Parallel Programming Machines
Map Reduce

Aakash Saravanan
Sahil Jaganmohan

Introduction

MapReduce is a model used to parallely process input sets and generate big data sets. Two main processes are involved in this model: *map* step to parse input sets and sort into various groups, and *reduce* step to perform a summary operation. In this implementation of MapReduce, input files are parsed to retrieve word counts across all input files.

To perform a parallel MapReduce, the input files are spread across processors where each process is responsible for either reading/parsing input files, mapping key/word pairs to respective reducer queues, and reducing word pair counts across processes.

MapReduce is implemented sequentially, using OpenMP to spread work across processes, and using MPI with OpenMP to run the OpenMP version across different nodes. With these implementations, performance analysis and isoefficiency analysis is performed to understand possible bottlenecks, efficiency of parallelism, and areas of future optimizations.

Sequential Implementation

The sequential implementation of MapReduce follows a standard process to concatenate several input files to generate a combined word frequency table. The process is split up into 3 main sequential parts that execute after the previous has completed. The parts include text parsing, handled by *readers*, word mapping, handled by *mappers* , and finally a word frequency reduction, handled by *reducers*.

A more detailed process follows shown pictorially in Figure 1:

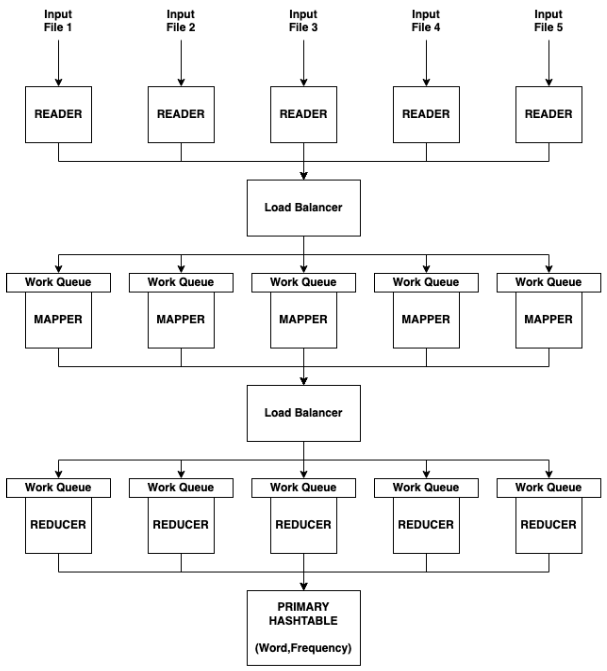

Figure 1:
MapReduce Sequential Implementation (15 iterations with 5 input files). The implementation can be scaled to any number of iterations where each iteration is defined as a reader, mapper, or reducer block.

MapReduce OpenMP Implementation (15 Threads with 5 input files). Each block defined is handled by a single thread. The program can be scaled to handle multiple files and can be scaled to various distribution of threads amongst processes.

Readers

Input files are parsed individually in a sequential fashion, reading in word by word from the files into a static sized buffer. Each word is processed to remove extraneous characters, and is pipelined into a corresponding queue for the next process' word mapping. Work queue load balancing was implemented but is not critical on sequential implementation. By distributing the same amount of slack for each queue, inserting the word on the least occupied queue.

It can be seen that each iteration takes a relatively equal amount of run time; however the total run time across all *mappers* will still remain the same regardless of load balancing as shown in Figure 2.

Number of Mappers	File Size (words)	Average Run Time (s)	Standard Deviation
5	47,865	0.0007822	1.6821414922651E-5
5	236,525	0.004342	1.8756E-9
5	2,835,000	0.0626788	0.0008692008743668

Figure 2: The run time and deviation of run between mappers for a variety of file sizes while the number of mapper iterations remain constant portraying expected load balancing

Mappers

Post text parsing begins the more arithmetic heavy portion of MapReduce where *mapper* iterations retrieve a word from their relative work queue, designated by iteration number. The word is then run through a hashing function to generate a corresponding hash number for it to be placed onto a reduction work queue. Once again load balancing was implemented to distribute words to each work queue. Words were placed in certain *reduction* queues each handling defined ranges based on the arithmetic hash code. The distribution demonstrates an even distribution across all the *reduction* queues as shown in Figure 3.

