MapReduce

ECE 563 Spring 2021 Project

Adhitha Dias: 0032780886 & Mohammed Moiz: 0032791643

Table of Contents

1. Introduction

1.1. Word Frequency Counter

In this project, we implement a Word frequency counter which counts the number of occurrences of different words in the given list of files. Word Frequency count has applications in Teaching/Learning new language[1]. Language research and Natural Language Processing. Processing large files can become time consuming as the number of files increase, and parallelization becomes a necessity. In this project, we have four implementations of word frequency counter: Serial, OpenMP, MPI w/o OpenMP and MPI w/ OpenMP. The runtimes are recorded for all four implementations and speedups are analyzed baselining against the serial execution. Finally, the bottlenecks and future improvements are summarized to further speedup this task.

1.2. Serial Execution

The serial execution of the word count problem can be understood from the below diagram.

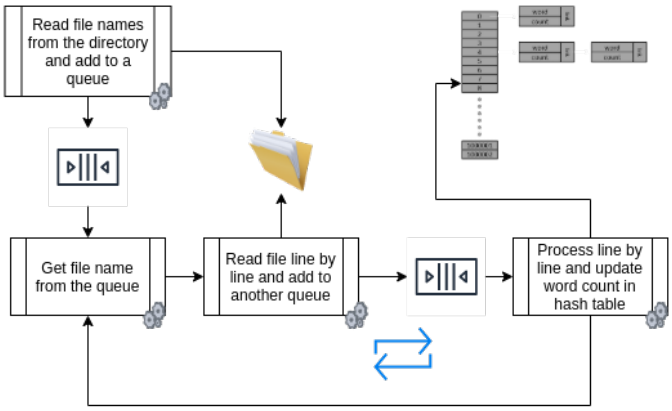

Figure 1 Serial execution flow for Word Frequency Counter

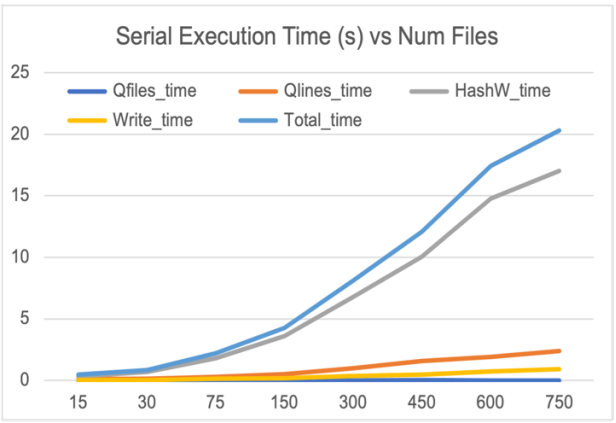

Figure 2 Serial Execution times for the various steps

Below are the steps followed for the serial execution:

1. Read the file names inside a folder and add it to a queue. (*Figure 2:* Qfiles)
2. Process each file in the file name queue one at a time and perform the below two tasks.
   a. Read the files line by line and add lines to a reader queue. (*Figure 2:* Qlines)
   b. Get lines from the reader queue, break down the lines into words, calculate the hash of each word and update the corresponding word count in a chained hash table. (*Figure 2:* HashW)
   c. This process is carried out until there are no more items left in the file name queue.

Figure 2. shows that the serial execution time increases proportional to the number of input files. This increase is linear with most time being taken by the Word Hash Mapping functionality (in gray)

1.3. MapReduce algorithm

As discussed previously, when the file size/number of files are large, it takes a lot of time to compute the word counts serially. Therefore, parallelizing this task will help us get the word counts faster. MapReduce is a programming model and an associated implementation for processing and generating big datasets with a parallel and distributed algorithm on a compute cluster. It has become very popular in part because of its use by Google, but it is an old parallel programming model. The significance of the MapReduce algorithm relies on the fact that it can be applied to any large data set and any problem that can be run in parallel in a distributed system. This report summarizes the MapReduce algorithm implemented both in OpenMP and MPI to speed up the word frequency counter.

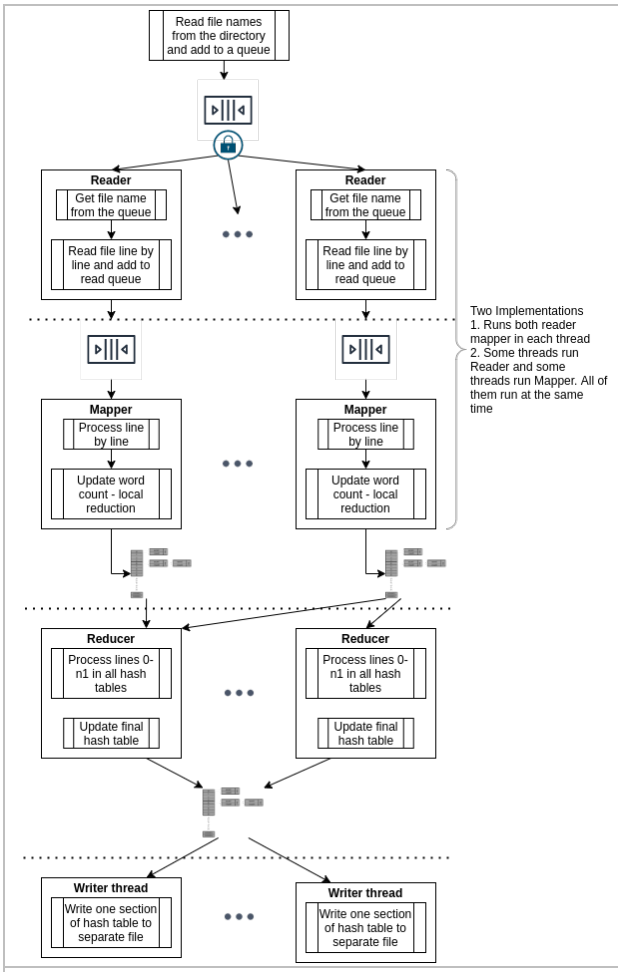

Figure 3. OpenMP execution diagram. We identified that reading and mapping takes most percentage of execution time. We implemented 2 programs – one where each thread runs reading and mapping sequentially, and the other one concurrently.

