Distributed MapReduce for Text Processing

Divya Amirtharaj
divya_amithraraj@college.harvard.edu

Siona Prasad
sionaprasad@college.harvard.edu

Katherine Zhang
katherinezhang@college.harvard.edu

1 Introduction

With large datasets becoming more prevalent, the need to process this so-called big data in an efficient or distributed manner is becoming increasingly necessary for tasks such as machine learning, recommendation algorithms, fraud detection, etc. Although individual computers with large amounts of compute power may exist, it is still often more time- and cost-effective for applications from small businesses to government programs to distribute their data processing across multiple machines. Additionally, processing data in parallel comes with the same advantages that any distributed system does – it is more scalable, reliable, and it harbors no individual point of failure.

As such, one prominent framework for performing distributed data processing is the MapReduce framework, which splits a data processing task into map tasks (which produce intermediate results) and reduce tasks (which aggregate these intermediate results). Patented by Google in 2004 and implemented in several open-source frameworks, including Apache Hadoop and CouchDB, MapReduce has often been touted as a solution to efficient, distributed data processing [3]. However, its implementation presents interesting trade-offs between computation and communication costs, as the need to split, send, and organize data that is sent between a central coordinator and several workers creates communication overhead. Thus, the communication time may actually make MapReduce less efficient for smaller tasks. Additionally, MapReduce has been criticized as much as it has been lauded for continuing to have a single point of failure in its central node/coordinator, as well as for not producing enough novelty in its field.

Both the positive aspects of a distributed system like MapReduce and the criticism towards it motivate our work, in which we seek to create our own MapReduce system for processing text. Specifically, we use Python and gRPC to build a distributed system with a central node (the server) that distributes mapping and reducing tasks to other nodes (workers) in order to count the number of times each word appears in several large book text files. Our implementation is also tolerant to faults in worker nodes and responds by having the server re-assign incomplete or pending tasks from a dead worker to another live worker node. This allows the MapReduce process to continue as long as the central server is alive. Finally, in order to address some of the questions about MapReduce's performance, we perform experiments with different numbers of workers, map tasks, and reduce tasks and time MapReduce's performance. As such, we not only design and build our own word counter using MapReduce, but we also evaluate its efficacy and the effects of the aforementioned trade-offs between communication and distribution.

2 Background

As mentioned above, the MapReduce framework was patented by Google in 2004 [1]. However, its origins go back to the advent of the parallel supercomputer known as the Connection Machine, as its compute power could make the map and reduce tasks highly efficient [2]. It is also based on the `map` and `reduce` operations from Common Lisp, which perform similar operations without parallelization.

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

Google's patenting of MapReduce and use of it in their Google File System then popularized its use for data processing tasks, leading to its implementation in the Apache Hadoop library. The Hadoop version of MapReduce has been utilized by eCommerce retailers, social media networks, and other businesses that need to sort or search through large amounts of data. We now turn to explaining the general functionality of MapReduce before discussing our implementation.

2.1 General MapReduce Framework

The general MapReduce framework simplifies the task of parallel processing by dividing the computation into two key phases: the Map phase and the Reduce phase. Figure 1 shows a visualization of how this framework functions for a word counting application:[1]

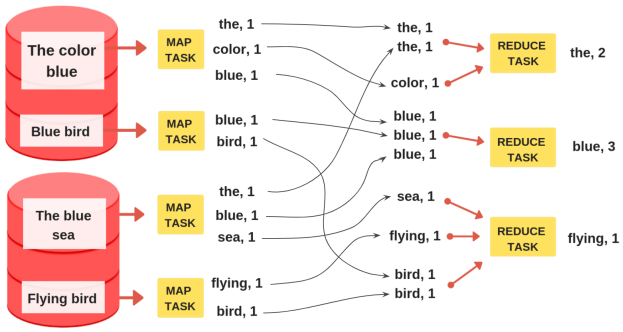

Figure 1: A rendering of the MapReduce framework for word counting, which is the task we seek to perform. This example contains 4 map tasks and 3 reduce tasks.