Number of Reducer Queues	File Size (words)	Average Queue Size (words)	Standard Deviation
5	236,525	47,118	8,271
5	2,835,000	565,450	114,591
10	473,050	47,118	10,098
10	5,670,000	565,450	152,975

Figure 3: Size of Reduction Queues and Deviation given different number of threads

Reducers

The final *reduction* stage is handled by different iterations, the process includes retrieving a word from the work queue, calculating the hash code, and inserting it onto the hashtable. Each insertion is stored as (word,count) to efficiently retrieve the final word frequency of all the input files combined. Since each iteration will handle words based on defined ranges, each *reducer* will handle a certain portion of the hashtable. The effects of load balancing can be seen in Figure 4 demonstrating the balancing of run time across *reducer* iterations.

Number of Reducers	File Size (words)	Avg Run Time (s)	Standard Deviation
5	236,525	0.0095008	0.0008139195046194
5	2,835,000	0.123196	0.0046195013150772
10	473,050	0.0106196	0.0012546168498789
10	5,670,000	0.2062479	0.090025634236533

Figure 4: Reducer average run time and std. deviation given different file sizes

Hash-Table Design

The hashing and hash table design is a crucial portion of MapReduce in order to reduce the number of collisions and increase efficiency in work distribution of *reducers*.

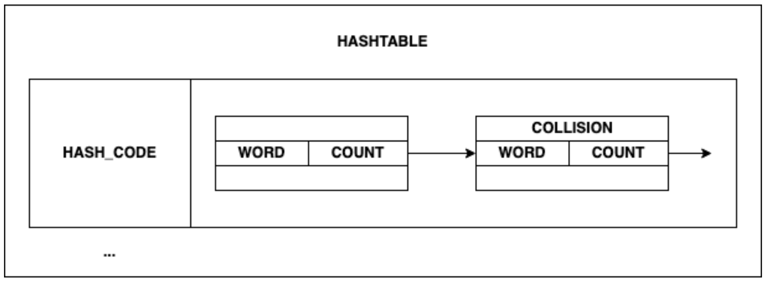

Figure 5: Hash-Table Design. The diagram shows an example for a hash code to word pair with a hash code collision.

The design is as follows:

Each word goes through a 8-bit hashing function that XORs sets of two characters producing a single hash value, giving a max 8-bit hash value of 65536. The hashtable is designed to store a linked-list for each hashcode where the length is dependent on the collisions. As shown in Figure 5 each node within the linked list contains a word and the relative count, this was the chosen design to make the output generation to be more efficient as the final hashtable contains all the word frequencies. The hashtable word insertion process accounts for three unique scenarios:

1. No Collision: Insertion of a node at the head of the corresponding linked list

2. Collision:
    a. Insertion of node at the tail of the corresponding linked list
    b. Count Incrementation of an existing node within linked list

The complexity of insertion into the hash-table is currently at O(n). The collisions to data ratio, the number of collisions per 1000 words, We have marked this search + insertion time as a possible speedup of our implementation based on data collected for a variety of file sizes shown in Figure 5. We can evidently improve our run time and the time spent running sequentially to insert elements into our hash-table in future updates.

OpenMP - Implementation

Utilizing parallel threads and processes, we are able to scale our MapReduce solution to larger file sizes and multiple files. Our implementation takes advantage of multiple threads on a single node to simultaneously parse through files, distribute words to work queues and reduce these multiple files to a final hash-table.

