Report (Assignment 3 - CS633)

-Yuvraj (180898), Vishwas Choudhary (180876)

1. Steps to run the code

- Make sure you place the file tdata.csv in the current directory
- Run the job script using python3 run.py which will compile and run the main code for all the (nodes, ppn) configurations for 5 iterations and dump the output in file dump
- Run the plot script using python3 plot.py to generate the plot files plot_intranode.jpg and plot_internode.jpg
- Run make clean to clean all the plot files, hostfile, exectuable and dumps
- If you wish to run the code on a customized use-case, follow the below steps :
  1. make
  2. mpirun -np X ./code tdata.csv

  Note : X is the no. of processes to spawn, you may also use a hostfile if you wish

2. Code explanation

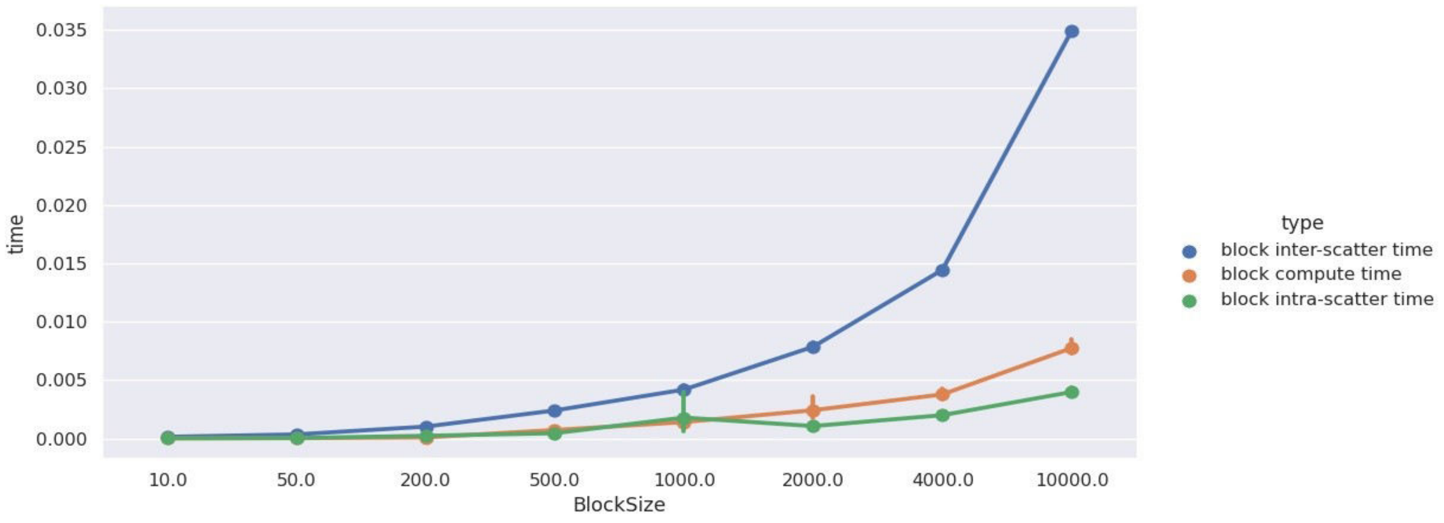
src.c : The code is quite understandable (is well commented) in the sense that it uses a naive algorithm in the case of a single process and for multiple processes, the code implements a data distribution strategy (explained later) from root process to leverage the possibility of parallelization and then the root retrieves back the processed data to compile the results.
Note : The variable 'block_size' stores the number of rows (and not the total data size) to be communicated to every process in every instalment.

