CPSC426 FINAL PROJECT: DISTRIBUTED MAPREDUCE IMPELEMENTATION

Troy Feng

Yale University

1. INTRODUCTION AND BACKGROUND

MapReduce is a common and applicable paradigm in programming and computing. In MapReduce, a user seeks to apply a Map function to a list of values, generating an intermediate list of key-value pairs. Then, these key-value pairs are aggregated, sorted, and provided to a Reduce function, which reduces all values grouped together by the same intermediate keys. Canonical abstractive use cases of MapReduce, albeit perhaps inefficient in a sequential non-parallel setting, include word count and sorting.

In 2004, Jeffrey Dean and Sanjay Ghemawat published the MapReduce paper, which adapted the MapReduce programming model to a large-scale distributed context at Google [1]. Because MapReduce use cases are abstracted, their distributed MapReduce model can be applied as a fault-tolerant, deterministic solution for efficiently computing MapReduce on input data distributed across large-scale distributed systems.

In this project, we design and implement a simple prototype for distributed MapReduce, closely following Dean et al's original paper [1].

2. DESIGN AND IMPLEMENTATION

We implement a simple MapReduce prototype following the original MapReduce paper. Our tech stack consists of Go and gRPC, on the basis of familiarity and Go's support for distributed programming via RPCs supported by gRPC.

At the highest level, our MapReduce implementation is invoked by a user, who wishes to apply the MapReduce programming model to a sequence of input files. We assume the user has access to several distributed machines (nodes). The user is able to dynamically select the number of worker nodes, as well as specify the worker nodes' addresses, in configuring MapReduce.

2.1. GFS

The original MapReduce paper assumes that the user's input files are already distributed across the machines that will become worker nodes, as in the Google File System (GFS) [2]. However, because we do not have access to the hardware or scale of a practical GFS-like distributed file system such as GFS, we chose to implement a mock in-memory service that merely mimics the functionality of GFS. We assume that the input files are originally hosted on the user's local machine, rather than across a true GFS-like file system. Our mock-GFS reads these local files, and then exposes a concurrent and contention-safe `Read`and `Write`API, with options to read or write files to a specific node or simply to write to any available low-workload nodes. As in the MapReduce paper, we assumed that GFS divides each file into chunks of fixed maximum size (up to 64MB in the real GFS), and that each chunk is usually replicated across three (several) nodes. We additionally exposed a `GetLocs`API in anticipation of locality ([1] Section 3.4), so that the leader node can access locality information for each input file chunk and consider such information when allocating Map tasks to workers. In invoking our MapReduce implementation, this mock-GFS service is started before any communication with worker nodes, passed information about the files it should initially host as well as all of the node identities in the distributed file system it is supposed to simulate (which will eventually be the worker nodes of MapReduce), and splits the provided files into distributed chunks across 3 nodes. Hereafter, requests to read from worker disks, to write intermediate files, and any file system operations are done through this mock-GFS API. It provides a guarantee of consistency and replication.

2.2. Starting Workers and Leader

As in the original paper, our implementation first splits the input file space into `M` chunks. Here, we make our first simplifying assumption: our chunking function for splitting the input file data is identical to that of GFS. This hugely simplified the logic of our GFS implementation, as we would not need to define complex mappings between partial file chunks and those hosted on GFS.

Our implementation then starts `N` workers as gRPC services, where `N` is specified by the user. At this time, the user also specifies custom `Map` and `Reduce` functions, assumed to be written and hosted locally as Go plugins. This was inspired by MIT 6.5840's Lab 1 [3] starter code implementation. Every worker initially has access to both `Map` and `Reduce`, and every worker initially hosts a subset of the input file chunks

troy.feng@yale.edu

as specified in the GFS section. In particular, this meant any worker could be assigned either Map or Reduce tasks. This allowed for abstractions in the leader implementation, wherein Map and Reduce tasks sometimes shared attributes, as well as flexibility in worker node selection and task delegation. By nature, since Map tasks temporally precede Reduce tasks, allowing workers to be assigned to both Map and Reduce reduced the amount of idle time.

