NTNU

Norwegian University of
Science and Technology

DEPARTMENT OF COMPUTER SCIENCE

TDT4200 - PARALLEL PROGRAMMING

Exercise 3

1 Introduction

This is the **graded** MPI exercise and can potentially give you 15 points towards your final grade. In this assignment, we will add a dimension to exercise 1 and work with a numerical solution for the 2D wave equation given by

$$\frac{\partial^2 u}{\partial t^2} = c^2 \cdot \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right), \tag{1}$$

where

- u is the wave's amplitude
- t is time
- x and y is space
- c is the wave's propagation rate (celerity)

visualised in Figure 1.

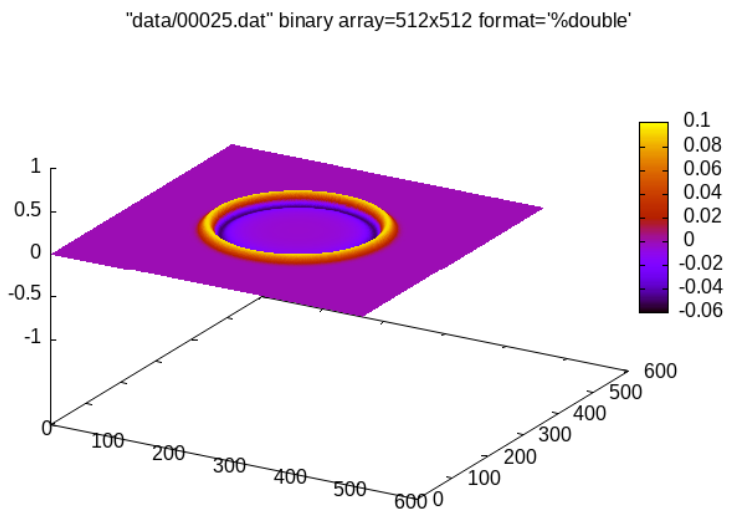

Figure 1: A visualisation of the 2D wave equation.

Equation 1 can be discretised in space with Finite Differences as follows

$$f''(x,y) \approx \frac{f(x-h,y) + f(x+h,y) + f(x,y-h) + f(x,y+h) - 4 \cdot f(x,y)}{h^2},$$

where the step size $h = \Delta x = \Delta y$. As in Exercises 1 and 2, the time domain will be approximated with the following central difference approximation

$$f''(t) \approx \frac{f(t-\Delta t) + f(t+\Delta t) - 2 \cdot f(t)}{\Delta t^2}.$$

2 Tasks

2.1 Programming

This exercise contains some new elements and concepts:

- An added dimension to the wave equation.
- User input for the simulation. This means that iterations, snapshots, and grid sizes can now vary. Grid size is also split into M (rows) and N (columns), meaning the grid isn't necessarily square.
- MPI I/O for writing the `.dat` files.
- Using a Cartesian communicator on the domain.

The ranks in MPI can be further split into sub-communicators, as shown in Figure 2. However, every rank will be part of the `MPI_COMM_WORLD`, default communicator.

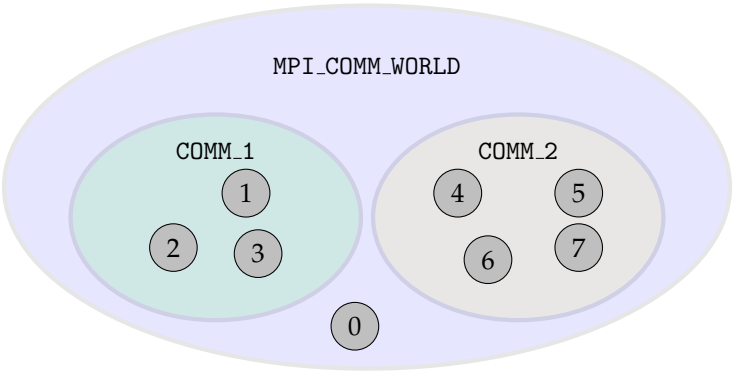

Figure 2: A model of an MPI universe split into two different communicators with rank 0 only being part of the default communicator.