3. Data distribution strategies and reasoning

- The first and the most naive strategy that we employed was the following :
  distribute data (row-wise) equally among all the processes -> every process does the required computation -> root retrieves the local computations
  This does not work simply because of the "idling" problem since data is large and a process can only begin computation after receiving its entire share of data.
- The second strategy employed was the following :
  ```
  split the data into many blocks (of rows) of equal size (=block_size)
  root process sends the first instalment of P blocks to the P processes
  for remaining instalments :
      root process issues isends of (n+1)th instalment of P blocks to P processes
      all the processes perform computation on blocks received in (n)th instalment
      wait call is issued to ensure receival of (n+1)th instalment of blocks
  ```
  This strategy gave good speedups but has the issue of root process being the bottlneck due to multiple isends issued by it in every instalment in a linear fashion.
- The third and final strategy is the refined version of the previous strategy :
  ```
  split the data into 'an optimal number of' blocks of equal size (=block_size)
  root process 'scatters' the first instalment of P blocks to the P processes
  for remaining instalments :
      root process issues 'iscatters' of (n+1)th instalment of P blocks to P processes
      all the processes perform computation on blocks received in (n)th instalment
      wait call is issued to ensure receival of (n+1)th instalment of blocks
  ```

Comments :

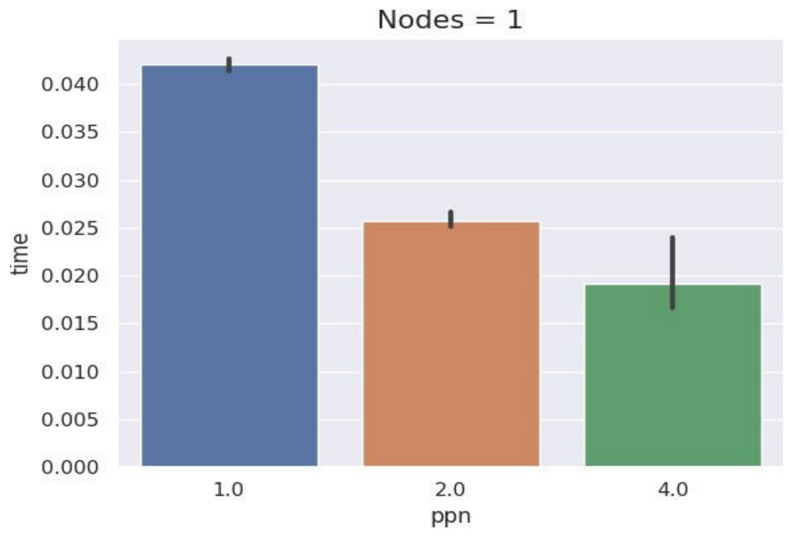
- Scatter is preferred over Send because it uses recursive doubling algorithm, which can avoid one process being the bottleneck and also minimize the number of commmunication steps (O(log P) vs O(P)).