In the Map phase, the input dataset is divided into smaller chunks, and each chunk is processed independently by a map function. The map function takes the input data and produces a set of intermediate key-value pairs. This phase is responsible for extracting and transforming the relevant information from the input data.

Once the Map phase is complete, the intermediate key-value pairs are partitioned and distributed across multiple machines in the cluster. This distribution enables parallel processing of the data in the next phase. The Reduce phase takes the intermediate key-value pairs as input and applies a reduce function to combine and summarize the data. The reduce function aggregates the values associated with each unique key and produces a final output.

3 Our Design

We designed a MapReduce algorithm for the word counting task. We leverage the gRPC (Google Remote Procedure Call) technology to create a server-client architecture. The server acts as the central coordinator, while multiple workers connect to the server and continuously request map or reduce "tasks" to perform.

To begin the word counting process, the server splits up the input text files into smaller chunks using a technique called chunking. Our chunking function reads the input data from files, divides it into smaller chunks, and assigns the chunks to the map tasks. The server manages the distribution of chunks to ensure load balancing across the available workers.

The map tasks are responsible for processing the assigned text chunks. Each map task reads its chunk and creates intermediate files that contain the words found in the chunk. The map task sorts words into buckets by the first letter of the word, and writes the buckets evenly to intermediate files.

Once the map tasks are completed, the reduce tasks begin. The server assigns reduce tasks to available workers. Each reduce task receives an intermediate file generated by the map task. The reduce task is responsible for combining the output of the map tasks by aggregating the counts of each word. They read the intermediate files, perform the necessary reduction operation (counting the occurrence of each word), and generate the final word count result.

[1] https://datawhatnow.com/mapreduce-shuffle-sort/

Throughout the process, the server handles the coordination and task assignment, ensuring efficient utilization of the available workers. The workers connect to the server, request tasks, and report the completion of each task. In case of failures or worker disconnections, the server can reassign the pending or incomplete tasks to other available workers to maintain fault tolerance and maximize processing efficiency.

3.1 Map and Reduce Classes

The Mapper class represents the map function in the MapReduce framework. It initializes the files dictionary, which will be used to cache file contents. The map method takes three parameters: `map_id` (the ID of the map task), chunks (a set of filenames representing the input text chunks), and `num_red_tasks` (the number of reduce tasks).

Within the map method, each chunk is processed individually. The text in each chunk is transformed by removing punctuation and converting it to lowercase. Then, the text is split into words. For each word, a bucket ID is calculated by taking the ASCII value of the first character modulo `num_red_tasks`. This bucket ID determines which reduce task will handle the word.

The map results are stored in the `map_results` dictionary, where the key is the bucket ID and the value is a list of words associated with that bucket. If a new bucket ID is encountered, a new key is added to the dictionary with an empty list. Each word is appended to the corresponding bucket list in `map_results`.

After processing all the chunks, the `map_results` dictionary is converted into a protobuf message. The MapResults message is then sent to the server to indicate the completion of the map task using gRPC communication.

The Reducer class represents the reduce function in the MapReduce framework. The `count_bucket` method takes the `map_results` received from the map tasks and counts the occurrences of each word across all the containers. The word counts are stored in a counter dictionary.

The reduce method takes two parameters: `bucket_id` (the ID of the reduce task) and `map_results` (the map results received from the server). It calls the `count_bucket` method to obtain the word counts. The word counts are then converted into a protobuf message. This message is sent to the server using gRPC communication to indicate the completion of the reduce task.

These map and reduce functions work together to process the input text chunks, generate intermediate map results, and perform the final word count aggregation in the MapReduce framework.

3.2 Server Design

The server functions as our equivalent of a central coordinator. We use gRPC to implement the server, as we have become familiar with it throughout this course. The workers act as clients, connecting to the server and continuously requesting tasks from it, while the server splits up the text files that the worker needs to perform their task by chunking them and sending them back to the client in a response. Once a worker finishes this task, they then send acknowledgement to the server in the form of their results. The server keeps track of the number of tasks performed so far and moves the state of the entire operation from "map" to "reduce" to "finished" as tasks are completed – thus, workers need not keep track of state themselves.