A `Cartesian communicator` is a communicator where processes are organised in a grid, exemplified in Figure 3.

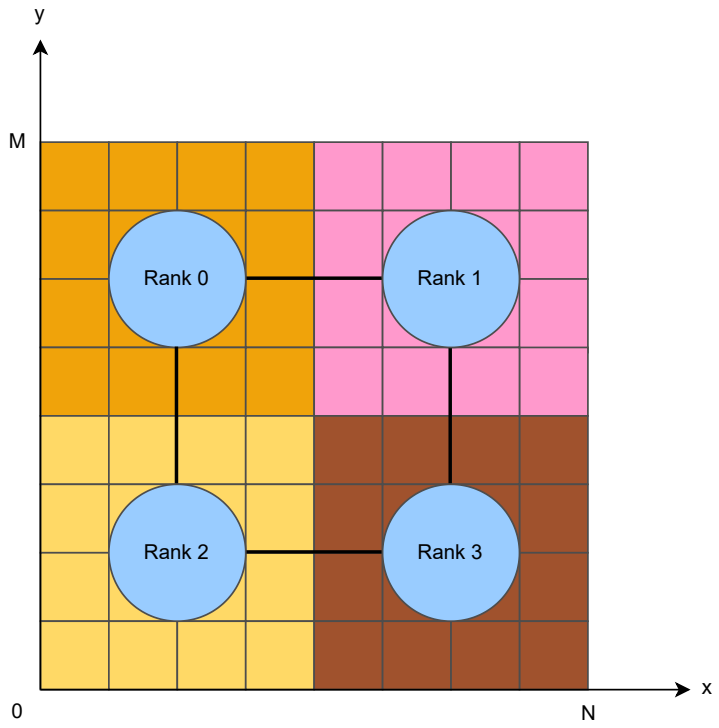

Figure 3: Cartesian communicator applied to a 2D grid. Note that both M and N represent space.

Cases you will not need to account for during coding:

- Having more processes than the size of the domain.
- You can always assume that the M and N will be divisible by the number of processes run. This means every process will get the same workload on the grid.

Note that the macros used for indexing the grid use `i` and `j`, representing rows and columns, not x and y.

You can run the following command to test your code with different input

```
mpiexec -n 4 ./parallel -m [rows] -n [columns] -i [max_iteration] -s [snapshot_frequency]
```

Listing 1: Run you code with different input

If you use non-default arguments for `m` and `n`, you need to use the following command to plot the images

```
./plot_image.sh -m [rows] -n [columns]
```

Listing 2: Create images with non-default grid size.

before running the

```
make movie
```

Listing 3: Combine images into a .mp4 file.

command. The default grid size is $M = N = 512$.

Hint Creating MPI datatypes might be useful in this exercise.

Before you start the exercise, it is recommended to run the `make check` command from the `Makefile` that will run the `compare.sh` script that compares the sequential and parallel solution. Running it on the handout files should produce a similar result as shown in Figure 4. If this is not the case, try to implement Task T1. If there are still differences, contact the staff.

```
find: 'data': No such file or directory
find: 'data': No such file or directory
mkdir -p data images
gcc wave_2d_sequential.c argument_utils.c -std=c99 -O2 -Wall -Wextra -o sequential -lm
mpicc wave_2d_parallel.c argument_utils.c -std=c99 -O2 -Wall -Wextra -o parallel -lm
wave_2d_parallel.c: In function 'main':
wave_2d_parallel.c:217:29: warning: unused variable 't_end' [-Wunused-variable]
  217 |     struct timeval t_start, t_end;
      |                             ^~~~
wave_2d_parallel.c:217:20: warning: unused variable 't_start' [-Wunused-variable]
  217 |     struct timeval t_start, t_end;
      |                    ^~~~~~~
mkdir -p data_sequential
./sequential
Total elapsed time: 1.183318 seconds
cp -rf ./data/* ./data_sequential
mpiexec -n 1 --oversubscribe ./parallel
./compare.sh


The sequential and parallel version produced mathcing output!


mpiexec -n 4 --oversubscribe ./parallel
./compare.sh


The sequential and parallel version produced mathcing output!


rm ./data_sequential/*
./sequential -m 2048 -n 512
Total elapsed time: 5.568541 seconds
cp -rf ./data/* ./data_sequential
mpiexec -n 16 --oversubscribe ./parallel -m 2048 -n 512
./compare.sh


The sequential and parallel version produced mathcing output!


rm -rf data_sequential
```

Figure 4: Running `make check` on a fresh project. The tests succeed as we haven't implemented any parallelisation yet, and even though we launch multiple processes, both versions are serial.