The design implemented parallelism by spawning threads for each of the processes in Figure 1. For a designated system, the number of threads can be predefined and the allocation of threads per processes. For a 15 threaded system we originally designate 5 *parser* threads, 5 *mapper* threads, and 5 *reducer* threads. As a contrast from the sequential implementation, each worker queue, mapper and reducer, can be accessed simultaneously from each of the parallel threads. When instantiated the threads do contain a shared memory for the two worker queues, and the final hash-table. However a main complication that occurs with having shared memory are the possibilities of race conditions. In order to prevent race conditions in our design we use locks on all shared memory structures. The load balancing logic between the *parser* to *mapper* and *mapper* to *reducer* allows each thread to access all other logic queues and therefore for each insertion on the queue, the current thread must acquire the lock beforehand and release once the insertion is completed. Finally each *reducer* thread must acquire a lock for the overarching hash-table in order to place a newly created word pair generated from the *reducer* queue. The next section will discuss the benefits of transitioning to a parallel implementation and the run-time improvement.

OpenMP - Complexity Analysis and Optimizations

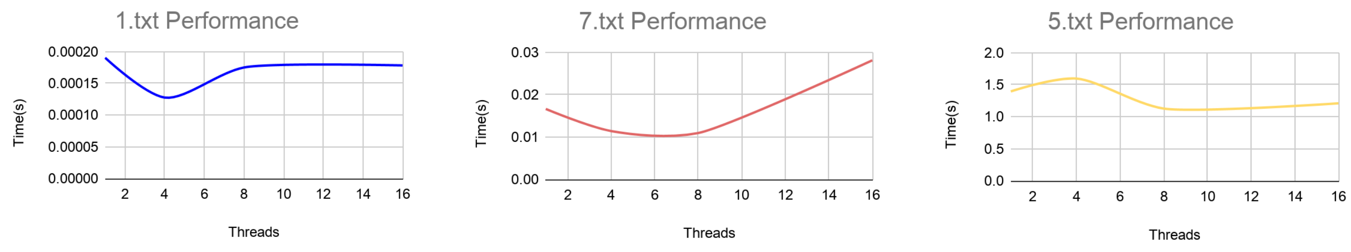

Figure 6: Time Performance of OpenMP with variable threads and multiple file sizes: 1.txt(47,865 words), 7.txt(236,525 words), 5.txt(2,835,000 words)

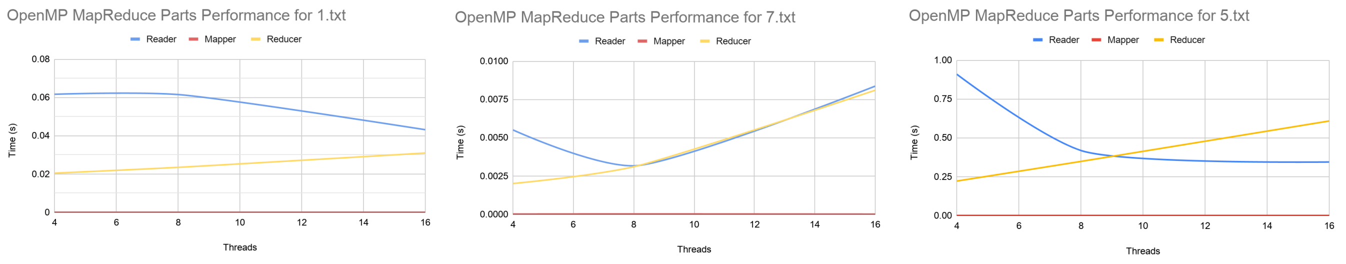

Figure 7: Time Performance of each part of MapReduce using OpenMP with variable threads and multiple file sizes: 1.txt(47,865 words), 7.txt(236,525 words), 5.txt(2,835,000 words)