The server is initialized with the following variables:

- `chunk_size`: Size of chunks of text.
- `num_map_tasks`: Number of tasks for mapping.
- `num_red_tasks`: Number of tasks for reducing.
- `workers`: A list of the unique ids of the workers running on this server.
- `cur_task_type`: Type of task currently being executed. Starts with a value of `map` and is changed according to the value of `task_count`.
- `task_count`: Keeps track of the number of tasks of `cur_task_type` that have been finished. Initialized at 0 and is set back to 0 every time the current task type is finished. This works because all map tasks must finish before any reduce tasks are executed.

- `task_id`: Keeps track of the id of the current task. Incremented whenever a new task (e.g. one that has not been done before by any worker) is assigned to a worker.
- `split_data`: Calls a method that chunks the file data and randomly splits them into `num_map_tasks` buckets for mapping tasks.
- `map_task_split` and `red_task_split`: Buckets the map and reduce tasks by worker id. Filled in as tasks are requested by workers.
- `map_task_backlog` and `red_task_backlog`: Records the backlog of undone tasks if a worker fails. See more in Section 3.3

There are four main RPC calls that can be made by workers to the server. Their behavior is as follows:

1. `get_worker_task`: Takes in a `Worker` proto, which contains a worker id. Depending on the current task type, retrieves a Map or Reduce task in a `Task` proto. Records that this worker id took the current `task_id`, increments it, and saves it in `map_task_split` and `red_task_split`.
2. `finish_map_task`: Invoked by a worker when a Map task has been finished. The worker sends over their results in a `MapResults` proto, which are keyed by reduce task id. These results are then ready to be reduced.
3. `finish_reduce_task`: Invoked by a worker when a Reduce task has been finished. The worker sends over their results in a `ReduceResults` proto, which are keyed by reduce task id. These results are then written to output files.
4. `worker_down`: Takes in a `Worker` proto, which contains a worker id of a worker that has been killed (through a KeyboardInterrupt). Upon worker death, the server places all of the tasks that this worker was responsible for in the backlog again, as none of the intermediate results are saved, so any work a worker was responsible for is lost.

Thus, the general workflow is as follows:

1. Initialize w workers by running `worker.py` w times. Each one will wait for the server to start. Then, start the server, giving the number of map/reduce tasks, a chunk size, and all of the worker ids in a command-line list.
2. Each worker continually calls a function `_ask_task`, which calls the server stub to get a task (as shown above).
3. The worker then performs Map tasks by calling the Mapper. It then calls the server stub to finish the task and sends over its results.
4. Once the Map tasks are all finished, the server resets its task counter. Then, the server begins assigning Reduce tasks to workers.
5. The workers then perform their assigned Reduce tasks by calling the Reducer. They then call the server stub to finish the task and send over the reduced results, which are stored in output files on the server side.

Now, we turn to discussing what happens when a worker fails, and how we achieve fault tolerance in our model.

3.3 Fault Tolerance

We handle fault tolerance through several mechanisms.

1. **Backlog and Redo**: The server maintains two backlogs, `map_task_backlog` and `red_task_backlog`, which store the task IDs of map and reduce tasks that need to be redone. When a worker sends a "goodbye" message indicating that it has been killed (`worker_down` method), the server updates the backlogs by adding the undone tasks. These backlogs ensure that tasks interrupted due to worker failures are rescheduled and completed when new workers join the system.

2. **Task Tracking**: The server keeps track of the tasks assigned to each worker using dictionaries `map_task_split` and `red_task_split`. These dictionaries map worker IDs to lists of task IDs that the worker is responsible for. When a worker fails, the server updates the backlogs and removes the failed worker's tasks from the respective task split dictionaries. This tracking mechanism allows the server to reassign the failed worker's tasks to other available workers.
3. **Idle State**: The server uses the `cur_task_type` variable to indicate the current type of task being executed. We keep track of when map tasks are finished by changing the state to reduce, and when the reduce tasks are finished by setting the state to shut down. This state tracking mechanism ensures that reduce tasks do not start until all map tasks are finished, and the server shuts down gracefully when all tasks are completed.
4. **Task Redo on Worker Failure**: When a worker fails and sends a "goodbye" message, the server updates the backlogs and task counts accordingly. The server then reschedules the undone tasks from the failed worker by reassigning them to other workers when new worker requests arrive. This process ensures fault tolerance by redistributing the workload and preventing the loss of completed tasks due to worker failures.