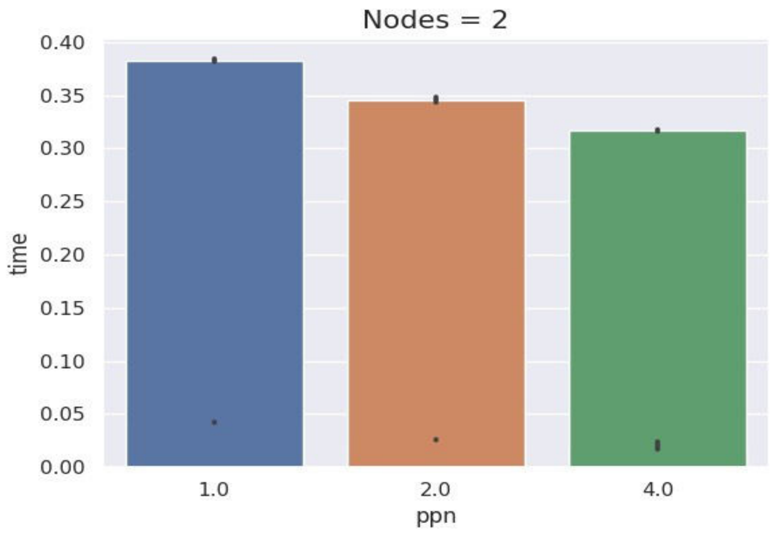


- Before undserstanding the significance of ‘block_size’ let us understand the above experimental plot, which (for a given block_size) compares the compute time for the block, inter-node and intra-node communication time taken to send the block. This comparison is important because for the best results, we wish the communication time at every instalment to be roughly equal to the compute time (so that communication through iscatter roughly completes with computation and the blocking time due to mpi_wait call is minimized).
  Now we can observe that for intra-node scatter, communication time for every block size is roughly equal to block computation time (which is great!), so we choose a smaller block size (to minimize the overhead due to first scatter call, which is blocking). For inter-node scatter, the communication time/computation time ratio becomes larger as block size increases, which again implies that a smaller block size will benefit us. Thus, we shall use smaller block size (<=1500) for further testing and refinement (refer ‘6. Additional testing’ for more details).

For retrieving back data, we have simply used mpi_reduce call which is the most feasible option considering the configuration we are working with (low (nodes, ppn) values).

4. Plots





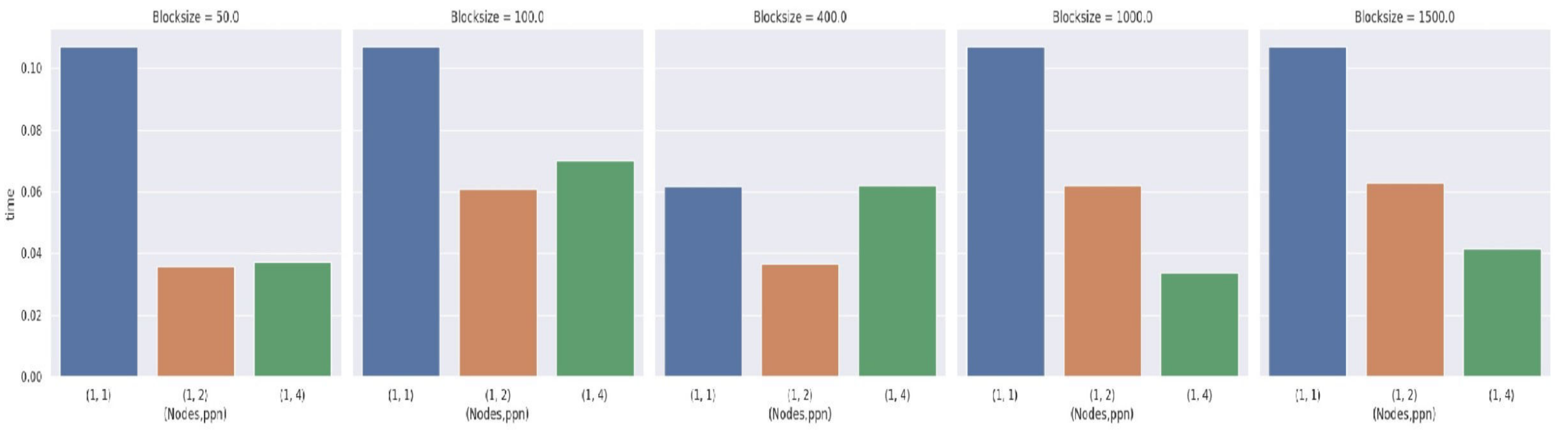
5. Observations

- The speedup observed for the first case is nearly ideal and is exactly what we expected. Minor latency comes due to the first scatter call and mpi reduce, otherwise communication tasks seem to be perfectly masked by parallel computation tasks.
- The performance for the second case is not at all up to the mark and is highly contrasting to what we expected. We tried playing around with the block sizes but this was the best we could obtain. A possible reasoning that we could think of is that probably inter-node latency is significantly high, to the extent that the compute time cannot even match it. This simply means that the mpi_wait call will block for a good amount of time, hence resulting in high delays. Thus, for such small computation, we cannot reap the benefits of inter-node parallelization because of very high communication overhead.

