1.0 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 merge 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 allow the original design to scale past the initial node.

2.2 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.

2.3 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.

2.4 MPI Summary

I used a very different approach when working on the MPI 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.

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 simultaneously 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.

2.5 Thread Summaries

2.5.1 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.

2.5.2 Mapper Thread

The mapper thread reads the queue generated by the reader thread and creates a local map of all the words, with the word and word count functioning as the key and value, respectively.

In the OpenMP version, the mapper thread reads from a single local queue. This is simple and does not cause any sequencing issues because the reader threads are complete before the mapper threads run.

In the MPI version, the reader thread actually creates two queues of words. 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 performed completely in parallel.

2.5.3 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 total counted 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, reducing 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 an inverted tree-like structure to reduce the hashes by combining them down the tree until the root node has a complete hash with every count.

2.5.4 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, and the final writer thread also writes a total word count 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.

2.6 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 hang 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 are additional load imbalances 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 correct for this imbalance. As written, the queues are divided into cores simply based on the starting character in a word.

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. If any mapping is still occurring and a given core moves on, the words that were put in that cores queue after it moved on would be lost. Reduction requires the mapping to be complete.

One of the strongest aspects of this algorithm is the way that reduction and writing is intertwined. Because mapping stores values to a core's queue, when the reduction finishes there is a guarantee that that thread's subset of the alphabet is complete. This means that the node with a complete reduction can immediately write those reduced values to memory and move on. As the number of cores increases, the time required for a core to write, and the delay waiting to write will both decrease. This design naturally schedules write access in an efficient manner.

3.1.2 OpenMP Iso-Efficiency

Assumptions: *n = total number of words*

$$\boldsymbol{T_P = Time\ to\ Count\ Words + Time\ to\ Map\ Words + Time\ to\ Reduce + Time\ to\ Write}$$

$$\boldsymbol{Time\ To\ Count\ Words = \frac{n}{P} * t_{read}}$$

$$\boldsymbol{Time\ to\ Map\ Words = \frac{n}{P} * t_{hash}}$$

$$\boldsymbol{Time\ to\ Reduce = Time\ to\ Send\ Words + Time\ to\ Merge\ Received\ Hashes}$$

$$\boldsymbol{Time\ to\ Reduce = \frac{n}{P} * \left((P-1) * t_{send} + t_{merge-hash}\right)}$$

$$\boldsymbol{Time\ to\ Write = \frac{n}{P} * t_{write}}$$

$$\boldsymbol{T_P = \frac{n}{P} * t_{read} + \frac{n}{P} * t_{hash} + \frac{n}{P} * \left((P-1) * t_{send} + t_{merge-hash}\right) + \frac{n}{P} * t_{write}}$$

$$\boldsymbol{T_P = \frac{n}{P} * \left(t_{read} + t_{hash} + \left((P-1) * t_{send} + t_{merge-hash}\right) + t_{write}\right)}$$

$$\boldsymbol{T_P = \frac{n}{P} * \left(t_{read} + t_{hash} + t_{write} + t_{merge-hash} + (P-1) * t_{send}\right)}$$

$$\boldsymbol{t_{work} = t_{read} + t_{hash} + t_{write}}$$

$$\boldsymbol{T_P = \frac{n}{P} * \left(t_{work} + t_{merge-hash} + (P-1) * t_{send}\right)}$$

