MAP-REDUCE implementation using OpenMP and MPI

1. Introduction

There are different parallel programming protocols like OpenMPI and MPI. These programming protocols are used for parallel computing where the execution of different processes are carried simultaneously for efficiency and speeding up the computational time. Large problems or functions are divided into smaller ones to solve them at the same time.

One important application of parallel programming protocols is MapReduce, a programming model for processing large data sets into a distributed fashion over several different machines. As the name suggests, input data or data to be processed is mapped into a collection of <key, value> pairs based on different properties and determined using “Hash Function”.A hash function is used to map/index original data of arbitrary size into data of fixed size. Once the data is collected as <key,value> pairs, they are reduced over with the same key.

In this project, we used the MapReduce programming model to count the numbers of various words that appeared in different input files. The output files contain each word with the number of times it appeared in the input text files.

2. Design

In our project, we have used two different parallel programming protocols to implement the MapReduce. The first is using OpenMP and the second is using MPI. The code is implemented using C++ using the Standard Template Library.

2.1 OpenMP

OpenMP comprises of the library that supports shared memory parallel programming which means all threads share memory and data. The part of the code that’s marked to run in parallel using OpenMP will cause threads to form. The main thread is the master thread. The slave threads all run in parallel and each thread executes the parallelized section of the code independently. When a thread finishes, it joins the master. When all threads finished, the master continues the main code sequentially. Figure 1 shows briefly how the MapReduce is implemented using OpenMP. Four different threads are used: reader, mapper, reducer, writer. Queues used to transfer data between threads with locks to help with concurrent thread safe access.

2.2 MPI

Message Passing Interface (MPI) is a standardized and portable message-passing standard that supports distributed memory parallel programming. The role of the different threads in MPI are as follows. Refer to Figure 1.

Reader Threads

Different text files are first read by Reader threads in parallel. They are read as a "Record" of data type pair<string,int> where string is the word and at this step "int" is just 1. They are then pushed into MapperWorkQueue where the index of the queue is generated using the rand function for each "Record". At this point the MapperWorkQueue might look like [<Cat, 1>, <hello, 1><place, 1>.....].

Mapper Threads

At this point we have reader threads writing the "Records" to the MapperWorkQueue.
Now, using a hash table, "Records" from the Mappers take records from their respective WorkQueues and create records of the form [<Cat, 2>, <hello, 5>, <place, 3>. Once all reading is complete, all the records are put into the SenderWorkQueue

Sender Threads

Sender threads gathers all the records from the various Mapper threads in a process and add it to a combined list for the process to send to the receiver threads.

Receiver Threads

The receiver thread is where all the MPI communication between processes happens. We use an MPI_AllGatherv to combine the various SenderWorkQueues in the various processes into one universal list of records. After this the receiver uses a hash function to send the records into the appropriate ReducerWorkQueue.

Reducer Threads

Reducer threads start accumulating records into single records with combined word counts. Each word is assigned to a unique reducer due to the hash function. After all Reducing is done, the Reducer assigns the record to be written to files randomly to the WriterWorkQueues

Writer Threads

Once all Reducer threads have finished doing their work and the ReducerWorkQueues are empty, the writer threads start writing the word counts to the output files.

