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, 2a and 2b, of which you are required to pick one** (although nobody will complain if you solve both).

Note: this text may evolve to improve its clarity.
You are reading version `1.0` of Jan, 15th, 2024

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 22.02.2023 at midnight***

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 8 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 can also implement an hybrid code, i.e. an MPI code in which the MPI processes spawn OpenMP threads, if you think this may help.

Implementing the algorithm in shared memory means that you write an OpenMP code that implements either a broadcast (from any thread to all the other threads) or an all-to-all using what you know of OpenMP.

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 hybrid parallel version of it that:

- when called with 1 MPI process will share the work among the threads. In the source file there is the code needed to generate a random data homogeneously distributed in `[0,1])` in OpeMP.
- when called with many MPI processes will share the work also among the MPI processes. In this case, you can choose among 3 possibilities:
    1. every MPI process will generate its own chunk of data
    2. a single MPI process generate data chunks and send them to each other MPI process
    3. the data are read from a file

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).