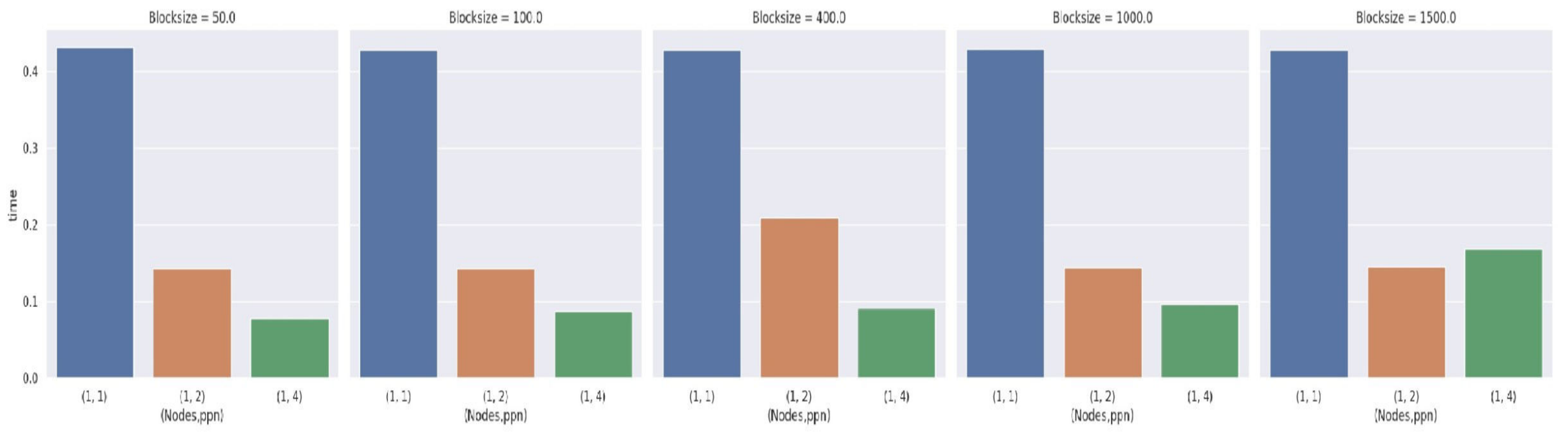
6. Additional testing

To figure out optimal block size values, we assumed that **block_size = f(rows, processes)**. Now to know on which factors optimal block_size really depends, we tested a few data files with the following specifications :
(rows, cols) = (100000, 100), (400000, 100), (700000, 100)
For every data file, we tested with block sizes : 50, 100, 400, 1000, 1500

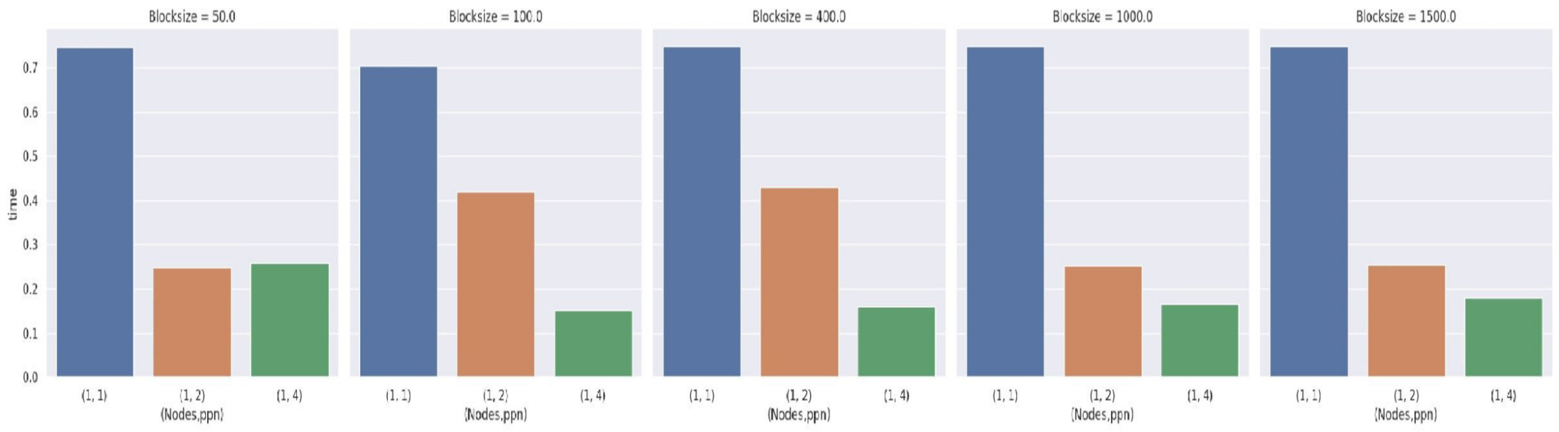
Intra-node (100000 rows)



