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

Master 1 Informatique and Master 1 MOSIG

UE Parallel Algorithms and Programming

Lab # 4

2021

Important information

- This assignment will be graded.
- The assignment is to be done by groups of at most **2** students.
- Deadline: **April 27, 2021**.
- 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 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.

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.

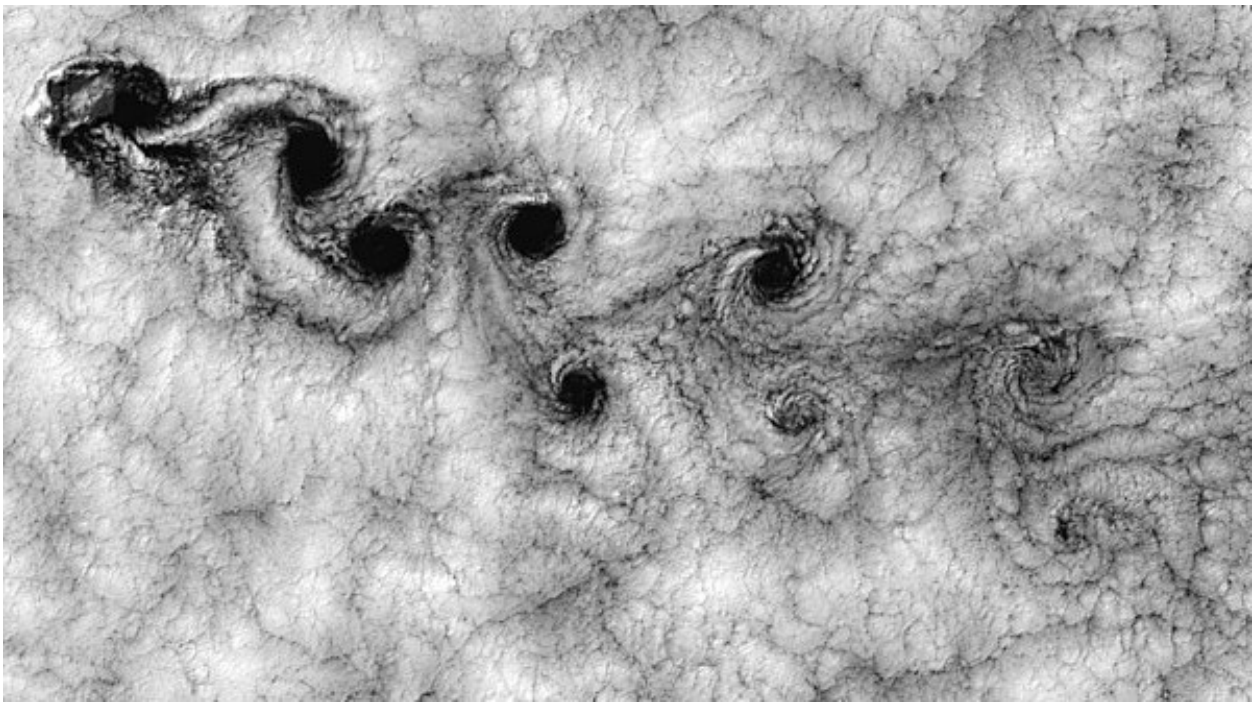

Figure 1: Karman street vortices observed on one of the Juan Fernandez islands by Landsat 7 from NASA. Source: `https://commons.wikimedia.org/wiki/File:Vortex-street-1.jpg?uselang=fr`.

The computational part is provided and 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 in abstract unit linked to the physical sizes via the Reynolds number.

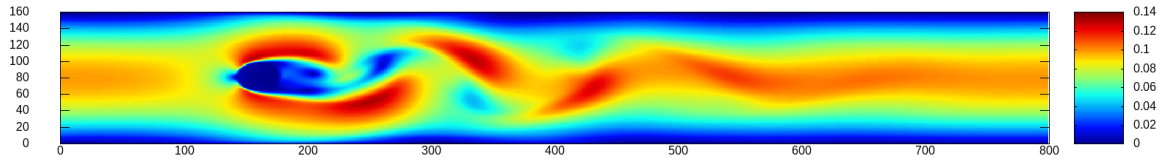

Figure 2: An example of output on a 800x160 mesh with Reynold of 200 after 6400 steps.

