Université Grenoble Alpes, Grenoble INP, UFR IM^{2}AG

Master 1 Informatique and Master 1 MOSIG

UE Parallel Algorithms and Programming

Lab # 4

2022

Important information

- This assignment will be graded.
- The assignment is to be done by groups of at most **2** students.
- Deadline: **April 26**.
- The assignment is to be turned in on Moodle

Your submission

Your submission is to be turned in on Moodle as an archive named with the last name of the two students involved in the project: `Name1_Name2_lab4.tar.gz`.

The archive will include:

- A short report (in `txt, Markdown or pdf` format[1]) that should include the following sections:
  - The name of the participants.
  - For each exercise:
    - A short description of your work (what you did and didn't do)
    - Known bugs
    - Any additional information that you think is required to understand your code.
- The archive (`lbm_sources.tar.bz2`) of your sources generated by running the command `make archive` in the `code` directory.

Expected achievements

Considering the time that is given to you to work on the assignment, here is a scale of our expectations:

- An **acceptable work** is one in which at least the 3 exercises for 1D splitting have been implemented.

[1] Other formats will be rejected

- A **good work** is one in which Exercise 4 on 2D splitting has been implemented in addition to the 3 exercises on 1D splitting.
- An **excellent work** is one in which Exercise 1-6 have all been implemented, and some of the proposed *extra* works have been done.

One possible *extra* work is to run scalability measurements in the Cloud. Note that this task can be done even if we have only implemented some algorithms (for instance, Exercises 1-3). See the last section of the document for more details on the proposed *extra* works.

1 Introduction

In this lab we will work on a CFD (Computational Fluid Dynamic) simulation. There are various methods to simulate a fluid. One of them is the Lattice Boltzmann Method (LBM) which is *simple* to parallelize in distributed memory systems.

The provided code is a basic implementation with a unique fluid phase and produces a simulation of the Kármán vortex street. It corresponds to the vortices generated by a wind blowing on an obstacle. An example of such a case is the vortices observed on one of the Juan Fernandez islands as shown by Figure 1.