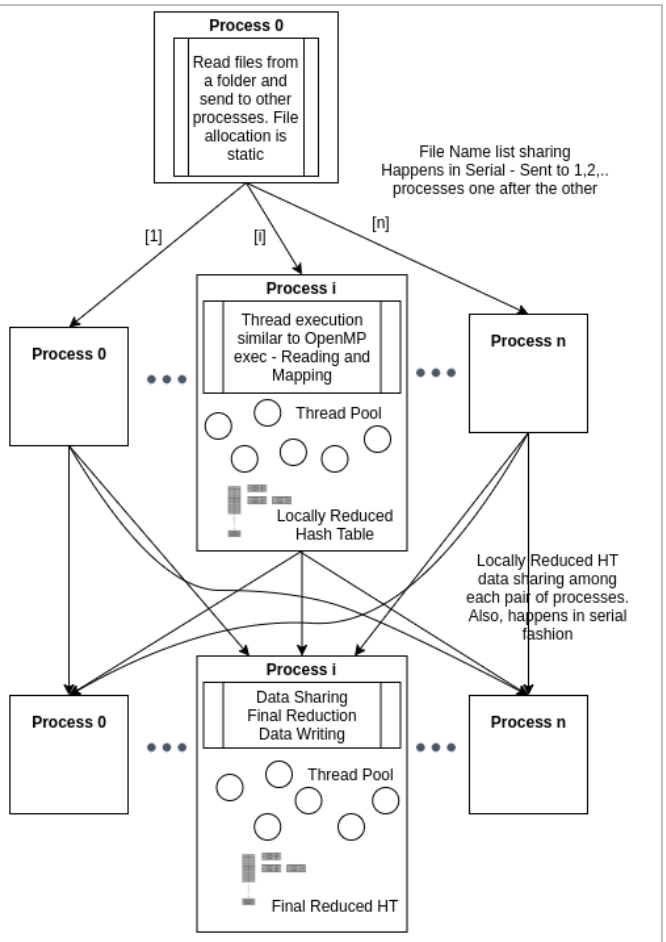

Figure 4. MPI w/ OpenMP program is identical to the OpenMP program execution. The only difference is that file names are sent from process 0 to all other processes. And after local reduction inside each process, they share the reduced word counts among the processes to calculate the final word counts.

2. OpenMP Implementation

We break down the implementation in to 4 separate parts. Reader threads, Mapper threads, Reducer threads and Writer threads.

1. Reader threads – Responsible for reading the files line by line and adding those lines as items into a queue.
2. Mapper threads – Responsible for reading line items from the reader queues, breaking the line strings to words, removing unnecessary characters by cleaning the words and updating a thread local hash map with the word counts.
3. Reducer threads – Responsible for accumulating the word counts in each thread local hash tables and updating the final counts in a final hash table.
4. Write threads – Responsible for writing a portion of the final reduced word count hash table into separate files.

We have 2 different implementations of the OpenMP code with minor semantic changes to the code.

2.1. Naïve OpenMP Implementation

The Naïve implementation using OpenMP parallelizes the serial program with the "#pragma omp" directives, each section of code for Reading, Mapping, Reducing and Writing. This implementation has input arguments to alter the program behavior without the need to recompile.

Figure 3. describes the different steps in OpenMP program. A typical run can be executed with the command ***'./parallel -d ./files -r 10 -t 8'***. The argument *'-d'* gives the input directory; *'-r R'* defines the number of repetitions (R) to make while reading files; *'-t T'* defines the number of threads (T) to be used. Below are the steps in the program:

1. Qfiles (single Queue) – Queue of input files in the input directory. This is repeated 'R' times.
2. Qlines ('T' queues) – Queue of lines in the files is created by reading the files one by one in parallel by T threads. Each thread locks the files queue and de-queues the topmost file in the queue. This file is then read, and the lines are added to their respective queue.
3. HashW ('T' Hashes) – T mappers populate the lines in parallel and perform mapping to their respective hash tables.
4. Reduce ('T' Reducers) – Each reducer processes 1/T section of all the T hash tables and collates the reduced results to a final hash table.
5. Write ('T' Writers) – The final hash table is written into T different files with each file having 1/T section of the result.

This implementation gives a top-level overview of which sections benefit the most by parallel runs and also point to possible bottlenecks. The detailed performance and speedups are recorded in OpenMP Evaluation section of this report. Figure 5 shows a good reduction in time for HashMap and Reduce steps with increasing threads. However, the step for reading lines from files shows no improvement. Moreover, the timing degrades in some cases. We conclude this is due to I/O read happening serially and some cache misses while trying to read with multiple threads which is causing high latency while reading files.