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 scales 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 consists in splitting the simulation space in a mesh of cells, in our case over the 2D plan. 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. Depending on the desired precision, we can choose to have more directions implemented. In our case we will consider 9 directions as shown by Figure 3. This scheme is called in the literature. D2Q9.

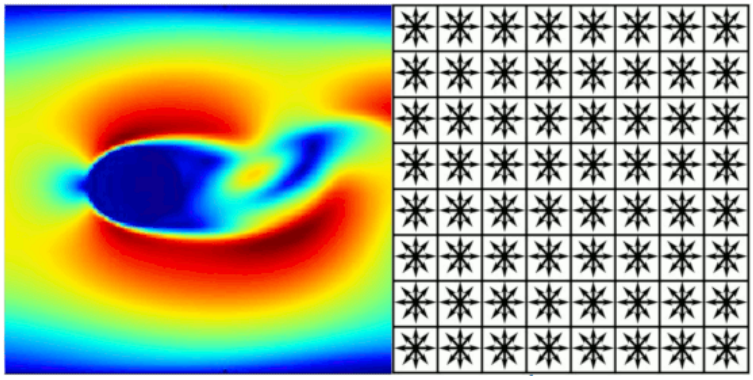

Figure 3: Fluid representation.

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

- Apply specific conditions (inflow, outflow, border, obstacle)
- Compute the effect of collisions between the fluid going in the 9 directions to redistribute the fluid over those directions.
- Propagate the fluid in the 9 directions to the neighbor cells to make it move over the mesh from cell to cell.

More details

If you want more details about LBM simulations, you can look on those documents:

- Lattice Boltzmann Method for Fluid Simulations (A relatively short document detailing the method and use cases).
- A thesis: From a lattice Boltzmann model to a parallel and reusable implementation of a virtual river

The provided LBM code

The lab sources provide a C implementation of the Lattice Boltzmann Method without the communication schemes you need to implement. All the work will be done in the `exercise_*.c` files. A sequential implementation is provided in `exercise_0.c` to serve as reference. **Do not edit this file**. The utility source code that you do not need to modify is stored in the `./src/` directory.

Compiling and Running

You can build the project using the provided `Makefile` by running the `make` command. It will produce a unique executable `lbm` containing all the exercises. You can choose which exercise to run by using the `-exercise` or `-e` option providing the exercise ID as parameter value (from 0 to 6).

The size of the problem you want to run and other parameters are defined in the `config.txt` file. Changing the default values can be useful to choose the right size when making scalability tests. You can provide an alternate config file using the `-c` or `--config` option. The default configuration defines a small size to be fast to run. When having a parallel version, you can easily multiply each dimension by 2 or 4 to get more precision while still having a reasonable compute time on 8 cores.

After producing the output file, you can render the result into an animated GIF file by using the script `gen_animate_gif.sh`. This script expects the raw data file as input and the name of the output GIF. You need to have the `gnuplot` command installed to be able to make the rendering. You can optionally get faster rendering for large simulations if you have `GNU Parallel` and `ImageMagick` installed.

Listing 1: Building and running

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

Validate communications

In order to ease communication debugging, we provide you with the `check_comm` executable which runs on a tiny mesh and displays it in the terminal with the splitting and colors to identify the ghost cells. It colors errors in red so you can progress quickly. You run it just like the main executable with the `-e/--exercise` option to select the exercise you are working on.

You can choose between different filling pattern via the option `-p/--pattern` to validate various aspects of the communication:

- `rank` fills the cells with the local rank to check whether the algorithm sends the data to the right process.
- `position` fills the cell with a different value for every cell to check if the algorithm sends and stores the right data at the right place. This is more precise but more verbose.
- `modulo9` and `modulo10` print shorter numbers than `position` to ease readability. The different modulo allow observing different patterns.