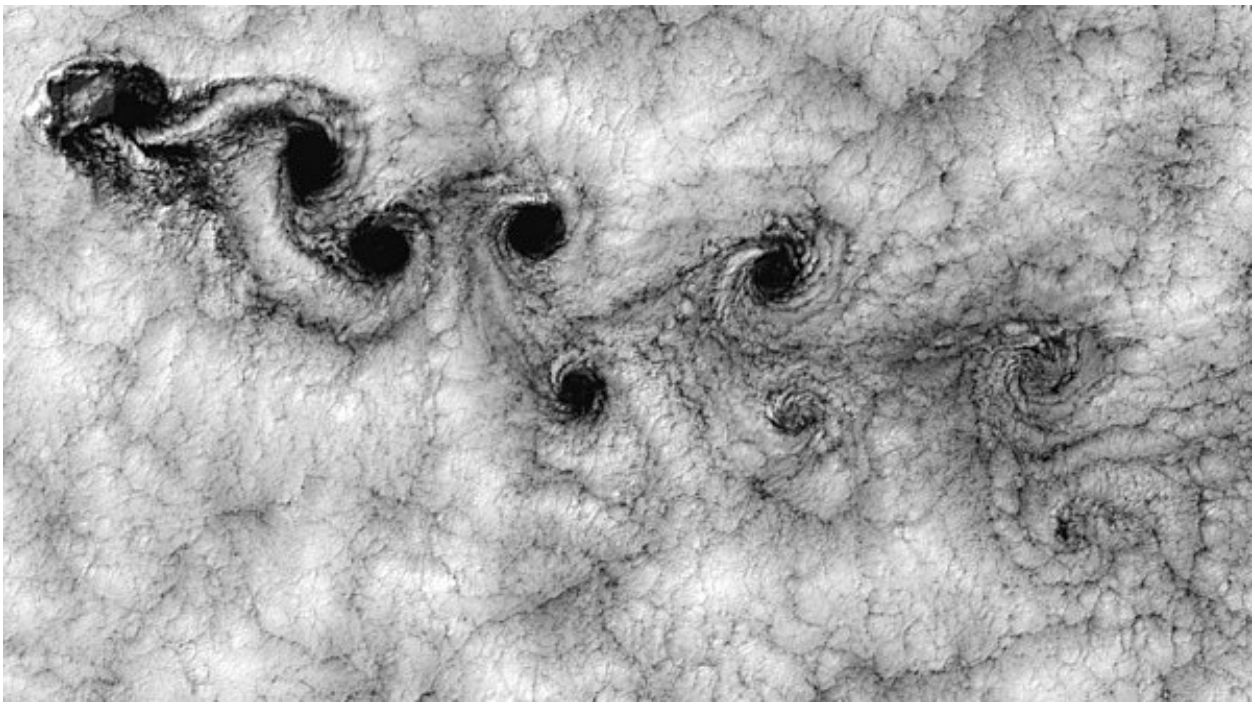

Figure 1: Karman street vortices observed on one of the Juan Fernandez islands by Landsat 7 from NASA. Source: `https://commons.wikimedia.org/wiki/File:Vortex-street-1.jpg?uselang=fr`.

For this lab, the computational part of the simulation is provided. You will be asked to implement the MPI communication scheme with various approaches. An example of the output generated by our application is provided in Figure 2. The colors represent the speed of the fluid at the given position. It is presented using an abstract unit linked to the physical sizes via the Reynolds number.

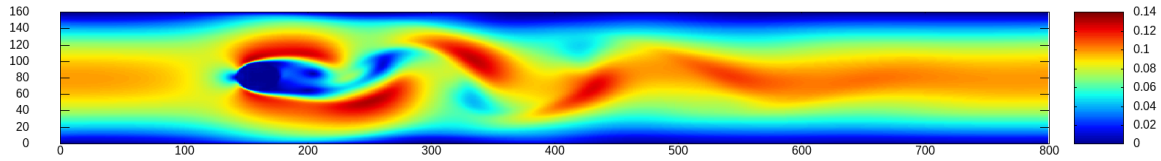

Figure 2: An example of output on a 800x160 mesh with Reynold of 200 after 6400 steps.

The result depends on the Reynolds number we are using. Shortly, in fluid dynamic, this dimensionless number is a ratio between the characteristic speed of the fluid, the characteristic size of the problem, and the viscosity of the fluid. In other words it defines the relative scale of the problem. Depending on the selected value, we are in the laminar, transient or turbulent domain. The vortices in this simulation arise with a Reynolds around 100-200 and disappear for a value lower than 80.

Some words on the Lattice Boltzmann Method

The Lattice Boltzmann Method splits the simulation space, in our case a 2D plan, in a mesh of cells. Each cell contains a part of the simulated fluid. The fluid is modeled in each cells by the quantity of fluid going in each direction. In our case we will consider 9 directions as shown in Figure 3. This scheme is called `D2Q9` in the literature.

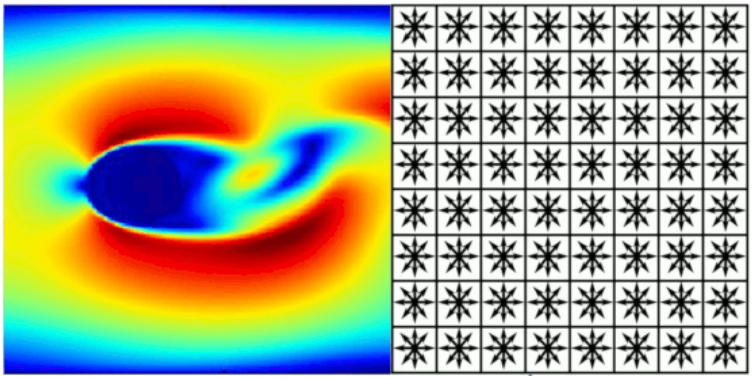

Figure 3: Fluid representation.

For each time step, the computation steps implemented in `src/phys.c` are:

- Applying specific conditions (inflow, outflow, border, obstacle).
- Computing the effect of collisions between the fluid going in the 9 directions to redistribute the fluid over those directions.
- Propagating the fluid in the 9 directions to the neighbor cells to make it move over the mesh.

