UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Parallel Computing (CPA)

Lab 3: MPI-based Parallelization

Year 2016/17

Contents

Introduction

This practical exercise will take 3 sessions. The following table shows the material needed for the implementation of each one of the sections.

Session 1	Computation of Pi	`mpi_pi.c`
Session 2	Fractals	`newton.c`
Session 3	Matrix-vector product	`mxv1.c`, `mxv2.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" 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` showing the identifier of the process and the number of processes.

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

int main (int argc, char *argv[])
{
  int rank, size;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("Hello world from process %d of %d\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

Figure 1: Program "hello world" in MPI.

You should compile and execute the program. 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 simply 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 -o hello hello.c
```

To execute the program, `mpiexec` command should be used. In our case, we should also use the queue system of the `kahan.dsic.upv.es` cluster. To do so, the most convenient way is to write a *script* containing the queue system options followed by the commands that should be executed. Figure 2 shows an example in which 4 nodes from queue `cpa` are allocated for a maximum time of 10 minutes, using the current directory (`.`) for executing the program. Program `hello` is executed using `mpiexec` (in other systems, it will be also necessary to indicate the number of processes as an argument of `mpiexec`, but the cluster is configured to take such value from the option (`-l nodes` of PBS). The line `cat $PBS_NODEFILE` is not actually needed, but it provides information about the nodes in the cluster where our program will be run.

```
#!/bin/sh
#PBS -l nodes=4,walltime=00:10:00
#PBS -q cpa
#PBS -d .

cat $PBS_NODEFILE
mpiexec ./hello
```

Figure 2: *Script* to execute using the queue system.

Next exercise will consist on modifying the program to include, along with the process identifier, the name of the node where the process is running. For retrieving this information, function `MPI_Get_processor_name` should be used. You can check that the queue system allocates each process in each one of the allocated nodes.

If we would like to execute all the processes in the same node, we should modify the script. In particular, we must allocate 1 node, indicating the number of processes per node (`ppn`), and also add a special option:

```
#PBS -l nodes=1:ppn=4,walltime=00:10:00
#PBS -W x="NACCESSPOLICY:SINGLEJOB"
```

1.2 Computation of PI

Next exercise will work on implementing 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 show in the first practical exercise. The interval $[0, 1]$ is split into n sub-intervals (rectangles), and each one of the p processors performs the computations associated to n/p rectangles. This computation can be computed in parallel and only requires summing the results from the partial sums, giving the final result. For this last operation, function `MPI_Reduce` should be used.

Execute the program for different values of n and different number of processes. You can replace the use of `MPI_Reduce` by point-to-point communications (`MPI_Send` in each process and a loop of `MPI_Recv` in process 0), leaving process 0 to compute explicitly the total value.

1.3 *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. Notwithstanding 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. A *ping-pong* program consists of two processes P_0 and P_1 where P_0 sends a message to P_1 which 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.

Implement a *ping-pong* program with the following features:

- The operations for sending and receiving the messages are implemented using the standard API: `MPI_Send` and `MPI_Recv`.
- Time is measured using `MPI_Wtime`, which returns the time in seconds.
- In order to get significant measures, the program should repeat the operation a certain number of times (.e.g 100 times or more) and will show the average time.
- The program will receive as a command line argument the size of the message to be sent, n (in bytes) and the number of repetitions.

Use such program to obtain a model for the communication time ($t_c = t_s + t_w n$). For this purpose, run the previous program for different message size (for example, starting from 0 and stopping at 1,000,000 with increments of 100,000). The latency t_s can be obtained with a message size of 0.

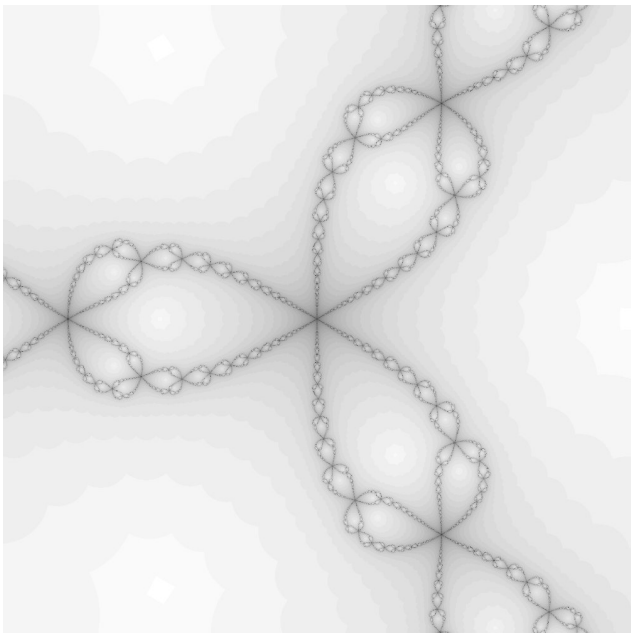
The results show the behaviour of the Infiniband network. If we want to obtain the performance of the Ethernet network we could force MPI to communicate through Ethernet using the option `mpiexec`: `--mca btl ^openib` (Note: this option is specific of Open-MPI and it is not available in other implementations of MPI).

The study should be repeated executing both processes in the same nodes. This will provide a comparison of the communication through Infiniband and directly using shared memory segments.

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



```
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).