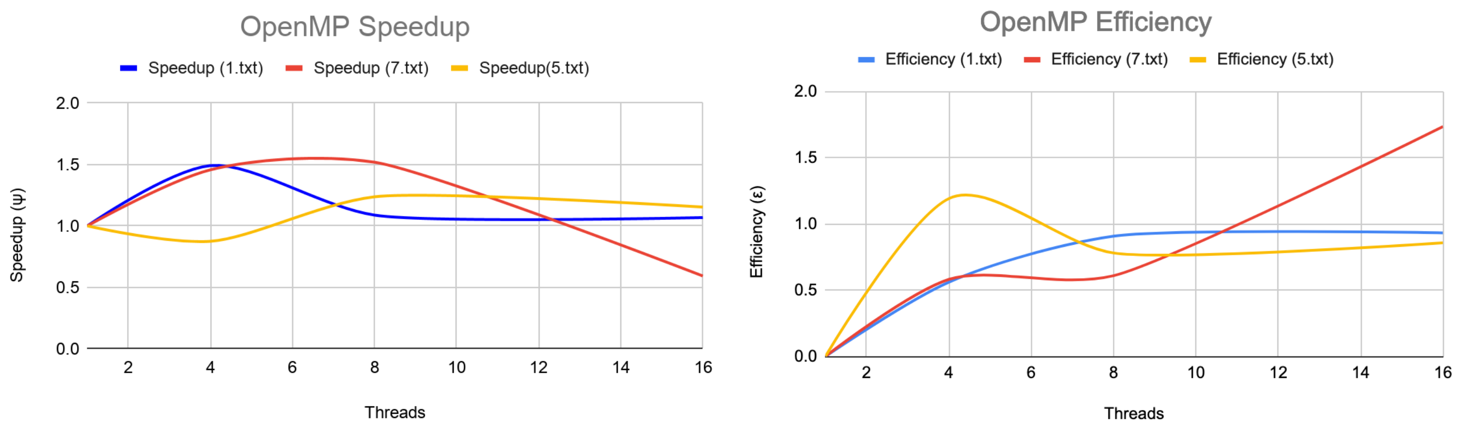

Figure 8: Karp-Flatt metrics - Speedup (Ψ) and Efficiency (ε) of OpenMP amongst variety of files sizes: 1.txt(47,865 words), 7.txt(236,525 words), 5.txt(2,835,000 words)

The run time analysis demonstrates that the parallelism of OpenMP allows for a much faster and efficient operation compared to the sequential MapReduce. The run times shown in Figure 6 and the speedup (Ψ) shown in Figure 8 shows how parallelism outputs faster than the sequential program. Using Karp-Flatt analysis we are able to see the communication effects and parallel overhead cost making the program in-efficient as expected. As seen in the largest file 5.txt as we increase the number of threads for reading, mapping, and reducing we generate a notable speedup when we reach 8 threads where processes are parallely processing information while our efficiency increases from 8 threads to 16 threads. As shown in Figure 7 we can also see that the time for each MapReduce process decreases as the number of threads available increases. Overall we can say that our program is able to parallely process large and multiple files efficiently.

We are able to generate these speedups for our OpenMP implementation because we are able to parallely read several files at once compared to a single file being read at a time sequentially. In addition we are also able to map and reduce several queues at the same time. Both of these reduce the amount being executed sequentially and as a result dramatically increases the program run time and efficiency. As shown in Figure 8 the karp-flatt metric graphs display that the implementation is able to handle larger files more efficiently than the smaller files. Along with that and the efficiency graph displaying reduction in serial execution we can conclude our implementation is highly scalable among large and multiple files

However we are able to identify bottlenecks and areas that can improve our speedup and efficiency for our multi-processor program. A big area is being able to acquire locks for different work queues for reading or writing. Therefore an optimization that would be implemented would include writing information to the queues in large quantities. By doing so the overhead of lock acquisition can be reduced drastically and increasing parallelism. Another big bottleneck includes being able to read from the file and place the word in the corresponding queue which, reading a single word from each file is extremely inefficient and increases run time significantly. Therefore the designed solution includes buffer reading reducing the amount of total reads and by doing so reducing run times for all programs shown in Figure 6. Lastly the biggest major bottleneck includes that the *mapper* threads are waiting for *reader* threads to finish as well as the *reducer* is waiting for *mapper* to finish. If this waiting is removed and each thread works independently we can increase our speedup and efficiency dramatically.

