Assignment 3

Mahi Agrawal (20111029) | Tamal Deep Maity (20111068)
maahi20@iitk.ac.in | tamalmaity20@iitk.ac.in

Contents

1. Running the Jobscript

To run the jobscript use the following command :-

```
python3 run.py
```

To run the task just for one configuration use the following command :-

```
make; mpiexec -np 8 ./src.x filename
```

where `X` is the number of nodes * number of cores and `filename` is the input file. For our case, it is `tdata.csv`

2. Code Explanation

- Initially the process with rank 0 reads the data from input file. It stores the data in array "`arr`". The rows in the input file are stored as columns and columns in the input file are stored as rows in "`arr`".

- Then the number of rows stored in the variable ROWS and number of columns stored in the variable COL are broadcasted to other processes.

- Rank 0 packs each ROW into 16384 (64KB) elements using `MPI_Type_Contiguous` and the new datatype is named as `float_type`. The number of data elements to be communicated together is named as `pack_size` in our code. For our case `pack_size` is 16384 as we chose to pack 64KB of data together but it can be changed to look for more efficient packing size.

- Each process gets $\frac{ROWS}{size}$number of rows where size is the number of processes. And from each row, the process designated for that row gets $\frac{COL}{pack\_size} * pack\_size$ number of columns. The rest of the columns and rows are with rank 0 only.

- The number of the rows may not be a multiple of number of processes, in that case process with rank 0 gets more rows as compared to other processes.

- Each process does the computation and then the data from each process is gathered at the process with rank 0 using `MPI_Gather()`.

- Then final computation is done at the rank 0 and output is written in the output.txt.
- We have used naive `MPI_Isend()` and `MPI_Irecv()` to send data from process with rank 0 and to receive data at other processes.
- We have used `MPI_Barrier()` just after rank 0 reads the data, so that all the processes enter the communication / computation parts simultaneously otherwise when rank 0 reads the data, other processes are idle and timer starts for them. Due to which the time we get will also include the idling time of ranks other than 0.
- We experimented with different pack_size and found out that 64KB takes the least time and hence we have used that in our code.

3. Data Distribution Strategies

- Initially we used a flow such that each process kept half of the data with itself and sends the rest of the data to the next process. In this, the total time taken was high as we realised most of the time, processes remained idle and the computation time was small so this code was not efficient.

- After this we decided that we will store columns as ROWS and rows as COLUMNS (COL), and then we packed the data using MPI datatype and transmitted to other processes. This code was efficient as compared to the decomposition mentioned in the above point.

- We stored the matrix as a transpose of the original matrix (where number of columns were the years) because it made for easier coding for sending all the data for a particular year to a process. Otherwise, we had to create a vector and send an entire column to a process which would have given us the same efficency w.r.t. time but would have added a few more lines of code.

- We decided to use a single `MPI_Scatter()` rather than using `MPI_Isend()` multiple times but then we realised that it wasn't possible to do according to the logic we had used. Because if the number of data points corresponding to a year wasn't a multiple of 16384 then those data points weren't sent to the process and their computation was taken care of by rank 0 only. As such data which was send to other process wouldn't necessarily be contiguous everytime which is a necessity while using `MPI_Scatter()` to send data.

- Then we thought of resolving this problem of ours by using a single `MPI_Scatterv()` as we can provide different displacement for each process, but here too, the data which was needed to be sent to each process needn't necessarily be contiguous going by the similar logic as explained above and hence this method couldn't be used.

- Then we realised that we can use `MPI_Scatterv()` in a for loop, such that in each iteration each process gets one row (all data points for a year) from the chunk of the rows it should get. But we observed that the pattern of time still remained same. So, we stuck with our original idea of doing multiple `MPI_Isend()`.

- We also tried to communicate data without any usage of `MPI_datatype` simply by doing `MPI_Scatter()` but we didn't find any improvement in time taken for communication for low number of processes. For higher number of processes, such a method could be useful.

We chose to decompose our data such that each process has more or less the same amount of computation to do. This was done because from our experiments with other types of decomposition such as when data to be sent to next process kept getting halved, the idling time of other processes was significant. Also, such a decomposition makes for good parallelism in code as all processes get their chunk of data at small intervals of time (because rank 0 can only send to one process at a time). And hence they finish their computation at certain time intervals too, depending on how early they got the data from rank 0, as the data size provided to them are the same. They again send back that data to rank 0 at certain time intervals.