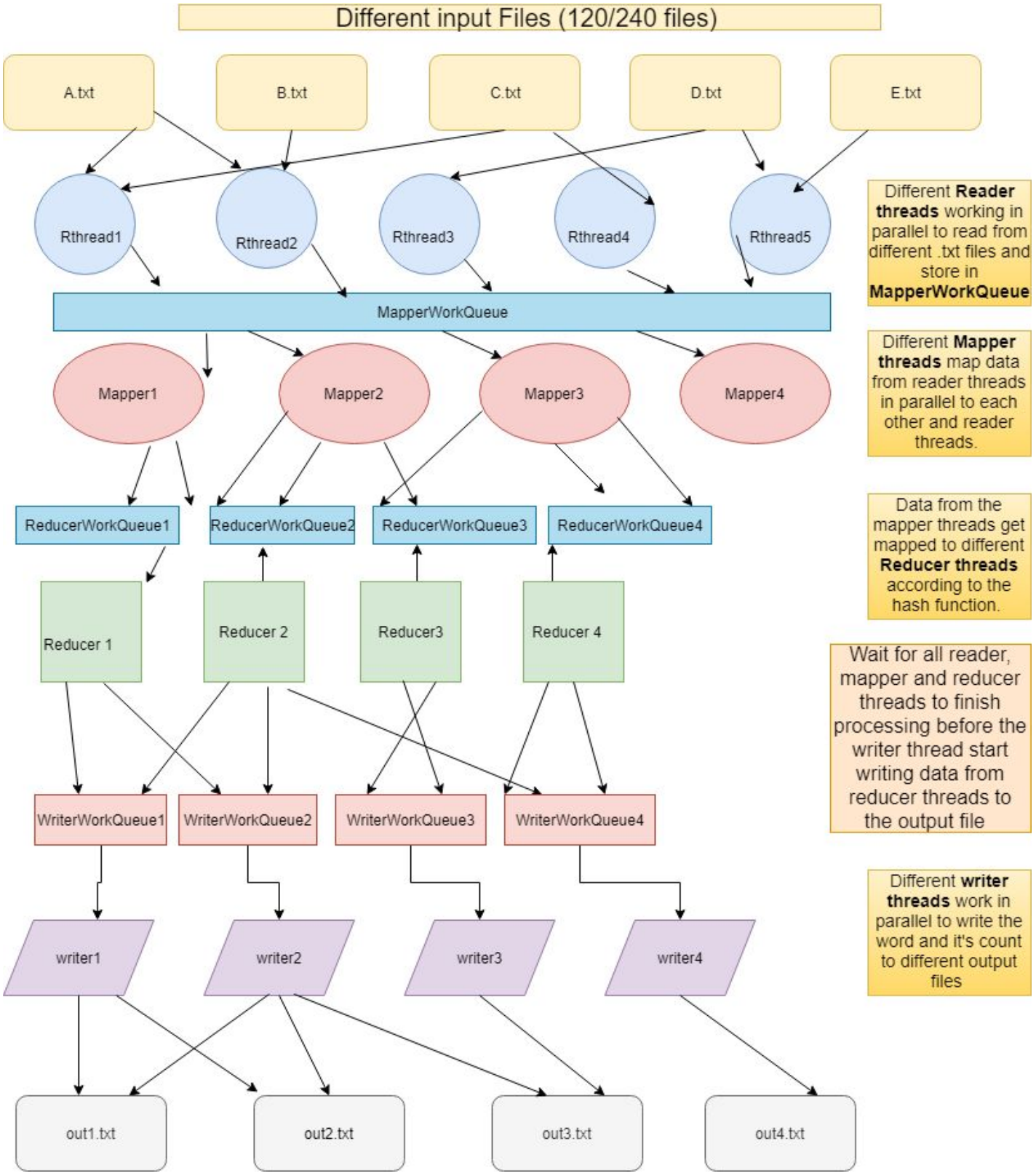

Figure 1: Flowchart showing the flow of data in MapReduce using OpenMP and MPI

3. Experiment

To test out our MapReduce implementation, we tested out OpenMP and MPI version on scholar with multiple nodes and processes. The details are described below.

3.1 Preparation of dataset

Initially, the text was processed with Python to clean punctuation and hyphenated words. Words with special characters were omitted to end in file parsing. The processed version of these text files split into 24 sub files randomly. The OpenMP version of our program was tested with one reader, mapper, reducer and writer thread each with the given data set. It was found that the total time under that test was ~ 3 secs. To increase the workload, multiple copies of the set of text files were used. It was found that ten sets of the given text files took about 30 seconds to run and that duplicated data set was used as the final dataset.

3.2 OpenMP Tests

The OpenMP version of MapReduce was run with various configurations (Table 1) of different thread types. Each configuration type was run at least ten times, and the average of these values was used as the final time taken.

Sl. No.	# of Reader Threads	# of Mapper Threads	# of Reducer Threads	# of Writer Threads
1	1	1	1	1
2	2	2	2	2
3	4	4	4	4
4	8	8	8	8

Table 1: Thread configurations tested for OpenMP version

For each run, the data in Table 2 in was collected. Note that Sender and Receiver thread times were only collected for MPI runs, as there are no sender and receiver threads in the OpenMP version.

Total Time	Shortest Sender ThreadTime	Longest Mapper Thread Time	Longest Reducer ThreadTime
Longest Reader Thread Time	Longest Receiver ThreadTime	Shortest Mapper ThreadTime	Longest Writer ThreadTime
Shortest Reader Thread Time	Shortest Receiver ThreadTime	Longest Sender ThreadTime	Shortest Writer Thread Time

Table 2: Data points collected per run

3.3 MPI Tests

For each configuration type of OpenMP mentioned in Table 1, the MPI version of our code was tested for processor count 1, 2, 4, 8 and 16. Data points similar to Table 2 were collected. As different processes had different numbers for each data point, the timing results were reduced across processes, with MPI max for longest time data points and MPI_MIN for shortest time data points. For MPI version, one sender and one receiver thread was used per process.

3.4 Checking accuracy of Word Count

To check the accuracy of the word counts, generated by MapReduce, the linux command `grep -roh <word> . | wc -w d` was used to compare with the output created by MapReduce.

4. Results

The best speedup of 3x was achieved with 4 processes each with 4 readers, 4 mappers, 4 reducers and 4 writers. All performance number for the graphs can be found in Table 3 and Table 4 under Section 7 Performance Numbers. The plots are in this section.

*NOTE: We use the notation "**config a, b, c, d"** to represent a reader threads, b mapper threads, c reducer threads and d writer threads. There is always 1 sender and 1 receiver thread.*

4.1 OpenMP

For OpenMP the plots below show the total time (Figure 2), SpeedUp (Figure 3), Efficiency (Figure 4) and Karp-Flatt Metric (Figure 5) vs the number of threads of each type below. As we can see in Figure 3, the speed up initially rises till 4 threads but stays constant after that.