1. We need to include MPI and do the basic preparations to get started.

   a Include the MPI header.

   b Declare variables that each MPI process will need. They can include, but are not limited to:

   - `world_size`
   - `world_rank`

   It would be nice to revisit this part later.

   c Initialise MPI and set the communicator size and communicator rank for the process for `MPI_COMM_WORLD`.

   d Finalise MPI.

2. **Time your code:**

   We want to track our speed-up and need to be able to time our code. Here, you can look for inspiration in the sequential code. Does every process need to time your code?

   > Implement functionality to time the code, specifically the `simulate` function.

3. **Prepare the Processes:**

   To prepare the processes, we must share the user's potential arguments between processes and set up the Cartesian communicator.

   > - Have only the root process parse the arguments the user gives.
   > - Distribute the arguments to the other processes.
   > - Set up a Cartesian communication.

4. **Initialise the domain:**

   As with the previous exercises, we need to initialise the domain. This includes dividing the domain between processes, setting the time step, and applying the initial condition. For the initial condition, we will use a Gaussian pulse at the centre of the grid described by

$$u(x, y, 0) = e^{-4\delta^2}, \tag{2}$$

   where δ is the radial distance adjusted for an MxN grid. Figure 5 shows an example of the initial condition.

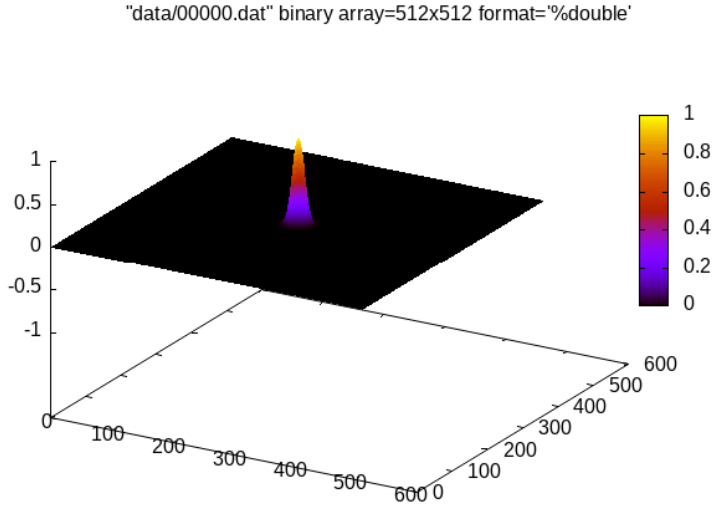

Figure 5: Initial condition of the 2D wave equation.

   As with the 1D wave equation, the time step in the wave equation is derived from the Courant-Friedrichs-Lewy (CFL) condition to ensure stability. For the 2D case, it is given by

$$\Delta t \leq \frac{\Delta x \Delta y}{c \cdot \sqrt{\Delta x^2 + \Delta y^2}}. \tag{3}$$

Change the function `domain_initialize`. The function should:

- Set up the three buffers for each MPI process.
- Apply the initial condition from Equation 2.
- Set the correct time step from Equation 3.

5. **Time Step**

To integrate over the time domain, we insert our approximation schemes into Equation 1 and rearrange to solve for the next time step

$$u_{ij}^{t+1} = -u_{ij}^{t-1} + 2u_{ij}^{t} + \frac{\Delta t^2 c^2}{h^2} \cdot (u_{i-1,j}^{t} + u_{i+1,j}^{t} + u_{i,j-1}^{t} + u_{i,j+1}^{t} - 4u_{i,j}^{t}) \tag{4}$$

Figure 6 visualises Equation 4.

