CS633 - Assignment 1

Group 42
Divyansh, Rajeev Kumar & Pranjal Singh
210355, 210815, 210744

1 Program Overview

The explanation of the code in `src.c` is as follows:

1.1 Includes and Global Variables

The code includes necessary header files and defines global variables to store the matrix dimensions (`cols`, `rows`), process decomposition along the x and y axes (`Px`, `Py`), halo region width (`width`), process rank (`myrank`), and matrices for data storage (`data`, `temp`). It also defines boolean variables to keep track of neighboring processes (`has_left_neighbour`, `has_right_neighbour`, `has_top_neighbour`, `has_bottom_neighbour`) and pointers for sending/receiving data to/from neighbors.

1.2 Helper Functions

- `fill_has_neighbours()`: This function determines whether the current process has neighboring processes along the left, right, top, and bottom directions based on its rank and the process decomposition.
- `communicate()`: This function handles the communication between neighboring processes. It packs data from the boundary regions of the local matrix into send buffers (`to_left`, `to_right`, `to_top`, `to_bottom`) and exchanges these buffers with neighboring processes using non-blocking `MPI_Send` and `MPI_Recv` operations. The communication pattern follows a specific order to avoid deadlocks.
- `swap()`: A simple function to swap the pointers of `data` and `temp` matrices.
- `get_val()`: This function retrieves the value of a matrix element at the specified indices (`i`, `j`). If the indices are out of bounds, it fetches the value from the corresponding receive buffer (`from_left`, `from_right`, `from_top`, `from_bottom`) if the neighboring process exists, or returns 0 otherwise.
- `compute()`: This function computes the new value of a matrix element based on its neighbors within the stencil radius (`width`). It uses the

`get_val()` function to retrieve the values of neighboring elements and performs the stencil computation.

1.3 Main Function

- The code reads command-line arguments for the number of processes along the x-axis (`Px`), matrix size (`N`), number of time steps (`num_time_steps`), seed for random initialization (`seed`), and stencil width (`stencil`).
- It initializes MPI and determines the process rank (`myrank`) and total number of processes (`P`).
- The matrix dimensions (`rows`, `cols`) are calculated based on `N`, and the number of processes along the y-axis (`Py`) is determined from `P` and `Px`.
- Memory is allocated for the matrices (`data`, `temp`) and communication buffers (`from_left`, `to_left`, `from_right`, `to_right`, `from_top`, `to_top`, `from_bottom`, `to_bottom`).
- The matrix `data` is initialized with random values using the provided `seed`.
- The main computation loop iterates over the specified number of time steps (`num_time_steps`). In each iteration, the `communicate()` function is called to exchange halo regions with neighboring processes, followed by the `compute()` function to update the matrix elements based on the stencil computation. The `swap()` function is used to swap the `data` and `temp` matrices, ensuring that the newly computed values are stored in `data`.
- After the computation, the maximum time taken across all processes is calculated using `MPI_Reduce` and printed by the root process (rank 0) along with the matrix size (`N`) and stencil width (`stencil`).
- Finally, the MPI environment is finalized with `MPI_Finalize()`.

2 Timing plot

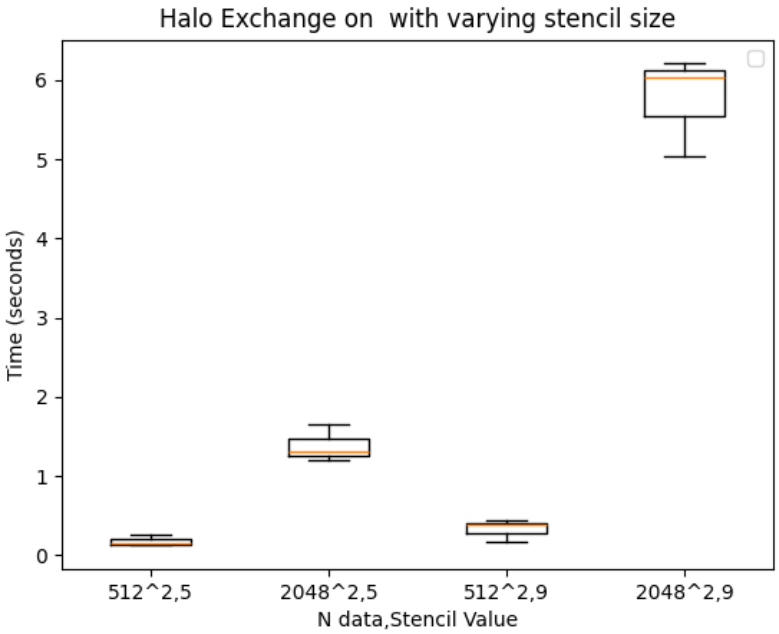

Figure 1: Timing Plot

3 Observations

Our observations from the plot[1] is as follows:

1. In case of smaller data size, we didn't find any clear outcome since few runs have `stencil=5` as the faster method while others have `stencil=9` as the faster one (although we expect `stencil=5` to be faster). This can also be because of the fact that the data size is small and it is getting affected by OS noise or similar things.

2. For the bigger data size, `stencil = 9` is taking more time which is expected since we are communicating twice the amount of data as compared to `stencil=5` and also there are more arithmetic operations.

4 Optimizations

We have performed the following optimizaitons:

1. While communicating the `width` times columns/rows we are first communication from the even ranks to odd ranks and then back from odd ranks to even ranks. This ensures the correct pairing of sends and receives as well as this doesn't make the communication sequential which would have been the case if in a row all the process where sending to their right and then receiving from left process.

2. This one is memory optimization, which receiving the packed data from any process we are receiving in such a manner that we don't need to unpack the data, rather we are accessing the data points from the buffer itself.