ECE 56300: Final Project Report

December 14, 2022

Yuan-Yao Lou (Mike, PUID: 33683217)
Yu-Ju Fang (Jeff, PUID: 33785918)
Cheng-Yang Tsai (Arno, PUID: 33797411)

1 Introduction: Map-Reduce

MapReduce is a programming model and an associated implementation for processing large datasets with a parallel and distributed algorithm on a cluster. MapReduce is a two-stage data processing paradigm that divides large datasets into smaller chunks and processes them in parallel. In the first stage, the "map" phase, the data set is divided into smaller subsets and a mapping function is applied to each subset to produce a set of intermediate key-value pairs. In the second stage, the "reduce" phase, the intermediate key-value pairs are grouped by key and a reducing function is applied to each group to produce a set of output values. MapReduce is often used in big data applications, where it is used to process large amounts of data in parallel across a cluster of computers.

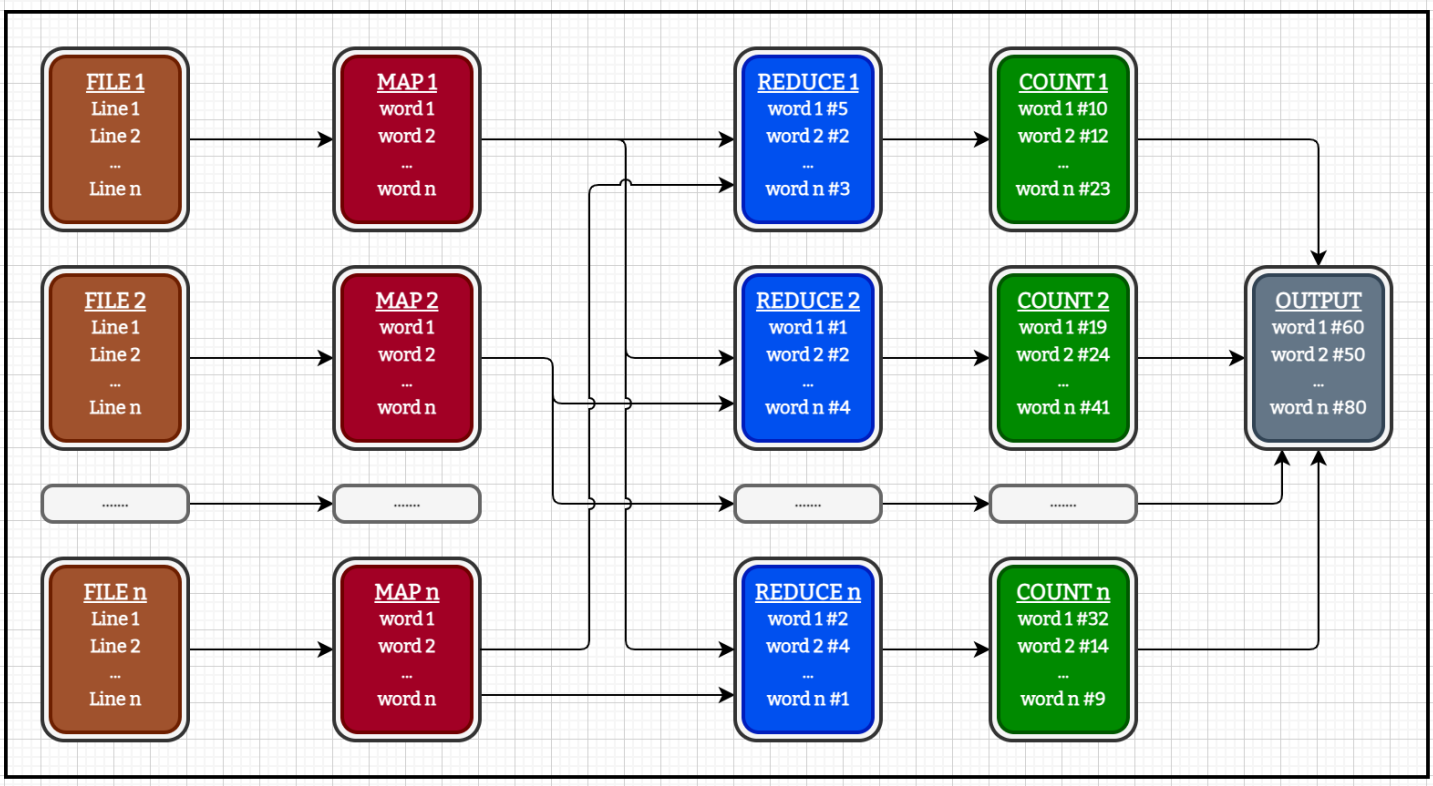

Fig 1. Word count problem illustration in MapReduce programming model

In this final project, we will address the **work count problem** utilizing the MapReduce data processing paradigm (Fig. 1). Word count is a typical example of a MapReduce problem. In a MapReduce job of word count problems, the input data is a set of text documents and the goal is to count the number of times each word appears in the documents. In the map phase of the word count MapReduce job, the content of text documents is divided into smaller subsets and a

mapping function is applied to each subgroup. The mapping function processes each segmentation of text document in each subset and outputs a set of intermediate key-value pairs, where each key is a word from the text and the value is the number of times that word appears. In the reduce phase of the word count MapReduce job, the intermediate key-value pairs are grouped by key and a reducing function is applied to each group. The reducing function sums the values for each key, producing a set of output values representing the total number of times each word appears in the input data.

The word count MapReduce job is a simple example of how MapReduce can be used to process and analyze large amounts of text data. By dividing the input data and applying the mapping and reducing functions in parallel, MapReduce can solve word count problems quickly and efficiently, even for very large datasets.