Intra-node (400000 rows)

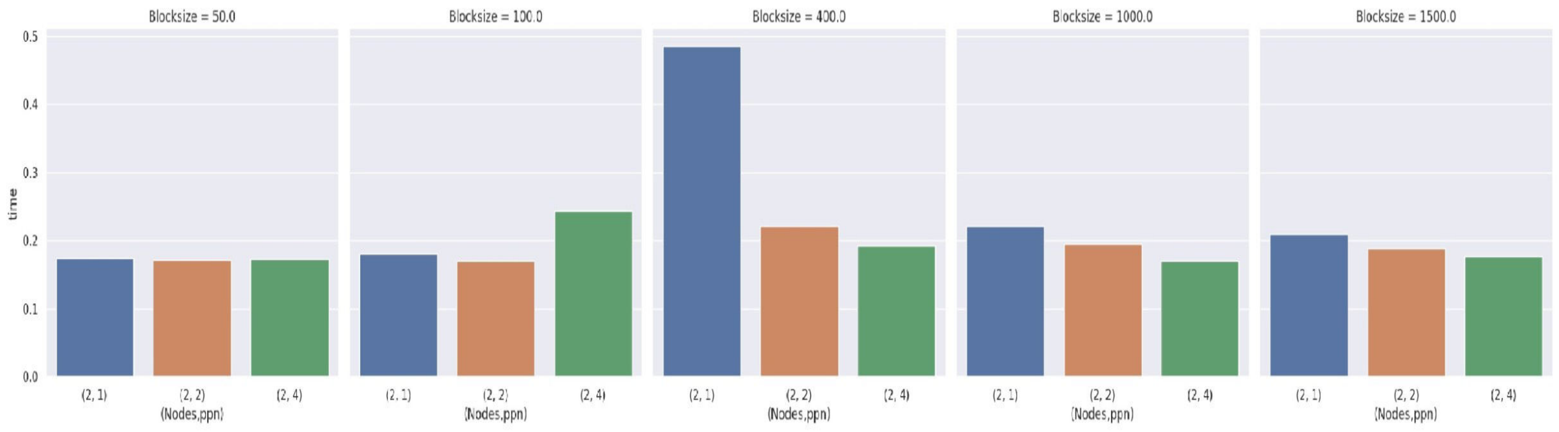


Intra-node (700000 rows)