Time at which processes get their data may not be nearly equal, it also depends upon the congestion and data size to be sent. The process on the same node as the rank 0 will take copying time from memory space of process 0 to its memory space. Whereas the process on the different nodes will take $\frac{i*numberofbytes}{bandwidth}$, where i denotes the number of process rank 0 sent data to, before it on different nodes. Plus, there can be some contention overhead and communication latency as well. If the distance between different nodes is less this time will be very less, so we can also perform node allocation such that the distance between nodes and hops between nodes is as few as possible. In our case, we are considering nodes from the same group and as the contention was also small, and as the problem size per process decreases with increase in number of processes, two consecutive processes receive data at nearly the same time.

The decomposition we chose also assigned anything that was non uniform to rank 0. These non uniformities were like when number of rows weren't equally divisible amongst all the ranks, some extra rows were assigned to rank 0 and all other processes were sent same amount of rows. Non uniformities were also found when the number of data points for a particular year wasn't an exact multiple of the data size we were contiguously communicating. As we already mentioned above that the communication time outweighs compute time, so leaving rank 0 to do these extra computations didn't affect the total time much.

4. Plots

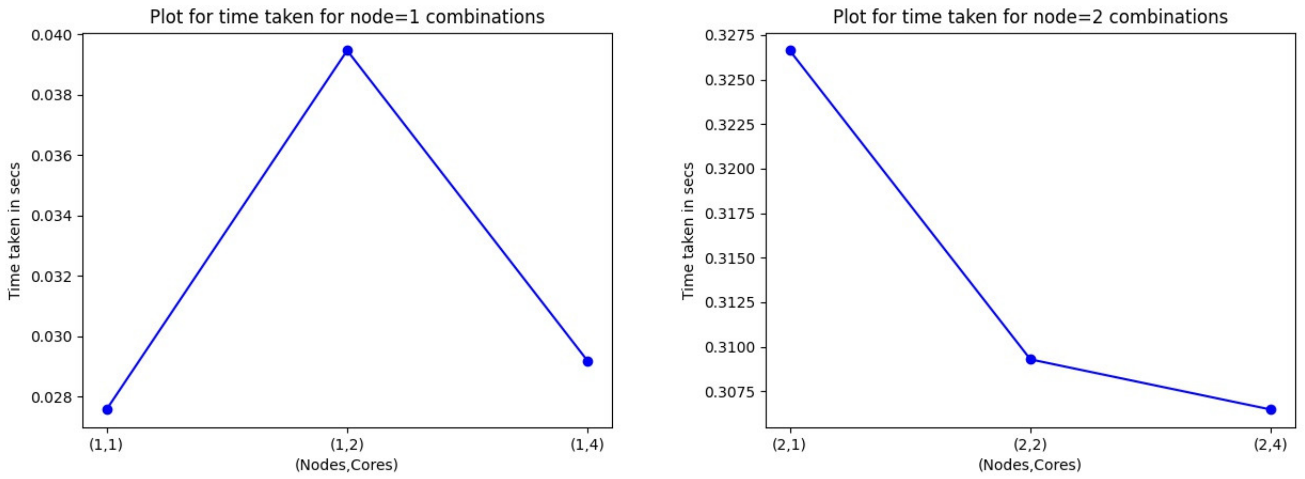

Figure 1. Separate plots per node combination

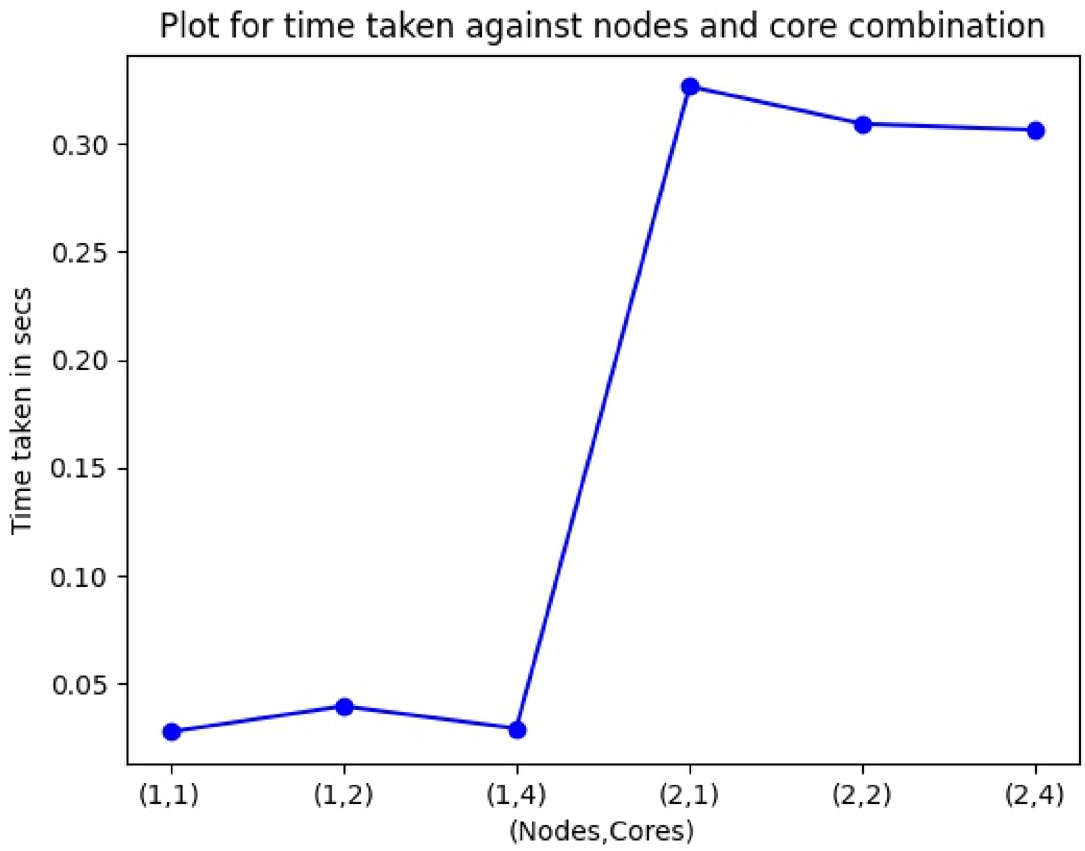

Figure 2. Timing pattern with MPI_Barrier()

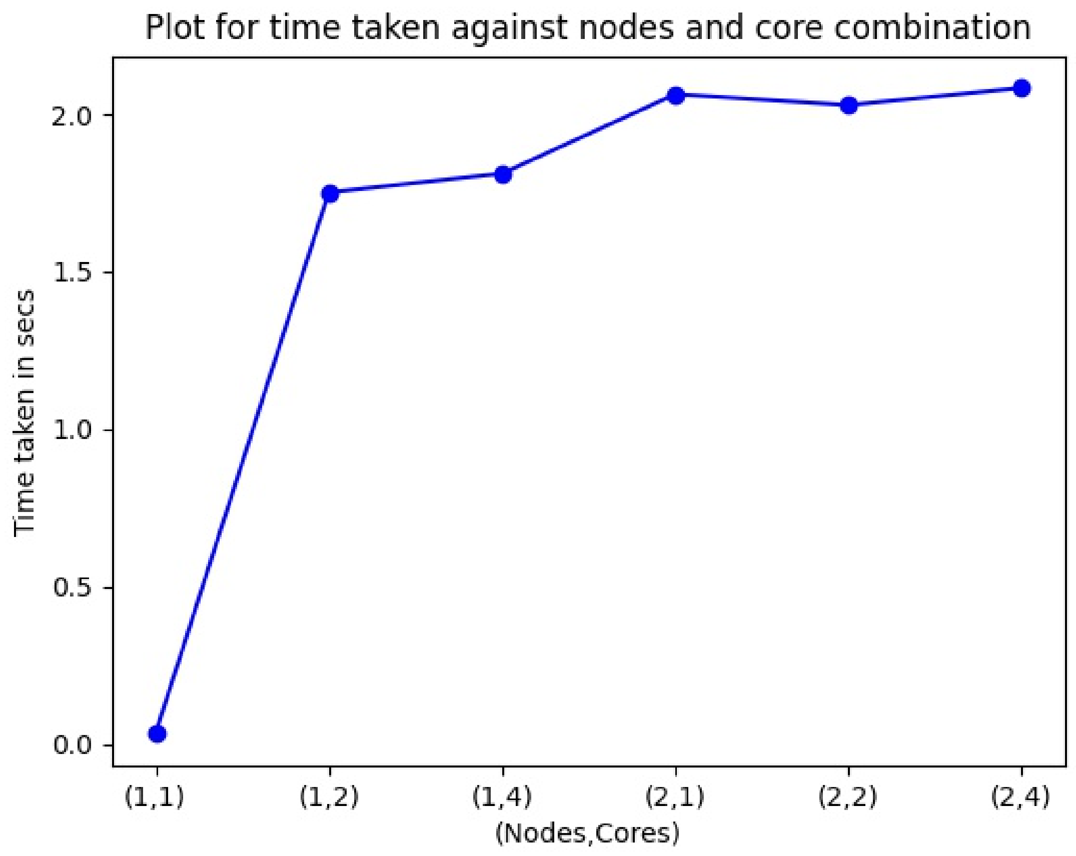

FIGURE 3. Timing pattern without MPI_Barrier()

