CS633: Assignment 1

Baldip Singh Bijlani(17807203), Paramveer Raol(170459)

February 23, 2021

1 Code Explanation

We decided to use non-blocking send and receives as the computation of the inner grid elements could be done while the communication amongst the process were going on. Briefly our algorithm/pseudo_code for the first part is as follow:

```
for time = 1 to TimeSteps(50 for our executions)
    if(left send required)
        for i=(1 to root(N))
            Isend(old_grid([i,0] element), dest = left_process,
            tag = left_process)
            Irecv(left_buff[i], src = left_process, tag = myrank)

    if(right send required)
        for i=(1 to root(N))
            Isend(old_grid([i,root(N)-1] element), dest = right_process,
            tag = right_process)
            Irecv(right_buff[i], src = left_process, tag = myrank)

    if(top send required)
        Isend(old_grid(root(N) elements of starting from[0,0]), dest = top_process,
        tag = top_process)
        Irecv(top_buff(N elements), src = top_process, tag = myrank)

    if(bottom send required)
        Isend(old_grid(N elements of starting from[root(N)-1,0]), dest = bottom_process,
        tag = bottom_process)
        Irecv(bottom_buff(N elements), src = bottom_process, tag = myrank)

    compute inner elements of the old_grid and store them in new_grid

    if(left send required)
        waitall for all left sends
        waitall for all left receives
```

```
if(right send required)
    waitall for all right sends
    waitall for all right receives
if(top send required)
    wait for  top send
    wait for  top receive
if(bottom send required)
    wait for  bottom send
    wait for  bottom receive

compute edge elements(old_grid, new_grid, left_buff, right_buff, top_buff, bottom_bu

swap(old_grid, new_grid)
```

Some important aspects of the code that should be noted are:

1. The tag being used in send is always the rank of the destination. This works and is not buggy because the receives from other processes can be distinguished by their source and the receives from the same source always arrive in order(the non-overtaking guarantee of MPI messages), and hence there is no ambiguity even if the tag in the MPI_Received call is always the same in every process.

2. The top_buff, left_buff, right_buff, bottom_buff are 1-d arrays of size root(N).

3. Each process has a myrow(= myrank/root(P)) and mycol(=myrank%root(P)) the left_process, right_process, top_process are w.r.t these myrow and mycol.

4. the grid is used over here is actually a 1-d grid of size(N) and the (i,j) element of the grid actually refers to the (i*N + j) element of the grid. 2-d version of the grid was implemented but then in the third part to effectively use vectors we had to convert it to 1-d and hence all methods now use this 1-d grid.

5. The "compute edge element" function does take all the buffers as input but based on myrank it uses only ones that are required for computing the edge elements.

6. Rather than copying the new grid to the old grid we just swap their pointers for efficiency.

1.1 Pack method

The changes to the normal algorithm for the pack method:

1. The first 2 left and right send's for a loop now is used to pack the elements into a single buffer(of size root(N) doubles) and then Isend and Irecv are written at the end of the for loop for the entire buffer.

2. The buffer is different in every direction.

3. The top and bottom sends, pack is done using for loop as all the elements are in the same row and then the Isend and Irecv are called.

4. The inner computations remain the same.

5. Instead of "waitall" for the left and right only "wait" is used as there would be one send and one receive, and n "unpack" statement is introduced for all the directions(left, ...).

6. Then the compute edges function is called and it remains the same as before.

1.2 Vector and Contagious method

Changes made for the vector and contagious method:

1. 2 new data-types with the following features were created

   - horizontal_type: using contagious function with the specification of N-doubles
   - vertical_type: using vector function with specifications count=N, blocklen=1, stride=N

