Tzi Siong Leong (20753794)
Jason Chao (21397806)

CITS3402 Project 2 Report

Design

Our combined openMP and MPI column wise matrix collisions program was implemented with the master and slave program structure in mind. This means that the master process (set as the process with rank 0) distributes the data columns to the different processes to generate the blocks for those columns. After the processes are done generating the blocks, the blocks are sent as an array to the master process where it pushes each block to the collision hashmap called collisionTable. This design tries to distribute work uniformly by distributing equal number of columns to each process. If the number of processes generated are not divisible by the number of columns in the data, it will keep decreasing the number of processes used until it finds the greatest common factor of the number of processes to work with the data columns. For example, since the number of data columns is 500 columns and 32 processes are generated to work with the data, the 500 columns are not divisible by 32 processes so the numprocs (number of processes used) is decreased to 25 processes. Rank 0 is always the master process which distributes work to the next 24 processes. The other 7 processes are not used for message passing or work distribution so they quickly end their processes. This system is needed in the case that the desired number of processes can not be manually indicated so it adapts by using as many processes as the data columns allow it to.

All of the openMP directives from the first project are still present in the second project. They mostly consists of worksharing constructs in the for loops where most has static scheduling. Static scheduling was used to divide loops into equal sized chunks for each thread to process work. OpenMP lock routines are also used when the blocks are being inserted into the hashmap collision table. This is so that the hashmap key bucket would not be accessed and written at the same time by multiple threads to insert blocks so that it would prevent read/write errors.

The master and slave program structure was chosen due to the fact that it naturally fit the idea that multiple processes can work on chunks of columns concurrently and independently without disturbing each other much to increase time performance. The master process distributes the columns to the other processes. The non-master processes mainly message to the master process when they have finished their work of generating blocks to let the master process push the blocks to the universal collision hashmap table. The program was purposely implemented to limit the number of times MPI_Send and MPI_Recv are used due to the fact that it can hinder time performance. Potentially, the more MPI_send and MPI_Recv are used, the more time it can spend blocking until the right conditions are met to send and receive thus increasing the overall time performance.

All the processes will have a copy of the keys data file when it uses the rows' key to generate the keysum when it generate blocks. The master process is the only process that have the data columns data, which it will be distributed to the different processes evenly using the blocking MPI_Send. The master task processes the first chunk of the columns while the other tasks would be receiving the equal number of data columns using the blocking MPI_Recv. Blocking send and receives are needed because of the sequential nature of generating blocks. For example, the chunk of data columns needs to be passed to a process timely in order to generate blocks from each data column in the data. Nonblocking sends and receives do not guarantee the timely receipt of the data so not all columns are guaranteed to have blocks generated from.

Tzi Siong Leong (20753794)
Jason Chao (21397806)