2 System Architecture

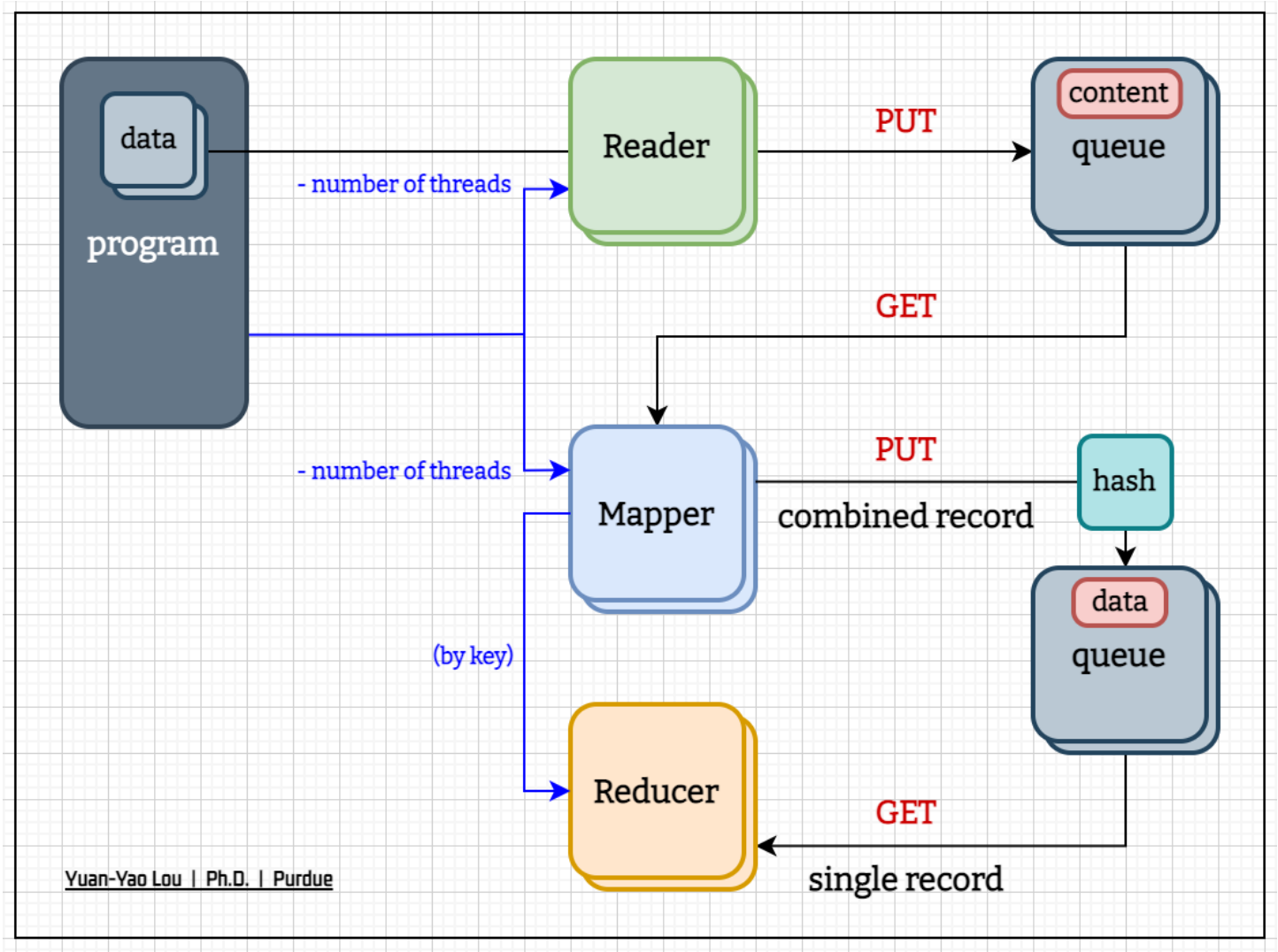

Fig 2. The system architecture of our programs; three different types of workers (1) Reader (2) Mapper (3) Reducer

The system architecture of our program is depicted in Figure 1. The data represents the text files and the programs access these text files to solve the word count problem. In our system, there are three types of workers and each of

which is associated with different phases of MapReduce implementation. First, **the readers** open the input text files and parse each file line by line and word by word. The readers push and transform the collected words into work items and push them to the content queue. To this end, we are entering the map phase from the input data processing phase. Next, **the mappers** retrieve the work items from the content queue and perform the combination, then transform the combined record into work items and push them to the data queue. Now we are entering the reduce phase from the map phase. Last, **the reducers** retrieve the work items from the data queue and count the occurrence of each combined record to generate the final answers.

3 Implementation

3.0 Project Environment

The project environment is illustrated in Fig. 3.