2. Horizontal data type was used to send top and bottom, and vertical data type for left and right

3. Just one receive was required for all the parts with the same buffers as mentioned in the pseudo-code.

4. The inner computation part remained the same.

5. The "waitall" was reduced to "wait" and just 1 send and receive was there.

6. The computation of outer edges remained.

2 Observation

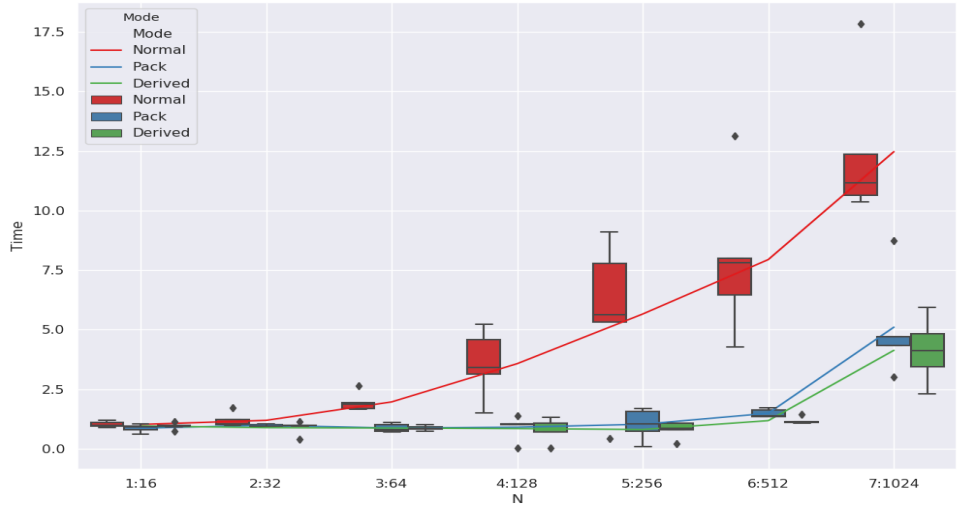

Figure 1: P=16

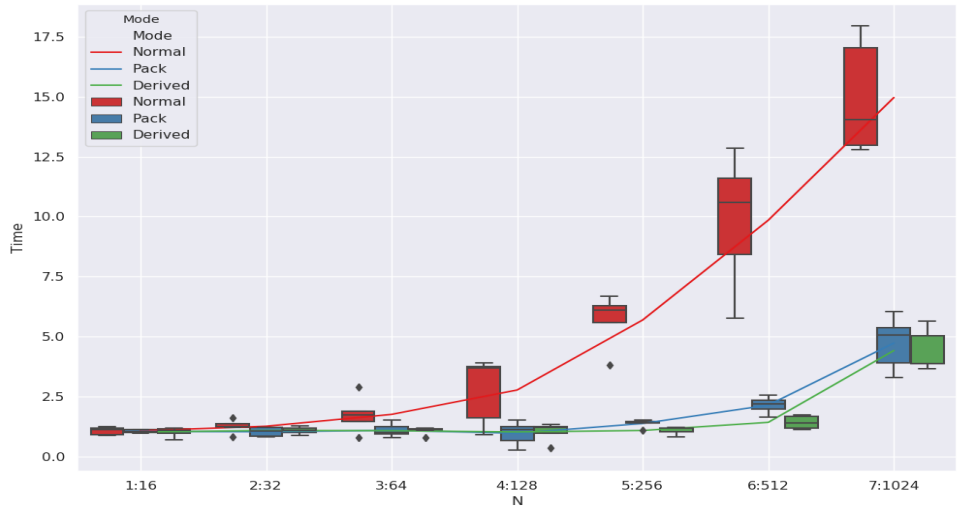

Figure 2: P=36

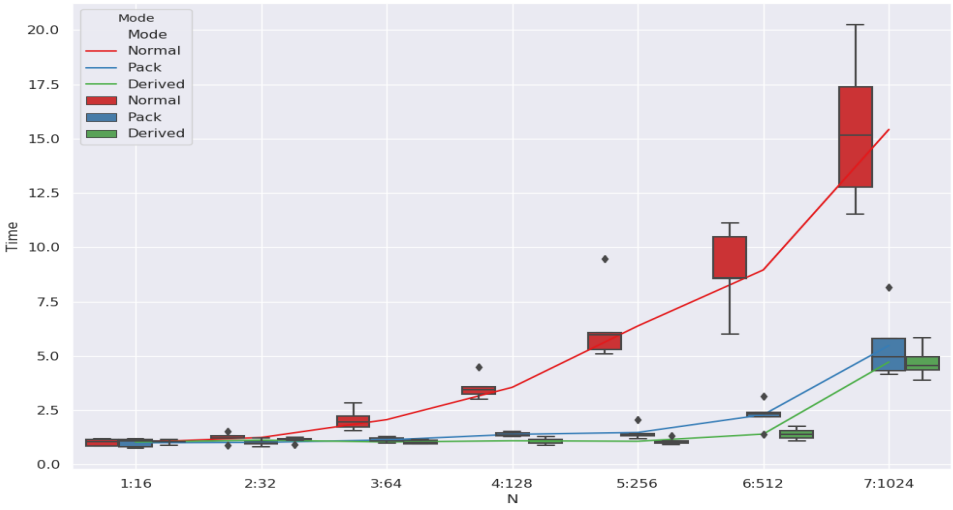

Figure 3: P=49

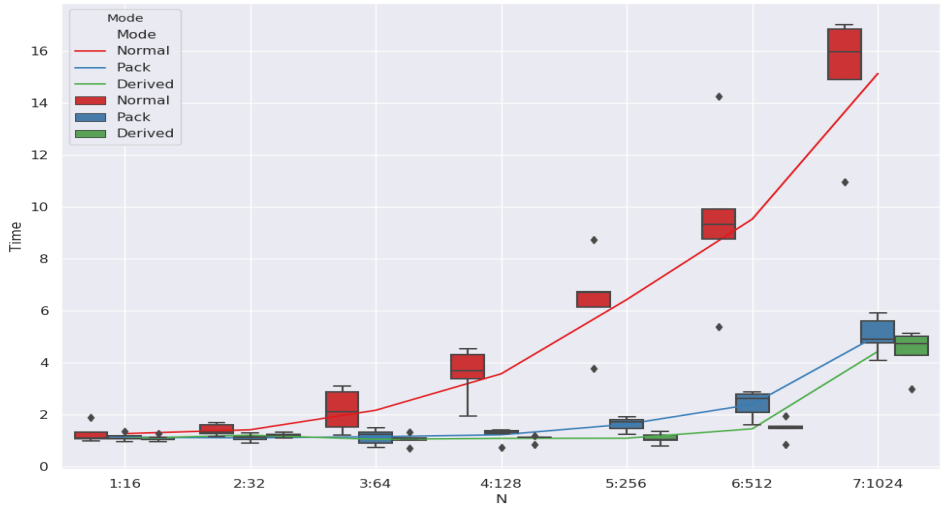

Figure 4: P=64

Some of the observations that can be drawn from the results obtained are as follows:

1. As the size of the grid increases there is an increase in either the number of sends, receives(normal method), or the size of data it sends, receives(normal, pack, and vector methods) increases which is the cause of the increase in time observed with the increasing grid size.

2. The time for the normal mode is much greater than that for pack or the derived mode. This can be used to conclude that more number of sends leads to more time even if the the total amount of data being sent is the same, as in the normal mode the number of sends is much higher even though the data being sent in all the three cases is same. This leads us to believe that it is more economical to send more per send.

3. The marginal better performance of the vector, contiguous method compared to the pack method can be attributed to the overhead that occurs due to for loops required to pack the data and the overheads that occur due to unpacking of the data.