Looking for more information?

If you would like to know more about LBM simulations, here are some references:

- Lattice Boltzmann Method for Fluid Simulations (A relatively short document detailing the method and use cases).

- From a lattice Boltzmann model to a parallel and reusable implementation of a virtual river (A PhD thesis)

Acknowledgments

This lab has been proposed and designed by Sebastien Valat.

2 The provided LBM code

You are provided with a C implementation of the Lattice Boltzmann Method. In this implementation, the code for splitting the problem into multiple parts that can be managed by different processes, and the management of the communication between these processes is missing. You will be in charge of implementing these missing parts. Note that:

- File `exercise_0.c` provides a working sequential implementation of the algorithm, to serve as reference. **Do not edit this file**.
- To implement the different exercises, you will only have to modify the files `exercise_*.c`.
- The utility source code is stored in the `./src/` directory. You do not need to modify it.

In addition to the LBM simulation, the provided code includes a dedicated program to test the communication patterns that you implemented. This additional program will help you during the debugging phase, as detailed below.

Compiling and running the programs

A Makefile is provided to build the project. Running `make` produces the unique `lbm` executable. When launching the program, you can choose which exercise you want to test with the `--exercise` or `-e` option (value from 0 to 6)

The file `config.txt` defines some parameters for the simulation (including the problem size). You can provide an alternative config file using the `--config` or `-c`. The default configuration defines a small problem size. With a parallel version of the code, you can multiply each dimension by 2 or 4 to get a more accurate simulation while still having a reasonable compute time.

Running the program produces an output file (`raw` file). You can render the result into an animated GIF file by using the provided script `gen_animate_gif.sh`. This script requires the `gnuplot` command. You can optionally get faster rendering for large simulations if you have `GNU Parallel` and `ImageMagick` installed. The script `gen_animate_gif.sh` takes 2 parameters: i) The name of the `raw` file to render; ii) The name of the `gif` file to produce.

Listing 1 illustrates the compilation of the program, its execution for Exercise 1, and the generation of the corresponding `gif`.

Listing 1: Building and running the simulation

```
make
mpirun -np 4 ./lbm -e 1
./gen_animate_gif.sh output.raw output.gif
```

Dedicating program to validate the communication

To ease communication debugging, we provide you with the `check_comm` program which runs on a tiny mesh to test the communication functions you implemented. It displays the results in the terminal, with errors colored in red.

In addition to the `--exercise` option, this program allows you to choose between different filling patterns for the mesh using the `-p/--pattern` option. These different patterns can help identifying different bugs:

- The `rank` pattern fills each cell with the rank of the MPI process that hosts it. It allows checking whether the algorithm sends data to the right processes.
- The `position` pattern assigns a different value to each cell. It allows checking if the algorithm sends and stores the right data at the right place. This is more precise but more verbose.
- The `modulo9` and `modulo10` patterns print shorter numbers than `position` to ease readability. The different values for the modulo allow observing different patterns.

Finally, you can use the `-s/--show` option to select what the program should display: `expected`, `current`, or `both`, as illustrated in Listing 2.

Listing 2: Debugging communication implementation

```
make
mpirun -np 4 ./check_comm -e 1
mpirun -np 4 ./check_comm -e 1 -p rank
mpirun -np 4 ./check_comm -e 1 -p position
mpirun -np 4 ./check_comm -e 1 -p position -s expected
mpirun -np 4 ./check_comm -e 1 -p modulo9  -s both
mpirun -np 4 ./check_comm -e 1 -p modulo10 -s both
```

Validation by computing a checksum

To validate that your solution is correct for each exercise, you can visually compare the obtained animation to the animation generated by the sequential sum. However, subtle bugs might be difficult to identify visually. Hence, to verify that the images produced by two solutions are exactly the same, we suggest you to use checksums.

With the help of the provided `display` program, you can compare the frames corresponding to the same time step, generated by two simulations. Listing 3 compares the result of the reference implementation (the sequential one) with the result for exercise 1 for time step 3.