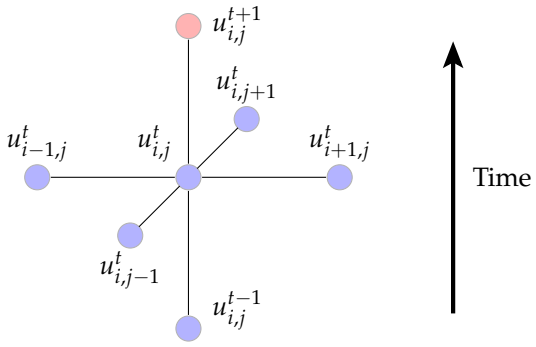

Figure 6: To calculate the red cell, $u_{i,j}^{t+1}$, we need 5 points at the current time step and 1 point at the previous time step.

Currently, the time step function in i covers the whole domain, MxN, but every process doing this will nullify the value of running multiple processes.

Change the function `time_step` so that each process only iterates over its sub-grid.

6. **Border Exchange**

If we project Figure 6 down to the 2D plane and look from the top, we get something similar to Figure 7.

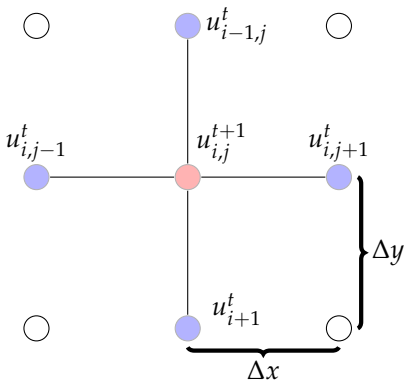

Figure 7: Dependencies in space. White cells are unused in the calculation of the red cell.

As in the last exercise, there is a dependency between the processes where a process needs to use values from the other processes' sub-grids to perform time step computations in their sub-grid, as shown in Figure 8. We, therefore, need to communicate the borders between processes.

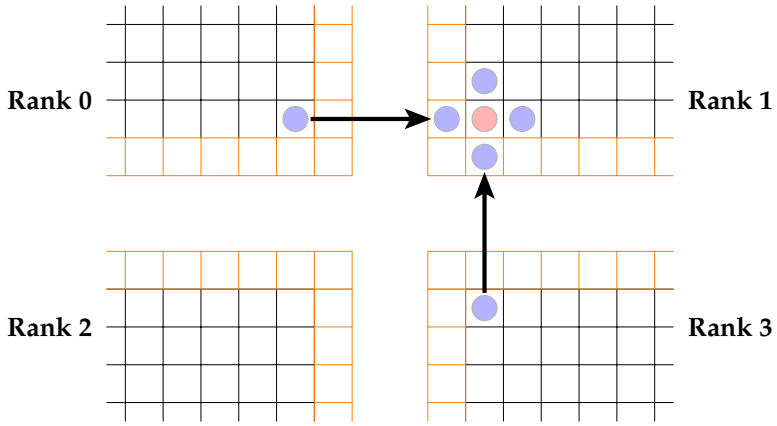

Figure 8: A process might depend on multiple other subgrids to calculate a cell in the next time step. To calculate the red cell at time $t + 1$, Rank 1 needs one value from Rank 0 and 3. Orange represents the ghost cells.

Implement the function `border_exchange` such that the ghost cells for each MPI process receive the data needed to calculate the next time step.

7. Boundary Condition

Which processes are close to the boundary and need to run the boundary condition? We will use Neumann boundaries, as with the 1D case, illustrated in Figure 9.

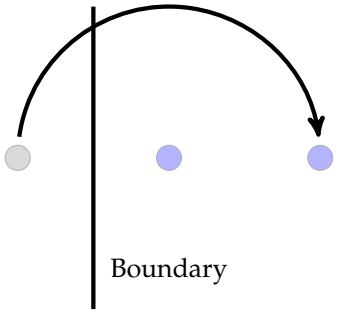

Figure 9: The ghost point outside the domain boundary (grey) copies the value of the cell one step inside the boundary as a reflection.