You can also choose to use the `-s/--show` options to select what to show (`expected`, `current` or `both`).

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
```

You can also validate the output of the `lbm` program by making a checksum of the displayed mesh. Use a run with `--exercise 0` as a reference and select a frame which was after at least one step to make the comparison. In the example below we chose to validate using the frame 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: Do not checksum directly the output file as its data layout depends on the MPI sub-domain splitting.

Regarding the implementation

The memory representation of a cell is composed of 9 doubles, each one corresponding to a direction. Those 9 (`DIRECTIONS`) doubles are contiguous in memory. 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 often used to store matrices in C**. This representation is illustrated by Figure 4.

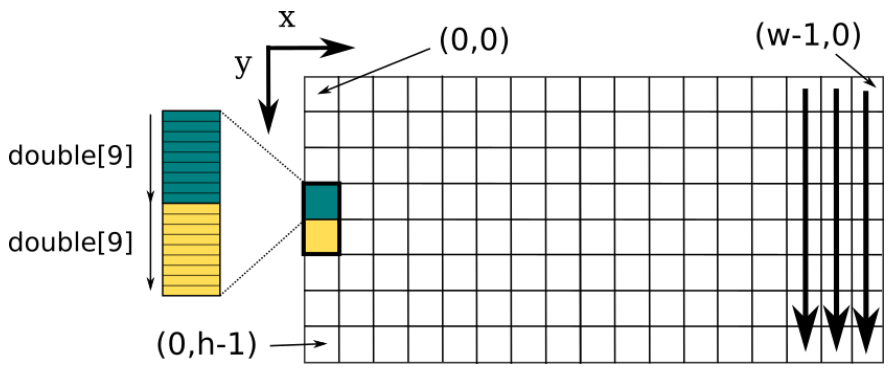

Figure 4: Mesh memory representation.

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 copies a value from a cell to the neighbor cells. In order

to simply manage the values that will be exchanged between processes at the border of the sub-domains, we created an extra cell layer around the subdomain. This is commonly called *ghost cells* and they are used for communication to be able to propagate the fluid from one domain to the next one: the *ghost cells* are used by one process to store the values it receives from its neighbors. It is presented by Figure 5 and Figure 6.

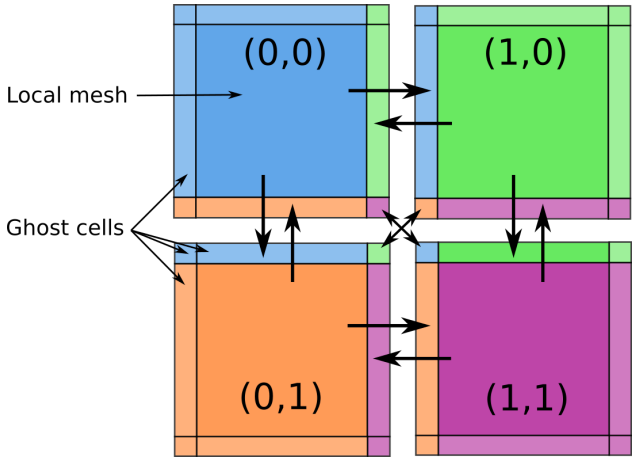

Figure 5: Sub-domain representation with their ghost cells used for communications.

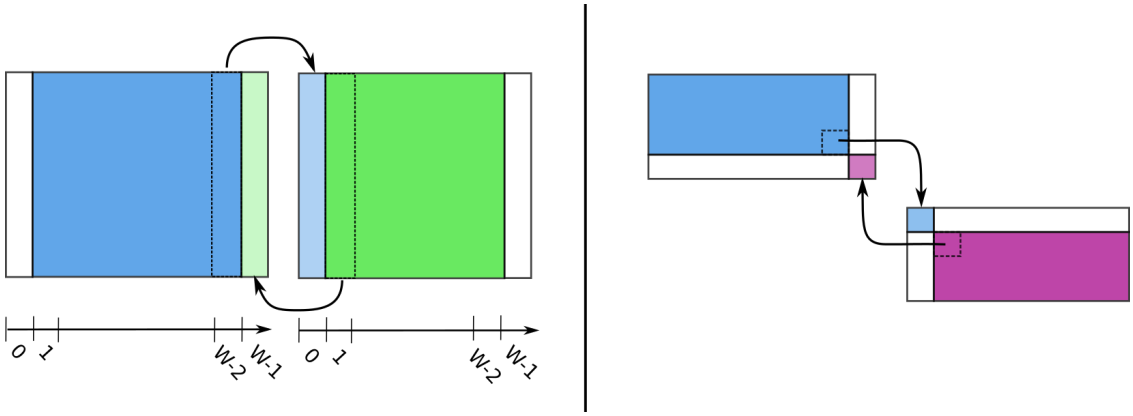

Figure 6: Zoom on ghost cells exchanges where W represents the width of the local domain (taking into account the ghost cells).

Cell accessor and comm structure

You can get the address of a particular cell using its local coordinate by using the `lbm_mesh_get_cell()` function. We remind here that a cell is composed of 9 (`DIRECTIONS`) contiguous doubles.

Listing 4: Cell accessor

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

You will also have to interact with the `lbm_comm_t` structure containing configuration information about the processes mapping and the local domain. You will have to fill some of its fields:

- `nb_x`: Number of tasks (processes) along the X axis. If `nb_x` does not divide the total width we should abort.

- `nb_y`: Number of tasks (processes) along the Y axis. If `nb_y` does not divide the total height we 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 (accounting ghost cells).
- `height`: Height of the local subdomain (accounting 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.

1D splitting

As 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. We will make three variants:

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

Comment: Note that 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.

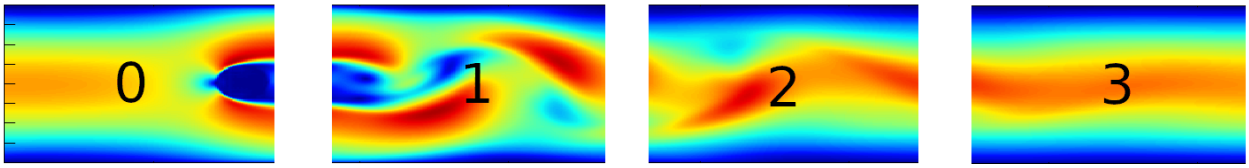

Figure 7: 1D splitting along X axis.

Exercise 1: 1D blocking communication

Implement the 1D splitting along X by using the MPI blocking communication functions. This will be implemented in the `exercice_1.c` source file. For this first step we will use blocking communication functions (`MPI_Send()` and `MPI_Recv()`).

Proceed by:

- implementing the domain splitting computation in function `lbm_comm_init_ex1()`.
- implementing the left and right communication exchanges in function `lbm_comm_ghost_exchange_ex1()`.

Don't forget that you can debug your communications using the `check_comm` binary.

Exercise 2: 1D odd/even communications

The previous implementation creates a dependency chain between the ranks so all the communications do not progress in parallel. This represents a large cost when running at large scale. In order to improve the communication, we will make a first fix by using an odd/even approach.

The idea is to split the ranks in two groups, odd or even depending on their rank ID. 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.

Proceed by:

- implementing the communication exchanges in function `lbm_comm_ghost_exchange_ex2()` in `exercice_2.c`.

Exercise 3: 1D non-blocking communication

Still to break the dependency chain between ranks you will now use non-blocking communication functions on the 1D splitting.

Proceed by:

- implementing the communication exchanges in function `lbm_comm_ghost_exchange_ex3()` in `exercice_3.c`.

2D splitting

In order to reduce the number of ghost cells when using more processes, we are interested in splitting the domain in 2D (along X and Y) instead of the 1D splitting we previously implemented. You will be asked to make three implementation:

- A blocking implementation using manual copy to the non contiguous cells.
- A variant using the MPI Datatypes for the non contiguous cells.
- A variant with non-blocking communication.

Exercise 4: 2D blocking communication, manual copy

We will split the domain over the X and Y axis to distribute the work over the MPI processes. Based on our memory representation of the mesh, you were previously able to directly communicate the ghost cells because you had only to send values stored contiguously along the Y axis. We now have to send the top and bottom lines which are not contiguous in memory. You will proceed by copying the data in a temporary buffer to be sent then copying back the received data into the corresponding ghost cells of the mesh.

Proceed in `exercice_4.c`. You can use the `MPI_Cart_*()` functions to compute the splitting. For our simulation, the mesh is **not periodic**.

We will use blocking communication as a first step. You will have to implement communication in the 8 directions: top, bottom, left, write, top left, top right, bottom left, bottom right.

Comments:

- The temporary buffers should be allocated 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`.
- You can store the new communicator in `comm->communicator`.

- You should also transfer corner cells during the left/right/top/bottom transfers to automatically handle the special case of the processes being on the border of the global mesh.

Proceed by:

- Initializing the splitting computation in function `lbm_comm_init_ex4()`.
- implementing the communication exchanges in function `lbm_comm_ghost_exchange_ex4()`.

Exercise 5: 2D noncontiguous via MPI Datatypes

In practice MPI offers a way to handle the non-contiguous communication. It goes through the use of the `MPI_Type` functions. Create a datatype to describe the ghost cells and make the communication via this type instead of making the manual copy.

Proceed by:

- Initializing the splitting computation in function `lbm_comm_init_ex5()`.
