UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Parallel Computing (CPA)

Lab 3: MPI-based Parallelization

Year 2019/20

Contents

Introduction

This practical exercise will take 4 sessions. Each section in the document relates to one session. The following table shows the material needed for the implementation of each one of the sections:

Session 1	Computation of Pi	`mpi_pi.c`, `ping-pong.c`
Session 2	Fractals	`newton.c`
Session 3	Matrix-vector product	`mxv1.c`, `mxv2.c`
Session 4	Systems of linear equations	`sistbf.c`

1 First steps with MPI

The objective of the first session of this practical exercise is to get familiar with the compilation and execution of simple MPI programs.

1.1 Hello world

Let's start with the typical "hello world" program in which each process will print a message on the standard output. The code in Figure 1 shows a minimal MPI program including initialization and finalization using MPI, and a `printf` statement showing a message.

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

int main (int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  printf("Hello world\n");
  MPI_Finalize();
  return 0;
}
```

Figure 1: Program "hello world" in MPI.

Copy the program in a text editor, save it, compile it and run it. To compile the program, the computer must have MPI installed, for which reason we will use `kahan` to compile. You can use any other computer where MPI has been installed. To compile it, you should use command `mpicc` as if you were using a standard compiler such as `gcc` (actually `mpicc` is a tool that invokes the default "C" compiler using the appropriate arguments for the specific installation of MPI of the computer - using "`mpicc -show`" such options are shown).

```
$ mpicc -Wall -o hello hello.c
```

As the execution of the program is very short, we can run it directly on the *front-end* of `kahan`. To execute the program, the `mpiexec` command should be used, with option `-n` to specify the number of processes we want to use. For example:

```
$ mpiexec -n 4 hello
```

All processes will be run on the same node.

This execution mode will be adequate for short executions, but in general we should use the queue system of the `kahan` cluster, so that the program is executed on the working nodes of the cluster, and not on the *front-end*. To do that, as you know, you must create a job file and use `qsub` to submit the job.

```
#!/bin/sh
#PBS -l nodes=2,walltime=00:10:00
#PBS -q cpa
#PBS -d .

cat $PBS_NODEFILE
mpiexec ./hello
```

Figure 2: Job file to execute using the queue system.

Figure 2 shows an example of a job file where 2 nodes are reserved. Previously, our jobs could only use 1 node, as they required the use of shared memory. With MPI, however, we can use several nodes for the same application. In the example, we run the program `hello` using the command `mpiexec`, but in this case, we do not need to specify the number of processes, as the system will run as many processes as reserved nodes (one process on each node). The nodes reserved for our execution can be obtained with the command `cat $PBS_NODEFILE`, although this is not necessary.

Given that a node in the cluster has 32 cores, it could be reasonable to launch several processes per node. In that case, we would modify the job file to specify the number of processes per node (`ppn`). An additional option (`-W x="NACCESSPOLICY:SINGLEJOB"`) must be added too. For example, to run the application on 2 nodes using 16 processes in each node we could use:

```
#PBS -l nodes=2:ppn=16,walltime=00:10:00
#PBS -W x="NACCESSPOLICY:SINGLEJOB"
```

Important: As `kahan` has only 6 nodes, we strongly recommend **to use no more than 1 or 2 nodes** for each job so that other users can submit jobs to the cluster at the same time.

Exercise 1: Perform several executions of the program in the queue system using a different number of processes and nodes.

Exercise 2: Change the program so that it shows the identifier of the process and the number of processes. You can use the functions `MPI_Comm_rank` and `MPI_Comm_size` for this purpose.

1.2 Computation of Pi

This section deals with the implementation of an MPI program that performs a computation in parallel of an approximation of π. The value of π can be computed using many different approaches, and one of them is solving the definite integral

$$\int_0^1 \frac{1}{1+x^2} dx = \frac{\pi}{4} .$$

The program `mpi_pi.c` computes an approximation to this integral using the method of rectangles we used in the first lab session. The interval $[0, 1]$ is split into n sub-intervals (rectangles), and each one of the p processes performs the computations associated to n/p rectangles. More precisely, the following `for` loop:

```
for (i = myid + 1; i <= n; i += numprocs) {
```

is the result of parallelizing the loop that processes the n rectangles:

```
for (i = 1; i <= n; i++) {
```

by distributing the iterations cyclically among the processes. Although OpenMP provides a way to automatically distribute the iterations of a loop among the threads (`#pragma omp for`), there is no such construction in MPI, so we must distribute the iterations explicitly.

In the previous loop, each process computes in parallel a partial sum, and we just have to accumulate the partial sums to obtain the final result. This is done by means of the collective operation `MPI_Reduce`:

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

This function sums the values of `mypi` of all the processes, and stores the result on the variable `pi` of process 0. Note that all the processes must perform the call to the `MPI_Reduce` function, because each of them has to provide its value of `mypi`. However, the final result is only available in process 0.

Exercise 3: Replace the use of `MPI_Reduce` by an equivalent code that only uses point-to-point communications (`MPI_Send` and `MPI_Recv`). For this purpose, every process should send its partial result (`mypi`) to process 0, which will receive each partial result and sum it up on the variable `pi`.

1.3 The program *ping-pong*

MPI programs use the Infiniband network of the cluster. This network has much better performance than a conventional Ethernet network, both concerning bandwidth and latency. Although the manufacturer publishes the specifications of the network, it is advisable to perform an experimental study to obtain latency and bandwidth by using a real program.

This can be achieved by means of a *ping-pong* program, which consists of two processes P_0 and P_1, where P_0 sends a message to P_1 who immediately returns the same message after its reception. P_0 measures the time spent since the sending of the first message to the reception of the second.

Exercise 4: Complete the provided program `ping-pong.c`, so that it works as explained above, taking into account the following considerations:

- The program receives as a command line argument the size of the message to be sent, n (in bytes).
- Time should be measured using `MPI_Wtime`, that returns the time in seconds. It works exactly in the same way as the OpenMP `omp_get_wtime()` function.
- To get significant measurements, the program should repeat the operation a certain number of times (`NREPS`) and show the average time.
- Use the standard operations for sending and receiving messages: `MPI_Send` and `MPI_Recv`, using `MPI_BYTE` as the data type to send/receive.

2 Newton fractals

A fractal is a geometric object whose basic structure repeats itself at different scales. In certain cases, representing a fractal implies a considerable computational cost. In this lab session, we are going to work with a program that generates Newton fractals.

We provide the source code of a parallel program that uses the master-slave (or manager-worker) scheme and the MPI communications library to compute different Newton fractals: `newton.c`. The objective of this session is to modify the communications of this program, changing them to use non-blocking communications, so that the master process can also compute part of the image.

2.1 Sequential algorithm

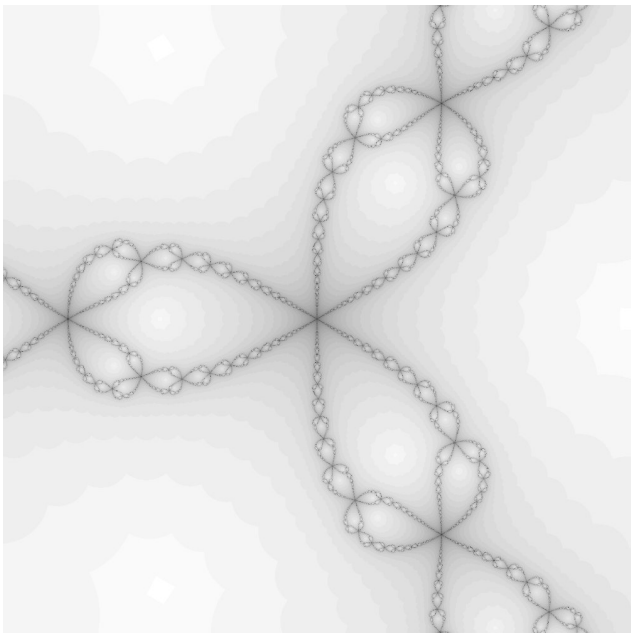
In the Newton fractals, the computation proceeds by searching for the roots of complex functions of a complex variable. Such roots are computed using the Newton root finding method, which performs more or less iterations depending on the initial value. Given a complex function, a zone in the complex plane is selected in which the fractal must be drawn, and a root of the function is computed starting from each of the points in this zone. The number of iterations necessary to reach a solution from each point determines the color in which that point will be painted in the fractal.

For instance, in Figure 3 (left) we can see a Newton fractal for function $f(z) = z^3 - 1$ in the zone with x and y between -1 and 1. An algorithm in pseudo-code for drawing a Newton fractal is shown in Figure 3 (right).

2.2 Classical manager-worker algorithm

The provided program (`newton.c`) is a parallel implementation following the manager-worker scheme to generate Newton fractals.

This program allows the computation of Newton fractals of different functions. It has multiple options that affect the generated fractal (can be seen directly in the source code). We highlight the option `-c` that takes the number of function to be used to generate the fractal. Currently, there are 4 functions, identified from 1 to 4, although the parameter also admits the case 5. Case 5 is actually a zoom of a white zone of case 4. It is not "very pretty", but it has a higher cost and it is, therefore, more useful to see the gain in