Listing 3: Validate application output

```
# make a reference run
mpirun -np 4 ./lbm -e 0
./display --gnuplot output.raw 3 | md5sum > ref.md5

# run the exercise you want to check
mpirun -np 4 ./lbm -e 1
./display --gnuplot output.raw 3 | md5sum -c ref.md5
```

Comment: The output files generated by two simulations cannot be directly compared because the data layout in the file depends on how the problem is parallelized.

3 Comments about the implementation

Memory layout

The memory representation of a cell is composed of 9 contiguous doubles (constant `DIRECTIONS`), each corresponding to one direction. To represent the mesh we chose to make the cells contiguous along the Y

axis (vertical) and non-contiguous along the X axis. This is called *column-major* order. **You will need to pay attention to this when implementing communication, especially since a *row-major* order is usually used to store matrices in C**. This layout is illustrated by Figure 4.

As shown in Figure 4, each cell can be identified by its X-Y coordinate. Cell `(0,0)` is at the top left. The memory layout implies that cell Cell `(0,1)` is stored next to cell `(0,0)` in memory, but cell `(1,0)` is not.

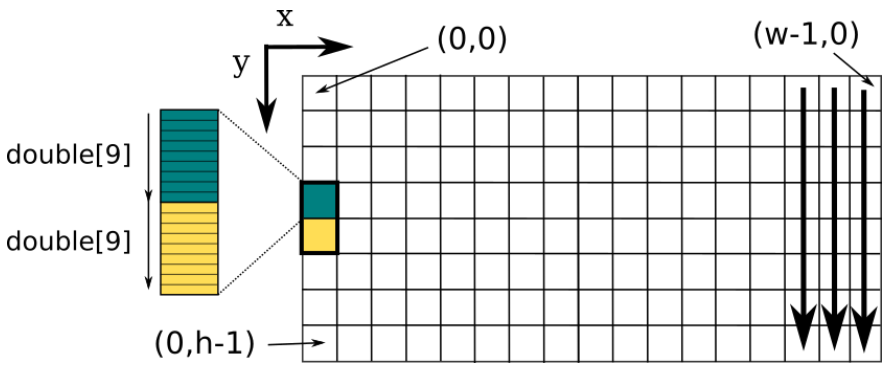

Figure 4: Mesh memory representation.

Ghost cells

To parallelize the code with MPI, we will split the global domain into multiple sub-domains. Each subdomain is hosted by one MPI rank.

The propagation step of the simulation requires that each cell knows the values of its neighbor cells. This implies communication between processes to exchange values at the border of the sub-domains.

To implement this communication, we will rely on a mechanism called *ghost cells*. It is illustrated by Figure 5 and Figure 6. Ghost cells are an extra cell layer created around each subdomain. This extra layer of cells should be used by each process to store the values it receives from its neighbors. In figures 5 and 6, colors show the source of the data to be stored in each ghost cell.

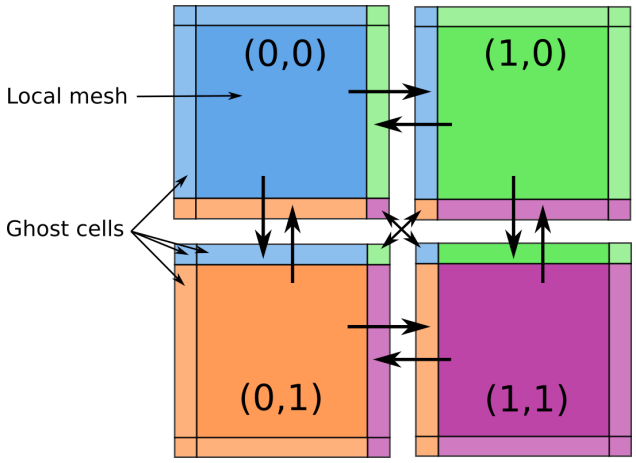

Figure 5: Sub-domain representation with their ghost cells used for communications.

