CS633: Assignment 3

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

April 21, 2021

1 Execution instructions

There are 2 files **src.c**(this contains the code of the methods implemented) and **filer&w.h**(this contains the for the file input-output operations that are performed to get the data and to write the data). One **make** a new executable **code** will be generated. To get the output ie. minimum temperature of every year ,global minimum temperature, time taken to execute the command **mpirun –np X ./code tdata.csv**.

NOTE: The output of the execution will be stored in **output.txt** it will not be displayed.

The run script is **run.py** the command to execute is **python3 run.py** it takes the data in **tdata.csv** and executes **code** executable on it. The cumulative data ie. the times and outputs of all the configurations(nodes = 1,2 and ppn = 1,2,4) will be stored in **data** file which is then used by **plot.py** to plot the box plots. The file name of the plot is **plot.jpg**.

NOTE: File **plot.py** is called by **run.py**. If called individually then make sure **data** file is there. And **please keep the "helper_script" folder as the files in it are used generate the hostfile**. Ensure that you have the following packages: **pandas, seaborn, numpy and matplotlib** .

2 Data Distribution Strategies

NOTE: We read the file in the following manner:

1. First get the number of rows and columns of the file.
2. Make array of appropriate size to store the entries in file.
3. Store the file (can be done in any format ie. row-major or column major).

We had tried 2 different data distribution strategies. Brief idea of each is as follows:

- First one is where the communication of contiguous rows to each process is done, and then computation of local minima of every column for the data received is done. Finally the local minimas are reduced at rank 0 process.

- The second strategy is where communication of contiguous columns is done to each process, and then the computation of the minimas of these columns is done. Finally the minimas of each set of columns are gathered at the rank 0 process

2.1 Row Wise Distribution of Data

The code for this approach can be found in the file Assignment3/methods_experimented.c in the function row_comm_normal. The pseudo-code for this strategy is as follows: The **row_data** is the data in row-major form.

```
--------------------------------------------------------------------------------
if(myrank == 0 )
    MPI_Bcast(dimensions, src = 0, MPI_COMM_WORLD); //broadcating the dimensions

//-------- PREPARING THE BUFFER TO BE RECIEVED
rows_2be_recved = num_rows / numProc;
recv_count = rows_2be_recved * num_cols
recv_buff = malloc(recv_count floats);

//-------- NUMBER OF  SPILL OVER ROWS WHICH WILL BE COMPUTED ON BY RANK 0
long rows_left = num_rows - rows_2be_recved*numProc;
long offset = rows_left*num_cols;

//------- THE SCATTERING OF DATA TO VARIOUS PROCESSES NON BLOCKING
MPI_Iscatter(row_data + offset,  recv_buff, root = 0, MPI_COMM_WORLD, req);

//------- PREPARING THE ANSWER BUFFERS WHERE COMPUTATIONS WILL BE DONE
reduce_buff = malloc((num_cols+1) floats);
for(i=0; i<= num_cols; ++i) reduce_buff[i] = FLT_MAX;

//------- SPILL OVER ROWS' COMPUTATION AT RANK 0
if(myrank == 0){
    for(long i=0; i<rows_left; ++i){
        for(long j=0; j<num_cols; ++j)
            reduce_buff[j] = min(reduce_buff[j] , row_data[get_pos(i,j,num_cols)]);
    }
}

//------- COMPUTATION ON THE BUFFERS RECIEVED FROM RANK 0
MPI_Wait(req);
for(long i=0; i<rows_2be_recved; ++i){
    for(long j=0; j<num_cols; ++j)
        reduce_buff[j] = min(reduce_buff[j], recv_buff[get_pos(i,j,num_cols)]);
}
//------- STORING THE MIN OF [0 TO NUM_COLS-1] IN INDEX NUM_COLS
```

```
reduce_buff[num_cols] = min_of_array(reduce_buff, 0, num_cols);

//------- FINAL REDUCTION THAT WILL GIVE THE ANSWER AT THE ROOT = 0
final_ans = malloc((num_cols+1) floats);
MPI_Reduce(reduce_buff, final_ans, num_cols+1, MPI_MIN, root = 0, MPI_COMM_WORLD);

--------------------------------------------------------------------------------
```

