High Performance Computing
Exercise Sheet 6

HS 25
Dr. Douglas Potter

http://www.ics.uzh.ch/
Teaching Assistants:
Cesare Cozza, cesare.cozza@uzh.ch
Mark Eberlein, mark.eberlein@uzh.ch

Issued: 24.10.2025
Due: 31.10.2025

This exercise sheet is special: Exercise 3 might take a while to solve so we extend the submission deadline of this particular exercise to be in two weeks. Exercises 1 and 2 are to be handed in as usual, in one week. If you only hand in Exercise 1 and 2 next friday you will get 2 points. If you hand in Exercise 3 in two weeks we update the points on this sheet to 4 given everything was answered.

MPI documentation: https://www.open-mpi.org/doc/current/

Exercise 1 [Ring and deadlocks]

In this exercise, you will write a MPI program which allows each process to communicate their rank around a ring and sum up all the ranks.

Assume, we have `n_proc` processes, each process will do the following:

(i) Store it's rank into `my_rank` and initialize the variables `my_sum` = 0 and `send_rank` = `my_rank`.

(ii) Calculate the ranks of the right and left neighbors. Assume that the last and the first rank are neighbours.

(iii) Loop `n_proc` times and do:

  (a) Send `send_rank` to it's right neighbour.

  (b) Receive the rank from the left neighbour and store it in `recv_rank`.

  (c) Add `recv_rank` to `my_sum`.

  (d) Update `send_rank` to `recv_rank`. So it sends the neighbours rank in the next loop iteration.

(iv) Print out the rank, the total number of ranks and the sum.

And the output should look like this, `I'm process 0 out of 4 and the sum is 6.`
`I'm process 1 out of 4 and the sum is 6.`
`I'm process 2 out of 4 and the sum is 6.`
`I'm process 3 out of 4 and the sum is 6.`

As a starting point, you can use the skeleton program `ring.c` in the folder `exercise_session_06/ring_and_deadlocks` of the course git repo.

Implement the ring communication following the above instructions. To begin with, use a synchronous send mode (`MPI_Ssend`) and the standard receive mode (`MPI_Recv`).

- What happens when you run the code? Why?

To overcome this problem, keep the sending and receiving modes but divide the processes into two groups: those with even ranks will first send the message and then receive it, while those with odd ranks will do otherwise.

- Run the code again. What changed and why did it solve the problem? Is it an optimal solution?

Commit: Push your code into your git repository. Name the source code `ring_1.c`.

Now let's try the non-blocking MPI communications. The difference with synchronous communication is that `MPI_Isend` and `MPI_Irecv` will complete even if the communication is not completed yet. Every non-blocking communication is assigned to a `MPI_Request` in order to be able to keep a track of the status of communication. To wait for a `MPI_Request` to complete, we then have to use `MPI_Wait` (for a single request) or `MPI_Waitall` (for multiple requests).

Commit: Try the combinations `Irecv-Isend-Waitall` and `Isend-Irecv-Waitall`. Why are these methods better solutions than synchronous communication for our ring problem? Push your code into your git repository and call the source code `ring_2.c`

Bonus: Decomposing a domain and computing the rank of the neighboring processes can be complicated and annoying, especially when you want to do it on multiple dimensions and if you have to consider boundary conditions. Fortunately, MPI provides routines that can do this task for us. In this exercise, you will create a 1D virtual Cartesian topology that allows communicating around the ring. After the usual MPI initialization, create a Cartesian topology communicator using the following routine

```
int MPI_Cart_create(
MPI_Comm comm_old,    // old communicator

int ndims,            // number of dimensions of the grid

int *dims,            // integer array of size ndims, specifies
                      // the number of processes in each dim.

int *periods,         // logical array of size ndims specifying
                      // whether the grid is periodic (true) or
                      // not (false) in each dimension

int reorder,          // ranking may be reordered (1) or not (0)
                      // (logical)

MPI_Comm *comm_cart); // new cartesian communicator
```

where `comm_old` is the "old communicator", in this case `MPI_COMM_WORLD`, `ndims` stands for the number of dimensions of your grid, `dims` is an array containing the number of processes per dimension, `periods` is an array of logicals specifying if the boundary conditions are cyclic or not for each dimension, `reorder` is an optimization option that you can set to `1` and `comm_cart` will be the new Cartesian communicator.

Recompute `my_rank` based on the new communicator `comm_cart`. From now on, there should be no more further usage of `MPI_COMM_WORLD`. Compute the rank of the right and left neighbors in the new topology using the routine

```
int MPI_Cart_shift(
MPI_Comm comm,  // communicator with cartesian structure
int direction,  // coordinate dimension of shift (integer)
int displ,      // displacement (>0: upwards shift,
                // <0: downwards shift (integer)
int *source,    // rank of source process (integer)
int *dest);     // rank of destination process (integer)
```

where `direction` is the dimension along which we want to communicate (from `0` to `ndims-1`), `displ` is the displacement of the shift (`1` if you look for the closest neighbors) and the routine will set `source, dest` to the ranks of the source and destination processes. Use the neighbor ranks you just computed to send and receive the messages in the ring.
(Note: to each process are also assigned coordinates related to its position in the virtual

topology, see Fig. 1. To compute these coordinates one can use the routine `MPI_Cart_coords`. However, in 1D the coordinate corresponds to the rank.)

Commit: Name the source code of this version `ring_3.c` and push it to git. Describe what advantage the new communicator provides.