MPI w/ OpenMP - Implementation

Our final addition to our implementation includes enhancing our single node parallel implementation to a multi-node parallel implementation. This addition designates each node to handle certain reduction functionality while also handling a much larger input size. This design is set to read multiple files simultaneously and develop partial filled hash-tables from each node to form a complete hash-table.

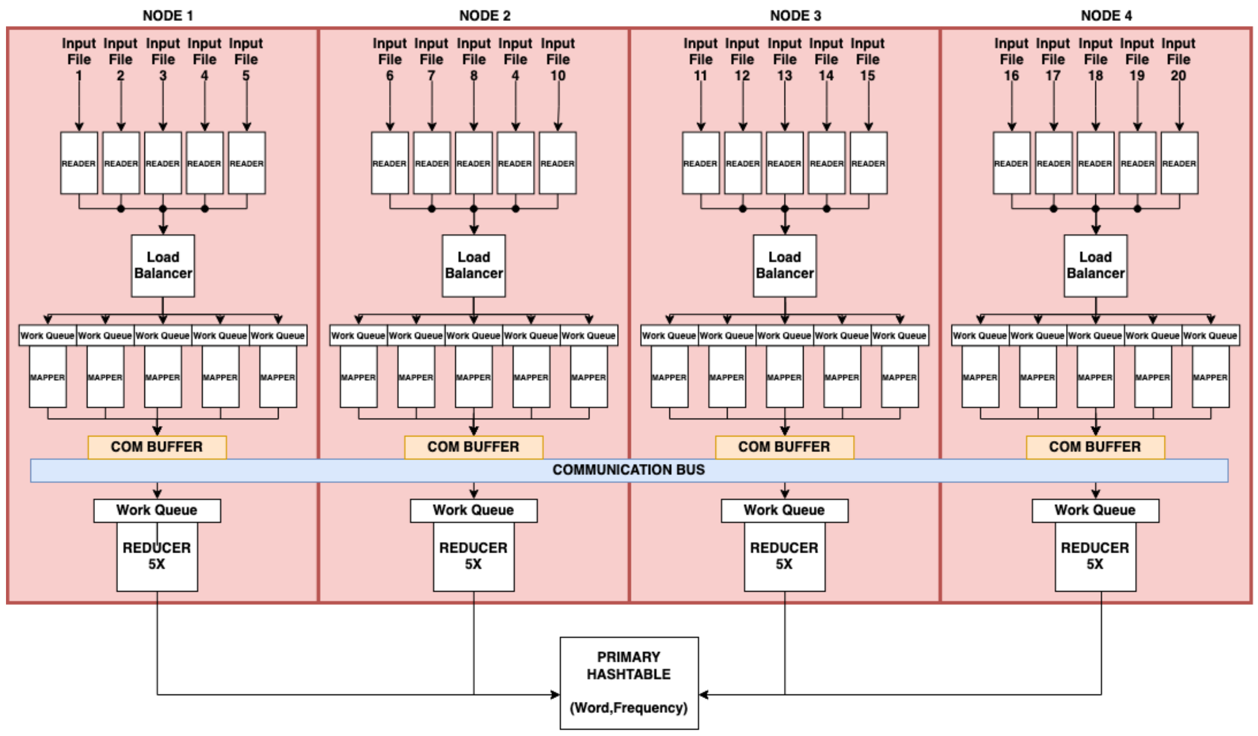

Figure 9: MapReduce MPI Implementation (4 Nodes, 15 Threads/Node, 20 Input Files)

As shown in Figure 9, the MPI implementation is designed to parallely process several input files at once. With the introduction of multi-node parallelism we treat phases 1 and 2 of our OpenMP process the same. Each node internally will spawn *reader* threads that will each take an input file and place them on mapper queues through a parallel load balancing logic. The *mapper* threads will retrieve words from the work queue and place them onto a communication buffer.