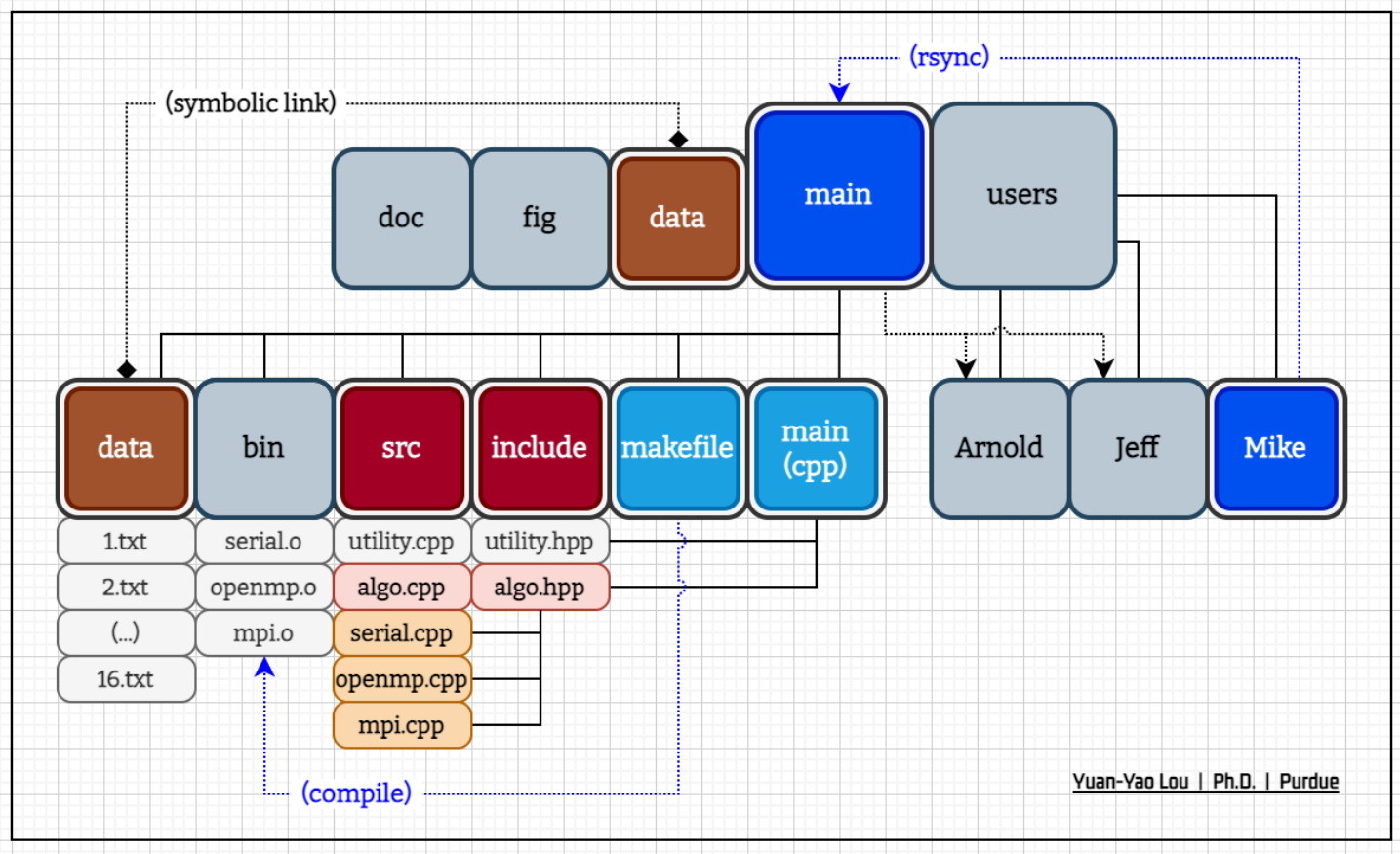

Fig 3. Project Environment / Folder Structure

To execute the programs, the following Makefile commands are helpful:

- **`make compile:`** compile all versions (i.e., serial, OpenMP, and MPI)
- **`make submit:`** submit jobs of serial version, OpenMP version with 16 threads, and MPI with 16 nodes
- **`make run:`** execute the above two commands sequentially

3.1 Serial Version

The algorithm of the serial version is clearly plotted in Fig. 2. The reader function will first parse all of the input text files and transformed them into work items and push them to the content queue. The mapper function starts after the readers finish parsing, combining the work items as records by retrieving work items in a fixed batch size (i.e., 20). The combined records will then be pushed to the data queue for the reduced operation. Note that in this engineering way, the serial algorithm simulates the parallel algorithm just as threads retrieve work items simultaneously (although there is only one thread available). After the mappers clear the content queue, the reducers start to grab combined records from the data queue and count the occurrence of each word and output the final answers.

