EPFL

Parallel and High Performance Computing

Dr. Pablo Antolín

Series 6

April 3 2025

Advanced MPI

General information

In this exercise we will practice three advanced MPI topics: derived datatypes, persistent communications, and MPI I/O.

1 Pi

Exercise 1.1: *Derived types*

Note: *This exercise is a very artificial and unrealistic application of derived MPI datatypes. It is only supposed to make you play around with them.*

- Take the Gather version of the π computation from Series 5 and consider a struct containing a double and an integer:

```
struct Sum {
    double sum;
    int rank;
  };
```

  Declare a derived data type for this struct in MPI.

  Note: *A starting code is available in the exercise repository in the file* `pi_gather.cc` *of the folder* `lecture_06/pi`.

- After having computed the local sum contribution in each process, create a `Sum` struct containing the local `sum` and the local MPI `prank`.

- Gather all the `Sum` structs on process 0 using the custom MPI datatype. Check that the gathered ranks are the integers from 0 to (`psize-1`), sorted in ascending order.

- Add up all the sum contributions with process 0 and Bcast the result.

Exercise 1.2: ***Persistent communications***

Use the asynchronous ring version as a starting point for this exercise (Series 5 Exercise 2.3), and modify it to use persistent communications instead.
Note: *A starting code is available in the exercise repository in the file* `pi_p2p_async_ring.cc` *of the folder* `lecture_06/pi`.

2 Write BMP

Exercise 2.1: ***MPI I/O***

- For this exercise we will use the code in the `write_bmp` folder. This code uses a `Grid` and fills it with data, and then writes this grid as a `bmp` image.
- To execute the code, after building the executable using the provided `Makefile`, you have to pass as argument to the executable the number of points per direction (one single integer).
- You will have to modify the dump function in `dumper.cc` to write the file in parallel. The way the dumper works is the following:
  - Allocates a vector `img` of `char` with length `h` $\times$ `row_size` where `h` is the height (m) of the grid and `row_size` the width (n) times 3 (for the 3 colors red, green, and blue), plus a padding required by the `bmp` specification.
  - The grid's data is transfered to the vector `img`.
  - Two vectors `bmpfileheader` and `bmpinfoheader` are filled with the required headers for the `bmp` file to be valid.
  - The headers are written in the file.
  - The image is written in the file.

  To parallelize the data writing, you can use `MPI_File_write_at` instead of `fout.write()` calls. The data to be written can be split along `h`. Thus, `row_size` will remain the same for each task but the total height `h` will be split among the tasks. Therefore, you will have to compute the proper offsets for each task. Notice that the header should be written only once at the beginning of the file.
- ***Optional:*** *This dumper file is the same as the one used by the Poisson code, you can try to modify it to use it with your parallel Poisson code. You will have to be careful to ignore the ghost cells when assigning the values of* `img`.

  Notice that in a naïve non-parallel dumping, the root process would gather all the Poisson values `u` *from all the processes, and the would dump the results to the file. This approach has two main problems: It requires global communication (all the processes send information to the root) and the writing is serial (only the root process writes the file).*

3 Poisson stencil with mpi4py

In this second part of the exercise, you will implement the Poisson stencil in 2D using Python. An explanation of the Poisson stencil can be found in Series 0 and a possible parallelization strategy is included in this week's slides.

Your task is to transform the provided serial Python code into a parallel version using the `mpi4py` package. Useful documentation can be found at https://mpi4py.readthedocs.io/en/stable/. The required modules can be loaded as

```
module load intel intel-oneapi-mpi
module load python py-mpi4py
```

Exercise 3.1: ***Blocking communications***

Consider first an implemenation based on blocking communications with *MPI_Sendrecv*.

Exercise 3.2: ***Non-blocking communications***

Now, try using non-blocking communications.