$$\boldsymbol{P * T_P = T_0 + T_1}$$

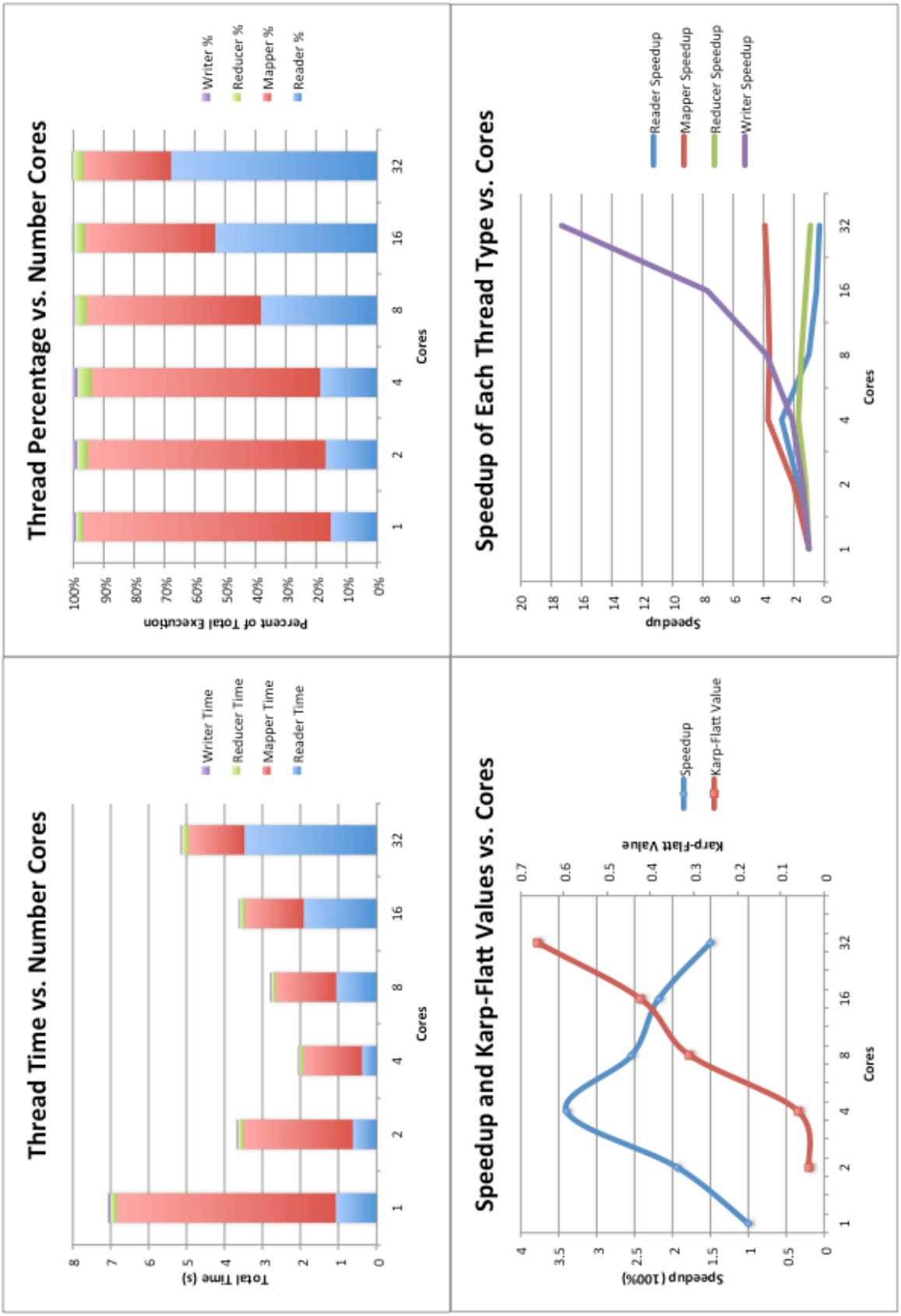
$$\boldsymbol{n * \left(t_{work} + t_{merge-hash} + (P-1) * t_{send}\right) = T_0 + T_1}$$

$$\boldsymbol{T_0 = n * \left(t_{merge-hash} + (P-1) * t_{send}\right)}$$

$$\boldsymbol{T_1 = n * t_{work}}$$

$$\boldsymbol{\theta\left(n * (P-1)\right) = Iso - Efficiency}$$

3.1.3 OpenMP Data



3.2 MPI
3.2.1 Summary of Results

In the graphs shown in the following section there is an interesting trend in the data. When studying the Karp-Flatt values, the general trend is decreasing serial code as the number of cores increases. Despite this trend, the actual speedup not increasing at a constant or even growing rate, instead the rate at which the speedup grows is decreasing. The answer to this problem can be found in the Reducer thread. The reducer thread is consuming an increasing amount of CPU time as the cores grow which makes sense due to the nature of a deepening reduction tree. Even though it is an efficient reduction, doing so causes a significant amount of idle time in threads. The steep growth of idle threads can be blamed for the decreasing speedup rates.