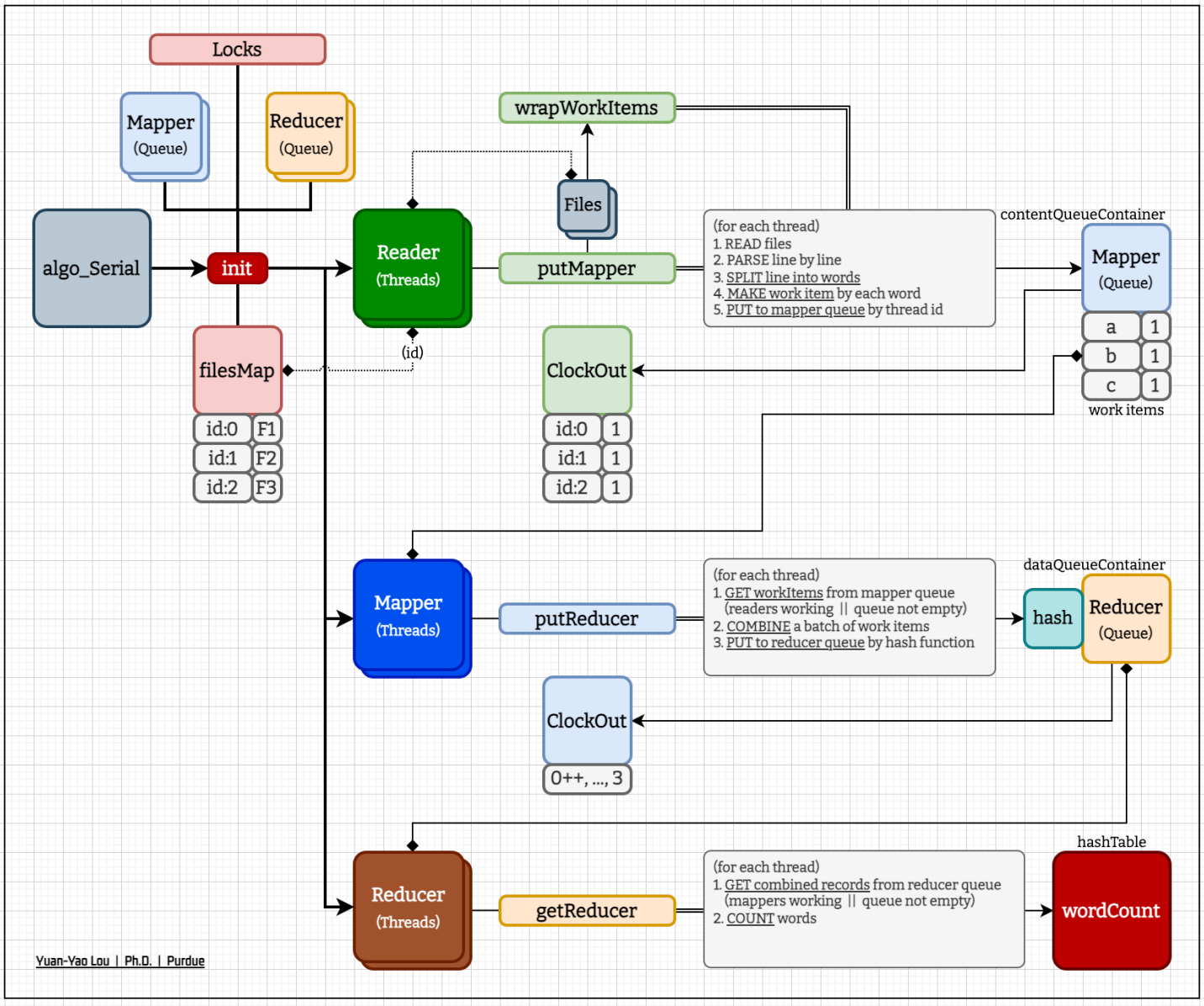

Fig 4. Flow chart of OpenMP algorithm

3.2 Parallel Version: OpenMP

The algorithm of the OpenMP version is plotted in Fig. 4 and is straightforward to implement in that the difference is we have to control multiple readers, mappers, and reducers to work at the same time (so as the number of content queues and data queues).