```
For y = y1, ..., y2
  For x = x1, ..., x2
    col = num_iterations_Newton(function,x,y)
    Draw point (x,y) with color col
  End_for
End_for
```

Figure 3: Newton fractal for the function $f(z) = z^3 - 1$ (left) and the algorithm used for its generation (right). Note: The fractal image is shown inverted (we see its negative).

parallel. We recommend using cases 1-4 to check (by looking at the generated image) that the program works correctly in parallel, and the case 5 to carry out performance measurements.

The program stores each of the computed fractals in a file named `newton.pgm` (unless this is changed with option `-o`). By default, it generates a gray scale image where the white color corresponds to the highest number of iterations required at any point of the image.

Exercise 1: Compile and execute the program using case 1. Make a copy of the image obtained (`newton.pgm`) using the name `ref.pgm`. This file will be used as a reference to check that the changes made to the program do not affect the results.

In the program `newton.c`, the function `fractal_newton` is the one that performs the algorithm for Newton fractals mentioned previously (Figure 3).

Given a matrix A of $w \times h$ pixels, this function draws a Newton fractal for the rectangle with corners (x_1, y_1) and (x_2, y_2). The roots of the function are computed with a tolerance given by `tol` and a maximum number of iterations given by `maxiter`. Furthermore, it computes and returns the maximum number of iterations that appears within the image, which will be used for the white color.

The provided code already implements a parallel version of the algorithm, following a classical manager-workers scheme. The corresponding pseudo-code is shown in Figure 4. Remember that in the classical manager-workers scheme one of the processes acts as the master (or manager) and is in charge of assigning work to the rest, the workers, who will send the results of the work done back to the master.

It is convenient that you study and understand the basic manager-workers scheme used in the provided code.

Exercise 2: Check the used algorithm (Figure 4) and its implementation in C available in function `fractal_newton` of the provided program. Answer the following questions:

- When a worker process sends a row to the master process, how does it indicate which row it is?
- How does a worker process know that there are no more rows to be processed?

```
If me=0 then (I am the master)

  next_row <- 0
  For proc = 1 ... np
    send request to compute row number next_row to process proc
    next_row <- next_row + 1
  End_for

  rows_done <- 0
  While rows_done < total_rows
    receive a computed row from any process
    proc <- process that sends the message
    num_row <- number of row
    send request to compute row number next_row to process proc
    next_row <- next_row + 1
    copy computed row to its place, that is row num_row in the image
    rows_done <- rows_done + 1
  End_while

Else (I am a worker)

  receive number of row to compute in num_row
  While num_row < total_rows
    process row number num_row
    send computed row to the master
    receive row number to compute in num_row
  End_while

End_if
```

Figure 4: Pseudo-code of manager-workers scheme.

2.3 Manager-worker algorithm with a working manager

In the classical manager-workers algorithm, the master just sends work to be done by the workers and collects the results, but the master itself does not do any work. If the algorithm is run with a processor reserved for the master, then we are underutilizing the computational power of that processor.

A way to avoid this is to make the master process also do useful work. For this, its communications must be non-blocking. In this way, instead of being blocked waiting for the response of any of the workers, it will be able to do work while it waits for that response.

In Figure 5 there is an algorithm in pseudo-code for making the master behave in this way. The code for the workers does not change.

Exercise 3: Read and understand the algorithm shown in Figure 5 and implement it on a copy of the original program, so that you can later compare both programs.

You should take into account the next points:

- When the master process receives a row, it will store it into array `B`. While the master process is waiting for the row, it should be able to process other rows. However it should not modify array `B`. Instead of that, it could modify array `A` directly.
- Array `A` is used by the master process to store the entire image. It is a unidimensional array, but it can be accessed as a matrix because the following macro is defined:

```
#define A(i, j) A[(i)*w + (j)]
```

```
next_row <- 0
For proc = 1 ... np
  send request to compute row number next_row to process proc
  next_row <- next_row + 1
End_for

rows_done <- 0
While rows_done < total_rows
  start non-blocking receive of a computed row by any process
  While nothing received and next_row < total_rows
    process row number next_row
    next_row <- next_row + 1
    rows_done <- rows_done + 1
  End_while
  If nothing received yet then
    wait (in a blocking way) to receive a message
  End_if
  proc <- process that sends the message
  num_row <- number of row
  send request to compute row number next_row to process proc
  next_row <- next_row + 1
  copy computed row to its place, that is row num_row in the image
  rows <- rows_done + 1
End_while
```

Figure 5: Pseudo-code of the master that also performs work (the code for the workers does not change).

so you can use `A(i,j)` to refer to the element in the `i`-th row and the `j`-th column.

Check that the new version of the program is correct. To do so, run it with the case 1 and check that the image obtained is the same as the one stored in the reference file (`ref.pgm`). Use the command `cmp` to compare both files.

Exercise 4: Use a fractal with a higher computational cost (case 5, for example), to measure the performance, comparing the results from both programs. Run both programs using, for example, 4 and 8 processes and check which version performs best.

3 Matrix-vector product

In this session, we are going to work with collective communication operations in MPI. For this, we will employ a computational kernel that is used very often: the product of a matrix by a vector.

3.1 Description of the problem

Many problems in numerical computing can be solved using what is known as iterative methods. In these methods, we start from an initial approximation of the solution and, by some operation that is repeated during multiple iterations, we obtain successive approximations to the solution, which under certain conditions tend to get closer and closer to the sought-after solution.

We are going to work with an iterative method based on the expression:

$$x^{k+1} = Mx^k + v,$$

where each iteration (k indicates the current iteration) computes a new approximation of the solution (x^{k+1} vector) based on a previous approximation of the solution (vector x^k). M and v are a matrix and a vector

```
Initialize M,x,v
For iter=1,2,...num_iter
   x = M*x+v
End_for
norm = sum of the absolute values of x
show norm
```

Figure 6: Basic iterative algorithm for `mxv1.c` and `mxv2.c`.

that are known. In this case, we compute each new approximation using a matrix-vector product and a vector sum, from the previous approximation.

This iterative expression is often used for solving linear systems of equations. The usual case is to perform as many iterations as necessary until it can be guaranteed that the obtained approximation is sufficiently close to the solution. However, for simplicity, in our case, we are going to carry out a fixed number of iterations.

We start with two parallel implementations (`mxv1.c` and `mxv2.c`) of the basic algorithm shown in Figure 6. In this code, we work with a random M and v, but they are built in such a way that the system will converge (the process tends to the solution). In the end, the program shows the 1-norm of the resulting vector x. This value can be used as a *hash* value to check that different executions are correct. (Each of the two versions shows a different result, because each version works with different M and v.)

The programs provided differ in the way data is stored and distributed among the different processes.

- In `mxv1.c` M is stored by rows and distributed by blocks of rows among the processes.
- In `mxv2.c` the matrix is stored by columns, and it is distributed by blocks of columns.

These different matrix distribution schemes imply that the communications required to carry out the computation are different in each case.

3.2 Program based on a block row-wise distribution (`mxv1`)

Exercise 1: Compile the program `mxv1.c` and run it. You can make a short run on a very small problem size directly on the *front-end*, as for example:

```
$ mpiexec -n 3 mxv1 -n 5 -i 5
```

The previous command will run the program with 3 processes, addressing a system with a matrix of 5 rows and columns (`-n 5`) and 5 iterations (`-i 5`). For longer executions, you should use the batch queue system.

Figure 7 shows the data distribution in the `mxv1.c` program, assuming an execution with 3 processes. Matrix M has n rows and n columns, vectors x and v have n elements. Each process stores its local part for M in `Mloc`, its local part for v in `vloc` and the whole vector x in `x`. Using these data, each process computes in `xloc` its block of the new vector x.

The dimension of the problem (n) may not be exactly divisible by the number of processes, for which reason the number of rows in M, v and the new x is extended (up to `n2` rows) to ensure that all processes have the same number of rows (`mb`). These additional rows (the dotted area in Figure 7) simplify the distribution of the data, but they are not considered when computing the matrix-vector product.

Exercise 2: Check the code of `mxv1.c` to understand how the matrix-vector product and the sum of vectors are performed.

Using Collective Communication Functions

The provided versions of the parallel algorithm for the iterative method have been developed using point-to-point communication operations only. However, as you should know, MPI offers many collective communication functions that ease the implementation of parallel programs that require such operations.

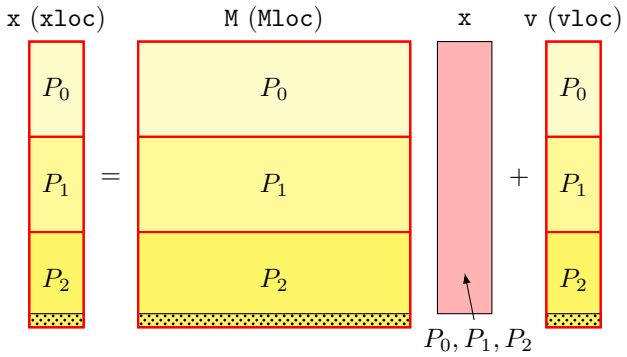


Figure 7: Data distribution in `mxv1.c`.

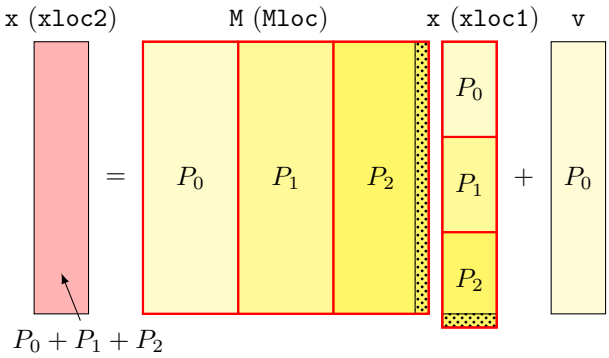


Figure 8: Data distribution in `mxv2.c`

Exercise 3: Modify program `mxv1.c`, substituting the point-to-point communications by collective communications, wherever possible. The code fragments needing modification are marked using a comment with the word `COMMUNICATIONS`.

We advise testing the program progressively after each communication operation is replaced, checking that the result of the program remains correct.

It is important to mention that although we have replaced point-to-point communications by collective communications, the usual approach is to use the collective communication operations from the beginning. The code will be easier to understand, easier to maintain and performance should generally be better, as collective communication operations are usually optimized for each MPI installation.

3.3 Program based on a block column-wise distribution (`mxv2`)

In program `mxv2.c`, matrix M is stored by columns, and it is distributed on a block column-wise fashion, as it is depicted in Figure 8 in an example using 3 processes. Each process stores its local part for M in `Mloc` and its local part for x in `xloc`. Using these data, each process computes in `xloc2` its local contribution to the new vector x. Such vector is then obtained by summing the vectors `xloc2` from all the processes, plus the vector v (stored in P_0).

The number of columns of M and the number of rows of vector x are extended (up to `n2` columns/rows) to ensure that every process has the same amount of rows from M (`nb`). These extra columns/rows (dotted area in Figure 8) ease the distribution of data, but again, they are not considered when computing the actual matrix-vector product.

Exercise 4: Modify program `mxv2.c` substituting the point-to-point communications by collective operations, wherever possible. The places needing modification have been marked using a comment with the word `COMMUNICATIONS`.

Once you have modified the parallel algorithms `mxv1.c` and `mxv2.c`, it is interesting to compare their execution time with respect to the original versions. However, you may not find a big difference between the modified and original versions for this particular problem.

4 Systems of linear equations

This last session focuses on the implementation using MPI of a more complex parallel algorithm. The problem selected is solving a system of linear equations. The primary objective of this session is to manage distributed matrices, both by blocks and cyclically.

The starting point is the file `sistbf.c`, which implements a parallel version where the matrix is distributed using a block row-wise distribution. The objective of this exercise is to modify it in order to use a **cyclic row-wise distribution**.

We advise you to write the row-cyclic implementation of the algorithm in a copy of the file `sistbf.c`, using e.g. the name `sistcf.c`.

4.1 Solving a system of linear equations

A system of linear equations can be described in matrix form as $Ax = b$, where A is the coefficient matrix and, b is the "right-hand side vector" (the vector with the independent terms), and x is the solution vector. We will assume that A is square, being n its dimension:

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix}, \quad x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}, \quad b = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix}. \tag{1}$$

Therefore, the first equation will be:

$$a_{0,0}x_0 + a_{0,1}x_1 + \cdots + a_{0,n-1}x_{n-1} = b_0. \tag{2}$$

The objective is to compute the vector x that satisfies the n equations simultaneously. The system has a single solution (it is compatible and determined) when matrix A has a determinant different from 0. There are several direct and iterative approaches to solve the system of equations. We select for this case a direct method based on the LU factorization (or LU decomposition).

The LU factorization obtains a pair of matrices, one of them unit lower-triangular (all elements above the main diagonal are zero, and the diagonal elements are one) and another one upper triangular (all elements below the main diagonal are zero), both of them of the same dimension as A, such that their product equals A.

Then, solving the linear system of equations is equivalent to solving two triangular systems:

$$\left.\begin{array}{l} A = LU \\ Ax = b \end{array}\right\} \quad \longrightarrow \quad LUx = b \quad \longrightarrow \quad \left\{\begin{array}{l} Ly = b \\ Ux = y \end{array}\right. . \tag{3}$$

The computation of the LU factorization can be performed through Gaussian elimination, using the diagonal elements of the matrix as pivots and making zeros below the diagonal in each column.

The triangular systems can be solved using forward elimination (for the lower-triangular matrix) and backward substitution (for the upper triangular one).

4.2 Initial parallel program

The program provided (file `sistbf.c`) generates and solves a system of linear equations $Ax = b$. To do so, it performs the following steps (outlined in the code with comments with the text "STEP"):

1. **Data generation**. Process 0 generates the entire matrix (`A`) and vector (`b`). Every process (including process 0) allocates memory for their local part of the matrix (`Aloc`).
2. **Data distribution**. The matrix is distributed by blocks of `mb` consecutive rows. Vector `b` is replicated in all the processes.
3. **LU decomposition**. Matrix `A` is overwritten by matrices L and U. The elements of L are placed in the lower triangle of `A` and the elements of U are placed in the upper triangle.
4. **Solving the lower triangular system** $Ly = b$. Vector y is stored in `b`, overwriting it.
5. **Solving the upper triangular system** $Ux = y$. Vector y, initially stored on variable `b`, is overwritten by x in this phase.

Exercise 1: Compile and run the program. Short execution tests can be performed directly on the *front-end* of `kahan`. For example, to solve a linear system of 5 equations using 3 processes:

```
$ mpiexec -n 3 sistbf 5
```

In this case, the system of equations solved is:

$$\begin{bmatrix} 25 & 4 & 3 & 2 & 1 \\ 4 & 25 & 4 & 3 & 2 \\ 3 & 4 & 25 & 4 & 3 \\ 2 & 3 & 4 & 25 & 4 \\ 1 & 2 & 3 & 4 & 25 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 35 \\ 38 \\ 39 \\ 38 \\ 35 \end{bmatrix}.$$

The content of the matrix `A` and the vector `b` in each process at different execution points will be shown on the screen. For example, after performing the distribution, we should have:

```
Matrix A:
---- proc. 0 ----
 25.000   4.000   3.000   2.000   1.000
  4.000  25.000   4.000   3.000   2.000
---- proc. 1 ----
  3.000   4.000  25.000   4.000   3.000
  2.000   3.000   4.000  25.000   4.000
---- proc. 2 ----
  1.000   2.000   3.000   4.000  25.000

Vector b:
---- proc. 0 ----
 35.000  38.000  39.000  38.000  35.000
---- proc. 1 ----
 35.000  38.000  39.000  38.000  35.000
---- proc. 2 ----
 35.000  38.000  39.000  38.000  35.000
```

In the end, vector `b` will store the solution of the system (a vector with all the elements equal to 1). The generated system is prepared so that, regardless of the system dimension, the solution is always a vector with

all the elements equal to 1. Finally, the program shows the error of the obtained solution, which should be zero.

The program should be modified to use a cyclic row-wise distribution, instead of a block row-wise distribution. This requires changing parts of the code in step 2 (data distribution), as well as in the steps 3, 4 and 5 (LU decomposition and triangular system solving). More details are given below.

4.3 Data distribution step

As mentioned above, this step (step 2) implements a distribution by blocks of consecutive rows. This is done using function `MPI_Scatter`.

Exercise 2: Change the data distribution, so that it is performed cyclically by rows. There are different ways to implement such distribution. A simple approach could be to use multiple *scatter* operations, as shown below.

In a row-cyclic distribution using p processes, each of the first p rows of the matrix A go to a different process, which corresponds to a *scatter* operation. The same happens with the next p rows, and so on. Therefore, a row-cyclic distribution of the whole matrix can be done with a loop where each iteration performs a *scatter* operation, paying attention to:

- The position (on the global matrix `A`) of the data to be sent at each *scatter*.
- The position (on the local matrix `Aloc`) where the data from each *scatter* should be received.

Once you have changed this step, run the program and check that the data distribution is properly performed. For example, if we run:

```
$ mpiexec -n 3 sistcf 5
```

Matrix `A` should have been distributed in the following way:

```
Matrix A:
---- proc. 0 ----
 25.000   4.000   3.000   2.000   1.000
  2.000   3.000   4.000  25.000   4.000
---- proc. 1 ----
  4.000  25.000   4.000   3.000   2.000
  1.000   2.000   3.000   4.000  25.000
---- proc. 2 ----
  3.000   4.000  25.000   4.000   3.000
```

The rest of the results that appear on the screen will not be correct, as you have not updated yet the LU decomposition and triangular system solving algorithms.

4.4 LU decomposition and triangular systems solving steps

This section describes the LU decomposition and triangular system solving algorithms, as well as their parallelization approach. We then describe which are the changes needed to adapt the algorithms to the new data distribution (row-cyclic).

LU decomposition (function `lu`)

The sequential algorithm for the LU decomposition is shown in Figure 9. It performs $n-1$ iterations (loop k), each one updating the block of the matrix located within the $k+1,\ldots,n-1$ rows and the $k,\ldots,n-1$ columns (see Figure 10) The modification of this block requires the row k (known as the pivot row).

```
For k = 0, ..., n-2
  if A(k,k) = 0 then exit
  For i = k+1, ..., n-1
    /* Modify row i (elements on columns k to n-1) */
    A(i,k) = A(i,k)/A(k,k)
    For j = k+1, ..., n-1
      A(i,j) = A(i,j) - A(i,k)*A(k,j)
    End_for
  End_for
End_for
```

Figure 9: Sequential algorithm for the LU decomposition.

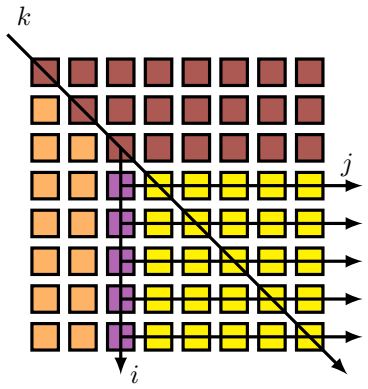


Figure 10: LU decomposition scheme that depicts the path of the three loops of the algorithm.

The parallel algorithm for the LU decomposition focuses on parallelizing loop i in the sequential algorithm, considering that each iteration updates a single row (i-th row), and that each row update is independent (they can be performed in parallel). In the parallel algorithm, each process updates those rows (between $k+1$ and $n-1$) that it owns, so that the entire matrix is updated among all the processes.

However, updating those rows requires using the pivot row (row k), which is stored in only one process. Therefore, the pivot row must be sent by the process that owns it to the rest of the processes (broadcast operation) before starting the update of the rows.

The parallel algorithm is shown in Figure 11, where `owner(i)` denotes the process which owns row `i`, and `iloc(i)` stands for the local index (in `Aloc`) for the `i`-th row of `A`.

Solving the triangular systems (functions `triInf` and `triSup`)

Once the LU decomposition is performed, we have to solve the associated triangular systems using the algorithms shown in Figure 12. The first algorithm solves a system $Lx = b$, where L is a unit lower triangular matrix. The second algorithm solves a system $Ux = b$, where U is an upper triangular matrix. In both cases, vector b is overwritten with the solution of the system.

Both algorithms are very similar. Each iteration of the `i` loop updates, by means of the `j` loop, the elements of vector `b` below element `i` (in the lower triangular system) or above it (in the upper triangular system). This update requires using the element `b(i)`.

The parallelization of both algorithms (Figure 13) is based on the parallelization of loop `j`, so that each process updates the elements in the rows owned by it. For that update, the value of `b(i)` must be previously propagated to the rest of the processes. In the end, all the processes end up with a full copy of the vector of unknowns.