Finally, our implementation starts one leader, also as a gRPC service. The leader is passed the locations of the original input files on the user's local disk, as well as the identities of its available worker nodes.

2.3. Map

During the Map phase, the leader selects worker nodes based on least congestion. To do so, the leader must maintain inmemory data structures tracking the status of each task (as in [1] Section 3.2), and additionally the status of each worker (i.e. the number of in-progress tasks currently being executed by that worker). While we did not have time to add locality consideration, this worker selection process should additionally include consideration of which workers locally host the input file chunks, so that they will not need to consume bandwidth by querying another worker node through GFS.

Here, we made another simplifying decision and assumption: we wait until all map tasks have completed before starting the Reduce phase below. This also followed the implementation suggested by MIT 6.5840 Lab 1 [3]. The alternative was to wait until all intermediate file tasks associated with only one map task have completed, before starting a Reduce task for those intermediate files, so that both Map and Reduce tasks can be run at the same time. While our decision simplified the logic of our leader implementation by separating the two phases, the tradeoff is that we must reckon with tail latency during the Map phase, since it blocks Reduce. With more worker nodes, assuming approximately normally distributed latencies and request processing times, it is likely that at least one worker node will take substantially longer to complete its map task, thereby blocking the Reduce phase from starting. However, our leader logic includes a hasty implementation of backup tasks as described in Section 3.6 of the original MapReduce paper [1]. In particular, our leader node monitors task statuses in a long-running goroutine, and after a fixed amount of time elapses, if a worker node fails or is still recorded as in-progress, the leader will optimistically reallocate a copy of the Map task to another idle worker. The first such task that completes will be recorded in a thread-safe manner. This also implements worker failure as described in Section 3.2 [1]. Scheduling backup tasks comes at the cost of increased network and server traffic for workers, but improves latency and availability guarantees by mitigating the effects of tail latencies and failing workers. We also minimize idle time, even towards the end of Map tasks, as having many idle workers only increases the probability that one backup task will finish in a reasonable amount of time. Thus, while implementing Reduce to begin in parallel with Map would have made full use of the parallelism available to us in distributed MapReduce, we chose to implement them in separate phases for understandability, simplicity of implementation, and because our backup task provisions mitigated the harm of tail latency.

Worker nodes produce intermediate key-value pairs after applying `Map`, which are then partitioned into `R` regions. These `R` partitions are then written to that worker's disk, and their locations are passed to the leader node via RPC as map tasks and partitions are completed.

2.4. Reduce

Once all Map tasks have been completed, the leader will have access to `R` sets of `M` intermediate key-value files, thanks to RPC notifications from workers. At this point, all workers are idle, and the leader begins the Reduce phase by similarly allocating Reduce tasks to lowest-burden workers, forwarding the locations of all `M` intermediate files to each worker. We make another simplifying decision here: instead of retrieving files from peer workers via RPC, the Reduce workers simply query GFS with the given node and intermediate-file IDs. Despite deviating from the original paper's specifications, we note that this still comes at the cost of an RPC call to the GFS service. The Reduce workers then aggregate all key-value pairs from all intermediate files, sort them to aggregate keys, and applies the `Reduce` function. Finally, the Reduce worker writes the full reduced output for its partition to an output file, and notifies the leader of its location via RPC as in the Map phase.

2.5. Return to User

The leader waits for all `R` Reduce tasks to complete, then reads the `R` output files off the disks of the Reduce workers via RPCs to GFS. It "returns" to the user by writing those `R` output files, in formatted key-value pairs, to a local file on the user's local machine, where it is assumed the user can pass those outputs to another MapReduce model or process them in a distributed manner.

Once again, the leader schedules backup tasks during the Reduce phase to mitigate tail latency effects: if a Reduce worker has not responded within a fixed timeout, the leader schedules another Reduce task for the most idle worker. The tradeoff is, again, between worker and network throughput and expected latency and availability/fault tolerance guarantees.

3. MORE TRADEOFFS

It is important to note that throughout our implementation, we were working on a far smaller scale than that of the original paper's assumptions. Google's GFS spans hundreds of thousands of machines, and is optimized for massive datasets on the scale of terabytes rather than a large number of smaller files [2]. While the efficacy and applicability of distributed MapReduce is perhaps more pronounced at such large scales, our implementation did not have access to such massive scales of data.

