M1 Info/MOSIG: Parallel Algorithms and Programming

Spring 2023

Lab sheet #4: MPI Lattice Boltzmann

04.04.2023

Contents

1) Details about submission

1.a) Submission instructions

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`.

Only submissions via the Moodle page will be accepted. You can do multiple uploads and the **last upload will be used for grading**. Submissions via Email are not allowed unless there are any technical issues.

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.
- **Programming in groups:** Regarding the programming part, you're encouraged to work in **groups of up to 2 students.** Please keep in mind that those of you who know the answer should **not tell the answer straight ahead**, but **help others to find the answer "on their own"**.

1.b) 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 assignments for 1D splitting (Sec. 6) have been implemented.
- A **good work** is one in which the assignments for the 2D splitting (Sec. 7) has been implemented in addition to the 1D splitting.
- An **excellent work** is one in which all non-optional assignments have been implemented (Sec. 8), and some of the optional assignments have been accomplished.

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. See the last section of the document for more details on the proposed *advanced* assignments.

1.c) Submission deadline

The **deadline** for the submission is 24.04.2023 at 23:59 Paris time.

[1] Other formats will be rejected

2) 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 understand and to parallelize in distributed memory systems.

The provided code is a basic implementation with a single 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.

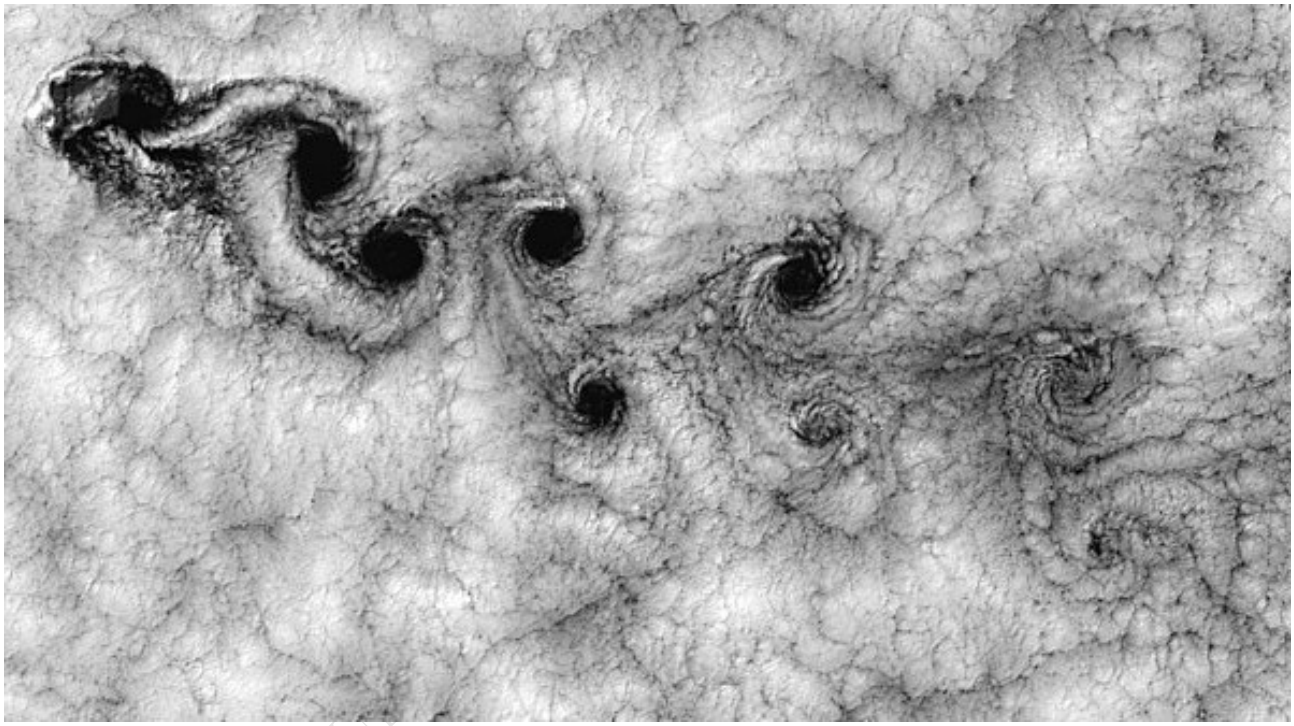

Figure 1: Karman street vortices observed on one of the Juan Fernandez islands by Landsat 7 from NASA. Source: `https://commons.wikimedia.org/wiki/File:Vortex-street-1.jpg?uselang=fr`.