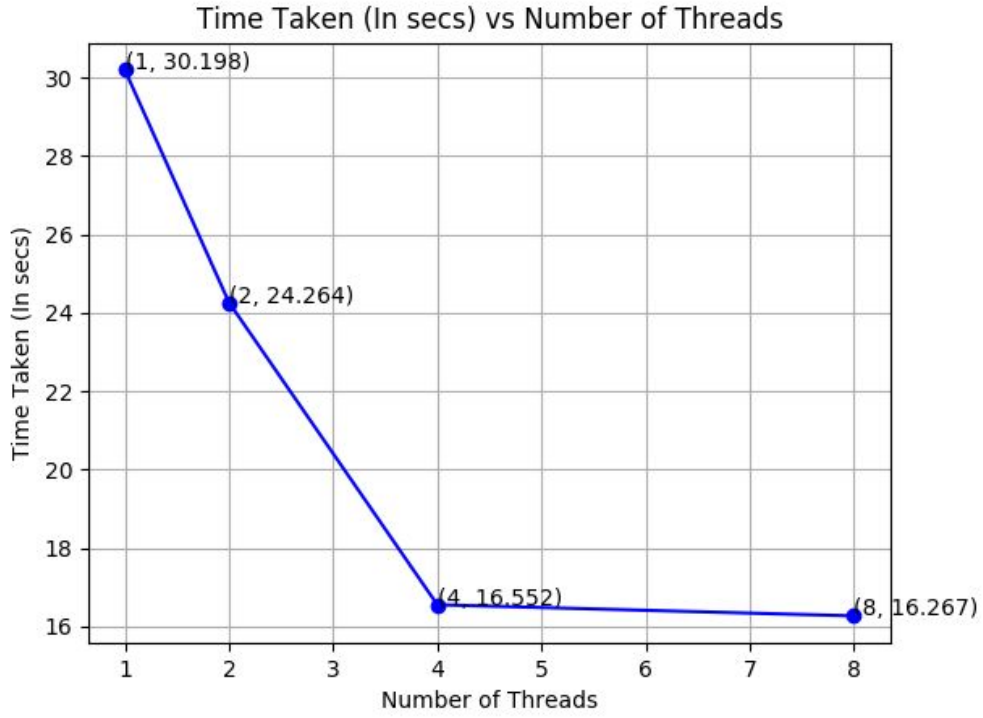

Figure 2: Total Time Taken vs Number of Threads

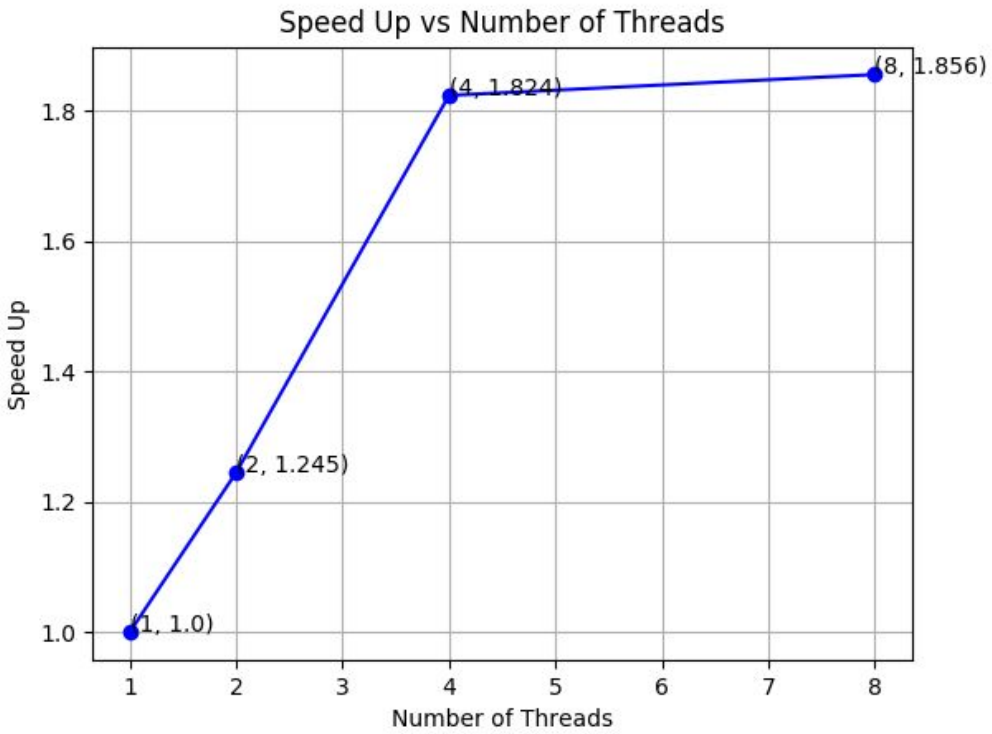

Figure 3: Total Time Taken vs Number of Threads

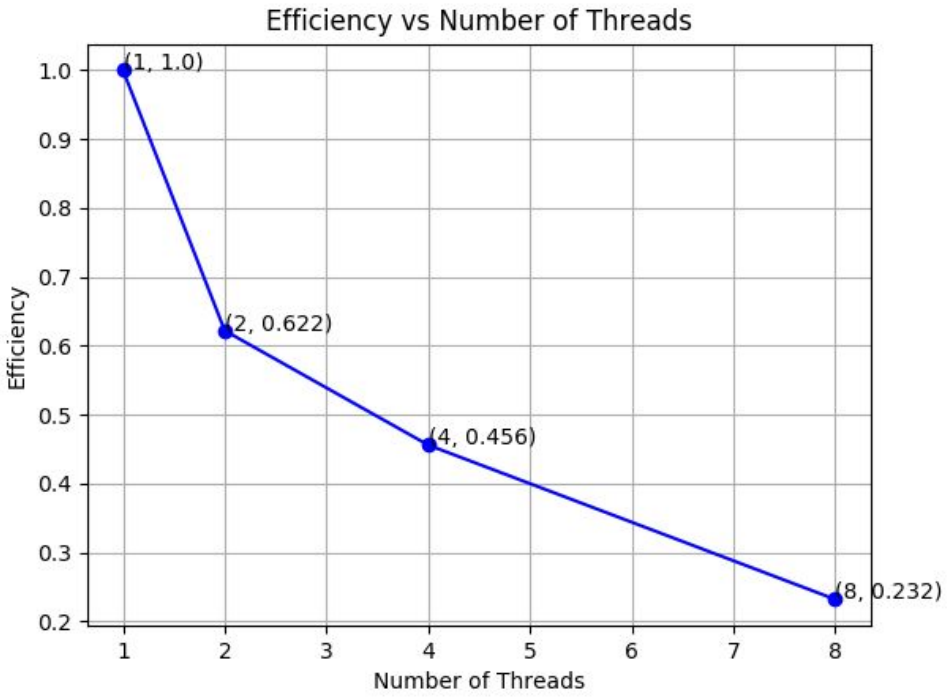

Figure 4: Total Time Taken vs Number of Threads

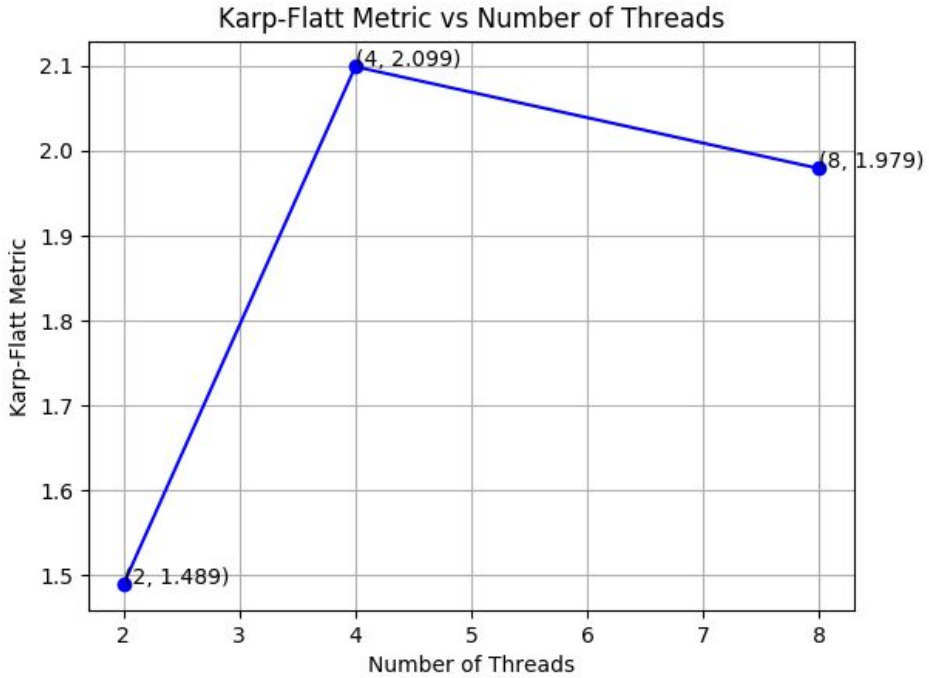

Figure 5: Karp-Flatt metric vs Number of Threads

4.2 MPI

The MPI version of our code was tested for processor count 1, 2, 4, 8 and 16. The plots for the 4, 4, 4, 4 configuration are below. The performance numbers are in Table 4 under section 7. The plots (Figure 6 - Figure 10) are below.