4. The performance for the same N across different P's is almost same, consider the time taken for P = 36, 49, 64 for N=1024(or at least there is not as high a dependence on P as compared to N). This can be attributed to the fact that the amount of data being send and received per process the same even if the total number of processes are different.

3 Issues faced

- Some time while running all the configurations a connection to the node was lost however the frequency of such an event was very less (in total it happened for 2 times).

- The part to check the correctness of the algorithm was quite difficult hence we created a framework for this which is in the "Assignment1_extra/correctness", the working details of it are mentioned in the next section.

- Apart from that there were some issues related to the mpi-installation , and NodeAllocator but they were appropriately resolved from the piazza queries.

4 Correctness Check Framework

In the directory "cs633-2020-21-2/Assignment1_extra/correctness" there is a file named run_and_check.py in that first of all the inputs are:

- numProcs: Number of processes.

- numCells: Size of the grid of individual process.

- timeSteps: Number of times stencil steps are performed for each cell

- mode: 1 for the normal method, 2 for the pack method, and 3 for the vector and contagious method.

The given python file first of all executes the entire method and simultaneously stores the (timestamp, row, col, value) entries for each process in a different file with the name as its rank in the all_data directory. Then this data is used to construct a timeGrid which basically has the format [timestamp, grid]. Then the gird at time=0 is given as input to

another program(halo_check) which runs on a single process and has (numProcs*numCells) as time=0 grid as the initial grid. Then this process also outputs (timestamp, row, col, value) into the file and then this new file's grid timeGrid2 is compared with the timeGrid to check the correctness