Concorrência e Paralelismo

MIEI 2018/2019

Practical sessions using MPI

Goals

Using a message-passing library (MPI)

General information

MPI (Message Passing Interface) is a standard used for most parallel applications targeting hardware configurations of distributed-memory memory machines. Please refer to the lecture slides for details.

MPI tutorials (using the C language) are available here:

- Lawrence Livermore National Laboratory (HTML)

  https://computing.llnl.gov/tutorials/mpi/

- Argonne National Laboratory

  https://www.mcs.anl.gov/research/projects/mpi/tutorial/

A PDF file with chapter 2 of the book *Introduction to HPC with MPI for Data Science* de F. Nielsen, Springer, 2016 is available

http://www.springer.com/cda/content/document/cda_downloaddocument/9783319219028-c2.pdf?SGWID=0-0-45-1544691-p177575298

The *hello, world* of MPI is as follow:

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

int main( int argc, char **argv)
{
  int myrank;
  int comm_size;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank);
  MPI_Comm_size( MPI_COMM_WORLD, &comm_size);

  printf("Hello %d from %d\n", myrank, comm_size);

  MPI_Finalize();
}
```

Several MPI implementations exist, namely

MPICH *www.mpi.ch.org*

OpenMPI *https://www.open-mpi.org*

In our tests, we will use openMPI. Programs will be developed and executed in *node9* (*10.170.138.240*) already used in previous laboratory classes. Access is through *ssh*, after logging in in FCT NOVA´s VPN.

Compiling MPI programs

Details about compiling and executing MPI programs are common across all the implementations. `mpicc` is a *wrapper script* that compiles the code, includes the headers, and links with the mandatory libraries. Syntax is as in *gcc*:

```
        mpicc -g -Wall -o mpi_hello mpi_hello.c
```

Executing MPI programs

MPI programs can be executed in:

- A single machine, exploiting its multiple CPUs (in our classes we will do that on *node9*)
- A set of machines, for example the lab PCs. This can be done, but lots of details must be taken care of.

To execute MPI programs, one uses the commands `mpirun` or `mpiexec` as below:

```
mpirun -np number_of_processes program_file argumentos

mpiexec -n number_of_processes program_file  argumentos
```

Please see more details in manual pages of `mpirun` and `mpiexec`

Work to do

1st part Calculating Pi value

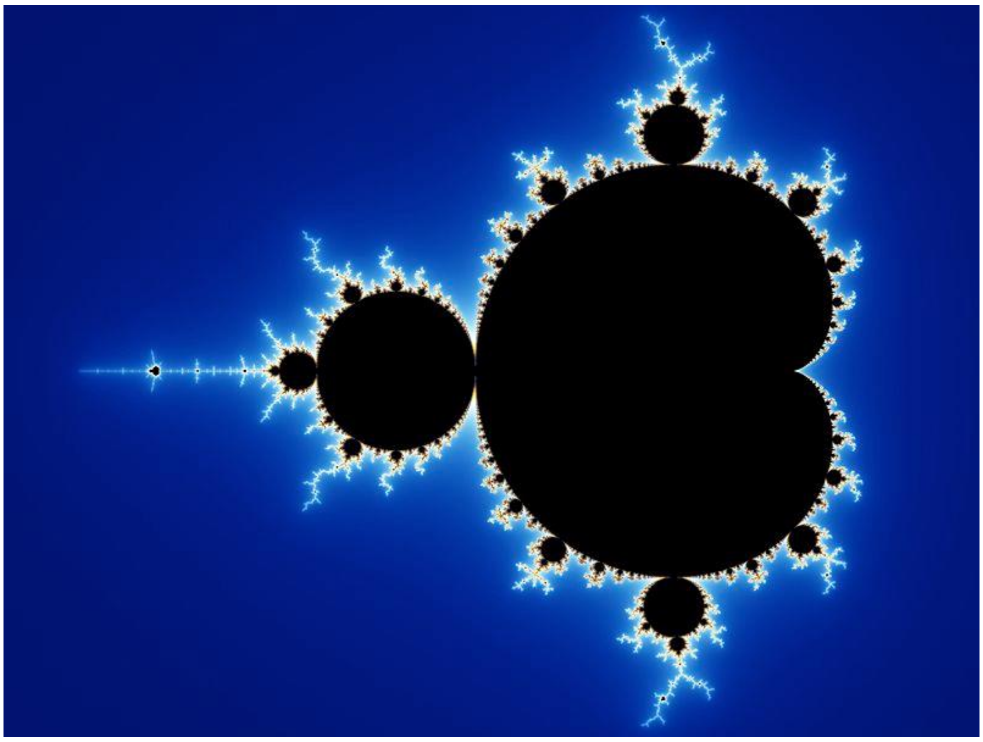
Remember the practical session no 2 (September, 19th) where a method to calculate an approximate value of pi was described and implemented using shared memory and threads. Adapt your solution to use C and MPI. Build two versions:

- Using *MPI_Send( )* e *MPI_Recv( )* operations
- Using *MPI_Bcast( )* e *MPI_Reduce( )* operations

Test your program with an increasing number of processors and annotate the elapsed time.

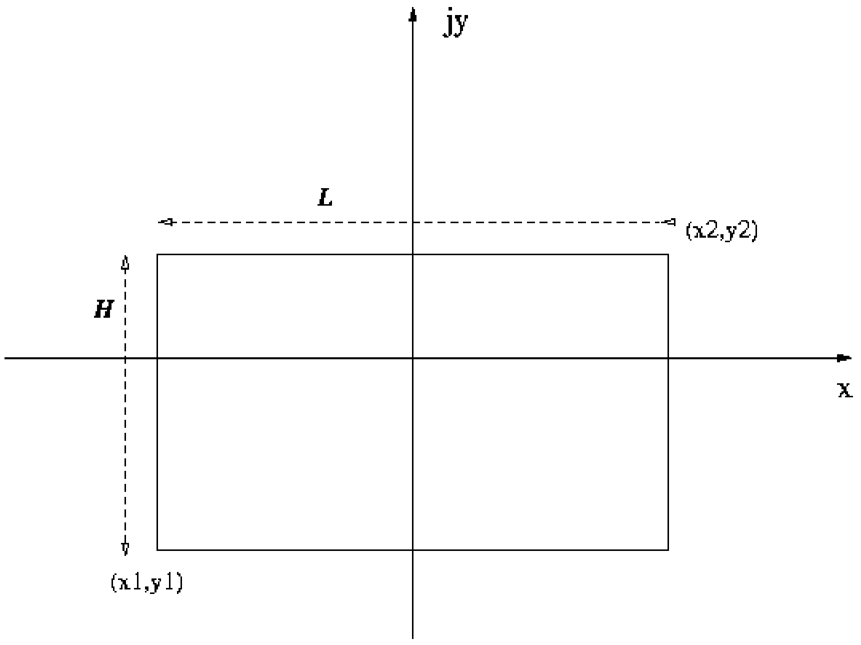
2nd part Calculation of MandelBrot figures

A Mandelbrot figure is a set of points in the complex plane



The fractal corresponds to the border of the figure defined by points. A complex number *c* belongs to the Mandelbrot set if starting from $x_0=0$ and repeatedly applying a recurrence formula the module of x_n never exceeds a certain limit, no matter how much *n* is increased.

The following figure presents the points (x_1, jy_1), (x_2, jy_2) in the complex plane.



Consider that the two previous points represent the vertices of a rectangle including a*L*H* points (where *L* represents the width, and *H* the height). For each of the points (*ci*,*cr*) in that rectangle, the algorithm in the following C code is executed:

```
int compute_point(double ci, double cr) {
        int iterations = 0;
        double zi = 0; double zr = 0; double nr, ni;
        while ((zr*zr + zi*zi < 4) && (iterations < max_iterations)) {
                nr = zr*zr - zi*zi + cr; ni = 2*zr*zi + ci;
                zi = ni; zr = nr;
                iterations ++;
        }
        return iterations;
}
```

This code is part of the file *mandel.c*, which is available in the course page in CLIP *Documentação de Apoio->Problemas*. That file includes:

- The C routines *compute* and *compute_point* that generate a matrix of points forming a Mandelbrot figure. These routines use the recurrence formula above and return a value between 1 and 255 corresponding to a level of grey (0- black, 255- white), or an index in a table of colors (in the file *color_map.c*, which is also available in CLIP.
- The routine *output_pgm* that uses the generated matrix to produce a colored image, or an image in levels of gray, in the PGM format. The files in this format have an ASCII header followed by integer values corresponding to the levels of gray, or color, of each pixel. More information complementing the self explaining code in this routine is available at the Wikipedia or from the URL *http://www.fileformat.info/format/pbm/egff.htm*

The program is to be invoked in the following way:

```
./mandel x1 y1 x2 y2  width height PGMfilename
```

where *x1 y1*, *x2 y2*, and *width height* have a meaning corresponding to what was explained before (the vertices and the points in the rectangle).

The goal of this exercise is to parallelize the calculation of the Mandelbrot image using the **MPI** library. An MPI version of *mandel* executed by *nprocs* processes will be invoked like this:

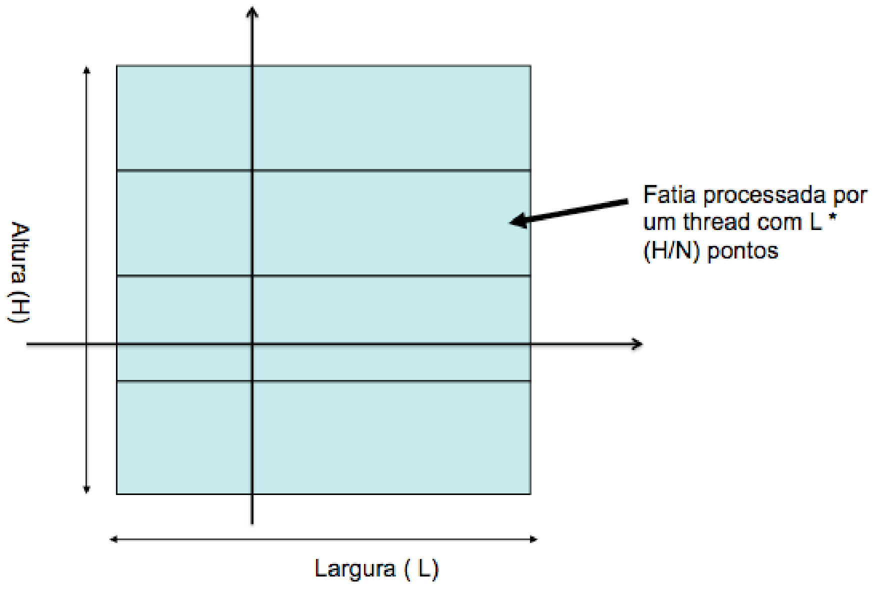
```
mpi-exec –n <nprocs> mandel-mpi x1 y1 x2 y2  width height PGMfilename
```

Assume that the values for *width* and *height* are multiples of *nprocs*. A possible approach to parallelize the code is to divide the rectangle in several bands/strips and each one becomes a unit of work. For instance, in the following figure and considering N processes, each one will process a band with L*(H/N) points.

Each thread will receive as input a band defined by its width (L – Largura), height (H – Altura) and the coordinates of the vertices (inferior on the left and the superior on the right). Each thread uses the recurrence formula described above to calculate the *bitmap* matching its band/strip. Each thread produces hence a *bitmap* which is part of the final image.

Parallelization strategies

a) *Strategy 1 – static distribution of work*: the surface to be calculated is subdivided in a fixed number of rectangles and each one is assigned to a process. The work associated with each point is variable (the work increases with the number of iterations that are necessary to obtain a value that diverges) meaning that this strategy may assign different workloads to each process.



b) *Strategy 2 – dynamic distribution of work*: the surface is divided into smaller work portions and each process requests a new one when it finishes the current work portion. The advantage of this approach is to have a better distribution of work since the processes that finish earlier may require an additional work portion. The goal of the work is, for both strategies, to measure the execution time resulting from modifying the number of threads. As a reference, use a figure with 2048 points in the dimensions of width and height and vertices located at (-2,-2) and (2,2).