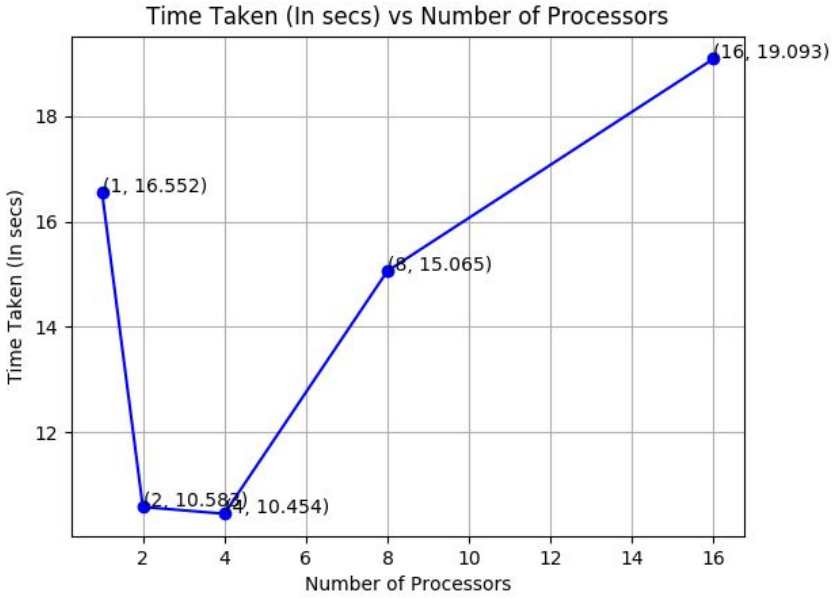

Figure 6: Total Time Taken vs Number of Processes

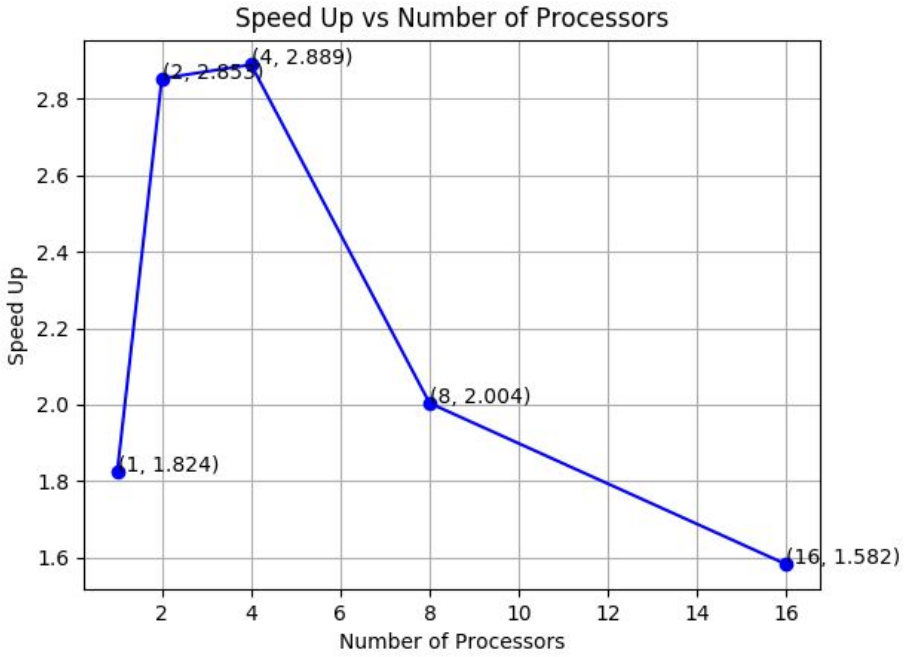

Figure 7: Speed Up vs Number of Processes

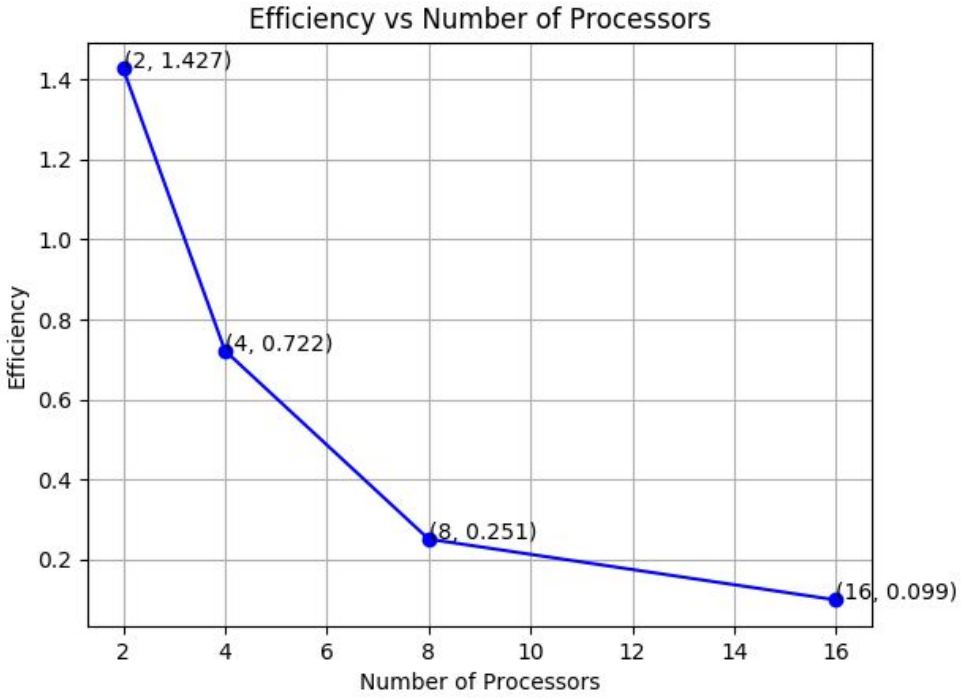

Figure 8: Efficiency vs Number of Processes

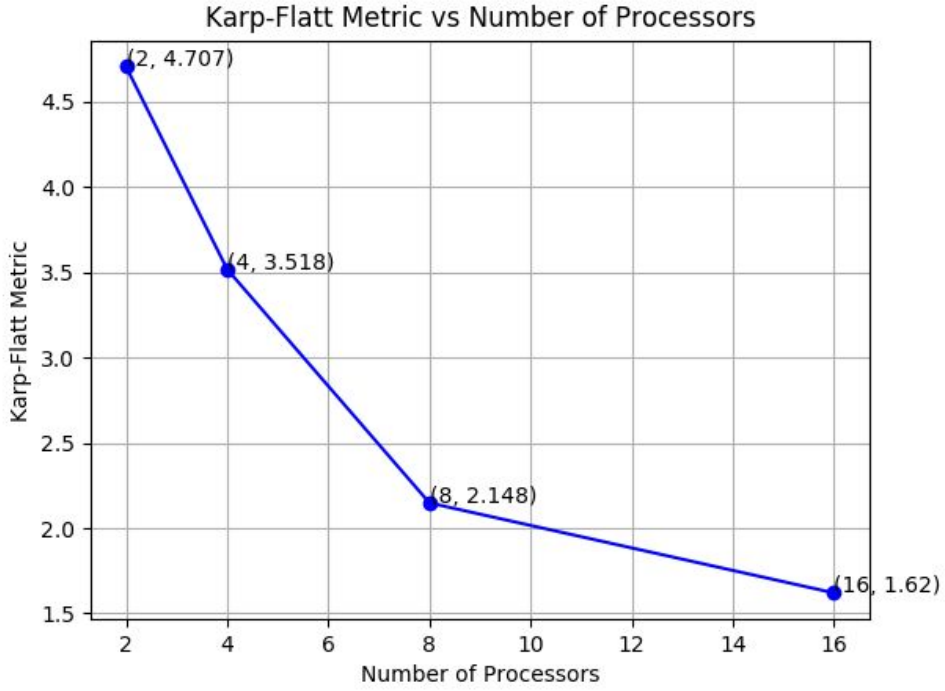

Figure 9: Karp-Flatt metric vs Number of Processes

5. Analysis

5.1 Explanation of speedup

Figure 10 shows the total time taken by the longest mapper, longest receiver, sum of mapper & receiver and the overall total time taken during execution of the MapReduce program for increasing number of processes. (Config 4, 4, 4, 4). From the graph, we can see that the mapper threads and the receiver threads account for the majority of the total time. We can see that the reducer and writer threads barely have an effect on the total time, as indicated by the almost horizontal yellow line in Figure 10.