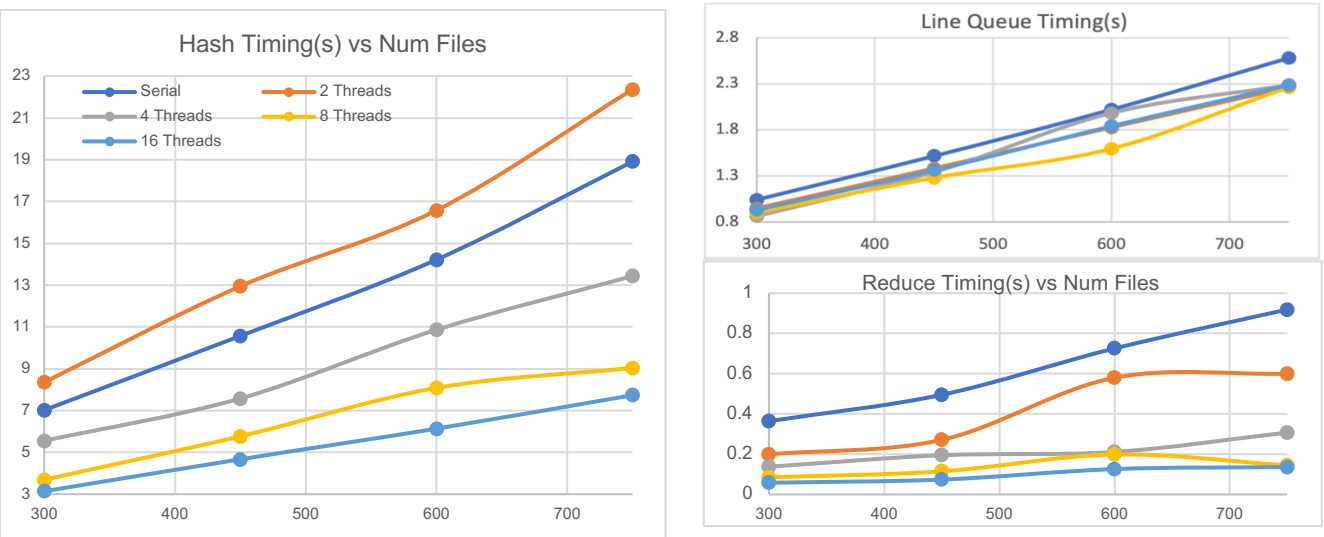

Figure 5 OpenMP Line Queuing, Hashing and Reduce timings with various threads

2.2. Parallel Execution of Reader-Mapper functionality

The drawback of the naïve approach is that we are first completing reading and then proceed to mapping. These steps are serial in nature. Therefore, we are using all the threads available exclusively for reading and subsequently for mapping. We can redesign the SPMD program such that some threads read and some threads map at the same time, leading to less load imbalances and speeding up the overall process. Furthermore, we would be using half the number of queues and hash maps for this implementation, which will reduce the memory footprint.

The Parallel Reader-Mapper functionality described above was also implemented using a single queue for storing the lines of all the files by multiple threads (As given by the used input). While the lines get populated to the queue, the Mapper threads will extract the lines from the queue and start mapping to hash tables simultaneously. This was implemented using 'tasks' with 'nowait' directive. However, this functionality led to an increase in the runtime with performance degradation. We suspect this is due to excessive locking by the Mapper threads on the Lines Queue causing a bottleneck. Therefore, we didn't proceed to evaluate this approach. Implementation of *MPI w/+ OpenMP*, we use the naïve implementation of OpenMP for evaluation.

3. MPI w/o and w/+ OpenMP Implementation

3.1. MPI w/o OpenMP Implementation

Naïve implementation of MPI program is simple in the sense that each MPI process will only have one thread of execution. The process of naïve implementation is as follows;

1. Master process (process 0) will read the names of all the files in the directory and divide the number of files evenly among 'P' processes. Then, it concatenates the file names to a big character array which is sent to other processes serially.
2. Once this chunk of file names is received by each process, it follows the same steps as explained in the serial execution of the word count problem.
3. Once every process finishes the local reduction, they send the ***first 1/Pth portion*** of the local hash table to 0^{th} process. Similarly, other processes send the ***second 1/Pth portion*** of the local hash tables to the 1^{st} process which is responsible for reducing the final word counts of the words that has hash values between [N/P, 2N/P] section. N is the hash table size.

3.2. MPI w/+ OpenMP Implementation

While we get enough speed up with single threaded MPI execution, we can get more speed up using multi-threaded MPI implementation, which we will explain below.

Once files have been received by each process, each process will spawn multiple threads inside each process to speed up the reading, mapping and local reduction steps. Figure 4 Explains the execution of MPI w/+ OpenMP implementation.

4. Evaluation and Analysis of Executions

4.1. OpenMP Evaluation

Figures 6, 7 and 8 show the variations in Run-times, Speedups and Efficiency metrics for each of the steps in OpenMP program for varying number of files. The timings are recorded for 150, 300, 450, 600 and 750 files. We see that as the number of files increase, the timings increase roughly linearly. We make the following observations on each step.

1. The Read step shows no speedup for the same number of files with increasing number of threads as I/O read is happening serially. In some cases, timing degrades which we think is due to cache misses which is causing high latency.
2. Map step shows a speedup of only **~3X** with 16 threads. This is maybe due to cache misses.
3. The Reduce step has low execution time to start with. But gives a very good speedup of ~9X for 600 files. The speedup here is seen to increase as the number of files are increasing. As the time taken to reduce the results is small to start, Overhead costs of creating more threads dominate and we get more speedup by increasing the input data size.