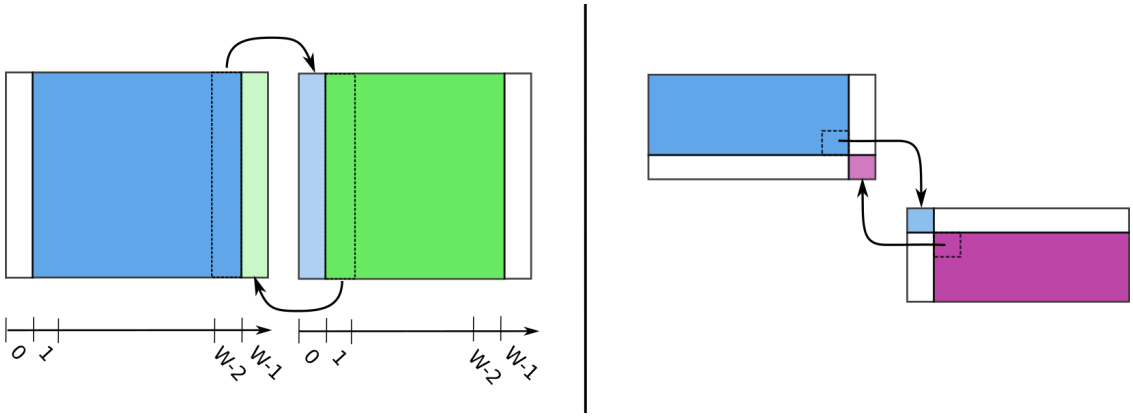

Figure 6: Zoom on ghost cells exchanges where W represents the width of the local domain (taking into account the ghost cells).

Helper function

To help you manipulating cells in each local domain, the function `lbm_mesh_get_cell()` is available. It returns the address of a cell based on its coordinated in the *local* domain:

Listing 4: Cell accessor

```
double * cell = lbm_mesh_get_cell(mesh, local_x, local_y)
```

We recall that a cell is composed of 9 (`DIRECTIONS`) contiguous doubles.

Your work

For each exercise, your work is:

1. To specify how the global domain should be split to run the computation in parallel (see function `lbm_comm_init*()` in `exercise_*.c`).
2. To implement the communication to exchange information between sub-domains (see function `lbm_comm_ghost_exchange*()` in `exercise_*.c`).

To specify how the global domain should be split, your work will be to initialize the `lbm_comm_t` data structure. The fields that need to be initialized are:

- `nb_x`: Number of tasks (=processes) along the X axis. If `nb_x` does not divide the `total_width`, the program should abort.
- `nb_y`: Number of tasks (=processes) along the Y axis. If `nb_y` does not divide the `total_height`, the program should abort.
- `rank_x`: Position of the current task along the X axis (in task number).
- `rank_y`: Position of the current task along the Y axis (in task number).
- `width`: Width of the local subdomain (taking into account ghost cells).
- `height`: Height of the local subdomain (taking into account ghost cells).

- `x`: Absolute position (in cells) of the local subdomain in the global one (ignoring ghost cells). We consider here the absolute position of the top left cell of the local mesh.
- `y`: Absolute position (in cells) of the local subdomain in the global one (ignoring ghost cells). We consider here the absolute position of the top left cell of the local mesh.

4 1D splitting

In a first step, we will split the global domain along the X axis to distribute the work over the MPI processes as shown by Figure 7. You will implement three variants:

- An implementation using blocking communication.
- A performance improvement using the odd/even communication pattern.
- An implementation using non-blocking communication.

Comment: If you want, you can start working on Exercise 7 (scalability measurement) as soon as you are done with the first 3 exercises. You can then extend your study with the algorithms based on 2D splitting.

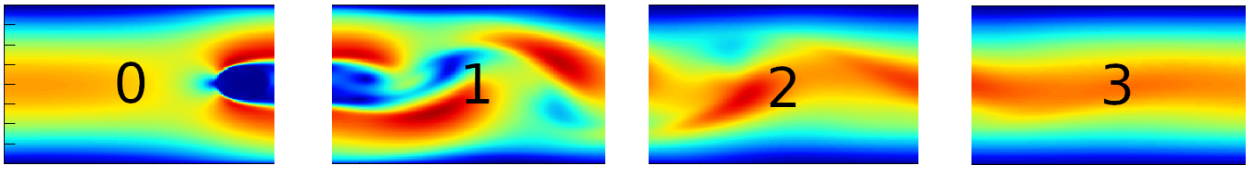

Figure 7: 1D splitting along the X axis.

