NTNU

Norwegian University of
Science and Technology

Department of Computer Science

TDT4200 - Parallel Programming

Exercise 2

1 Introduction

This exercise will serve as an introduction to MPI and some basic concepts. We will be using the 1D wave equation from Exercise 1, given by

$$\frac{\partial^2 u}{\partial t^2} = c^2 \cdot \frac{\partial^2 u}{\partial x^2},$$

where

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

Recall that the 1D wave equation can be visualised as a straight line in space, which we can discretise as in Figure 1.

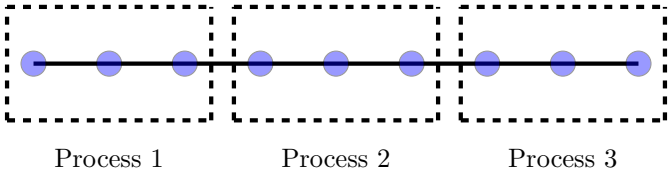

Figure 1: Example of a 1D wave discretisation where the domain is divided between 3 processes.

We would need to communicate the borders between `Process 1` and `Process 2`, and `Process 2` and `Process 3` in this example. Every sub-process, therefore, needs to allocate space for two ghost cells to account for the neighbouring value, the same as the boundary condition in Exercise 1, as shown in Figure 2.

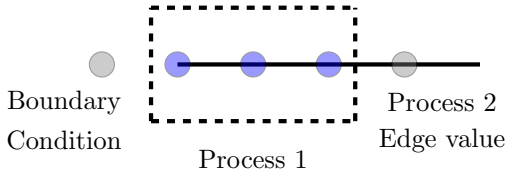

Figure 2: The domain of Process 1 with two ghost cells.

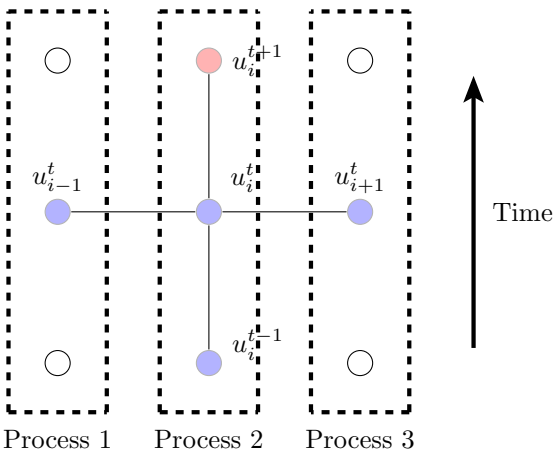

Figure 3: 3 processes working on the domain. Process 2 needs the value from Process 1 and Process 3 to calculate the next point.

Notice in Figure 3 that only the space points at the current time iteration need to be communicated.

2 Tasks

2.1 Programming

Cases you will not need to account for during coding:

- Having more processes than the size of the domain.

Besides this scenario, you should be able to produce the correct output for any number of processes. To get the base code going, starting with multiples of 2 before going more advanced is recommended.

In the `Makefile`, you will see a new command `make check` that checks whether your solution produces the same output as the sequential version by running the `compare.sh` script. You should test this command to make sure everything is installed correctly. It should produce an output similar to that in Figure 4.