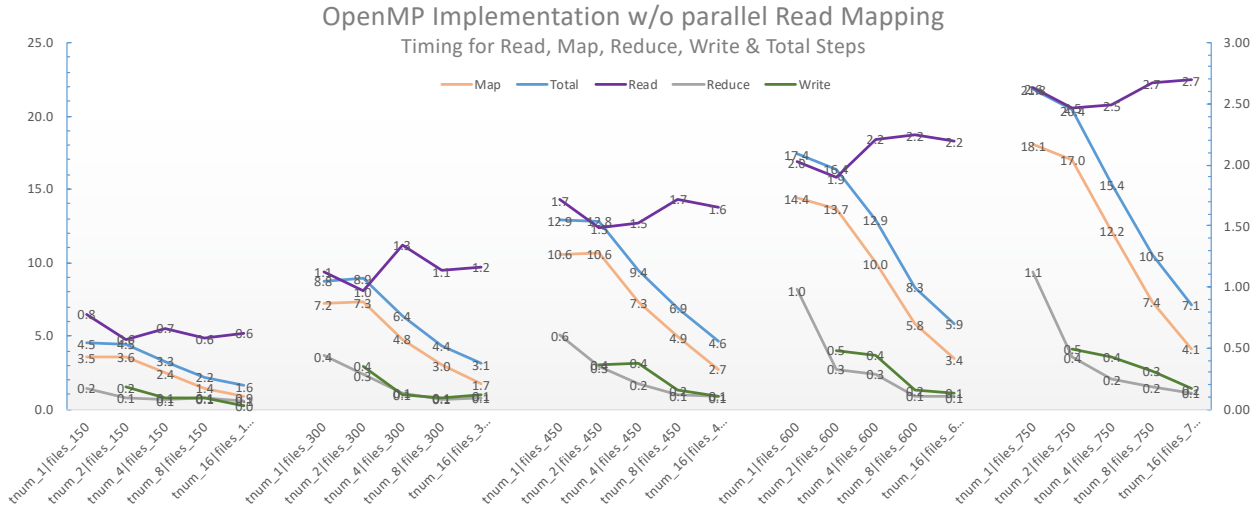

Figure 6 Timings for various steps in the OpenMP program across varying number of files. X axis contains information about the number of threads and the number of files used.

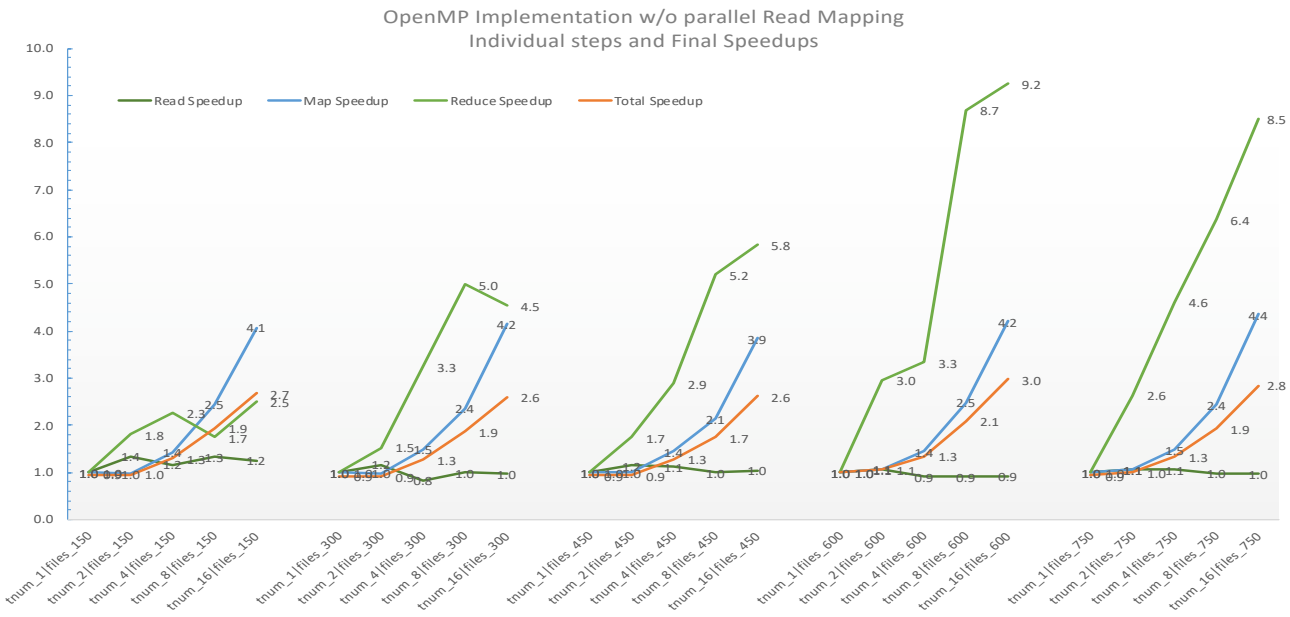

Figure 7 Speedups for various steps in the OpenMP program across varying number of files

The efficiency and Karp-Flatt metrics are shown below for the OpenMP Program. The efficiency is calculated as the amount of speedup per thread, and we expect it to be close to 1 to denote effective usage of the requested resources. Here, we see that the efficiency of OpenMP code goes to around 20% with 16 threads. This is due to the bottlenecks discussed above.