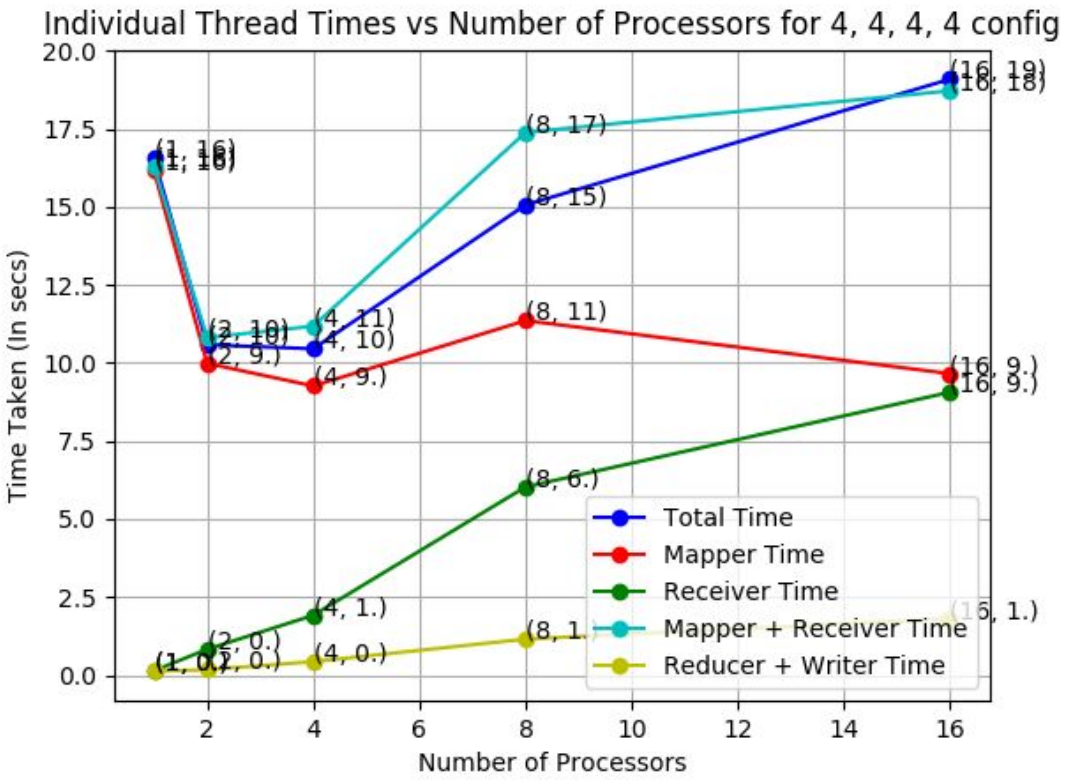

Figure 10: Individual Thread times vs Number of Processors

In fact, the total time is a trade off between drop in mapping costs, vs rise in communication costs in receiver. The SpeedUp curve in Figure 7 can be explained on the basis that till 4 processes the drop in mapping costs outweigh the rise in communication costs in receiver. Beyond 4 processes the drop in mapping costs slows down, as can be seen in Figure 10, but communication costs keep rising, therefore the speedup starts dropping.

Also, for a constant process count, the increase in speedup comes from reduced reading & mapping times with increasing thread count which is the true core of all speedup. This can be seen in Figure 11. The cost for mapper and reducer keeps dropping with rising thread count with almost a constant reducer and writer time.

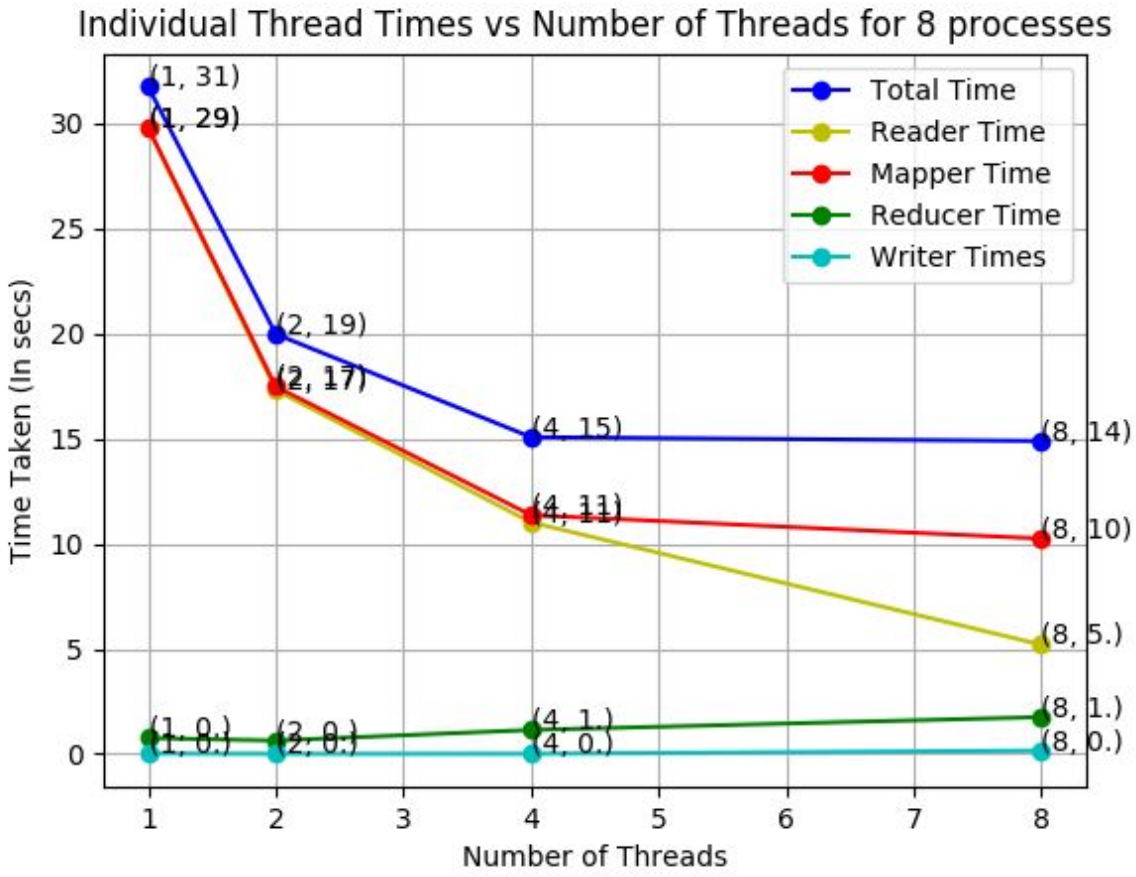

Figure 11: Individual Thread times vs Number of Threads for 8 Processes

5. 2 Bottlenecks

There are two major bottlenecks in our code. First as shown in Figure 12, for a variety of configurations, the gain in Mapper SpeedUp slow down after a certain number of processes.