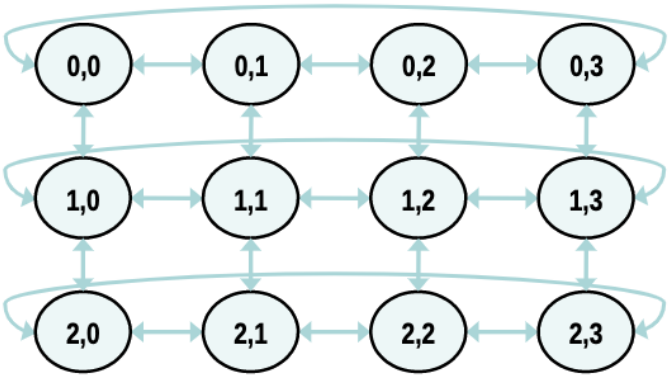

Figure 1: Cartesian virtual topology on a 2D domain, where `ndims=2`, `dims=[3,4]` and `periods=[0, 1]`.

Exercise 2 [π revisited]

Write a MPI program which computes $\pi = 3.141\,592\,653\,589\,793...$.
To do that, we can use the fact that π can be expressed as the sum of an infinite series (Leibniz formula):

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ... \tag{1}$$

and therefore, its computation can be easily parallelized. Of course we can't compute π exactly, but summing the first $N = 1\,000\,000\,000$ terms will do the job.

The code should the following three things:

a) Each process can take care of N/n_{proc} terms and calculate a partial sum of the series.

b) Each process will send the result back to the master process.

c) The master process will add the partial sums and output the result.

Commit: Commit your approximated value of π and your code.

Exercise 3 [MPI Poisson solver (due on 07.11.2025)]

In this exercise, you will parallelize with MPI the 2D Poisson solver which was part of last week's exercise session (first serial and then OMP version). We provide you a skeleton of the code, which you can find in the folder `exercise_session_06/poisson_MPI` of the git repository, and in the following we will guide you through four tasks you need to address in order to parallelize the code. For each task we describe different methods you can use to perform your parallelization: the first method is the simplest, while the others are progressively more sophisticated but also more performing. Please use comments in your code to document what your code is doing and which method you use. If you decide to implement different methods, you can use different branches in your git repository.

1. **Initialization**
   In the skeleton code we added a new module, `mpi_module.cpp`. In this module, you'll initialize the MPI environment and decompose the domain as a function of the number of processes and the topology you choose. Moreover, you will need to modify accordingly the subroutine `init.cpp` to take into account the size of each sub-domain and the communication haloes. We propose you three methods to decompose your domain:

   I. Decompose your domain in one direction only (let's say in the x direction, see Fig. 2) and compute the size of each process' slice size by `nx/n_proc`. Remember to add the remnant of the division to the slice of the last process. For each process, compute the indices `xmin` and `xmax` corresponding to the beginning and end of their subdomain in the x direction (potentially also `ymin` and `ymax`).

   II. Do the same as above, but instead of adding the remnant to the last process, share it within different processes. In this way you will avoid the overloading of the last process.

   III. Create a cartesian topology (see *Bonus* point of Exercise 1 and Fig. 1) and decompose your domain in both directions (compared to Fig. 1, you would need periodic boundary conditions in both dimensions).

2. **Halo communication**
   In parallel computing, the halo of a process refers to an ensemble of its neighboring processes. Each process has to communicate data with its halo to perform a Jacobi iteration. You will have to set up this communication in the `halo` routine in order to build the solver. In the case of a domain decomposed in the x direction, processes exchange arrays containing their outermost column with their neighbors. The `MPI` library allows multiple ways of achieving this, here are some options you can try:

   I. Use the blocking `MPI` routines `MPI_Ssend` and `MPI_Recv` with odd and even process groups to avoid deadlocks (see Exercise 1).

   II. Use non-blocking `MPI` communication with the routines `MPI_Isend`, `MPI_Irecv` and `MPI_Wait`.

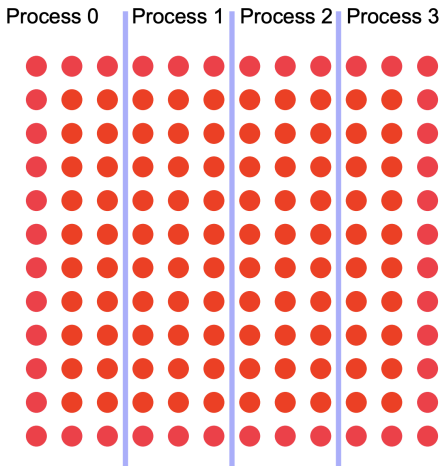

Figure 2: Domain decomposition over 4 processes of a 12×12 grid along the x direction.

III. As an attempt to reduce the overall waiting time of processes, an option is to first call the non-blocking `Irecv-Isend` routines, then perform computations on the inner local domains - where no `MPI` communication is required. Finally `Wait` for the halo arrays from neighbors, and perform the computations on the local boundaries.

3. **Computation of error**
   Each process can compute errors between two consecutive iterations on their local domain using the function `norm_diff`. Use the routine `MPI_Allreduce` to evaluate errors over the entire grid.

4. Bonus: **Output**
   In the skeleton version of the code, each process will output its local domain in a separate file. Pick out a master process that will collect and combine other processes' data. The master then writes the entire domain into one single file.

In the folder `exercise_session_06/poisson_MPI` you'll also find a python script `simple_mpi.py`, which will automatically collect the different output files and produces the plots. To use it, you must specify the number of processes used and the output number you want to see. For example, to see the 140th output of the run with 8 cores, you should do `python simple_mpi.py 8 140`. Instead, if you do the *Bonus* exercise, you should be able to output the results with last week's python script.

Commit: Parallelize the code and run some tests. You should achieve the same solutions as in the case of serial (and OMP) version. If you have time, try different methods and check which one is most performing. You may try different numbers of MPI process and comment on the performance.