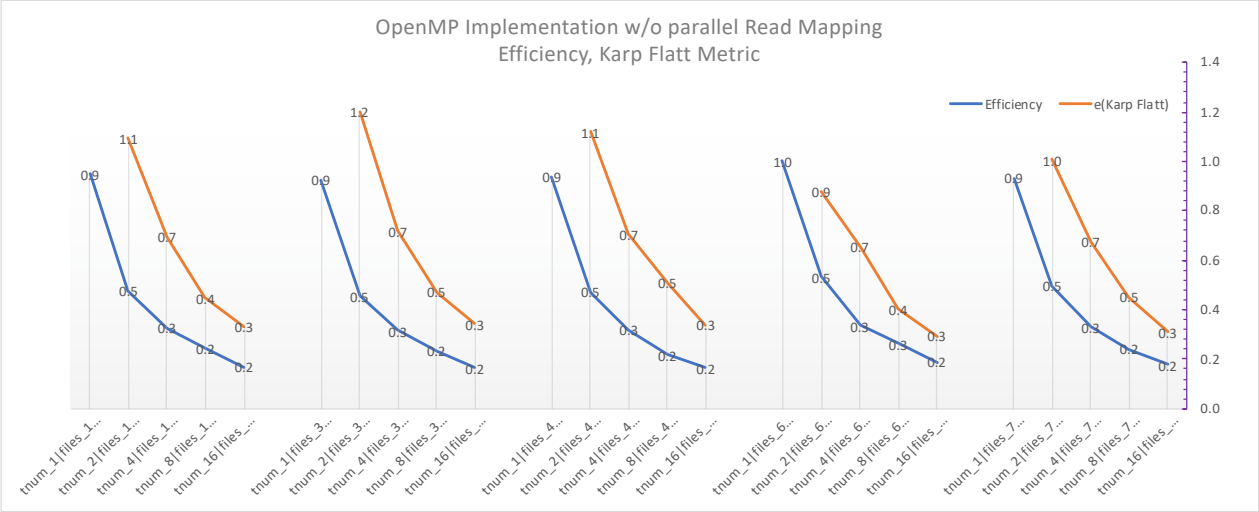

Figure 8 Efficiency and Karp-Flatt Metric for OpenMP program across varying number of files

4.2. MPI Evaluation

4.2.1. MPI w/o OpenMP

As shown in Figure 9, we see that the program has more speedup with increasing number of processes. The program is a pure implementation of MPI without OpenMP. We observe that the execution time goes up with the file count as expected but the final speedup is almost same for different number of file counts.

The Karp-Flatt metric is decreasing and in accordance with the speedup achieved. Therefore, the reader can argue that more speedup is possible with more processes until it reaches a maximum due to contentions.

P	1	2	4	8	16
Total	18.24	9.75	5.23	2.80	1.64
Speedup	0.96	1.79	3.34	6.23	10.10
Efficiency	0.96	0.89	0.83	0.78	0.66
e	-	0.12	0.07	0.04	0.03

Table 1Karp-Flatt Metric for MPI w/o OpenMP

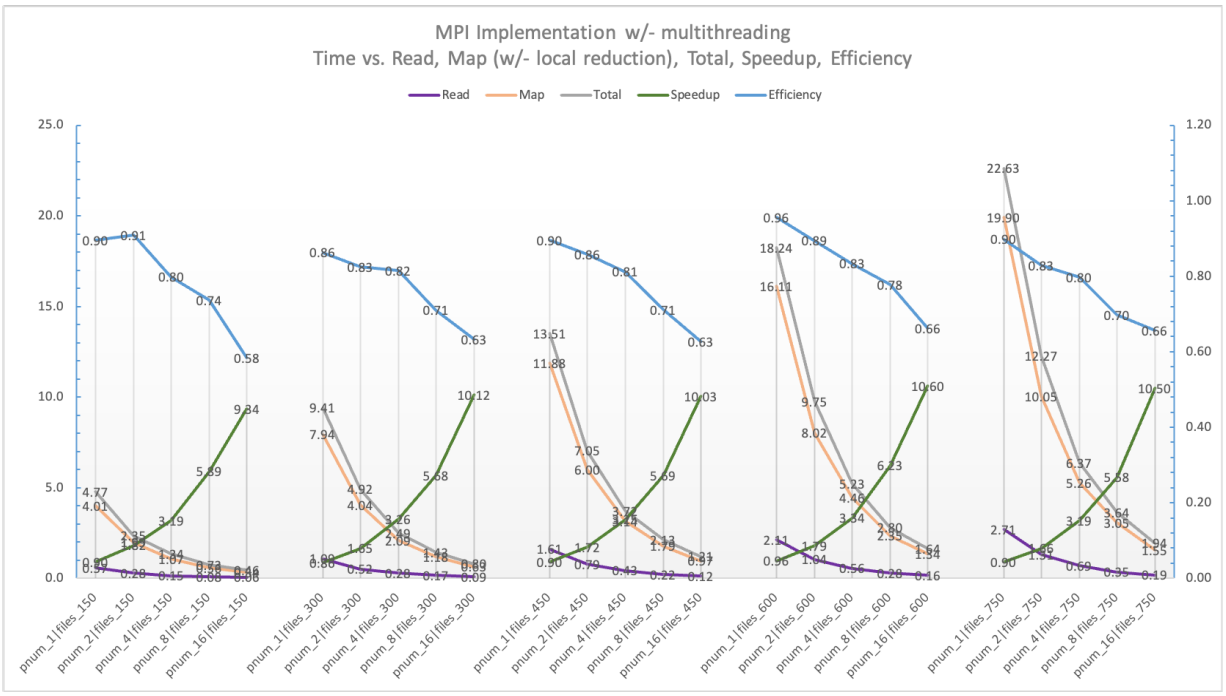

Figure 9 MPI Execution w/- multithreading. The figure contains read, map, total, speedup vs. time on the left vertical axis and efficiency vs. time on the right vertical axis. X axis contains information about the number of processes and the number of files used.

