1. **Executive Summary:**

This report analyzes the design decisions made during the creation of two iterations of a parallel word count algorithm. The base requirement was that the software designed had to be able to calculate word count across a variety of files. The initial version of the algorithm needed to be implemented to support a shared memory model using OpenMP. The second iteration of the program needed to be able to take the initial algorithm and using MPI spread the OpenMP algorithm across a variety of nodes. Both programs produce a single output file that contains word count for every word, and a total word count.

The OpenMP version performed better and had much more reliable speedup than the MPI version. The biggest difference between the two designs was the way communication was orchestrated. When using MPI, there can only be one MPI communication from a given node at a time. This makes sense, because a processor only has so much hardware available for actual communication on a network, but regardless this introduces a limitation into the number of simultaneous operations that can be performed.

Furthermore, due to the nature of sharing data between cores and sharing data across a network, queuing is much simpler and much faster when using OpenMP. Further details and performance metrics will be analyzed in the report that follows, but from a high level both algorithms operate in a very similar manner. Each design utilizes reader threads to pull data from files, mapper threads to generate a local word count, reducer threads to combine word counts across threads/nodes and finally writer threads to return the results.

**2.0 Algorithm Design:**

This section will break down the way that the word count of the files was calculated and explain the differences between the two versions.

**2.1 Types:**

Two parallel algorithms were developed. A shared memory model and a message-passing model that utilized shared memory on a given node. The second design pulled from the first for a given node, but added logic to make the original design able to scale past the initial node.

**2.1 Basis Steps**

1. Read all of the files
2. Parse each word and add to a queue.  
   (Clean each word so that case and other random punctuation marks do not skew results)
3. Process the queue to generate a count of each word, as well as a total word count.
4. Return the results to the user.

**2.2 OpenMP Summary**

OpenMP allows communication from any node at any time. This is because the communication can easily be done through memory, and no specific communication hardware is required (past basic cache coherence). Without heavy communication hardware hits, I came to the conclusion that all the threads should attack a problem and then move on to the next task. The result of this is that the number of cores is equal to the number of reader threads, then mapper threads, reducer threads, and finally writer threads.

**2.3 MPI Summary**

I used a very different approach when working on the OpenMP version. Initially I wanted the same approach as the OpenMP version, where each node would have a reader on each thread, then a mapper, and finally a reducer. Doing so would require a mass reduction at the end to communicate between nodes, but ultimately that would be a log2(nodes) hit and is affordable if the rest of the process is efficient enough.   
  
Unfortunately, after designing this system I learned that only one MPI communication operation could be performed at a time. This meant that I could not have massive reductions and mapping between nodes without having to have much more complicated logic to prevent deadlocks. Ultimately I decided that it would be simpler to have one reader on a node, and a sleeping mapper and reducer. The two sleeping threads would wake up when work existed for them, with the goal that those threads would complete at the same time that the reader threads finished or shortly thereafter, leaving the final step being communication between cores to reduce the final count.

**2.4 Thread Summaries**

**2.4.1 Reader Thread**

This thread requests a filename, then the master responds with the file that the thread should analyze. Should no thread exist, the reader thread would either switch focus and perform additional actions, or close itself.

The main goal of the reader thread is to process a given file. Processing a file means opening the file and reading every word in the file. Each word is then “standardized,” which means it is made lowercase and special characters are removed. The clean word is then added to a queue for further processing.

**2.4.2 Mapper Thread**

The mapper thread reads the queue generated by the reader thread and creates a local map of all the words, which the key being the word and the word count being the total number of that specific word read.

In the OpenMP version, the mapper thread reads from a single local queue. This is easy and will not cause any issues because the reader thread is complete before the mapper thread runs.   
  
In the MPI version, the reader thread actually creates two queues. If one queue is empty, the reader switches and starts to fill the other, and the mapper switches to the one the reader was just filling. This simple buffering allows for the reading and mapping to be processed completely in parallel.

**2.4.3 Reducer Thread**

The purpose of this thread is to take multiple hash maps (of word counts) and combine them into one. This allows for the final result to have accurate totals for each given word. While the overall total would be the same if you summed the hashes across each core, it would be possible to have a word listed multiple times in the output with values from each core. The desired output is for a word to be listed once with the summed value from every process.

In the OpenMP version, the mappers place a complete map from each local core in a given destination core. This ultimately means that a given core will be responsible for reducing part of the alphabet. The reducer thread then combines all of the hashes from each thread into a single hash with accurate and complete counts for some subset of the alphabet.

In the MPI version, mapping is much more complicated because it requires collective communication across all the nodes, and because MPI does not allow for a complicated datatype like a hash to be sent. Instead the MPI version uses a tree-like structure to reduce the hashes by combining them down the tree until the root node has a complete hash with every count.

**2.4.4 Writer Thread**

The goal of the writer thread is to take the reduced thread(s) and save them to memory so that the user can analyze the results. This also involves calculating a master total that is saved and printed to the output file after all of the words.

