can easy check the correctness of your program:

```
> make Jacobi_vr
mpicc Jacobi_vr.c  -O3 -march=native   -o Jacobi_vr  -lm
> sbatch submit-mpi.sh Jacobi_vr 5
  ... (*)
> diff Jacobi_vr_p5r13-out.txt Output-Jacobi_vr-5.txt

  (*) On your laptop: mpirun -np 5 Jacobi_vr > Output-Jacobi_vr-5.txt
```

Let's inspect the values of some variables to see: 1) how the *load is balanced* across the different processes; 2) a variable amount of data is transfered with `MPI_Gatherv` and stored in the proper positions in array `x`.

[6]See the definition and usage of `getRowCount` in the example of matrix multiplication within the slides on Distributed Memory Systems. Note that "/" is an integer division; and % is evaluated before the comparison; thus, the result of the modulus "%" operation is compared > against the rank. The comparison returns a 0 for FALSE and a 1 for TRUE. Thus, (rowsTotal % mpiSize > mpiRank) will add one row to the initial rowsTotal % mpiSize processors and none to the rest of the processors.

```
Values of some variables for the parameters set at the beginning of the file:
#define maxn 13
#define P 5

   Process 0: nrows=3, i_first=2 i_last=3
   Process 1: nrows=3, i_first=1 i_last=3
   Process 2: nrows=3, i_first=1 i_last=3
   Process 3: nrows=2, i_first=1 i_last=2
   Process 4: nrows=2, i_first=1 i_last=1

   recvcnts[0]=39      displs[0]=0
   recvcnts[1]=39      displs[1]=39
   recvcnts[2]=39      displs[2]=78
   recvcnts[3]=26      displs[3]=117
   recvcnts[4]=26      displs[4]=143
```

Note that: 13/5=2 and 13%5=3. Thus, by default we assign 2 rows per process. And we distribute the 3 remaining rows among the first three processes (with rank 0, 1 and 2), one row to each.

Variable `i_first` is the index of the first row to process in the segment. And `i_last` the index of the last one. However, in this problem, the *halo* makes somehow more difficult to understand the code. The halo is the reason to set `i_first=1+1=2` in Process 0; and to set `i_last=nrows-1` for the last process. But, you don't really need such details for completing this assignment.

Each element of `recvcnts` has to be set to the total number of elements in the segment of the corresponing process, i.e. `nrows*maxn`: e.g. 3*13=39 for P0, P1 and P2; 2*13=26 for P3 and P4.

Vector `displs` contains indices of the initial positions of each segment starting from the beginning of array `x`.

Deliverable

After working on this laboratory assignment, and before starting the next one, you will have to upload **two files**:

1. A report in `PDF` format (other formats will not be accepted) containing the answers to the questions stated at the end of this document;
2. a packed file in either `tar.gz` or `ZIP` format containing all the **source C codes** with the final code. PLEASE, **DO NOT** SEND OBJECT AND EXECUTABLE FILES.

Submissions which do not follow these indications can reach at most 80% of the marks corresponding to this assignment.

Your professor will open the assignment at the moodle aula ESCI website and set the appropriate delivery dates for the delivery. **Only one submission per group** must be done through the moodle aula ESCI website.

Important: In the front cover of the document, please clearly state the name of all components of the group, the identifier of the group (username `cpds1XYZ`), your UPC e-mail addresses, title of the assignment, date, academic course/semester, ... and any other information you consider necessary.
Important: Failure to provide your complete data in the front cover of the pdf file and/or upload the pdf file as a separate file to Moodle Atenea will imply a penalty so that you can only get below 80% of the marks corresponding to the assignment.

As part of the document, you can include any code fragment, figure or plot you need to support your explanations. In case you need to transfer files from boada to your local machine (laptop or desktop in laboratory room), or vice versa, you can use the secure copy `scp` command. In `scp` first we provide the source file and last the destination. For example, consider you want to copy to your local machine (your laptop, or desktop), a file named `foo.txt` kept inside directory `a1` in your home directory of boada. And you want to copy it to your local current working directory which is referred to as ".". Then, you need to execute in a shell in your local machine `"scp cpds1XYZ@boada.ac.upc.edu:a1/foo.txt ."`

Section 1.1

1. Write the solution for 12 statements, identified as `S1` ... `S12` in the source codes, that have some missing parameters in some calls to MPI primitives. Indicate the identifier of the statement and how you have filled it in. For instance:

```
S1: MPI_Comm_rank( MPI_COMM_WORLD, &rank );

S2:

...

S12:
```

Section 1.2

1. Why do we need to use `MPI_Allreduce` instead of `MPI_Reduce` in all the codes in section 1.2?

2. Alternatively, which other `MPI` call would be required if we had used `MPI_Reduce` in all the codes in section 1.2?

3. We have said that we have a *halo* so that some of the values in the matrix are read but not written during the computation. Inspect the code and explain which part of the code is responsible for defining such halo.

4. Explain with your own words how the *load is balanced* across the different processes when the number of rows is not evenly divided by P.

5. Write the solution for the statements identified as `S13` ... `S24` in the source codes, which have errors or some missing parameters in some calls to MPI primitives. Indicate the identifier of the statement and how you have filled it in:

```
S13:

S14:

...

S24:
```