Exercise 2 for the course of High Performance Computing

This second exercise, given for the course of 2023/2024 academic year, requires more coding than exercise 1.
There are two possible choices:

1. MPI-only code + OpenMP-only code
   You choose either ex. 2a or 2b, and implement them with MPI only, AND you implement ex. 2c with OpenMP only
2. MPI+OpenMP code
   You implement the code required by ex. 2c with an hybrid MPI+OpenMP code. In Appendix II, at the end of this page, you find a simple advice about how to write an hybrid code.

Note: this text may evolve to improve its clarity.
You are reading version `1.1` of Feb, 14th, 2024

Changes from v1.0

- clarified the OMP roles in ex 2a and 2b: none, pure MPI codes are required for those exercises
- added the choice of ex 2c; it may be chosen as OpenMP exercise, or as a unique exercise for MPI+OpenMP
- limit of pages for the report increased to 10

Rules

The same rules than for Ex. 1 apply, obviously.

- Exercise should be done individually.
- Materials (code/scripts/pictures and final report) should be prepared on a github repository, starting with this one and sharing it with the teachers.
- A report should be sent by e-mail to the teachers at least seven days (a week) in advance: the name of the file should `YOURSURNAME_report.pdf` if the report deals with both exercises, or `YOURSURNAME_exN_report.pdf` with `N={1,2}` in the opposite case.
- Results and numbers of the exercises should be presented (also with the help of slides) in a max 10 minutes presentation: this will be part of the exam. A few more questions on the topic of the courses will be asked at the end of the presentation.

deadlines

You should send us the e-mail at least one week before the exam:
`luca.tornatore@inaf.it`
`stefano.cozzini@areasciencepark.it`

For the first two scheduled sessions this means:

- exam scheduled at 06.02.2024 ***deadline 01.02.2023 at midnight***
- exam scheduled at 27.02.2024 ***deadline 26.02.2023 at midnight, considering the change to v1.1***

In the email please

- indicate the exam session for which you candidate (note: if you take the first one and you're not satisfied with your result, you can also take the following ones without having to redo the work - unless required by the examiners)
- include the URL of the github
- attach a report, in pdf

The report that you have to attach to the email must

- be limited to 10 pages
- state which one you have chosen between 2a and 2b
- Introduce the problem and the algorithm that you choose/use
- Explain your strategy
- Present the relevant details of your implementation
- Present the results, i.e.
    - the strong scalability
    - the weak scalability
    - the comparison with the algorithm found in the MPI library (for ex. 2a)
- Draw conclusions and possible improvements from your results

Exercise 2a

Implement a broadcast algorithm or a all-to-all algorithm (you choose either one), both in distributed memory and in shared memory.

Implementing the algorithm in distributed memory means that you write an MPI code that *using point-to-point calls only* implements either a broadcast (from any single process to all the other processes) or an all-to-all collective call among MPI processes.

You may use the reference given in Ex. 1 to have some insight about the algorithms, or you may find different ones at your convenience.

Exercise 2b

Write a parallel implementation of the QuickSort algorithm.

You find in this folder the file `quicksort.c` which is a standard implementation of the famous divide-and-conquer algorithm. You are provided only with a serial implementation of the vanilla quicksort algorithm, which consists in a recursive calling of a partitioning routines.
Feel free to improve whatever aspect you may spot in the code.

The requirement is to implement a parallel version that distribute the work among the MPI processes.
You can choose among 3 possibilities:

- every MPI process will generate its own chunk of data
- single MPI process generate data chunks and send them to each other MPI process
- the data are read from a file

It is important that you **do not assume that the data set can fit in the memory available on a single node**.
To simplify the implementation in distributed memory, you can assume that the data are always homogeneously distributed. You can also assume that they live in the range `[0,1)`.
The data to be sorted are double-precision floating point; however, to allow you to study the memory efficiency, the array entries are a structure of double of which one is used to sort the entries (see the code for the details).

Exercise 2c

The Mandelbrot set

The Mandelbrot set is generated on the complex plane $\mathbb{C}$ by iterating the complex function $f_c(z)$ whose form is

$$f_c(z) = z^2 + c$$

for a complex point $c = x + iy$ and starting from the complex value $z = 0$ so to obtain the series

$$z_0 = 0,\ z_1 = f_c(0),\ z_2 = f_c(z_1),\ \ldots,\ f_c^n(z_{n-1})$$

The $\mathcal{M}andelbrot\ Set\ \mathcal{M}$ is defined as the set of complex points c for which the above sequence is bounded. It may be proved that once an element i of the series is more distant than 2 from the origin, the series is then unbounded.
Hence, the simple condition to determine whether a point c is in the set $\mathcal{M}$ is the following

$$|z_n = f_c^n(0)| < 2 \ \text{ or } \ n > I_{max}$$

where I_{max} is a parameter that sets the maximum number of iteration after which you consider the point c to belong to $\mathcal{M}$ (the accuracy of your calculations increases with I_{max}, and so does the computational cost).

