Indian Institute of Technology Kanpur

CS633

PARALLEL COMPUTING

Assignment 2

Author:
Group 36
Kalika(210482)
[kalika21@iitk.ac.in]
Monil Lodha(210630)
[monill21@iitk.ac.in]
Akshat Rajani(210812)
[rajanias21@iitk.ac.in]

April 15, 2024

1 CODE

1.1 Inititalization

We start by initiializing buffers which we will use in send/receive, pack/unpack, gather and scatter operations. We also initialize the MPI environment and get the rank and size of the process.
We take in the arguments given in the command line and store them in variables `Px`, `D` which is $\sqrt{N}$, `num_time_steps` and `seed`. We calculate `Py` by dividing number of processes `P`, by `Px`.
We defined an integer variable `x`, which has value 2.
The matrices we defined:

1. `arr` : The main array in which we store randomly initialized values usind seed given. It's size is `D` $\times$ `D`.

2. `new_arr` : The array in which we store the updated values of `arr` after each time step.It's size is `D` $\times$ `D`.

3. `sbuf` : It is the send buffer, which is used to send the values of the boundary elements to the neighbouring processes. It's size is `D` $\times$ `x`.

4. `rbuf` : It is the receive buffer, which is used to receive the values of the boundary elements from the neighbouring processes. It's size is `D` $\times$ `x`.

5. `rarr_top` : For computing the top row of a process, we require elements from the process directly above it, specifically the bottom row(s) of the process above. Thus, the `rarr_top` matrix is utilized to store these values. It comprises two rows. Its dimensions are `x` $\times$ `D`.

6. `rarr_bot` : For computing the bottom row of a process, we require elements from the process directly below it, specifically the top row(s) of the process below. Thus, the `rarr_bot` matrix is utilized to store these values. It comprises two rows. Its dimensions are `x` $\times$ `D`.

7. `rarr_left` : When computing the element the left column of a process, we require elements from the process immediately to its left. Specifically, we need the right column(s) of the process to the left of it. Therefore, this matrix is used to store those values. It consists of two columns. Its size is `D` $\times$ `x`.

8. `rarr_right` : When computing the element the right column of a process, we require elements from the process immediately to its right. Specifically, we need the left column(s) of the process to the right of it. Therefore, this matrix is used to store those values. It consists of two columns.Its size is `D` $\times$ `x`.

9. `gather_array_top` : It is used to gather all the top rows of the processes into the root of each sub communicator which is the row. It's size is (`P_x` $\times$ `x`) $\times$ `D`.

10. `gather_array_bot` : It is used to gather all the bottom rows of the processes into the root of each sub communicator which is the row. It's size is (`P_x` $\times$ `x`) $\times$ `D`.

11. `scatter_array_top` : It is used to scatter the top rows of the processes from the root to each process in sub communicator which is the row. It's size is (`P_x` $\times$ `x`) $\times$ `D`.

12. `scatter_array_bot` : It is used to scatter the bottom rows of the processes from the root to each process in sub communicator which is the row. It's size is (`P_x` $\times$ `x`) $\times$ `D`.

We perform Communication and Computation for each time step.

1.2 Communication

We have used 6 MPI calls for Communication:

1. `MPI_Send` : It is used to send the boundary elements of the process to the neighbouring processes.
2. `MPI_Recv` : It is used to receive the boundary elements of the neighbouring processes.
3. `MPI_Pack` : It is used to pack the boundary elements of the process to be sent to the neighbouring processes.
4. `MPI_Unpack` : It is used to unpack the boundary elements received from the neighbouring processes.
5. `MPI_Gather` : It is used to gather the top and bottom rows of the processes into the root of each sub communicator which is the row.
6. `MPI_Scatter` : It is used to scatter the top and bottom rows of the processes from the root to each process in sub communicator which is the row.

The communication is done in the following way:

1. The order of send/receive operations we have followed begins with row-wise communication followed by column-wise communication. Initially, processes with even row indices communicate with their immediate right neighbors, followed by processes with odd row indices performing the same operation.
2. Before sending the boundary elements, we pack them in the `sbuf` using `MPI_Pack` and then send them to the neighbouring processes.
3. For the top and bottom communication, we first gather the top and bottom rows of the processes to the root of the sub communicator in the arrays `gather_array_top` and `gather_array_bot` respectively. Then we communicate with the top and bottom neighbours only for the root node in each row. Afterwards, we scatter the top and bottom rows from `scatter_array_top` and `scatter_array_bot` respectively to the processes in the sub communicator storing the elements in `rarr_top` and `rarr_bot` respectively.
4. Similarly, we receive the boundary elements in `rbuf` and then unpack them in `rarr_left` and `rarr_right`.
5. As the name suggests `rarr_top` stores the rows needed for computation of the top row of the process, which is the bottom row(s) of the process just above it. Similarly for `rarr_bot`, `rarr_left` and `rarr_right`.