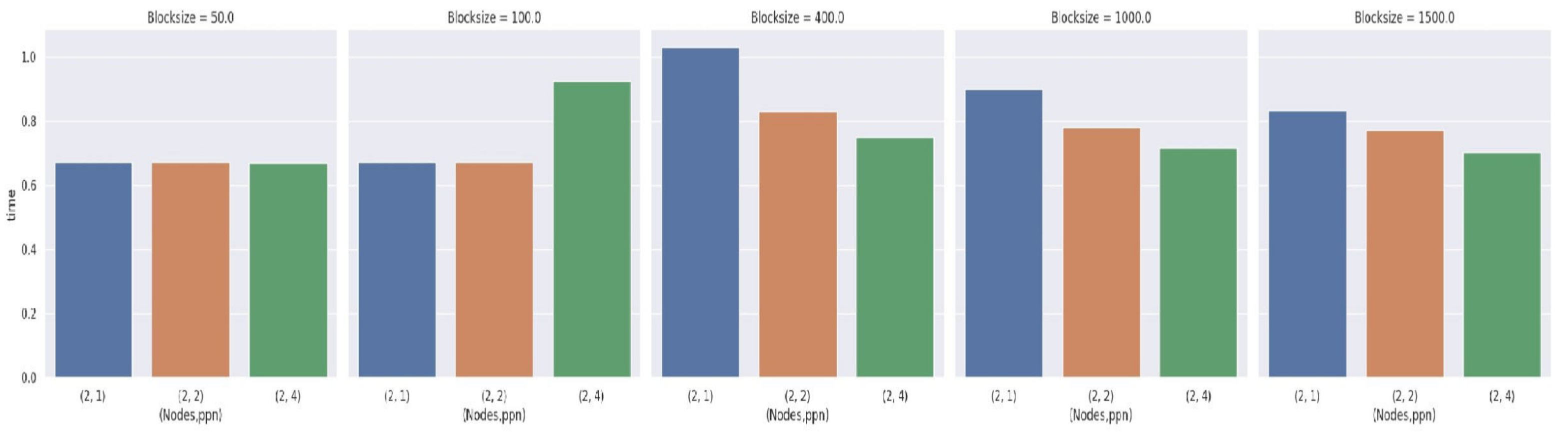


We observe that for intra-node case, we obtain the best results at block size of around **1000** across all data files, which makes us conclude that optimal block_size does not depend on number of rows. Also, there is not much dependence on the number of processes either.

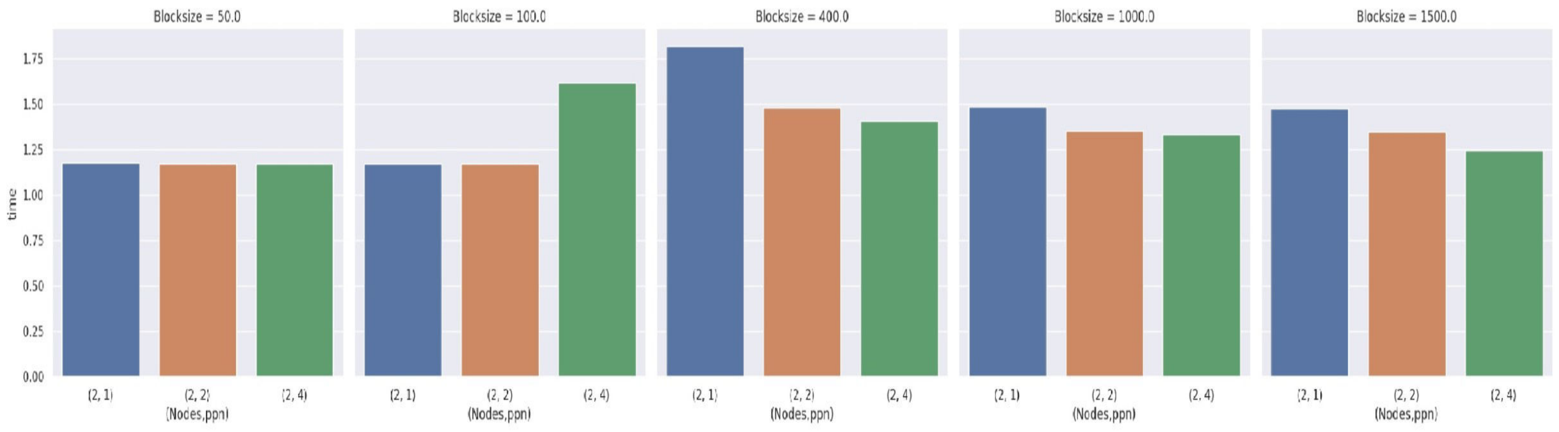
Inter-node (100000 rows)



Inter-node (400000 rows)



Inter-node (700000 rows)



We observe that for inter-node case, we obtain the best results at block size of around **1000** across all data files, which makes us conclude that optimal block_size does not depend on number of rows. Also, there is not much dependence on the number of processes either.

These insights make us conclude that for both the cases, **optimal block_size = 1000**
Hence, we have used block_size = 1000 in our implementation.