Some key points regarding the optimizations that we made in the code as compared to naive implementation, are as follows:

- The scatter over here is done in a non-blocking manner so that the computations at rank 0 and some buffer initialization at other processes can be done in non-blocking manner.
- The computations of min are done in a manner such that consecutive elements of the buffer are accessed while iterating this leads to fewer cache misses as compared to the case when the elements accessed are at **num_cols** apart. We did analysis for this and this indeed should be a process level optimization as the L1 cache is of size 32K and a non optimized traversal will lead to re-storing of row elements in cache if the number of rows are in the range of hundred thousands which is the Case given.

2.2 Column Wise Distribution of Data

The code for this approach can be found in the file Assignment3/src.c in the function col_comm_normal. The pseudo-code for this strategy is as follows: The **col_data** is the data in column-major form.

```
--------------------------------------------------------------------------------
if(myrank == 0 )
    MPI_Bcast(dimensions, src = 0, MPI_COMM_WORLD); //broadcating the dimensions

//-------- PREPARING THE BUFFER TO BE RECIEVED
cols_2be_recved = num_cols / numProc;
long recv_count =  cols_2be_recved * num_rows;
recv_buff = malloc(recv_count floats);

//-------- NUMBER OF  SPILL OVER COLUMNS WHICH WILL BE COMPUTED ON BY RANK 0
cols_left = num_cols - cols_2be_recved*numProc;
offset = cols_left*num_rows;

//-------- THE SCATTERING OF DATA TO VARIOUS PROCESSES NON BLOCKING
MPI_Iscatter(col_data + offset,  recv_buff, root = 0, MPI_COMM_WORLD, req);

//------- SPILL OVER COLUMNS' COMPUTATION AT RANK 0
float* final_ans;
```

```
if(myrank == 0){
    final_ans = malloc((num_cols+1) floats);
    for(long i=0; i < cols_left; ++i)
        final_ans[i] = min_of_array(col_data, i*num_rows, num_rows);
}

//------- COMPUTATION ON THE BUFFERS RECIEVED FROM RANK
proc_mins = malloc(cols_2be_recved FLOATS);
MPI_Wait(req_flag);
for(long i=0; i<cols_2be_recved; ++i){
    proc_mins[i] = min_of_array(recv_buff, i*num_rows, num_rows);
};

//------ GATHERING THE MINS THAT WERE COMPUTED BY OTHER PROCESSES AT RANK 0
MPI_Gather(proc_mins, final_ans+cols_left, root = 0, MPI_COMM_WORLD);
if(myrank == 0){
    //---COMPUTING THE GLOBSL MINIMA
    final_ans[num_cols] = min_of_array(final_ans, 0, num_cols);
}
--------------------------------------------------------------------------------
```

Some key points regarding the optimizations that we made in the code as compared to naive implementation, are as follows:

- The scatter over here is done in a non-blocking manner so that the extra-columns' computations at rank 0 can be done in non-blocking manner.
- The computations of min are done in a manner such that consecutive elements of the buffer are accessed while iterating this leads to fewer cache misses as compared to the case when the elements accessed are at **num_cols** apart. This is similar case as of the row-communication's implementation.

2.3 Other Topology Aware Methods

After observing and analyzing the results of the above two methods,(these analysis and observations are listed in the subsequent section) and finding which are the most time-consuming steps of the whole algorithm. We tried to optimize those specifically. This is mentioned in subsection 3.2, since we want to explain the observations before the reader reads that.

3 Observations

Note: Both the row and column methods have the same ccn(atleast for the scatter part) so the only point of difference will be because of the reduce and gather, or becuase of some non-blocking computations, for this we did some experiments mentioned below.