The trends seen in the mapper and reader threads are responsible for the speedup that is observed. The steady drop in real time spent in those tasks is based on the fact that those static operations are being spread evenly among an increasing number of cores.

In the MPI version, the reader threads are not competing for access to the hard drive's read head, because these threads are spread across distributed machines instead of all residing locally. The bottleneck in this case is not the memory hardware of the system, but is instead the communication hardware. In the MPI model, there is a master thread that is fully responsible for distributing the work (files) that need to be read. This thread manages the master list of files that need to be processed, and because only one MPI operation can be performed at a time, the thread can only service one node at a time.

To fix the bottlenecks caused by the reducer threads and actually take advantage of the performance gain created through the reader and writer threads, a much more intelligent communication model needs to be created. All of the hashes are sent manually, with a starting message, the word, and then the value. This is extremely expensive as a simple object is converted into a complex series of communications that has to deal with network traffic and various other induced latencies. To correct for this, the data needs to be sent in a larger package with much fewer requests to avoid paying communication costs multiple times. The easiest way to move to a proper implementation is to define custom MPI data structures, or use a library like Boost that has already done it instead of manually transmitting hashes.

3.2.2 Iso-Efficiency Analysis

Assumptions: *n = total number of words*

$$T_P = Time\ to\ Count\ Words + Time\ to\ Map\ Words + Time\ to\ Reduce + Time\ to\ Write$$

$$Time\ To\ Count\ Words = \frac{n}{P} * t_{read}$$

$$Time\ to\ Map\ Words = \frac{n}{P} * t_{hash}$$

$$Time\ to\ Reduce = Time\ to\ Send\ Words + Time\ to\ Merge\ Received\ Hashes$$

$$Time\ to\ Reduce = (3 * n * \log_2 P * t_{send}) + \sum_{i=1}^{\log_2 P} \frac{n}{2^i} * t_{hash}$$

$$Time\ to\ Reduce = (3 * n * \log_2 P * t_{send}) + n * \frac{P-1}{P} * t_{hash}$$

$$Time\ to\ Write = n * t_{write}$$

$$T_P = \frac{n}{P} * t_{read} + \frac{n}{P} * t_{hash} + (3 * n * \log_2 P * t_{send}) + n * \frac{P-1}{P} * t_{hash} + n * t_{write}$$

$$T_P = \frac{n}{P} * (t_{read} + t_{hash}) + (3 * n * \log_2 P * t_{send}) + n * \frac{P-1}{P} * t_{hash} + n * t_{write}$$

$$t_{work} = t_{read} + t_{hash}$$

$$T_P = \frac{n}{P} * (t_{work}) + (3 * n * \log_2 P * t_{send}) + n * \frac{P-1}{P} * t_{hash} + n * t_{write}$$

