High Performance Computing
Exercise Sheet 7

FS 21
dr. Douglas Potter

`http://www.ics.uzh.ch/`

Teaching Assistants:
Jozef Bucko, jozef.bucko@uzh.ch
Hugues de Laroussilhe, hugues@ics.uzh.ch

Issued: 23.04.2021
Due: 30.04.2021

Exercise 1 [MPI Poisson solver]

In this exercise, you will parallelize with MPI the 2D Poisson solver which was part of last week's exercise session. We provide you a skeleton of the code, which you can find in the folder `exercise_session_07/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. 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. 1) 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 `myimin` and `myimax` corresponding to the beginning and end of their subdomain in the x direction.

   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. (*Bonus*) Create a cartesian topology (see *Bonus* point of Exercise 1 from previous week's exercise sheet and Fig. 2) and decompose your domain in both directions.

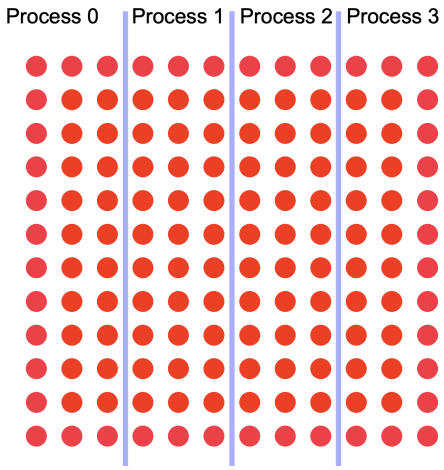

Figure 1: Domain decomposition over 4 processes of a 12×12 grid along the x direction.

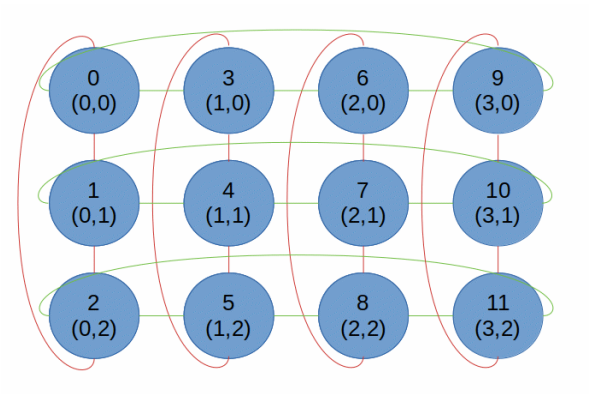

Figure 2: Cartesian virtual topology on a 2D domain, where `ndims`=2, `dims`=[4,3] and `periods`=[1, 1].

2. **Halo communication**

   In parallel computing, the halo of a process refers to an ensemble of its neighboring processes. Each process have 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 from previous session).

   II. Use non-blocking `MPI` communication with the routines `MPI_ISEND`, `MPI_IRECV` and `MPI_WAIT`.

   III. (*Bonus*) 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_07/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. If you have time, try different methods and check which one is most performing.

Exercise 2 [OpenMP Poisson solver]

In this exercise, you will have to parallelize the Jacobi Poisson solver with `OpenMP`. Obviously, the most time consuming operation is updating the grid components for each jacobi step, then comes the estimate of difference and error. Start with the serial version of the code, that you can find in the `exercise_session_07/poisson_OMP` in the git repository, then:

- Update the Makefile in order to compile with the `OpenMP` library.
- Insert `OpenMP` instructions at the appropriate spots to parallelize the `jacobi_step` and the `norm_diff` computations.
- Submit jobs with 1 and 4 threads and make sure that you get same answers.
- (*Bonus*) Update your `MPI` Poisson solver to make it compatible to both `MPI` and `OpenMP` (see this link for some help.)

Commit : Push your OMP parallel codes into git repo

Exercise 3 [Speedup tests]

Now that you have both a `MPI` and an `OpenMP` (and eventually a hybrid `MPI` + `OpenMP`) versions of the Jacobi Poisson solver, it is time to evaluate the performances of your implementations.

- Add a timing function to measure how long it takes to solve the Poisson problem.
- Measure and plot the speedup of the code using $[1, 4, 16, 36]$ `MPI` tasks or `OpenMP` threads on one single multicore node of Daint, with a grid sizes of $[128^2, 256^2, 512^2, 1024^2]$. If you implemented any advanced `MPI` strategy last week, do the experiment with

  Hint: Bash scripting can be helpful there !
- (*Bonus*) With a hybrid (`MPI` + `OpenMP`) version, test the following configurations of `MPI` tasks and `OpenMP` threads per task on one multicore node of Daint:

ntasks	nthreads per task
1	36
2	18
4	9
9	4
18	2
36	1

Commit : Push your codes and speedup plot into git repo

Exercise 4 [OpenMP and race conditions]

In this exercise you will parallelize with `OpenMP` a simple code which computes the maximum value and the number of `0` in a long array of integers (see file `num.txt`). You can find a serial version of the code in the git repository `exercise_session_07/race_conditions`. Remember that, by default, all variables in the `!$OMP PARALLEL` regions are `shared`. Therefore pay particular attention to which variables you want to keep `shared` and which ones you want to make `private`.

- First task, parallelize the first loop which computes the maximum value of the array. Use a simple `!$OMP DO` region and keep the same structure. Run it multiple times with different number of threads and check that it always outputs the correct result.
- Now parallelize in the same way also the second loop, counting the occurrences of `0` in the array. Run it again multiple times. What happens?

- ***Commit :*** If you weren't cautious enough in the precedent tasks, you probably run into a race condition. One way to solve it is to use the `ATOMIC` or `CRITICAL` directives. Where is it best to insert these directives in order to avoid a serialization of your code?
- Another way to parallelize the two loops is to use `REDUCTION` clauses. Try this option.
- (*Bonus*) Check that your code also compiles in serial.