Specifically, we need additional tricks to keep it working properly. First, given that the number of available threads is varying in each run, we assign the input files for each thread **(statically)**. Following, the varying number of mapper threads and reducer threads require a certain level of synchronization to ensure correctness and robustness. It is obvious that the termination of reader threads is the EOF. On the other hand, the termination of mapper and reducer threads depends on two factors. First, the termination of readers (for mappers) and mappers (for reducers). Second, the occupancy of the content queue (for mappers) and data queue (for reducers). Concretely, the mapper threads terminate when all the readers are terminated and the content queue is empty. The reducer threads terminate when all the mappers are terminated and the data queue is empty.

To this point, the synchronization between three workers is established, and we notice that there is no "task wait" required in our system for different phases in MapReduce. In other words, the only sequential part of the algorithm is the initialization function. Concretely, the readers keep parsing the input files and pushing the work items to the content queues. The mappers keep retrieving the work items, combining them into records, and pushing combined records to the data queues. Finally, the reducers continue grabbing combined records from the data queue and counting the final answer. There is no deadlock or race condition in our system among readers, mappers, and reducers. Although this seamless synchronization mechanism boosts the overall performance, the speedup given by supporting more threads and such improvements are not significant if the workload is not too much (e.g., 15 files in our evaluation).

3.3 Parallel Version: MPI

3.3.1 Method 1

Unlike the OpenMP version, in our first implementation method, we have to separate the reader operations from the mapper and reducer operations. This is because we need to make sure each node gets the files broadcasted from the root node. Hence, First, we start with using a reader (on the root node) with a for loop to read all the text files into temporary buffers, the way we read is by using OpenMP to read parallel. After that, we combine all the temporary buffers into a final large buffer. Second, the root node uses MPI_ Bcast to send the input to worker nodes. Third, worker nodes split the sentences and maps the word into the hash table using a while loop. The termination factor for this file loop is to check the buffer is not NULL, which means there are still words waiting to be mapped. The while loop here also uses OpenMP to achieve. Next, each node sends the mapping result back to the root node. In addition, we use **`MPI_Waitall`** and **`MPI_Irecv`** to make sure we get all the map results back from the worker node, and then the root node writes the result to an output file using the writer and prints the result out. After we run some tests on this version, we realize that this method has a really bad performance due to the reader's behavior (i.e., readers spend around 170 seconds). It appears that we read, buffer, broadcast, and parse almost two times. This is really not necessary and consumes the runtime. The second method changes the implementation, being simpler but much more effective.

3.3.2 Method 2

In method 1, we use **`MPI_Bcast`** to send the input text files so that each node gets the input text files. Here, we follow the same flavor of the OpenMP version to **statically** assign each node with the input text files based on the SIZE and RANK. Next, each node executes the same algorithm of the OpenMP version to parse, map, reduce and collect the results. Finally, each node sends the local counter to the root node, and the final answers are combined.

4 Results

4.0 Parameters

- **`# of Input Files`**: 15
- **`Batch size`**[1]**:** 20
- **`Threads(OpenMP)`**[2]**:** 1, 2, 4, 8, 16
- **`Nodes(MPI)`**[3]**:** 1, 2, 4, 8, ~~*16 (resource not available)*~~

4.1 Serial Version

The serial version without any support of OpenMP and MPI gives the overall runtime (including the output of final answer) around **2.93295 seconds** in ten runs of experiments. The average initialization time is 0.00349 seconds and the algorithm part is around 2.92946 seconds.