Overall, the server implementation addresses fault tolerance in the MapReduce framework by maintaining task backlogs, tracking task assignments, rescheduling failed tasks, and gracefully handling worker failures. These mechanisms allow the system to recover from failures and continue processing tasks efficiently, ensuring fault tolerance in distributed word counting operations.

3.4 Experiments

Due to the communication overhead associated with distributing tasks over multiples workers, it is necessary to find the point where the benefits of distribution outweigh the costs of the server-worker communication and data organization. To this end, we calculated the speed at which the word counting task was performed while modifying the following variables:

1. Number of workers
2. Number of map tasks
3. Number of reduce tasks

The results of these experiments are detailed in section 4.

3.5 Testing

We wrote unit tests to verify the functionality of individual methods in our Server and Worker found in `test_server.py` and `test_worker.py` respectively. The server tests cover the setup process of the server, task retrieval for workers, handling of worker death scenarios, and the correct progression of map and reduce tasks. They ensure that the server behaves as expected and maintains the appropriate state during the MapReduce computation. The worker tests validate the initialization process of the worker and its ability to retrieve tasks from the server stub. They help ensure that the worker behaves as expected in terms of its setup and task retrieval functionalities.

4 Experimental Results

4.1 Testing across multiple machines

We ran two worker on one machine and a server on a separate machine with 2 map tasks and 2 reduce tasks. The system took ≈ 7 seconds to complete the word counting task, which demonstrates that the overhead created by communication between different machines is much higher for this relatively small task.

4.2 Testing different numbers of workers, map tasks, and reduce tasks

We have provided a sample of our results below.

Workers	Speed
2	0.73
3	1.19
4	1.72
5	1.46

Map Tasks	Speed
2	1.86
3	2.23
4	0.91
5	0.73

Reduce Tasks	Speed
2	0.62
3	0.70
4	0.99
5	0.73
6	0.87
7	0.99
8	1.18

We first varied the number of workers while keeping the number of map tasks and reduce tasks constant (at 5). There does not appear to be any conclusive trend in the speed of completion when more workers are added. We see a general trend in which speed decreases as workers are added, with 5 workers being the outlier. It is possible that this might also be due to fluctuations from our dataset being relatively small. Experiments with more variations in the number of workers could also determine more conclusive results.

We also varied the number of map tasks and reduce tasks. Again, we see no conclusive trends in the results. In general, a greater number of map tasks seems to correspond with a higher speed (with one outlier), while an increased number of reduce tasks seems to correspond with a lower speed (with one outlier). We hypothesize again that we may need much larger data sets to see the impact of distributing the data processing.

5 Conclusion

Overall, this code provides a basic implementation of the MapReduce paradigm, demonstrating the distribution and parallel processing of tasks across multiple workers. Some future directions include expanding the system to handle more complex data processing tasks. For example, we could try extending the project to support different input and output formats, such as handling structured data (e.g., JSON, XML) or working with databases. This would involve developing input and output modules that can parse various data formats and integrate them into the MapReduce workflow. Another optimization is integrating a combiner function into the map phase to perform a partial reduction of the intermediate key-value pairs. This optimization can help reduce network traffic and improve the overall efficiency of the MapReduce process. Lastly, we could integrate a distributed file system (e.g., HDFS) to handle large-scale datasets that span across multiple machines. This would involve adapting the code to read and write data from a distributed file system, allowing for efficient data storage and processing.

In conclusion, by utilizing the MapReduce framework with gRPC-based server-client communication, we achieve a scalable, fault-tolerant and distributed word counting solution. The server splits the text files, assigns tasks, and manages the overall process, while the workers perform the map and reduce tasks in parallel. This design enables efficient processing of large-scale text datasets and allows for easy scalability by adding more workers as needed.

References

[1] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[2] W. D. Hillis. *The connection machine*. MIT press, 1985.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.