> Change the function `boundary_condition` such that only the processes on the boundaries deal with the boundary condition.

8. **Store the Domain**

In Exercise 2, you sent the complete domain to the root rank, which assembled it and then wrote a `.dat` file. Now, we will utilise MPI I/O to parallelise this as well. You will need to **open** the file, **write** the data, and then **close** the file.

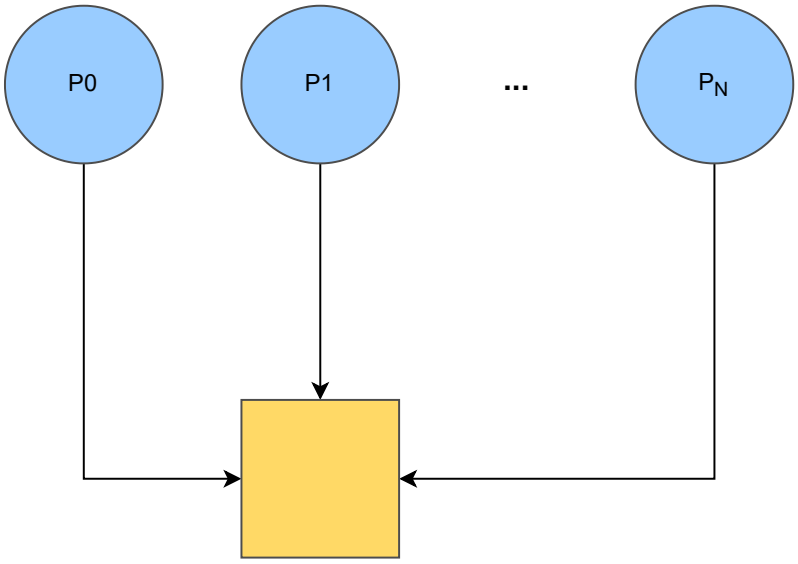

Figure 10: Multiple MPI processes writing to the same file.

> Change the function `domain_save` so that it uses MPI I/O to store the state of the domain.

2.2 Theory Questions

1. Discuss how using `MPI_Allgather` versus `MPI_Gather` will impact performance of a program.

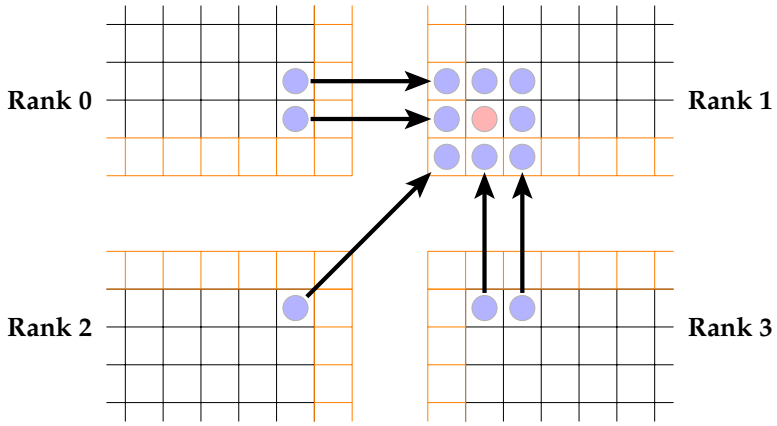

Figure 11: Rank 1 depends on multiple other processes to calculate the red cell for the next time step. Orange represents the ghost cells.

2. Let's say we wanted to use a 9-point stencil for approximation on the current time step instead of just 5, as illustrated in Figure 11. How would you communicate the value needed from Rank 2?

   Hint There are multiple solutions to this question.

3. Try to run the code with $M = 2048$ and $N = 512$. What changes, and why does this happen? (The sequential code will produce the same behaviour if you haven't fully implemented the parallel version.)

4. What is the difference between *weak scaling* and *strong scaling?*

3 Deliverables

Please deliver `.zip` file with the following contents containing your changes to the code:

- `wave_2d_parallel.c`
- A document with the answers to the theory questions.