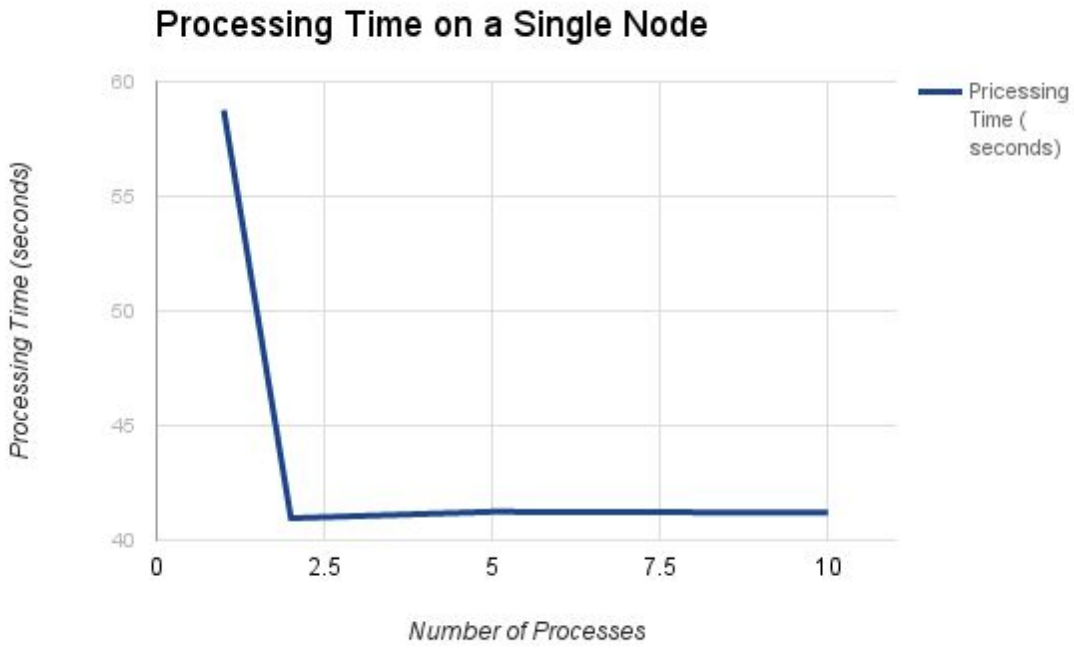
CITS3402 Project 2 Report

Analysis

WIth our combined OpenMP and MPI code, we ran it on the given test cluster which consisted of 29 nodes, with each node containing twelve cores, which gave us a total amount of 348 MPI usable processes. We performed some tests with our MPI code to see how well it scales. The tests that we carried out include: using one node to do all the processing, using all nodes available in the cluster and testing a fixed number of processes with varying amount of nodes.

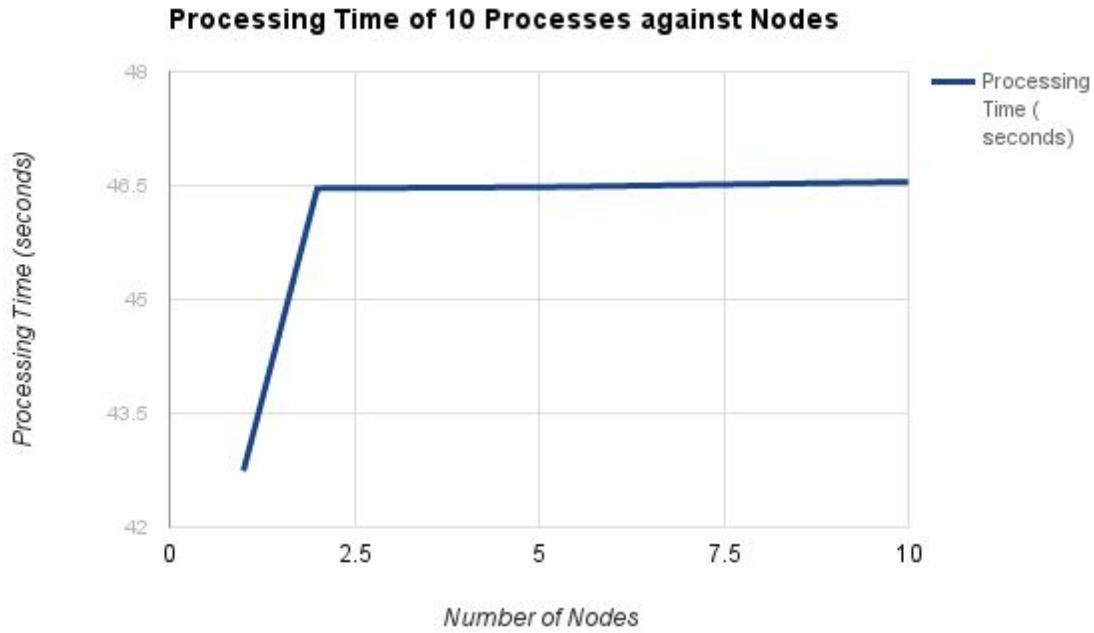
From our testings in Figure 1, the results show that when running our code with one node only while using one process the wall clock time was 58.768 seconds. Increasing the process count to 10 in the same one node decreased the wall clock time to 42.3111 seconds. The decrease in time shows that the workload was spread out to the other cores of the node for parallel computation of the data.

Figure 1. Processing time on a single node with varying number of processes.