4.2 Parallel Version: OpenMP

Threads	1	2	4	8	16
Overall Runtime	3.07431 s	2.87530 s	2.43633 s	2.01972 s	1.78369 s
Initial Execution	0.00003 s	0.00004 s	0.00004 s	0.00005 s	0.00008 s
Parallel Execution	3.07428 s	2.87526 s	2.43629 s	2.01967 s	1.78361 s
Speedup	---	1.069	1.262	1.522	1.724
Efficiency	---	0.5345	0.3155	0.1902	0.1077

4.3 Parallel Version: MPI ($N_{Files} > N_{nodes} * N_{threads}$)

Nodes	1	2	4	8	16
Overall Runtime	3.29252 s	1.53281 s	1.03501 s	1.03795 s	N/A [4]
Sequential Execution	1.03387 s	0.52717 s	0.35873 s	0.39211 s	---
Parallel Execution	2.25865 s	0.99564 s	0.67628 s	0.64584 s	---
Speedup	---	2.14802	3.18114	3.17215	---
Efficiency	---	1.07402	0.79528	0.39652	---

[1] The number of work items retrieved by mappers and reducers
[2] The total number of the reader, mapper, and reducer threads
[3] The total number of the reader, mapper, and reducer threads
[4] Resources are never available on Scholar

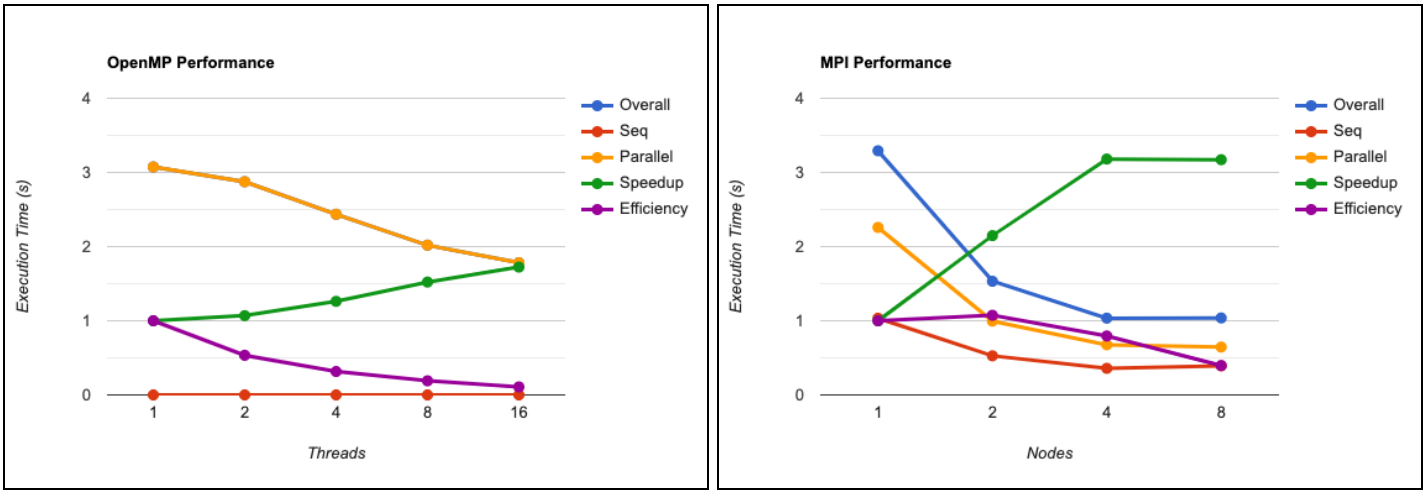

Fig 5. Execution time vs. Threads (OpenMP) / Nodes (MPI)