Exercise 1: 1D blocking communication

In file `exercise_1.c`, implement the 1D splitting along the X axis by using MPI blocking communication functions (`MPI_Send()` and `MPI_Recv()`).

Proceed as follows:

- Implement domain splitting by completing function `lbm_comm_init_ex1()`.
- Implement the left and right communication exchanges, taking into account corner cases, in function `lbm_comm_ghost_exchange_ex1()`.

Remember that you can debug communication using the `check_comm` binary.

Exercise 2: 1D odd/even communication

The previous implementation might create a dependency chain between the ranks that imply that communications do not progress in parallel. This is a problem when running at scale. To have parallel communication with blocking function calls, we will use an odd/even approach.

The idea is to split the processes in two groups, odd or even, depending on their rank. The odd ranks will first send then receive and the even one will first receive and then send. For this approach we still use blocking communication functions.

- Implement the odd/even communication pattern in `exercise_2.c`.

Exercise 3: 1D non-blocking communication

An alternative to implement parallel communication is to use non-blocking communication functions.

- implement the communication exchanges using non-blocking functions in `exercise_3.c`.

5 2D splitting

In order to reduce the number of ghost cells when using more processes, we are now interested in splitting the domain in 2D (along X and Y), as illustrated by Figure 5. We will consider 3 implementations:

- A blocking implementation based on a manual copy of data for managing non-contiguous cells.
- A variant using MPI Datatypes to manage non-contiguous cells.
- A variant with non-blocking communication.

Exercise 4: 2D blocking communication, manual copy

In this exercise, you should split the domain over the X and the Y axis to distribute the work over the MPI processes. Because of the data layout in memory, when working in 1D you were able to directly send/receive the cells on the border of the local domains: You only had to send values stored contiguously along the Y axis. We now have to send the top and/or bottom cells of each local domain, which are not contiguous in memory.

To send non-contiguous data, in this exercise you will have to use a temporary buffer into which you will *manually* copy the non-contiguous data before sending. The data will also be received in a temporary buffer and will be then copied into the correct ghost cells.

Implement the solution in file `exercise_4.c`, taking into account the following instructions:

- Use blocking communication functions.
  - You will have to implement communication in the 8 directions: top, bottom, left, write, top left, top right, bottom left, bottom right.
- You can use the `MPI_Cart_*()` functions to split the mesh in 2D and to organize the processes in a Cartesian topology.
  - For our simulation, the mesh is **not periodic**.
  - You can use the field `comm->communicator` to store the Cartesian communicator you create.
- Allocate the temporary buffers once in `lbm_comm_init_ex4()`.
  - Their address can be stored in `comm->buffer_recv_down`, `comm->buffer_recv_up`, `comm->buffer_send_down` and `comm->buffer_send_up` fields.
  - you should free them in `lbm_comm_release_ex4`.
- Tip: To simply deal with the case of the processes being on the border of the global mesh, we recommend you to transfer corner cells during the left/right/top/bottom communication.

Exercise 5: 2D blocking communication using MPI Datatypes

MPI offers a way to handle non-contiguous data without manual copies, using `MPI_Type` functions.

In `exercise_5.c`, implement a new version of the 2D splitting where the communication of ghost cells is managed using MPI Datatypes.

- Create the new Datatype in `lbm_comm_init_ex5()` and store it in `comm->type`.
- Free the Datatype you created in `lbm_comm_release_ex5()`.

Exercise 6: 2D with non-blocking communication

Just like we did for Exercise 3, implement a new version of the 2D solution using non-blocking communication functions:

- Implement the solution in `exercise_6.c`.
- Tip: Depending on your design choices, it might be the case that in the previous implementation, you were sending twice the corner ghost cells. This could create an issue here. To avoid the problem, we suggest you to run the communication in two batches, one for the left/right/top/bottom directions and another one for the diagonal communication.