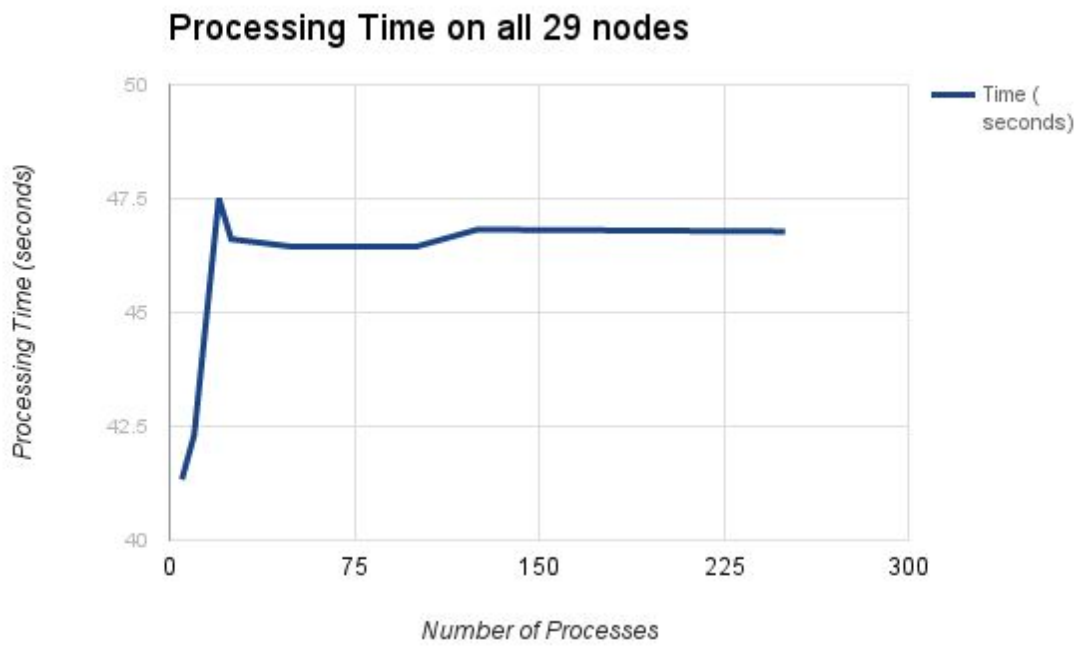
The first test carried out was using a single node in the cluster and changing the amount of processes used to run our code. The amount of processes used are the numbers that are factors of 500. In this case, the amount of processes used are one, two, five and ten processes for a single node. Figure 1 shows that when we increased the number of processes from one to two, the time decreased from approximately 58 seconds down to 42 seconds. Though from increasing the amount of processes above two, we observed no improvements in time performance.

Figure 2. Processing time on ten processes with a varying amount of nodes.



The second test we performed involved having a total of ten processes to be run across the nodes, in configurations of one node and ten processes, two nodes and five processes on each node, five nodes and two processes on each node and one process on each of the ten nodes. That is, the total number of processes generated is fixed while the number of nodes and processes per node varies. From the results shown in Figure 2, we can see as we go from one node to two, performance in terms of processing time actually decreases, and once we go above two nodes, the processing time stays constant. The increase in processing time can be due to network buffers and network loads between communicating nodes, although this may not be the case as when we increase the amount of nodes the processing time does not drastically increase.

Figure 3 Processing time using all 29 nodes and processes which are factors of 500



The final test that we performed used all 29 nodes in the cluster while specifying the amount of processes we want to generate in the cluster to share among all the nodes, the number of processes generated are factors of 500. Figure 3 shows that there is an increase in processing time when we increase the number of processes from approximately five to twenty processes. And the processing time roughly stabilized around 47 seconds.

Conclusion

From all the tests that we performed, the results generally show that the more number of nodes are used, the processing runtime generally increases. In addition, the more processes are used, the better the time performance of the processes, although only shown in the case of using a single node and also up to a limit. What we can conclude from the results is that as the number of nodes increases, the runtime of the program increases as well suggesting that internode communication might hinder on the performance of the program (Layton, 2016).

References

Layton J. 2016 “Why Isn’t Your Application Scaling?“
Available from: http://www.admin-magazine.com/HPC/Articles/Failure-to-Scale