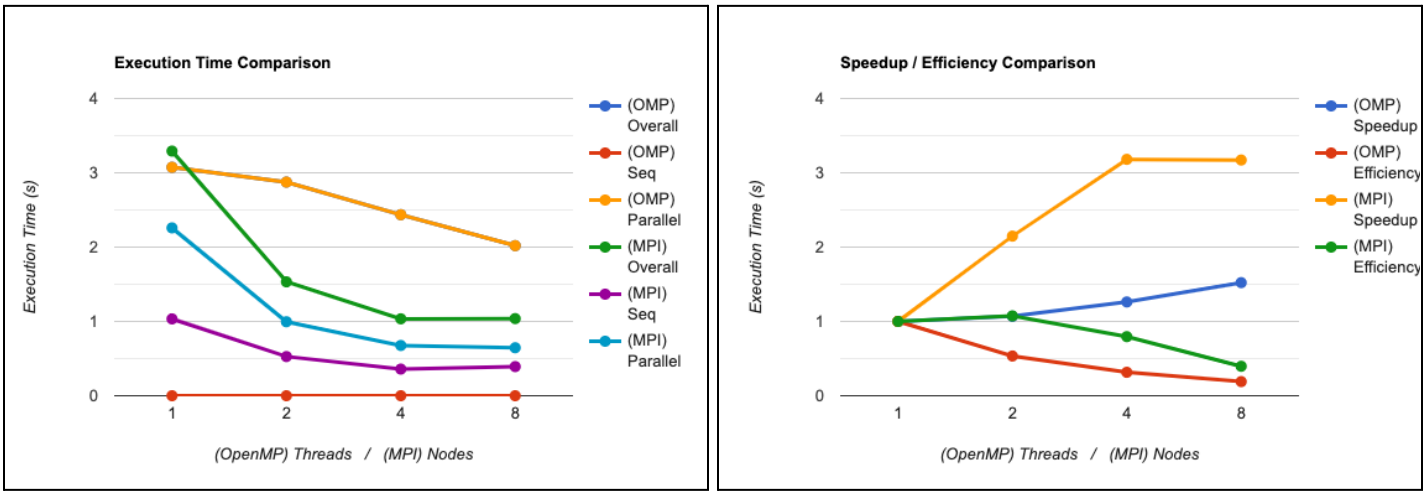

Fig 6. Comparison between OpenMP and MPI regarding execution time, speedup, and efficiency

4.4 Analysis

Overall speaking, the performance of the MPI version outperforms the OpenMP version. We first look at the OpenMP version (left plot of Fig. 5). Given that our OpenMP algorithm only involves a very short initialization time (i.e., sequential part), the overall execution time overlaps with the parallel execution time. Note that the efficiency gets lowers as the number of threads increases. This is because, again, the workloads (i.e., 15 input text files) are too light. For instance, we initialize sixteen threads, and each thread reads one or two files. This is apparently not worth it.

On the other hand, from the right curve plot of Fig. 5, we notice that the speedup level increases a lot until the case of eight nodes. Once again, this is because the workload does not increase as we give more resources to the programs. Also, Fig. 5 indicates the sequential execution time is much higher than the OpenMP version, and this is due to the initialization processes. Yet, the parallel execution time of the MPI version beats the OpenMP version very well. The detailed comparison between OpenMP and MPI versions is depicted in Fig. 6.

Last but not least, our implementations have an obvious issue, the unbalanced workloads between each thread or each node, and this is decided after the internal discussion. We have two reasons regarding this: (1) The number of input data files in our evaluation scenario is not huge (2) Simplified implementation approaches might reduce the runtime overhead.

5 Conclusion

It is really interesting to solve a practical problem by utilizing what we've learned from the lectures and the associated homework assignments during the whole semester. The word count problem, especially, is a typical application among various cloud servers. Throughout the whole progress, from the project kick-off meeting, the system and architecture design phase, to the development stage and the final verification and performance analysis, we learned a lot, including but not limited to the insights of parallel programming development and the core concepts of MapReduce and data processing. We also give our best to draft this report to share what we've developed in effective visualized ways. We all know, however, the optimization of such problems never ends. For example, we can tune the batch size or increase the workloads (i.e., the number of input text files). In this way, we can further verify the performance bottleneck and go back to revise our algorithms. And of course, we would love to explore more literature to see how various optimization techniques developed in the future if time allows.