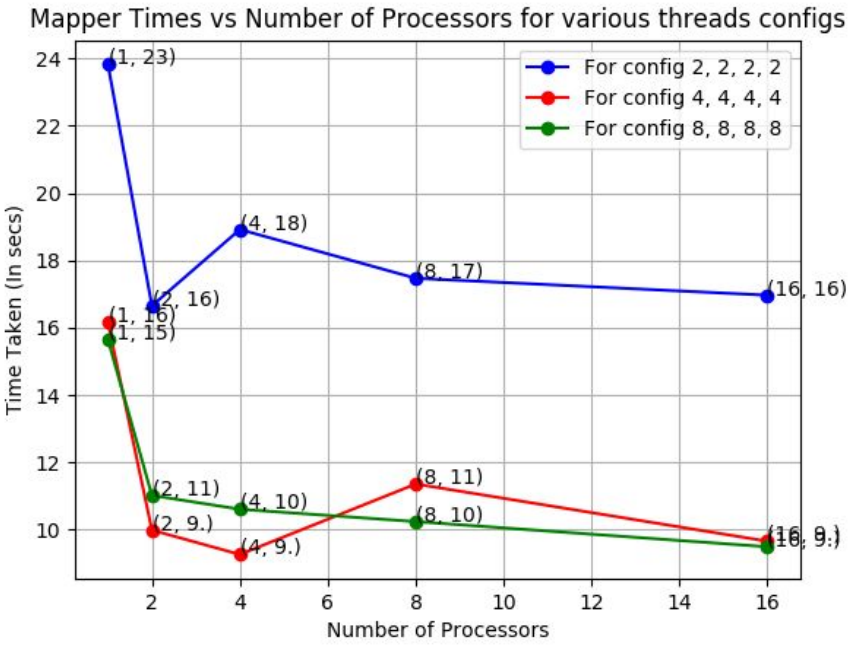

Figure 12: Mapper Times for different thread configs vs Number of Processes

However, as can be seen in Figure 13, for every configuration tested, communication cost rises linearly with process count. Therefore, we needed a better communication model to achieve better speedups lower than $\Theta(p)$, where p is the number of processes.

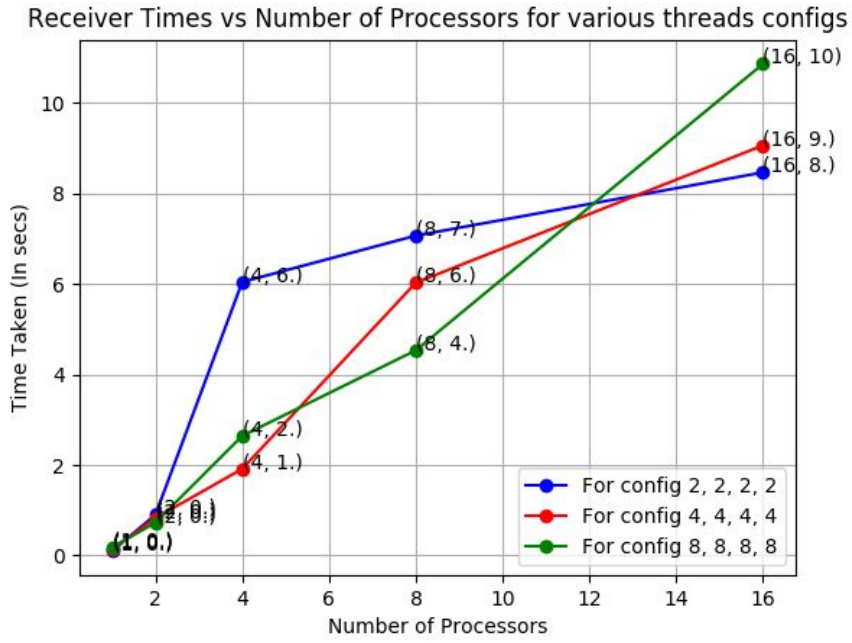

Figure 13: Receiver Times for different thread configs vs Number of Processes

5.3 Positives & Negatives

Our MPI project achieves higher speedups than just the OpenMP version indicating the MPI works. In addition, the intricacies of AllGather are handled perfectly. In the future, we would like to improve the communication to a better system that doesn't have to wait for all mappers to complete. In addition, we would like to consider the node the process is running on properly to only run threads on processes in different nodes

6. Conclusion

We built two versions of MapReduce, one using OpenMP and MPI. The first version only uses OpenMP and uses one processes. The second version uses MPI and can run on multiple nodes with multiple processes. The best speedup of 3x was achieved with 4 processes each with a 4, 4, 4, 4 config. Future work would involve improving the communication method to reduce the growth of communication costs.

7. Performance Numbers

# of Threads reader and mapper threads	Time Taken (In secs)	SpeedUp	Efficiency	Karp-Flatt Metric
1	30.198	1.0	1	-
2	24.264	1.245	0.622	1.489
4	16.552	1.824	0.456	2.099
8	16.267	1.856	0.232	1.979

Table 3: OpenMP Performance Numbers

# of Processes	Time Taken (In secs)	SpeedUp	Efficiency	Karp-Flatt Metric
1	16.552	1.824	1	-
2	10.583	2.853	1.427	4.707
4	10.454	2.889	0.772	3.518
8	15.065	2.004	0.251	2.148
16	19.093	1.582	0.099	1.620

Table 4: MPI Performance Numbers per Process for Config 4, 4, 4, 4