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.

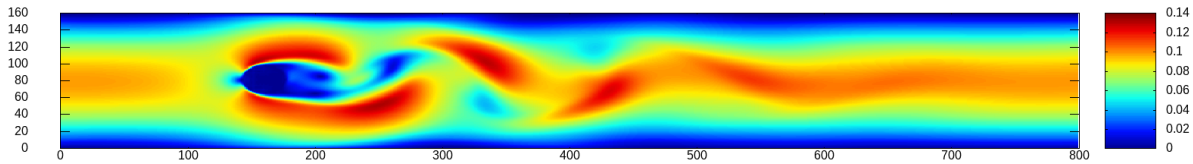

Figure 2: An example of output on a 800x160 mesh with Reynold of 200 after 6400 steps.

The result depends on the boundary conditions, initial state and parameters of the equation (Reynolds number) we are using. The Reynolds number is a dimensionless number (no units such as "seconds") and represents the 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.

2.a) Some words about 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 the probability density distribution vectors where the length of each vectors describes the state of the fluid in this cell. The fluid is modeled in each cells by the probability of fluid going in each direction which is then related to its quantity. In our case we will consider 9 directions as shown in Fig. 3. This scheme is called `D2Q9` in the literature.

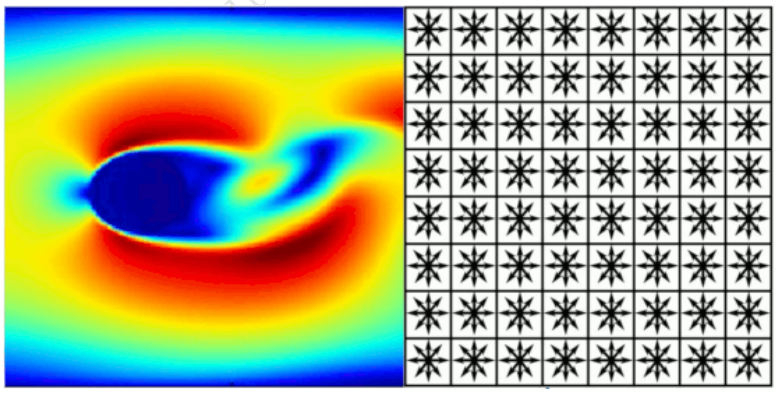

Figure 3: Fluid representation.

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, see Fig. 4

- Propagating the fluid in the 9 directions to the neighbor cells to make it move over the mesh.

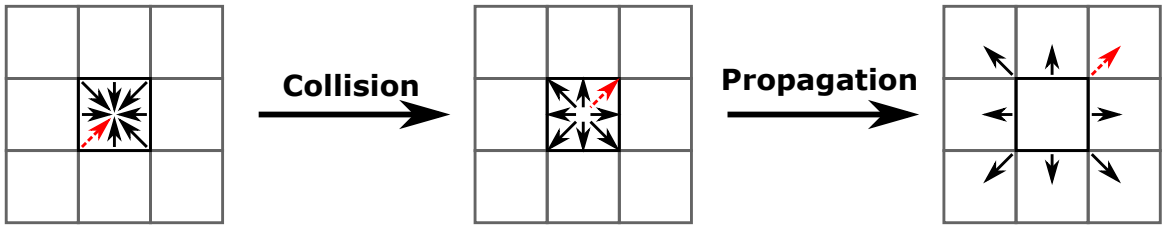

Figure 4: Collision and propagation operator of the LBM method.

2.b) Related work

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

- Yuanxun Bill Bao & Justin Meskas, "Lattice Boltzmann Method for Fluid Simulations" (A relatively short document detailing the method and use cases) `https://web.archive.org/web/20150418141146/http://www.cims.nyu.edu/~billbao/report930.pdf`.
- Alexandre Dupuis, "Hybrid From a lattice Boltzmann model to a parallel and reusable implementation of a virtual river" (2002) `https://web.archive.org/web/20190305083629/http://cui.unige.ch/~chopard/CA/aDupuisPhD.pdf` (PhD thesis)
- M. Schreiber "GPU based simulation and visualization of fluids with free surfaces" (2010) `https://www.martin-schreiber.info/data/research_diplomathesis/thesis_2010_06_08.pdf`
- C. Riesinger, et al., "A holistic scalable implementation approach of the lattice Boltzmann method for CPU/GPU heterogeneous clusters" (2017) `https://www.mdpi.com/2079-3197/5/4/48/pdf`

2.c) Acknowledgments

This lab was proposed and developed by Sébastien Valat.