Given a portion of the complex plane, included from the bottom left corner $c_L = x_L + iy_L$ and the top right one $c_R = x_R + iy_R$, an image of $\mathcal{M}$, made of $n_x \times n_y$ "pixels" can be obtained deriving, for each point c_i in the plane, the sequence $z_n(c_i)$ to which apply the condition (???), where

$$\begin{aligned} c_i &= (x_L + \Delta x) + i(y_L + \Delta y) \\ \Delta x &= (x_R - x_L)/n_x \\ \Delta y &= (y_R - y_L)/n_y\,. \end{aligned}$$

In practice, you define a 2D matrix `M` of integers, whose entries `[j][i]` correspond to the image's pixels. What pixel of the complex plane $\mathbb{C}$ corresponds to each element of the matrix depends obviously on the parameters $(x_L, y_L), (x_R, y_R), n_x,$ and n_y.

Then you give to a pixel `[j][i]` either the value of 0, if the corresponding c point belongs to $\mathcal{M}$, or the value n of the iteration for which

$$|z_n\left(c\right)| > 2$$

(n will saturate to I_{max}).

This problem is obviously embarrassingly parallel, for each point can be computed independently of each other and the most straightforward implementation would amount to evenly subdivide the plane among concurrent processes (or threads). However, in this way you will possibly find severe imbalance problems because the $\mathcal{M}$'s inner points are computationally more demanding than the outer points, the frontier being the most complex region to be resolved.

Requirements:

1. your code have to accept $I_{max}, c_L, c_R, n_x \text{ and } n_y$ as arguments. Specifically, the compilation must produce an executable whose execution has a proper default behaviour and accept argument as follows:
   `./executable n_x  n_y  x_L  y_L  x_R  y_R  I_max`
2. the size of integers of your matrix `M` shall be either `char` (1 byte; $I_{max} = 255$) or `short int` (2 bytes; $I_{max} = 65535$).
3. your code must produce a unique output file. You may use MPI I/O if you choose to implement the MPI+OpenMP version, **directly producing an image file** using the very simple format `.pgm` that contains a grey-scale image.
   You find a function to do that, and the relative simple usage instructions, in Appendix I at the end of this page.
   In this way you may check in real time and by eye whether the output of your code is meaningful.
4. you have to determine the strong and weak scaling of your code. If you are developing a hybrid MPI+OpenMP code (option 2 at the beginning of this file), the scalings must be conducted as follows:
   - *OMP scaling* : run with a single MPI task and increase the number of OMP threads
   - *MPI scaling* : run with a single OMP thread per MPI task and increase the number of MPI tasks. Use at least two nodes, preferably four.

   The corresponding plots will be part of your report.

> **Note 1:** Mandelbrot set lives roughly in the circular region centered on $(-0.75, 0)$ with a radius of ~ 2.

> **Note 2:** the multiplication of 2 complex numbers is defined as $(x_1 + iy_1)\ \times\ (x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1)$

Appendix I

Writing a `PGM` image

The `PGM` image format, companion of the `PBM` and `PPM` formats, is a quite simple and portable one.
It consists in a small header, written in ASCII, and in the pixels that compose the image written all one after the others as integer values. A pixel's value in `PGM` corre
sponds to the grey level of the pixel.
Even if also the pixels can be written in ASCII format, we encourage the usage of a binary format.

The header is a string that can be formatted like the following:

```
printf( "P5\n%d %d\n%d\n", width, height, maximum_value );
```

where "`P5`" is a magic number, `width` and `heigth` are the dimensions of the image in pixels, and `maximum_value` is a value smaller than `65536`.
If `maximum_value < 256`, then 1 byte is sufficient to represent all the possible values and each pixel will be stored as 1 byte. Instead, if `256 <= maximum_value < 65536`
, 2 bytes are needed to represent each pixel (that is why in the description of Exercise 1 we asked you that the matrix `M` entries should be either of type `char` or of ty
pe `short int`).

In the sample file `write_pgm_image.c` that you find the `Assignment03` folder, there is the function `write_pgm_image()` that you can use to write such a file once you hav
e the matrix `M`.

In the same file, there is a sample code that generate a square image and write it using the `write_pgm_image()` function.
It generates a vertical gradient of N_x x N_y pixels, where N_x and N_y are parameters. Whether the image is made by single-byte or 2-bytes pixels is decided by the max
imum colour, which is also a parameter.
The usage of the code is as follows

```bash
cc -o write_pgm_image write_pgm_image.c
./write_pgm_image [ max_val] [ width height]
```

as output you will find the image `image.pgm` which should be easily rendered by any decent visualizer .

Once you have calculated the matrix `M`, to give it as an input to the function `write_pgm_image()` should definitely be straightforward.

Appendix II

A note about hybrid MPI+OpenMP

Although we did not yet discuss this topic in the class, at the level you may use it here that is quite straightforward. As you have seen, it is obviously not a requirement
 but just an opportunity for those among you that like to be challenged.

As long as you use OpenMP regions in a MPI process for computation *only* and *not* to execute MPI calls, everything is basically safe and you can proceed as usual with bot
h MPI and OpenMP calls and constructs.

It may be safer, however, to initialize the MPI library with a call slightly different than `MPI_Init()`:

```c

int mpi_provided_threaD_level;
MPI_Init_threads( &argc, &argv, MPI_THREAD_FUNNELED,
&mpi_provided_thread_level);
if ( mpi_provided_thread_level < MPI_THREAD_FUNNELED ) {
     printf("a problem arise when asking for MPI_THREAD_FUNNELED level\n");
   MPI_Finalize();
   exit( 1 );
}

...;  // here you go on with BaU

MPI_Finalize();
return 0;
```