5. Observations

- When the number of nodes is one and number of cores increases, in that case only intranode communication takes place, i.e only data has to be copied from data space of one process to another process and hence time remains almost same. Speedup achieved here, for combinations with one node, is sublinear as throwing more processes at the task doesn't speed up the entire operation (Figure 1 , node=1).

- We noticed that there was a hike in time that means we noticed negative speed up when number of nodes = 1 and number of cores were increased from 1 to 2, where as we noticed positive speed when number of cores were increased to 4 from 2. The reason behind this is that when there is only 1 process, there is no communication involved and hence the time is to perform the computation by 1 process, when number of cores increases, the communication time(to copy the data from the memory space of one process to another) is an overhead, and each process has to do the half of the work, and hence there is hike in the time. When number of cores are increased to 4, the communication volume increases, but the work per process decreases by a significant amount.

- We noticed a different scenario when nodes were increased to two. On increasing the cores, the performance kept improving. The reason behind this is that the communication volume which is sent to different node is always the same, as always almost half of the volume is sent to the other node. As the data being sent to other node remains same, the workload per process decreases on increasing the number of cores, which decreases the computation time and hence gives a better performance (Figure 1 , node=2).

- For configurations where node=1, the time taken was less than for configurations where node=2 because even though the number of processes are more in the latter, the task at hand is not computation intensive. Hence, the time incurred while doing internode communication outweighs the benefits of reducing the compute data size per process (Figure 2).

- The communication time is an overhead as the computation time of the data is small due to which when data is transmitted to other node, the overall time increases.

- We deduced that when the data to be transmitted, from the process reading the file to other processes not in the same node as the file reading process, increases, the total time increases as internode communication is involved.

- The given problem is not CPU intensive as the computation time is not significant and hence keeping the problem size same and increasing the nodes won't do any good. It just adds on to the overhead of communication time. This essentially means that Strong scaling cannot be observed for this problem.

- If both the number of processes and data size are increased and if the total time taken by single process to perform the task is more than time taken to communicate the data to processes on other processors, in that case increasing the number of processes would be beneficial. It essentially means that for such cases weak scaling could be observed for the given problem.

- If parallel I/O was used for reading the data from file, all processes would read the data intended for them. In that case, as the communication time is non existent before computation and minimal after it, it would have performed better than the single process performing the I/O and communicating the data to other processes.

- We also think if the processors allocated belong to different group then the time may increase more as the distance increases and the number of hops also increase. (Considering the topology of our csews clusters)

- The given problem is highly scalable but only if parallel I/O is allowed.

- As of now the number of processes are less and hence naive `MPI_Isend()/MPI_Irecv()` works better than a single `MPI_Scatter()` but if the number of nodes/processes is large in that case `MPI_Scatter()` will perform better. Because when the data size is increased by the order of N, the communication latency increases by order of log N in `MPI_Scatter()` and by order of N in `MPI_Isend()/MPI_Irecv()` and the transmission time will increase by the order of N in both the versions.

- Performing scatter takes a lot of time when number of processes is only 1, as the process sends all the data to itself and then does the computation. Due to which we observed speed up when cores are increased as the communication volume remains same, but problem size per process decreases.

- We also noticed that there was no speedup achieved when we didn't use barrier, as when process 0 was reading the data all the other processes start their timings while 0 does not complete reading (Figure 3)

- We took multiple runs for the timing of all the combinations but we didn't notice any significant change in time upto 4 places of decimal, so we just took one run and plotted those times.

6. Experimental Setup

(1) **Headers included in C program:**

- stdio.h, stdlib.h, math.h, string.h, assert.h, float.h, mpi.h (MPICH version 3.3.2)

(2) **Python packages/modules used:**

- matplotlib (version 3.3.4)
- os
- sys (version 3.8.5)