- implementing the communication exchanges in function `lbm_comm_ghost_exchange_ex5()`.

Comment: As before you can create the type into `lbm_comm_init_ex5()`, store the type into `comm->type` and release it in `lbm_comm_ghost_exchange_ex5()`.

Exercise 6: 2D with non-blocking communications

Just like we did for Exercise 3, replace the blocking communication from Exercise 4 or 5 by non-blocking communication. Apply the changes in `exercice_6.c`.

Proceed by:

- implementing the communication exchanges in function `lbm_comm_ghost_exchange_ex6()`.

Comment: In the previous implementation we send twice the corner ghost cells. This could create an issue here. To avoid the problem, we suggest you to run the communication in two batch, one for the left/right/top/bottom directions and another for the diagonal communications.

Possible 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 measurement for the various communication schemes we 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/`

When speaking about scalability measurement we can consider two approaches. Strong scaling refers to measuring the execution time on a use case which has a fix size for every number of processes. Ideally we should observe an execution time which divides itself by a factor two each time we double the number of processes. The limit of this approach is the sequential reference run which can take a long time when we want to use a problem size tuned for a large number or processes.

The other approach, called weak scaling, consists in growing the problem size while we increase the number of processes to keep a fixed amount of work for each tasks. 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 `--scale` option to specify a scaling factor. With this option, the provided executable will automatically increase the mesh size to fit the requested scaling. Note that the sizes are sometimes adapted here to ensure that the mesh can be split in sub-domains.

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)` - Initialize MPI
- `int MPI_Finalize (void)` - Cleanup 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)` - Send 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)` - Receive 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)` - Make 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)` - Send 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)` - Receive 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)` - Create with cartesian topology
- `int MPI_Cart_rank (MPI_Comm comm, int *coords, int *rank)` - Determine rank from cartesian coordinates
- `int MPI_Cart_coords (MPI_Comm comm, int rank, int maxdims, int *coords)` - Determine cartesian coordinates from rank
- `int MPI_Cart_shift (MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)` - Determine ranks for cartesian shift.
- `int MPI_Cart_sub (MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)` - Split into lower dimensional sub-grids

MPI Types

- `int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)` - Create a MPI vector type.
- `int MPI_Type_commit(MPI_Datatype * datatype)` - Commit the MPI type to be ready to use on all ranks.
- `int MPI_Type_free(MPI_Datatype *datatype)` - Release the MPI types 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 task ID to use to skip the communication: `MPI_PROC_NULL`