$$P * T_P = T_0 + T_1$$

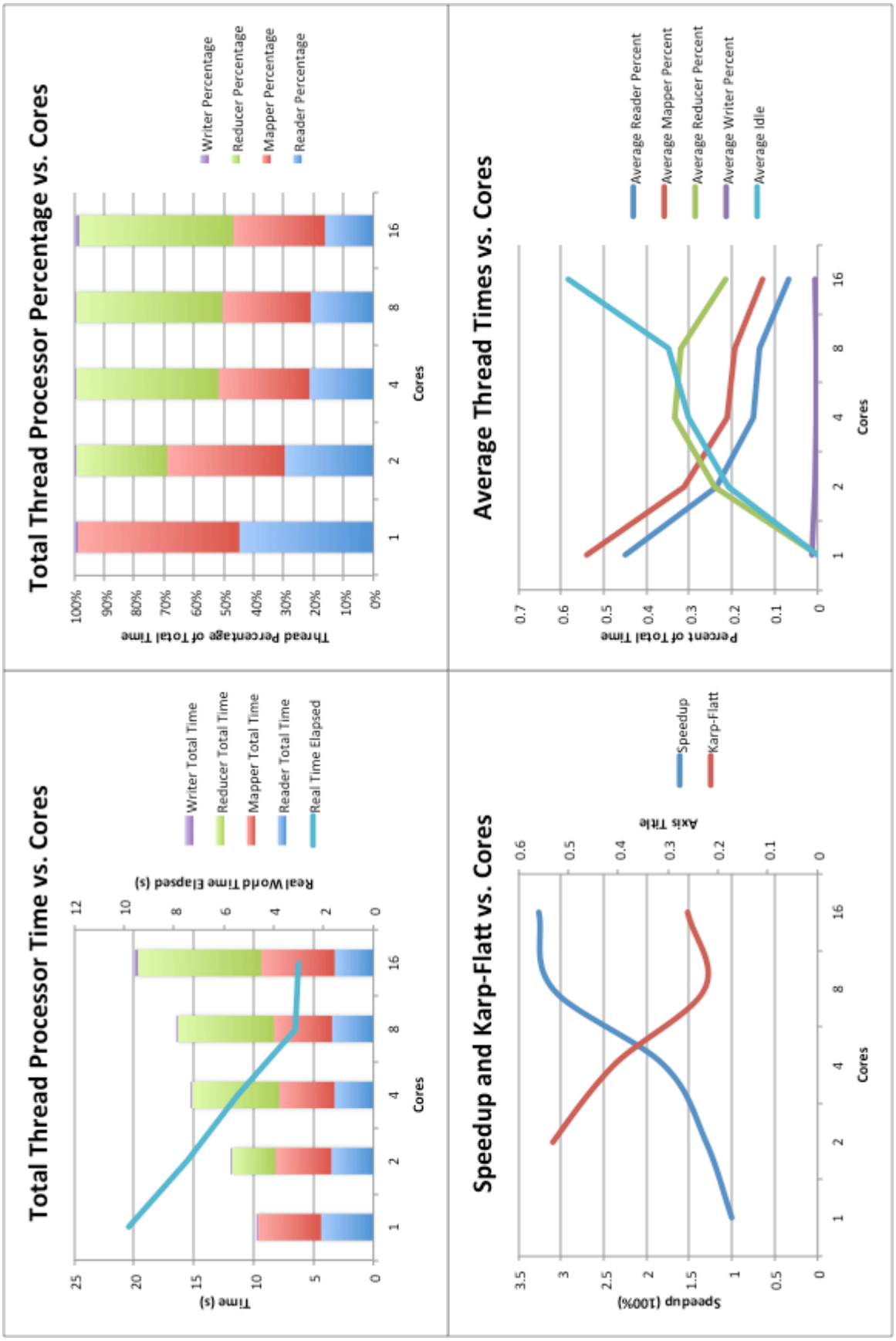
$$n * (t_{work}) + P * (3 * n * \log_2 P * t_{send}) + n * (P-1) * t_{hash} + n * P * t_{write} = T_0 + T_1$$

$$T_0 = P * \left(3 * n * \log_2 P * t_{send}\right) + n * (P-1) * t_{hash}$$

$$T_1 = n * t_{work} + n * P * t_{write}$$

$$\theta(n * P * \log_2 P) = Iso - Efficiency$$

3.2.3 MPI Data


Total Thread Processor Percentage vs. Cores
Writer Percentage
Reducer Percentage
Mapper Percentage
Reader Percentage
Thread Percentage of Total Time
Cores
Average Thread Times vs. Cores
Average Reader Percent
Average Mapper Percent
Average Reducer Percent
Average Writer Percent
Average Idle
Percent of Total Time
Cores
Total Thread Processor Time vs. Cores
Writer Total Time
Reducer Total Time
Mapper Total Time
Reader Total Time
Real Time Elapsed
Time (s)
Real World Time Elapsed (s)
Cores
Speedup and Karp-Flatt vs. Cores
Speedup
Karp-Flatt
Axis Title
Speedup (100%)
Cores