3) The provided LBM code

A C implementation of the Lattice Boltzmann Method is provided. Here, the code for splitting the problem into multiple parallelizable 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 which serves as a reference solution. **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.

3.a) Compiling and running the programs

A Makefile is provided to build the project. Running `make` produces the `./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 raw output file (`output.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` (using output.raw for input and output.gif for output as default parameters).

```
$ make
$ mpirun -np 4 ./lbm -e 1
$ ./gen_animate_gif.sh
```

Listing 1: Building and running the simulation

3.b) 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 in red color.

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 runs 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 output: `expected`, `current`, or `both`, as illustrated in Listing 2.

```
$ 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
```

Listing 2: Debugging communication implementation

3.c) 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** (4th argument).

```
# 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
```

Listing 3: Validate application output

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.

4) Comments about the implementation

4.a) 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 5.

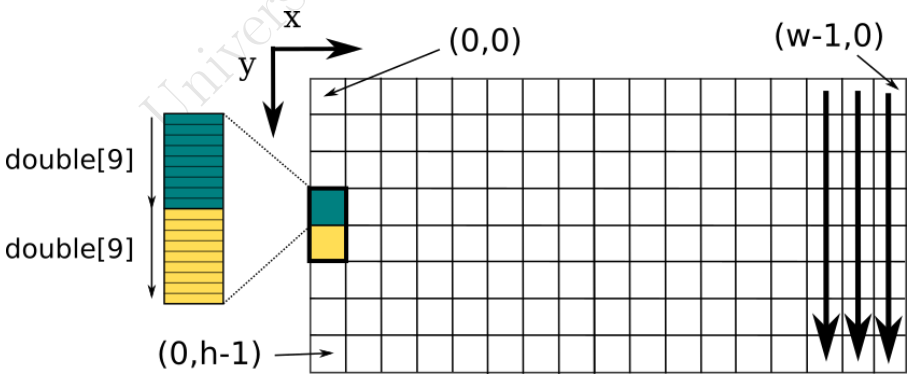

Figure 5: Mesh memory representation.

As shown in Figure 5, 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.

The 9 values relating to the fluid properties in each cell are stored consecutively in memory as shown on the left side of Fig. 5.

4.b) 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 6 and Figure 7. 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 6 and 7 we use colors to show the source of the data to be stored in each ghost cell.

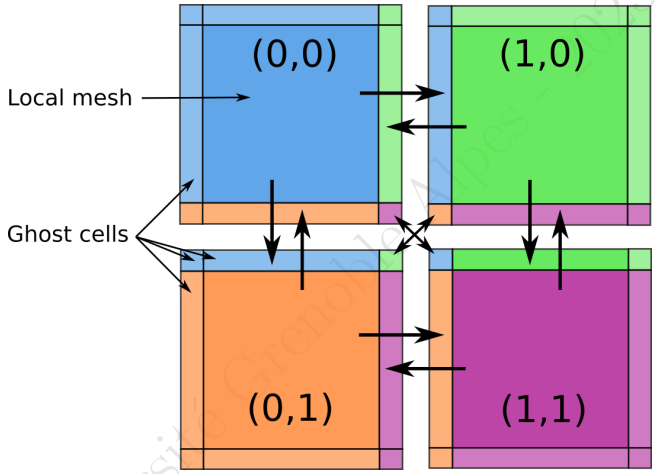

Figure 6: Sub-domain representation with their ghost cells used for communications.