We see a steady increase in speedup and a steady decrease in efficiency. Both the reading time of the files and mapping without local reduction is in line with the speedup achieved. Since the graph has an upward slope even with 16 processes, we could expect to achieve even higher speed-up with increased number of processes.

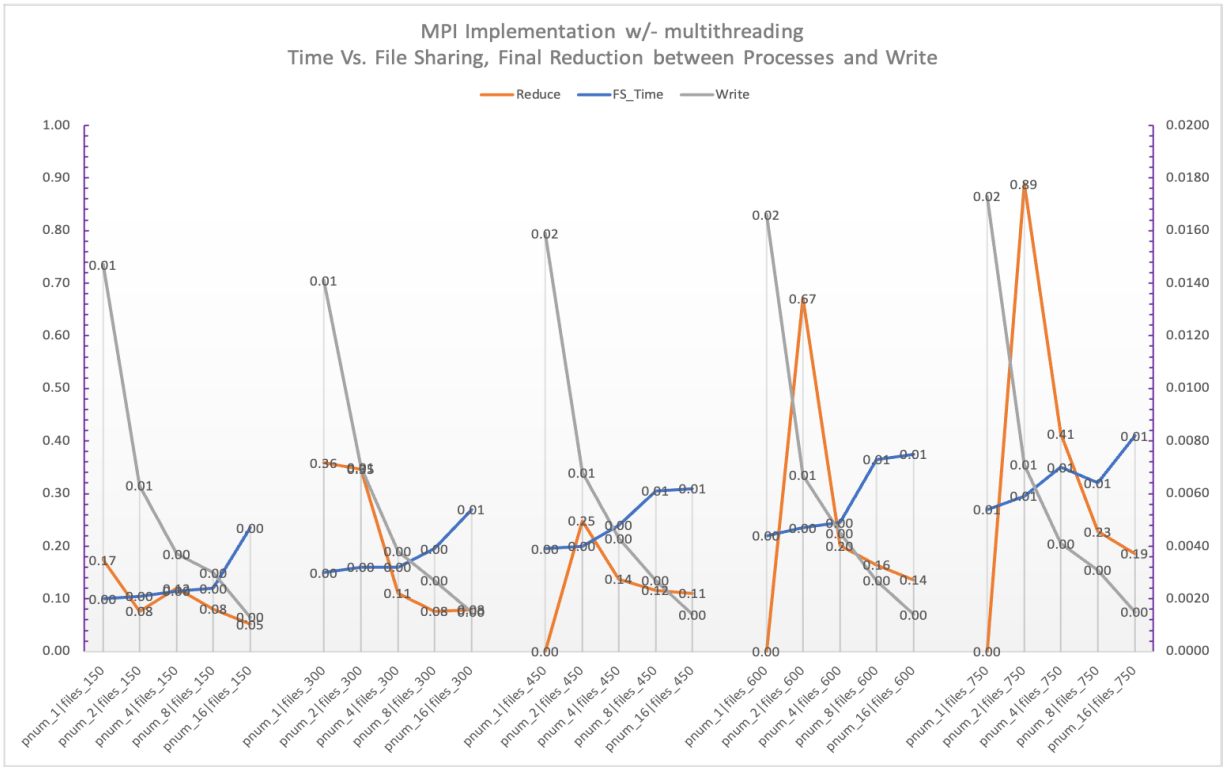

Figure 10 MPI Execution w/- multithreading. The figure contains final reduce between processes vs. time on the left vertical axis and time taken for file allocation to the processes at the beginning, write vs. time on the right vertical axis. X axis contains information about the number of processes and the number of files used.

It can be seen that file name sharing at the beginning of the execution, final reduction after each process has completed their respective local reductions and writing the counts in to files takes much less time compared to read and map time. Therefore, the timing for these steps is shown separately in figure 10

4.2.2. MPI w/+ OpenMP

According to Figure 11 we can see that we are getting more speedup with increasing number of processes and threads, the Karp-Flatt metric is going down when we increase the number of processes and threads. We notice that majority of the time is occupied by reading and mapping functionalities.

From Figure12 we see that file sharing, writing, local reduction, final hash table data word-count pair sharing, and final reductions take much less time compared to the reading and mapping.

Processes	1	1	1	1	2	2	2	2	4	4	4	4	8	8	8	16	16
Threads	2	4	8	16	2	4	8	16	2	4	8	16	2	4	8	2	4
Total (P*T)	2	4	8	16	4	8	16	32	8	16	32	64	16	32	64	32	64
Time	16.0	9.6	5.7	3.4	8.7	5.0	2.9	1.8	3.9	2.6	1.6	1.1	1.6	1.0	0.7	1.4	0.8
Speedup	1.09	1.83	3.05	5.20	2.00	3.53	6.06	9.60	4.48	6.82	11.19	16.09	10.92	16.95	25.41	12.38	21.00
Efficiency	0.54	0.46	0.38	0.33	0.50	0.44	0.38	0.30	0.56	0.43	0.35	0.25	0.68	0.53	0.40	0.39	0.33
e	0.83	0.4	0.23	0.14	0.33	0.18	0.11	0.08	0.11	0.09	0.06	0.05	0.03	0.03	0.02	0.05	0.03