We monitored the average time taken(for 10 test) to execute the algorithm ie. the one that find the year-wise minima and the global minima for the following parameters:

- **Row Size:** 50K, 100K, 200K, 400K, 600K, 800K. As it was mentioned that number of rows will be couple of hundred.
- **Column Size:** 20, 40, 60, 80, 100. As it was mentioned that maximum number of columns will be 100.

The methods for which the time was monitored are as follows:

- **Normal method**: does the entire computation at just at rank=0 process.
- **Row method**: Algorithm mentioned in the 2.1 section.
- **Column method**: Algorithm mentioned in the 2.2 section.

The best-performing method for all the parameters sets(ie all row,col pairs) at various process configurations is as follows:

ppn ->	1	2	4
1 node	column (same for all methods)	column	column
2 nodes	normal	normal	normal

The key insights that we got from this are as follows:

- Row and Column method always perform better than the single process in case of a single node.
- Between Row and Column The column-method always performs better.
- On 2-nodes neither of the method performs better than the normal-method.

3.1 Computation Vs Communication Time

We investigated the reason for poor performance on 2 nodes as compared to on 1 node, and found that the computation on a single node takes far less time as compared to even a single communication between 2 nodes. For finding this we had written a code that computes global minima of a buffer of a given size and timed it. The same code transfers size/2 floats to the other node in the same group(if the group is not the same then the performance would be even worse) and timed this process. The results that we received are as follows:

size	compute time	simple send time
10000	0.0001285	0.0002901
100000	0.0007751	0.0019752
1000000	0.0070702	0.0173154
10000000	0.0705102	0.1710467
100000000	0.7047669	1.7090403

As at-least one send will be there in the scatter' binomial algorithm (present in both the methods) of data worth size/2 to the rank (numProc/2) present at another node hence for the data sizes we are interested in(10^7 to max 10^8) the communication time will always dominate over computation time in case of inter-node communication. However as the data size is increased a it is possible that the speed up will increase for the column and row-methods because the communication and computation times will start becoming comparable. The is evident from the figure shown below .

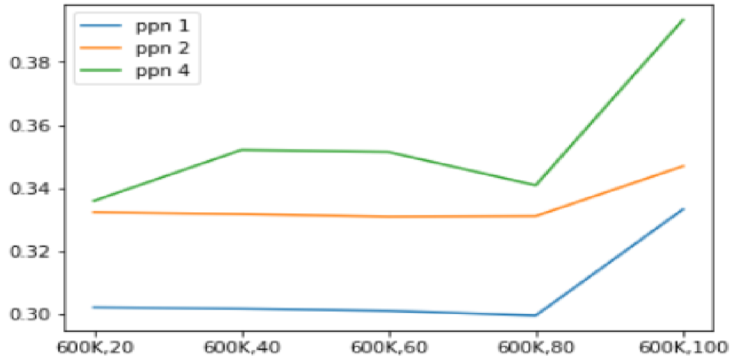

Figure 1: x-axis=(row, col), y-axis = speedup of col-method , nodes = 2

As you can see with data-size the speed-up increases for number of nodes = 2, but the speed up is still not more than 1 for the data sizes of interest.

After looking at the send-compute table we came up with another idea which is as follows:

3.2 Parallel sends method

The code for this approach can be found in the file Assignment3/methods_experimented.c in the function row_2_node. As stated previously, we found that the send to the other node was taking the majority of the time. And we try to optimize that time. To understand the method lets take the following example. The total amount of data is 80MB, the number of nodes are 2 and ppn is 4. The method runs in the following manner:

1. First of contiguous data of 20MB is sent from rank=0 to rank 1,2,3 using scatterv.
2. Then 10 MB data is sent in the following manner $0 \rightarrow 4$, $1 \rightarrow 5$, $2 \rightarrow 6$ and $3 \rightarrow 7$,
3. Then after performing the computations the entire computed data, is reduced at rank 0 (communications are row-wise).

The intuition behind this method was the the number of communications to the other node will happen in parallel, and with lower amount of data (as in scatter binomial(recursive halving and distance doublling) 40MB of data would be sent by 0 to 4 where as in this case 10MB of data is sent in parallel). However the performance worsened, this we figured out was because of the single link between the nodes(ie. the tree not fat tree topology of CSE network) there would be congestion between these 2 nodes, hence the performance deteriorated and this method was not given as the final submission.