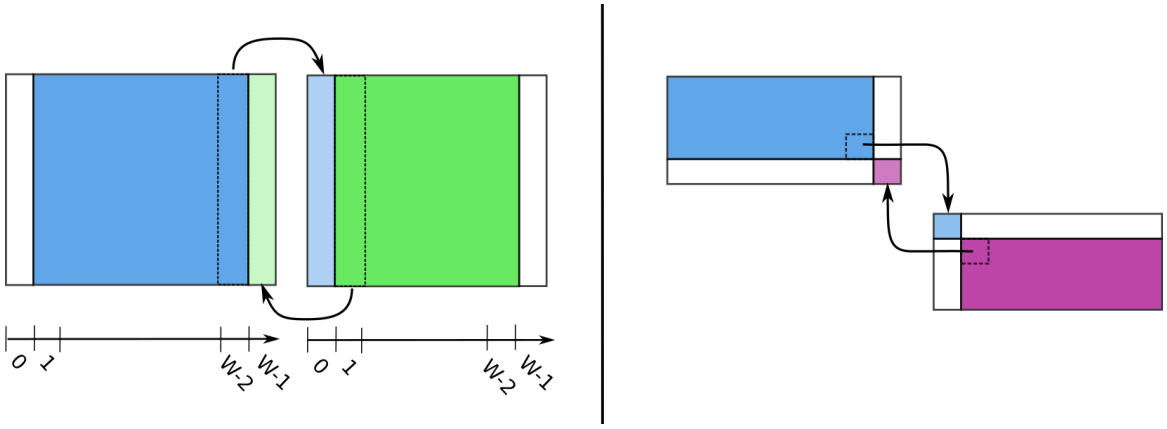

Figure 7: Zoom on ghost cells exchanges where W represents the width of the local domain (taking into account the ghost cells).

4.c) 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:

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

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

5) Assignments: General comments

For each exercise, your work is:

(a) To specify how the global domain should be split to run the computation in parallel (see function `lbm_comm_init*()` in `exercise_*.c`).

(b) 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 MPI processes along the X axis. If `nb_x` does not divide the `total_width`, the program should abort.
- `nb_y`: Number of MPI 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.

6) Assignment 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 8. You will implement three variants (explained in the following sections):

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

Hint: If you want, you can start working on the scalability measurements (Sec. 8.1) as soon as you are done with the first 3 exercises. You can then extend your study with the algorithms based on 2D splitting.

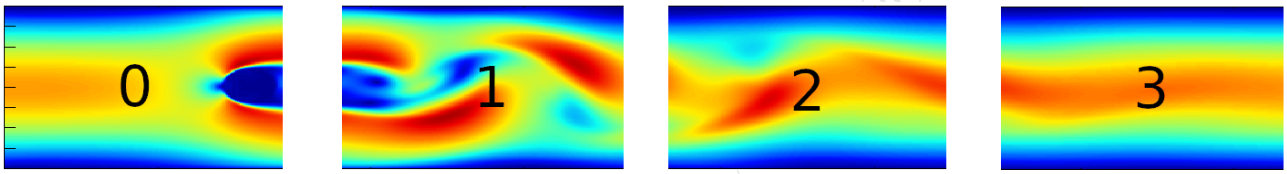

Figure 8: 1D splitting along the X axis.

6.a) 1D blocking communication (exercise 1)

Assignment **(Exercise 1)** 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.

6.b) 1D odd/even communication (exercise 2)

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.

Assignment **(Exercise 2)** Implement the odd/even communication pattern in `exercise_2.c`.

6.c) 1D non-blocking communication (exercise 3)

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

Assignment **(Exercise 3)** Implement the communication exchanges using non-blocking functions in `exercise_3.c`.

7) Assignment: 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 Fig. 6.

The following variants will be explained in more depth in the following sections:

- 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.

7.a) 2D blocking communication, manual copy (exercise 4)

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.

Assignment **(Exercise 4)** 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`.

Hint: 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.

7.b) 2D blocking communication using MPI Datatypes (exercise 5)

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

Assignment **(Exercise 5,)**

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()`.

7.c) 2D with non-blocking communication (exercise 6)

Similar to exercise 3, implement a new version of the 2D solution using non-blocking communication functions:

Assignment **(Exercise 6)** Implement the solution in `exercise_6.c`.

Hint: 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.

8) Advanced assignments (optional)

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.

8.a) Scalability measurement (exercise 7)

Assignment 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.

8.b) 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.

9) Message Passing Interface - Quick Reference in C

9.a) MPI warmup/shutdown

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

9.b) Basics about communicators

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

9.c) 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.

9.d) 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

9.e) 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

9.f) 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.

9.g) 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`