The communication buffer is designed to dynamically allocate for the total elements tasked to communicate from the current node to each of the other respective nodes. In order for the communication to take place between all nodes a *MPI_Barrier()* is placed after each *mapper* thread has populated the communication buffer to verify all mapping is complete. To instantiate the communication between all nodes, each node will generate a metadata structure to inform the other nodes of the incoming communication. An *MPI_allToall()* is used to send and receive the metadata from each node. Once the metadata is received, the node will dynamically allocate the required size of a data_buffer in order to handle the transaction. Finally to complete the data transaction an *MPI_allToallV()* is used to send the data from node to node. After the communication is completed the received data will be placed on the *reducer* work queues to be parallely processed to generate subsections of the final hash-table. Each node will handle certain ranges of the hash-table, dependent on the number of *reducer* threads per node. Therefore each node knows where to send the data received from the *mapper* work queue and will build a partial hash-table to be combined to produce the MapReduce.

MPI w/ OpenMP - Complexity Analysis and Optimizations

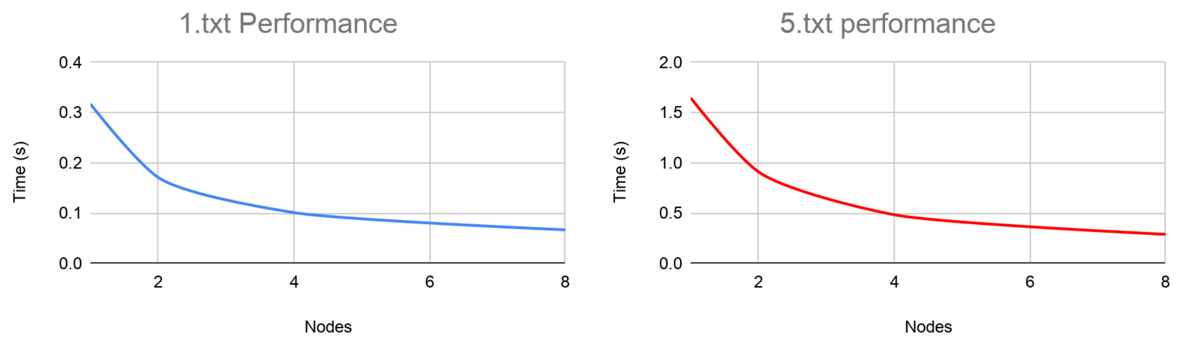

Figure 10: Time Performance of MPI w/OpenMP with variable threads and multiple file sizes: 1.txt(47,865 words), 5.txt(2,835,000 words)

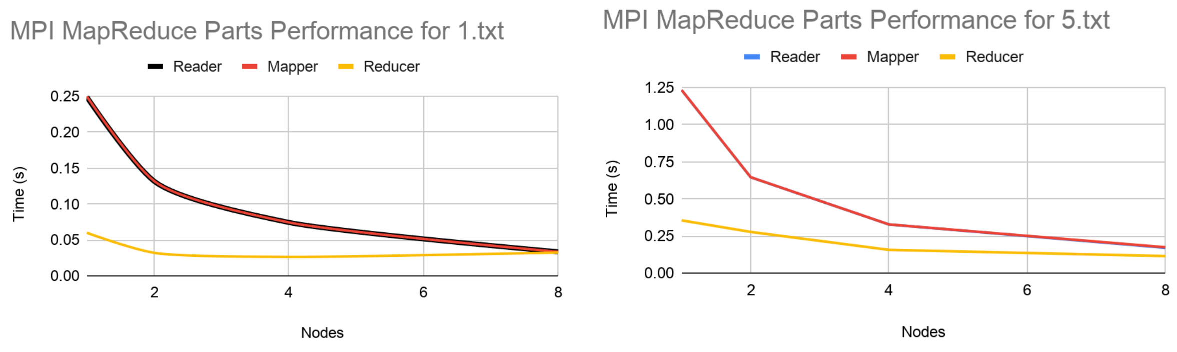

Figure 11: Time Performance of each part of MapReduce using MPI w/OpenMP with variable threads and multiple file sizes: 1.txt(47,865 words), 5.txt(2,835,000 words)