However, because our implementation followed the original specifications, our implementation is extremely flexible and scalable. On just one machine, we were able to run MapReduce with 15 workers and 40 reduce tasks with no loss of correctness or latency, only encountering upper limits as a result of our own machine's CPU and network limits. Instead of running locally, changing one input configuration allows the user to specify worker nodes across other and arbitrarily many machines with more computational power, and our implementation remains fault-tolerant and consistent insofar as guaranteed by the original MapReduce paper.

4. TESTING AND RESULTS

We test our distributed MapReduce implementation by simply providing it with input files and a Go plugin, with Map and Reduce functions defined to implement particular use cases as mentioned before. We then compared it against gold outputs generated by sequential, non-parallel and non-distributed MapReduce outputs with the same inputs and plugins. We ran our MapReduce implementation under the simple abstracted use case of word frequency counting, on free publicly available .txt files taken with attribution from Project Gutenberg and from MIT 6.5840 Lab 1's tests [3]. We aggregated the outputs of the `R` outputs using a custom Reduce script, and compare with the output of sequential MapReduce on the same input files to test for correctness.

Our implementation was correct when tested on one local machine on four input files, with a number of workers equal to a range of values between 1 and 15, and on a range of partitions between 3 and 30. Moreover, our implementation was also fault-tolerant: by providing a higher number of workers to the leader than we actually instantiated, some nodes were essentially non-existent (simulating downtime). In this case, the leader would simply reallocate failed Map tasks to other idle workers upon failure to connect to the client, proceeding with the Map and Reduce tasks until completion.

5. RELATED WORK

As mentioned throughout, our implementation adhered closely to the specifications of the original MapReduce paper, but drew inspiration from MIT 6.5840 Lab 1 in terms of detailed implementations, and in particular the decision to separate the Map and Reduce phases. MIT 6.5840 Lab 1 is the most prominent implementation of MapReduce that operates on a smaller scale as compared to ours. Regardless, our implementation makes several changes relative to Lab 1. For example, Lab 1 reads and writes all files, including input, intermediate, and output files, to the local disk of the machine running the user's MapReduce program. While this was probably done for lightweight installation and testing, it disregards the presence of GFS in the original paper, as well as potential benefits of data locality. Our implementation remains flexible with respect to the underlying file system. Our mock-GFS service is exposed merely as an abstracted service: any service that exposes a Read and Write API can be substituted in place of it. Thus, our implementation lends itself far more conveniently to scalability to much larger distributed file systems and thus much larger computations on input files spread across those file systems, as our implementation is only limited by the compute resources of the worker nodes in question.

Another well-known open-source implementation of MapReduce at scale is Apache Hadoop, which provides APIs to facilitate large-scale distributed dataset computing [4]. Hadoop exposes a MapReduce library which operates similarly to that described by the original MapReduce paper: it operates over the Hadoop Distributed File System (HDFS), and assumes that input files are already hosted on nodes in HDFS. Thus, our implementation is an extremely simple prototype that offers more flexibility in allowing users with smaller local files and smaller machine clusters to quickly apply the MapReduce programming model, but which scales to larger distributed systems and can be integrated with distributed file systems such as HDFS.

6. CONCLUSION

In this writeup, we implement and discuss our implementation of distributed MapReduce, largely following specifications of the original paper. We present our design choices to mimic GFS, as well as to implement worker and leader logic. Future work might seek to continue implementing features that we have not yet incorporated from the original MapReduce paper, such as locality and leader fault tolerance; test our implementation on more abstractive use cases; test our implementation at scale with multiple machines instead of multiple services running on one local machine; and further improve on the flexibility and general cleanliness of our code and implementation.

7. REFERENCES

[1] Jeffrey Dean and Sanjay Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150.

[2] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The google file system," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, 2003, pp. 20–43.

[3] "Mit 6.5840," 2023, Accessed on May 8, 2023.

[4] "Apache hadoop: Mapreduce tutorial," 2023, Accessed on May 10, 2023.