6 Extra works

This section describes some directions to extend you work. You can choose to follow one or several of these directions. The two main directions are:

- Running a performance evaluation of your algorithms in the Cloud.
- Implement some Matrix product algorithms using MPI

Exercise 7: Scalability measurement

Use your Google cloud account to make scalability measurements for the various communication schemes you implemented. Note that you can disable the output file writing to measure only the communications and computation performance by using the `-n/--no-out` option.

We recall that:

- You received instructions to obtain Google Cloud credits earlier in the semester
- A Web site presenting a set of tutorials about Google Cloud is accessible here: `https://roparst.gricad-pages.univ-grenoble-alpes.fr/cloud-tutorials/`
- On this Web site, there is a tutorial dedicated to running MPI applications: `https://roparst.gricad-pages.univ-grenoble-alpes.fr/cloud-tutorials/mpi/`

For scalability tests, we can consider two approaches. Strong scaling refers to measuring the execution time for a fix global problem size. Ideally the execution time should be divided itself by two each time we double the number of processes. One problem with this approach is measuring the sequential time, which can take a long time when we want to use a problem size tuned for a large number of processes.

The other approach, called weak scaling, consists in growing the problem size while we increase the number of processes to keep the local domain size constant. In other words we double the size of the mesh if we double the number of processes.

You are free to choose the approach you want. You can use the `--scaling` option to specify a scaling factor. With this option, the provided executable will automatically increase the global mesh size to fit the requested scaling. Note that the sizes are sometimes adapted here to ensure that the mesh can be split in equal sub-domains.

After running scalability measurements, you should:

- Provide on your report a description of the experiments you made (including a description of the hardware configuration).
- Present the results of your measurements using graphs.
- Provide an analysis of the results you obtained.

Exercise 8: Matrix products

We published on Moodle the description of an extra lab about the implementation of different algorithms to compute a matrix product in parallel. You can pick one or several of the proposed algorithms and try to implement them.

Message Passing Interface - Quick Reference in C

Environmental

- `int MPI_Init (int *argc, char ***argv)` - Initializes MPI
- `int MPI_Finalize (void)` - Cleans up MPI

Basic communicators

- `int MPI_Comm_size (MPI_Comm comm, int *size)`
- `int MPI_Comm_rank (MPI_Comm comm, int *rank)`

Blocking Point-to-Point Communication

- `int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)` - Sends a message to one process.
- `int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)` - Receives a message from one process
- `int MPI_Sendrecv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)` - Makes a send and receive operation in one call.

Non-blocking Point-to-Point Communications

- `int MPI_Isend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)` - Initiates the sending of a message to one process.
- `int MPI_Irecv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)` - Initiates the reception of a message from one process

Communicators with Topology

- `int MPI_Cart_create (MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)` - Creates a cartesian topology
- `int MPI_Cart_rank (MPI_Comm comm, int *coords, int *rank)` - Determines rank from cartesian coordinates
- `int MPI_Cart_coords (MPI_Comm comm, int rank, int maxdims, int *coords)` - Determines cartesian coordinates from rank
- `int MPI_Cart_shift (MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)` - Determines ranks for cartesian shift.
- `int MPI_Cart_sub (MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)` - Splits into lower dimensional sub-grids

MPI Types

- `int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)` - Creates a MPI vector type.
- `int MPI_Type_commit(MPI_Datatype * datatype)` - Commits a MPI type to be ready to use on all ranks.
- `int MPI_Type_free(MPI_Datatype *datatype)` - Releases a MPI type before existing.

Constants

Datatypes: `MPI_CHAR`, `MPI_SHORT`, `MPI_INT`, `MPI_LONG`, `MPI_UNSIGNED_CHAR`, `MPI_UNSIGNED_SHORT`, `MPI_UNSIGNED`, `MPI_UNSIGNED_LONG`, `MPI_FLOAT`, `MPI_DOUBLE`, `MPI_LONG_DOUBLE`, `MPI_BYTE`, `MPI_PACKED`

Reserved Communicators: `MPI_COMM_WORLD`
Ignore status for asynchronous MPI calls: `MPI_STATUS_IGNORE`
Target rank to use to skip a communication: `MPI_PROC_NULL`