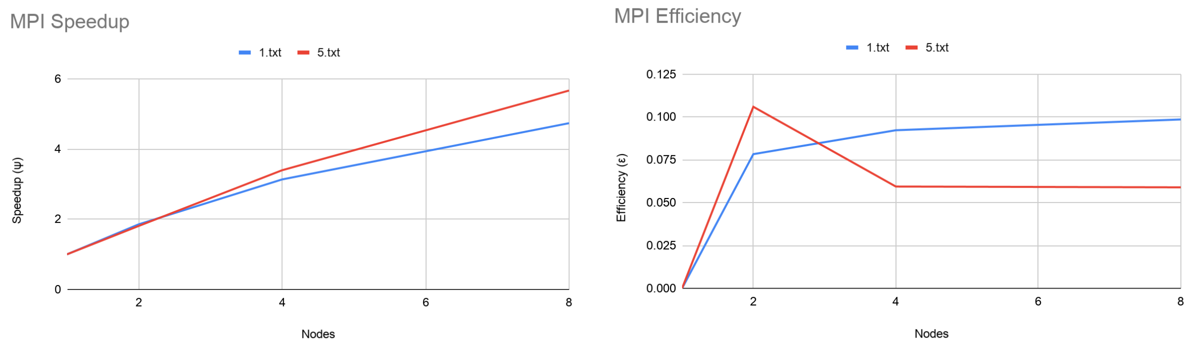

Figure 12: Karp-Flatt metrics - Speedup (Ψ) and Efficiency (ε) of MPI w/OpenMP amongst variety of files sizes: 1.txt(47,865 words),5.txt(2,835,000 words)

The run time analysis demonstrates that the parallelism of MPI w/OpenMP allows for a much faster and efficient operation compared to the sequential and the OpenMP MapReduce implementation. The run times shown in Figure 10 and the speedup (Ψ) shown in Figure 11 shows how running the OpenMP program across multiple nodes thru MPI enables better runtimes compared to running the OpenMP program on one node. Using Karp-Flatt analysis we are able to see the communication effects and parallel overhead cost making the program in-efficient as expected. As seen in the largest file 5.txt, as the number of nodes is increased, the speedups increase as well as the efficiency. As shown in Figure 11, we can also see that the time for each MapReduce process decreases as the number of nodes the OpenMP program runs on increases. From this, we are able to conclude that increasing the number of nodes the OpenMP program runs on outweighs the added communication and overhead costs associated with the MPI implementation.

We are able to generate these speedups for our MPI w/OpenMP implementation because we are able to parallely read several files at once on each node as opposed to reading several files on just one node. When considering Figure 12, it is important to note the efficiency difference between parsing the smaller 1.txt file and the larger 5.txt file. As the number of nodes used increases, the efficiency of parsing the 5.txt file decreases and is less than the efficiency of the 1.txt file. This can be attributed to the communication costs and overhead of communicating with each node. With a larger file, more data needs to be properly communicated across all nodes which adds to the run time. However, with the MPI w/OpenMP implementation, more files are able to be parsed and this outweighs the costs of communicating across nodes.

An area of optimization is improved communication amongst nodes to process the overall word count. In the current implementation, data that needs to be communicated must be first packed into an array of strings with each element of the array being one word (having word count of 1), communicated via MPI, and unpacked once the data has been received. A trivial optimization is to communicate words with their respective word counts. Another optimization is to utilize parallel programming techniques to have certain threads process data as it is being received and sent. Therefore streamlining the results as the wait times of threads and in turn nodes are decreased immensely.