```
For k = 0, ..., n-2
  If owner(k) = me
     if A(iloc(k),k) = 0 then exit
  End_if
  broadcast row k
  For i = k+1, ..., n-1
    /* Update row i (elements on columns k to n-1) */
    Si owner(i) = me
      A(iloc(i),k) = A(iloc(i),k)/A(k,k)
      For j = k+1, ..., n-1
        A(iloc(i),j) = A(iloc(i),j) - A(iloc(i),k)*A(k,j)
      End_for
    End_if
  End_for
End_for
```

Figure 11: Parallel algorithm for the LU decomposition.

Lower Triangular

```
For i = 0, 1, ..., n-1
  For j = i+1, ..., n-1
    b(j) = b(j) - L(j,i)*b(i)
  End_for
End_for
```

Upper Triangular

```
For i = n-1, ..., 0
  b(i) = b(i)/U(i,i)
  For j = i-1, ..., 0
    b(j) = b(j) - U(j,i)*b(i)
  End_for
End_for
```

Figure 12: Sequential algorithms for solving the triangular systems.

Lower Triangular

```
For i = 0, 1, ..., n-1
  broadcast b(i)
  For j = i+1, ..., n-1
    If owner(j) = me
      b(j) = b(j) - L(iloc(j),i)*b(i)
    End_if
  End_for
End_for
```

Upper Triangular

```
For i = n-1, ..., 0
  If owner(i) = me
    b(i) = b(i)/U(iloc(i),i)
  End_if
  broadcast b(i)
  For j = i-1, ..., 0
    If owner(j) = me
      b(j) = b(j) - U(iloc(j),i)*b(i)
    End_if
  End_for
End_for
```

Figure 13: Parallel algorithms for solving triangiular systems of equations.

Changes to be performed

The parallel algorithms previously described are valid for different types of row distributions. You just have to change the definition of the functions `owner` and `iloc` to adapt the algorithm to different data distributions (`iloc` is named `localIndex` in the provided code).

Exercise 3: Modify functions `owner` and `localIndex` to change the block row-wise distribution into a row-cyclic distribution. The behaviour of those functions must be the following:

- Given the `i` row in the global matrix `A`, the function `owner` should return the index of the process which has this row in its local matrix `Aloc`.
- Given the `i` row in the global matrix `A`, the function `localIndex` should return the index of such row in the local matrix `Aloc` in the process that owns such row.

We must also change function `numLocalRows`, which returns the number of local rows in a process (this number could be different from `mb`, as the number of rows may not be divisible by the number of processes). In this case, the changes are already included in the code, so you just need to uncomment the part of the code that corresponds to the row-cyclic distribution and to comment (or remove) the other part.

After changing these functions, check that the algorithms for the LU decomposition and for solving the triangular systems work correctly. For example, if the program is executed with 3 processes for a system of 5 equations, the result of the LU decomposition and of the triangular systems will be:

```
Matrix LU:
---- proc. 0 ----
 25.000   4.000   3.000   2.000   1.000
  0.080   0.110   0.140  24.074   3.352
---- proc. 1 ----
  0.160  24.360   3.520   2.680   1.840
  0.040   0.076   0.108   0.139  24.071
---- proc. 2 ----
  0.120   0.144  24.131   3.373   2.614

Vector b after triInf:
---- proc. 0 ----
 35.000  32.400  30.118  27.426  24.071
---- proc. 1 ----
 35.000  32.400  30.118  27.426  24.071
---- proc. 2 ----
 35.000  32.400  30.118  27.426  24.071

Vector b after triSup (system solution):
---- proc. 0 ----
  1.000   1.000   1.000   1.000   1.000
---- proc. 1 ----
  1.000   1.000   1.000   1.000   1.000
---- proc. 2 ----
  1.000   1.000   1.000   1.000   1.000

Total accumulated error: 0.000000
```

Exercise 4: Once the row-cyclic version is implemented, run both versions comparing their performance. You should use a reasonably large problem dimension (between 1000 and 2000). You must also avoid the display on the screen of matrices and vectors, as this will considerably delay the execution. To do so, just comment the line:

```
#define VERBOSE
```

Compare both versions and analyse which one is more efficient. Try to find the reason for the difference in performance.