1.3 Computation

1. The outer loop iterates over the rows(`'i'`) of the matrix and the inner loop iterates over the columns(`'j'`).
2. We check for the boundary condition to determine whether `i` is equal to 0, `i` is equal to 1, `j` is equal to 0, `j` is equal to 1, `i` is equal to `D-1`, `i` is equal to `D-2`, `j` is equal to `D-1`, or `j` is equal to `D-2`. If any of the conditions is met, we initialize the variable `count` to 1 and assign the value of the current element to the variable `temp`. For each boundary condition that is not satisfied, we increment `count` by 1 and add the value of the neighboring element to `temp`.

3. We start by checking if the element is not at the bottom row, which is one of the conditions we check by comparing `i < D-1`. If this condition is met we add the bottom element to the `temp` variable and increment the `count` by 1. Subsequently, we get the following sub-conditions:

   - If `i` is not equal `D-2` indicating that the element is not in within the halo region, we add the value of the bottom element to `temp` and increment the `count` by 1.
   - Else if the element is in the halo region, we add the value from `rarr_bot` to the sum. We only add the top row of `rarr_bot` to the sum.

   Another condition is that the element is initially within the halo region, indicated by `i` being `D-1`, but `row` is not equal to `Py - 1`. In this case, we add the values of both rows from `rarr_bot` to the sum stored in `temp` and increase the `count` by 2.

4. We do the same for the top, left and right elements.

5. Next, we proceed with the calculations, determining the average by dividing the sum stored in `temp` by the `count`.

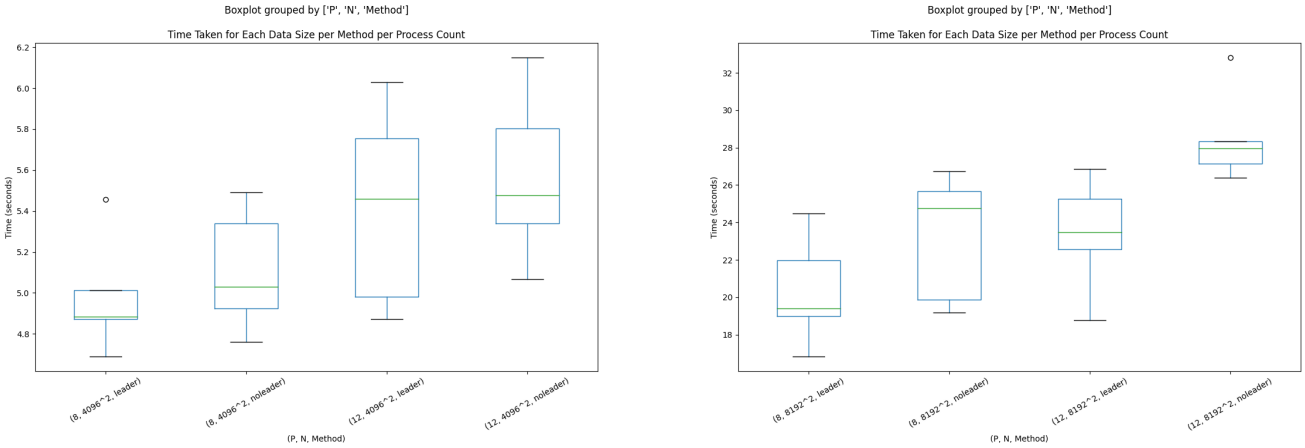
6. Afterwards, we store the calculated value in `new_arr` and proceed with the process for the next element.

7. If the boundary condition is not met, we directly determine the value of `new_arr` by computing average of the current element and elements surrounding it.

After completing these steps, our next action is to copy all the elements from `new_arr` to `arr`, and then we repeat the process for the next time step. This repetition is performed for a total of `num_time_steps` times. Additionally, we utilize `MPI_Barrier` to ensure that all processes have finished their computations before proceeding to the next time step.

Finally to print the final time stamp we use `MPI_Reduce` to get the maximum time taken by any process and print it only from the root process.

2 PLOT ANALYSIS



(a) D = 4096 (b) D = 8192

Figure 1: Time vs P,N,Method

Observations from the plots:

1. The time taken with the leader is less than the time taken without leader. This is because we have reduced the inter-row communications, earlier they were done by each subprocess in the row, now only the root process or the sub communicator leader does it. Therefore, increasing the intranode communication and decreasing the internode.

2. When the data size is increased, the time taken increases.

3 OPTIMIZATION

During the communication phase , we pack two rows together and send them in a single send operation. This strategy reduces the number of sends and receives, consequently decreasing the time taken for communication.

4 CONTRIBUTION

Kalika(210482), Monil Lodha(210630) and Akshat Rajani(210812) contributed equally to the assignment. We wrote the code and report together.