session is to modify the communications of this program, changing the 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 the 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 admits also the case 5. Case 5 is in reality a zoom of a which zone in case 4. It is not "very nice," but it has a higher cost and it is therefore more useful to see the gain in parallel. We recommend to use cases 1-4 to check (by looking at the generated image) that the program works correctly in parallel, and then 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 which color corresponds to the highest

```
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

  done_rows <- 0
  While done_rows < 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
    done_rows <- done_rows + 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.

number of iterations required in any point of the image. However, as a curiosity, the program allows to use a palette of different color in the generated fractals. For this, one must use option `-p` followed by a positive number (increment to be used between the components of consecutive colors in the palette) or negative number (used as a seed to generate a random palette). In this case the output file has the extension `.ppm` corresponding to a color image.

Exercise 1: Compile and execute the program observing the 5 basic fractals that can be generated (you will have to run it at least 5 times: with options `-c1`, `-c2`, `-c3`, `-c4` and `-c5`). You can generate some fractal with "psychedelic" colors. Try with option `-p50` or with `-p-11`.

In the program `newton.c` it is the function `fractal_newton` the one in charge of performing the algorithm for Newton fractals mentioned previously (Figure 3).

Given a matrix A of $w \times h$ pixels, this function draws a Newton fractal generated for the area with coordinates x, y in 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 an image, that will be used for the white color.

The provided code already implements a parallel version of the algorithm, following a classical master-slave (or manager-worker) scheme. The corresponding pseudo-code is shown in Figure 4. Remember that in the classical manager-worker scheme one of the processes acts as the master 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.

Exercise 2: Check the used algorithm (Figure 4) and its implementation in C available in function

```
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

done_rows <- 0
While done_rows < 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
    done_rows <- done_rows + 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
  done_rows <- done_rows + 1
end_while
```

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

`fractal_newton` of the provided program.

2.3 Manager-workers 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 useful 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 making the master process also perform useful works. For this, it is necessary that its communications are non-blocking. In this way, instead of being blocked waiting for the response of any of the workers, it will be able to make 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. Use one of the original fractals to check that the new version is correct (the generated picture is the same). Then, use a more costly fractal (case 5, for instance) to obtain performance measurements comparing the results of both programs. To obtain other costly cases you can pass the following options to the program:

```
-c4 -r0.01 -x0.1 -y0.2
-c4 -r0.5 -x0.4 -y0.4
-c4 -r0.5 -x-0.12 -y0.4
```

Make tables and plots for times, speedups, efficiencies... for both programs.

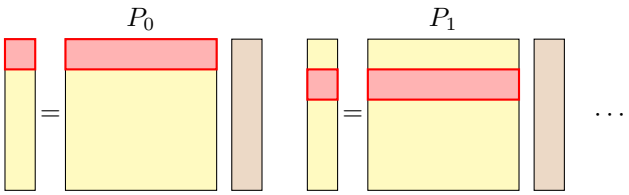


Figure 6: Matrix distribution among the processes in the case of a block-row scheme.

3 Matrix-vector product

In this session we are going to learn how to use 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 by means of 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 to solve *linear systems of equations*, that is, given a square matrix $A \in \mathbb{R}^{n \times n}$ and a vector $b \in \mathbb{R}^n$, we want to compute another vector $x \in \mathbb{R}^n$ that satisfies

$$Ax = b.$$

In the case of the resolution of linear systems of equations, the usual expression used in the iterative process is

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

where x^k represents the approximate solution at iteration k, and M and v are a matrix and a vector obtained from A and b, respectively. In this case, we compute each new approximation by means of a matrix-vector product and a vector sum, from the previous approximation.

The usual case is to perform as many iterations as necessary until a certain stopping criterion is met, which guarantees 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 will start from a code that has already been parallelized, that performs all the process until it obtains a vector x that is the final approximation to the solution. In this code we work with a random M and v, but they are build in such a way that the system will converge (the process tends to the solution). At the end, the program shows the 1-norm of the resulting vector x. This value can be used as a *hash* value in order to check that different executions are correct.

We have at our disposal two versions that differ from each other in the way the store and distribute data among the different processors. (Results among the two versions are not comparable, because each of them works on a different problem.)

- In the first version (`mxv1.c`) we store matrix M by rows and it is distributed by blocks of rows among the processors (see Figure 6).
- In the second version (`mxv2.c`) the matrix is stored by columns and it is distributed by blocks of columns among the processors.

These different matrix distribution schemes make that the communications required to carry out the computation are different in each case. We recommend the student to analyze in detail both versions in order to understand how the matrix-vector product and vector sums are carried out in both cases.

3.2 Task to be developed

The supplied versions of the parallel algorithm for solving a system of equations by an iterative method have been developed using only point-to-point communication operations. However, as the student knows, in MPI there are many collective communication functions that make programming easier when this type of communication is required.

The student must replace all communications that are amenable to be carried out by means of collective operations with the corresponding calls to MPI functions for collective communication. In the code, these operations are marked with a previous comment indicating `COMMUNICATIONS`.

In general, this is what should have been done from the beginning. Normally, it is convenient to use MPI collective operations whenever possible, since they will be optimized to perform the communication in an efficient way.

Once the two versions are available with and without collective operations, it is interesting to compare execution times of each of them. Although it is probable that no significant differences are found for this particular problem.

As always, the student is advised to do the development incrementally, checking the correctness of the program after changing each of the communications.