In the OpenMP version, each thread writes the subset of the alphabet that it was responsibly for back to memory.

In the MPI version, the master thread writes everything back to memory because at the end of the reduction the master thread has the complete data.

**2.5 Load Balancing**

In my testing I used 35 files with the file size ranging from 24 KB to 4.9 MB. Across that wide of a range, there is enormous opportunity for the disparity in sizes to negatively impact performance. For example, if the largest file was read last, it is quite possible that the rest of the application would be hung up waiting for it to complete. To combat this, the best method is to read the largest files first. If you have a large subset of files with a relatively balanced distribution, then starting with the largest files allows for the smaller files to fill in and correct for the load imbalance between threads.

It would be quite complicated to write a C++ application that fully integrated with the local file system to analyze files and their sizes in addition to the rest of the program, and it would further skew the parallelizable nature of the software. Instead, my solution was much simpler. I used existing command line tools to automatically generate a complete list of the files I wished to read, and then passed that list of files in as sorted command line arguments. My program simply read the files passed in, in the order the parameters were set. An example call to my program can be seen below:

echo $(ls -xS docs/\*.txt) | xargs mpiexec -n 16 ./mpi

There is additional load imbalance between the reducer threads caused by an uneven distribution of words within each subset of the alphabet. Using a more complicated hashing algorithm to divide the queues out could solve this load imbalance.

**3.0 Results**

**3.1 OpenMP**

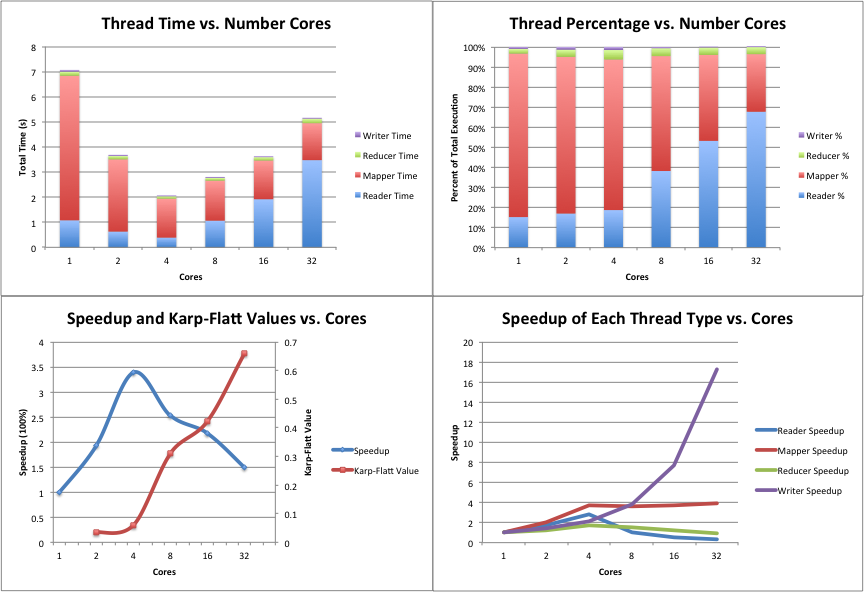
**3.1.1 Summary of Results**

The OpenMP program scaled well from 1 to 4 cores, then quickly saw a drop in performance after 4 cores were passed. When looking at the time breakdown of the programs, after 4 cores, the reader time drastically increased. This is likely because OpenMP runs locally with shared memory, and after 4 cores the cores began to compete for read head time for the different files. If a given thread is waiting for access to a computer’s read head, then not only will there be delays from the fact that the hard drive must be hit, but there will be even more delays from the complete loss of parallelization of the program. Reading is a hardware operation that occurs on one element at a time, and when it becomes the bottleneck, the entire performance will suffer.

The increase in sequential code is reflected in the Karp-Flatt values of the program. The values for two and four cores are both under 0.1, while 8 cores is around 0.3 and by the time you hit 32 cores, the value has risen to 0.66.

The total reducer and writer times remain relatively constant for each iteration of the program. The reducer time is going to be quite efficient because it is all local concatenation of hashes, and the writing time by nature will always be fully sequential. In the OpenMP version the write time will gain some overhead as additional cores hit barriers, but overall it is relatively constant. There is a growth in the time required to complete mapping however. This makes sense, as the mapping logic waits for all of the mapping to conclude before it progresses to make sure that nothing is lost in translation over the course of the program. However the loss of parallel speedup in this section of the program also contributes to the rising Karp-Flatt values. When analyzing speedup however, the writer times will see enormous growth, because there is no barrier between the proceeding reducer logic. When one reducer finishes it is immediately allowed to go on and write to the file. This means that the time waiting to write and the time required to write (because the subset of the alphabet a cores has to write) will both decrease.

**3.1.2 Data**

**3.2 MPI**

**3.2.1 Summary of Results**

**3.2.2 Data**