Table 2: Karp-Flatt for MPI w/+ OpenMP

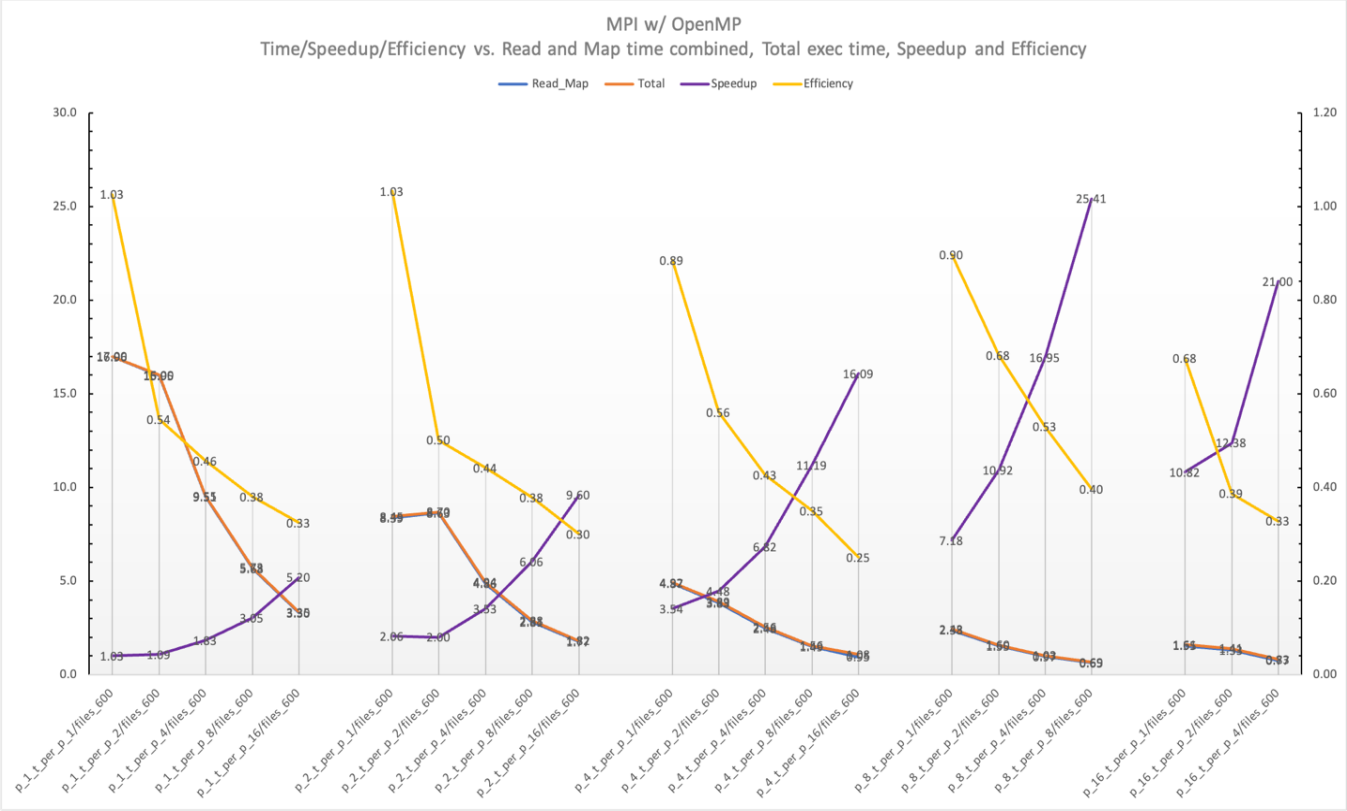

Figure 11 MPI Execution w/ multithreading. The figure contains read and map, total, speedup vs. time on the left vertical axis and efficiency vs. time on the right vertical axis. X axis contains information about the number of processes, threads and the files used.