3.3 Final Method used

Finally we decided to read the data in column-major form and apply the column-method. The results of time vs configuration on **tdata.csv** are as follows:

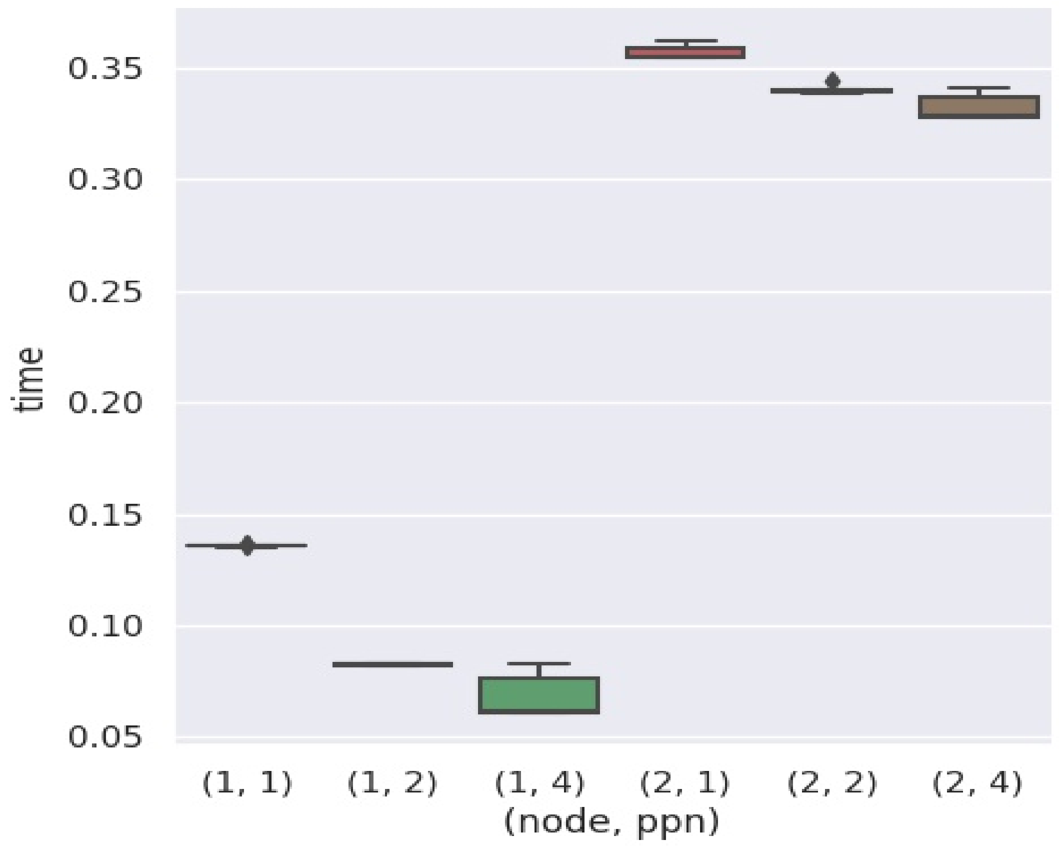

Figure 2: The box-plot is formed from 10 executions' time

As discussed above on one-node there is speed up as the ppn is increased, but on 2 nodes because of the inter node communication cost the time taken becomes too high. This will drop as the data size(ie. the computation time as mentioned in 3.1 section) increase, and there is a marginal improvement as ppn is increased.

4 Correctness Check

For testing purposes all the computations were done only on the **rank=0** without distributing the data, and then finally the output received after the distribution of data was matched with the rank 0's local answers. All the configurations and different data-sizes were extensively tested and no error was found. We also took the diff of our outputs, with a small python code for processing CSV's and finding the minimum of each column, to be more rigorous.

5 Problems Faced

We did not face any significant problems in this assignment from implementation's point of view.