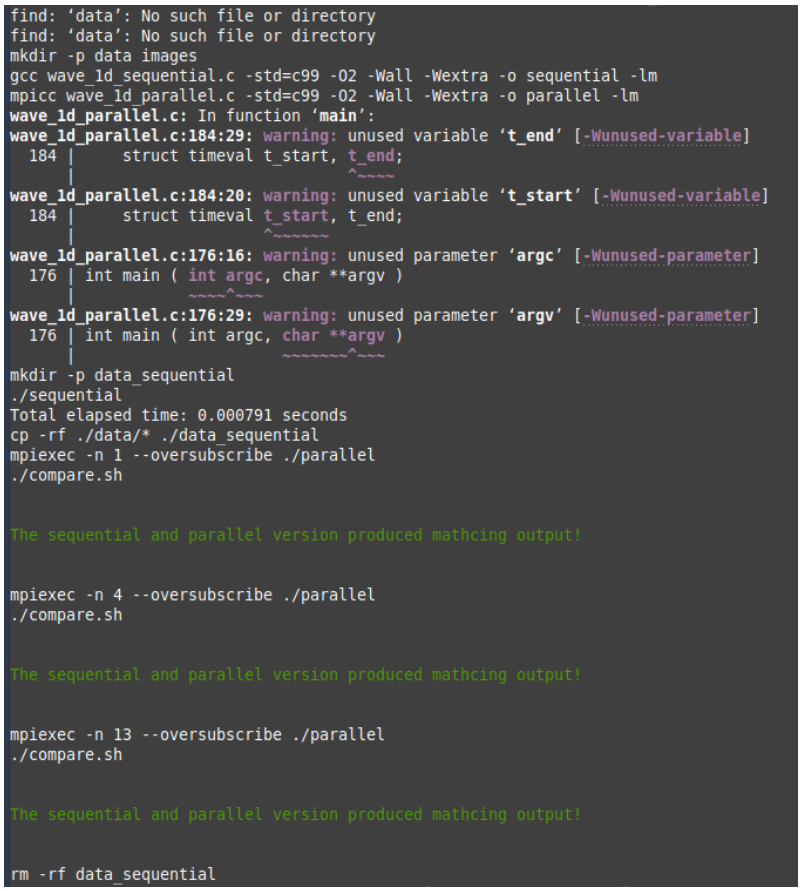

Figure 4: Running `make check` on a fresh project. The tests succeed as we haven't implemented any parallelisation yet, and even though we launch multiple processes, both versions are serial.

1. **Initialise MPI:**

   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. **Initialise the domain:**

   The first thing we need to do is to initialise the domain and make it ready for simulation. As in Exercise 1, we apply

   $$u(x, 0) = \cos(\pi x) \tag{1}$$

   as the system's initial condition, where x is a point in space, note that every process does not need to allocate space for the whole domain, as it will only perform calculations on parts of it. The root rank, however, will need the answer from all the other ranks later and must be able to store the entire domain.

   > Change the function `domain_initialize`. The function should:
   >
   > - Allocate memory for three sets of spatial domains: the previous, the current, and the next domain.
   > - Divide the total domain, N, between the processes.
   > - Apply the initial condition from Equation 1.

4. **Time Step**

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

   > Make the `time_step` function only iterate through the number of points you have allocated the process.

5. **Border Exchange**

   As shown in the Introduction, we need to correctly communicate the borders between processes to calculate the next time step.

   > Implement the function `border_exchange` such that the ghost cells allocated in task **T3** for each MPI process receive the data needed to calculate the next time step.

6. **Boundary Condition**

   Which processes are close to the boundary and need to run the boundary condition?

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

7. **Send the Data to the Root Process**

   The root rank should have allocated memory for the entire domain so that it eventually can be the only process doing I/O and writing a file.

   > Implement the function `send_data_to_root`. The function should communicate the results of the other processes to the root rank and assemble it in the root rank buffer

8. **Save the Domain**

   Currently, every process is writing a file to the `data` folder. This creates a race condition between ranks, but only the root rank has the complete picture.

   > Change the function `domain_save` such that only the root rank writes a file to the `data` folder.

2.2 Theory questions

1. What speed-up/expected speed-up did you get by parallelising the code?
2. What is the name for this type of parallelisation (splitting the domain between processes and using ghost cells to communicate)?
3. What is the difference between point-to-point communication and collective communication?
4. Given the following code:

```
1 int* a, b;
```

   Which type is `b`?

3 Deliverables

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

- `wave_1d_parallel.c`
- A document with the answers to the theory questions.