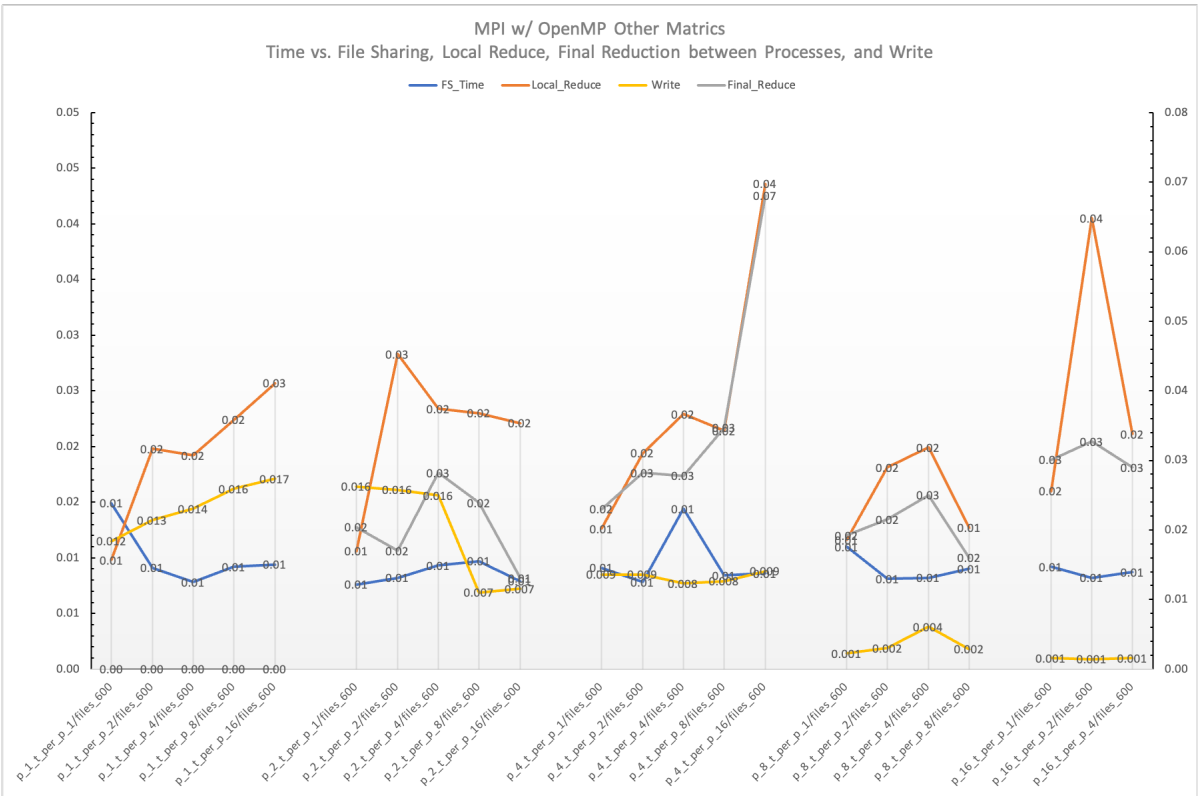

Figure 12 MPI Execution w/+ multithreading. The figure contains final reduction between processes w/ many threads vs. time on the left vertical axis and time taken for file allocation to the processes at the beginning, write vs. time on the right vertical axis. X axis contains information about the number of processes, threads and the number of files used.

5. Discussion, Performance Bottlenecks and Improvements

5.1. OpenMP

1. Further Reduce Load Imbalances: We have used many queues in the OpenMP implementation. By using only one reader queue, we can have better controllability over the number of reader and mapper threads that can be used. However, proper locking should be implemented to not encounter excessive locking.

2. We dynamically allocate files to the threads, but load imbalance is unavoidable due to different lengths of the files. The below table lists minimum and maximum execution times in different threads for a file count of 450. Here, the maximum time is reducing with increasing number of threads but the gap between max an min is increasing which worsens the overall performance/speedup.

# threads	2	4	8	16
Max (s)	11.4062	7.4442	3.9986	2.4317
Min (s)	11.3449	7.3756	3.8500	2.2438
Diff (s)	0.0613	0.0686	0.1486	0.1879
Diff (%)	0.54%	0.93%	3.86%	8.37%

3. We see that most of the execution time is spent on Mapping. We believe this is a result of using a queue data structure to dynamically store character array “word lines”. Since multiple queues are populated by many threads, data points may not be contiguous in the memory. Since cache miss rate is high when mapping processes read data from discontinuous memory locations, the algorithm speedup and efficiency is compromised.

5.2. MPI

1. Execution times were different when the number of nodes used in the batch script is different. This may be due to caching effects when many nodes are used vs. only one node is used. This maybe a result of having more L3 cache when more nodes are used.

2. File sharing happens at the very beginning and each process gets equal number of files. Therefore, the number of files processed by each process is static for a given number of input files. This leads to load imbalances. We believe the

code could be improved by dynamically requesting the files from the master process. We believe that this will impact the performance depending on the lengths of the input files.
For example, the first process may get 10 files with 2k lines while the 2nd process may get 10 files with 200k lines. The same can be achieved by randomly shuffling the files before sharing the files with other processes. Since reading and mapping takes majority of time in the execution, we collected the execution times of each thread and each process for those.
The table lists timing for 450 files with different number of processes with only one thread in each process. Max min time difference is increasing with the number of processes, and the percentage difference is getting much larger than the increase itself, which worsens the speedup.

# Processes	2	4	8	16
Max (s)	6.1993	3.1271	1.6131	0.9931
Min (s)	6.1218	3.0364	1.5088	0.7909
Diff (s)	0.0775	0.0907	0.1043	0.2022
Diff (%)	1.27%	2.99%	6.91%	25.5%

3. All the points discussed under OpenMP applies here as MPI w/+ OpenMP program runs an OpenMP threads inside each process.
4. Final data sharing between processes happens in a serially. First all the processes will send data to process 0, then all the processes send data to process 1, etc. Since this process happens serially, it is not the ideal maximum speedup achievable.
5. We do a local reduction inside each process before sending data to other processes. This dramatically decreases the communication overhead. For example, sending [{"word", 1}, {"word",1}, etc.] length 100 array one by one vs. sending just 1 {"word", 100} results in much higher speedup. We believe our code is able to achieve a higher speedup since this strategy is used.

6. References

1. Word frequency data. Accessed on: Feb 15, 2021. [Online]. Available: https://www.wordfrequency.info/uses.asp
2. Queue – Linked List Implementation. Accessed on: Feb 15, 2021. [Online]. Available: https://www.geeksforgeeks.org/queue-linked-list-implementation/
3. Getopt – Passing string parameter for argument. Accessed on: Feb 15, 2021. [Online]. Available: https://stackoverflow.com/questions/17877368/getopt-passing-string-parameter-for-argument
4. OpenMP: Divide all the threads into different groups. Accessed on: Feb 28, 2021. [Online]. Available: https://stackoverflow.com/questions/25556748/openmp-divide-all-the-threads-into-different-groups
5. MPI_Init() vs MPI_Init_thread(). Accessed on: April 28, 2021. [Online]. Available: https://stackoverflow.com/questions/34851903/mpi-init-vs-mpi-init-thread