6.033 System Critique: MapReduce

David Amirault

March 16, 2018

1 Introduction

MapReduce is a system for automating the parallelization of programs that create and parse large datasets. As a programming model, MapReduce utilizes a functional programming style in which programs must be expressed as two functions: Map and Reduce. MapReduce imposes a strict structure on the inputs, outputs, and interplay between these two functions (2.2). The implementation of MapReduce then takes care of scheduling the execution of the program across many machines. The key features of this implementation are efficient management of inter-machine communication, graceful handling of machine failures and “stragglers” that would otherwise interfere with the computation, and maximizing performance on Google’s custom distributed file system and computing environment (3). MapReduce provides a scalable and easy to use framework for the parallelization of programs that is accessible to software designers without knowledge of the nuances and messy details of fault tolerance and load balancing in distributed applications, thus leading to its popularization. This system successfully addressed a gap in the area of automating parallel-processing applications, as it had no precursor that was as widely accessible. However, MapReduce is critically limited in the space of tasks that can be expressed in map-reduce form, which often implies that MapReduce is only applicable to a narrow subset of possible tasks.

1.1 Scope

In order to minimize the complexity of using MapReduce, the system only supports the parallel execution of tasks that can be expressed in terms of a Map and a Reduce function. Map takes an inputted key-value pair and produces intermediate key-value pair(s), while Reduce takes a list of all the values corresponding to a single intermediate key and combines these values into a typically smaller list of values. Such programs only produce deterministic outputs when the ordering of the Map tasks and Reduce tasks are irrelevant (3.3). Although this structure is applicable to a number of problems, including counting tasks, distributed sorting, and reversing large directed graphs (2.3), in practice this constraint is MapReduce’s most severe limitation. Implementing a MapReduce application is remarkable simple, but it only works on a specific subset of tasks.
1.2 Fault Tolerance

MapReduce takes advantage of a master-worker architecture in which the master copy of the program is responsible for distributing the Map and Reduce tasks between the remaining worker machines and redirecting Map outputs to the corresponding Reduce inputs. By pinging each worker periodically (3.3), the master is able to handle large-scale worker failures by re-assigning dropped tasks to other workers. Although this model cannot recover from a master failure, this case is sufficiently rare that it is not cause for concern. As the Map and Reduce tasks near completion, the master dual-assigns incomplete tasks to idle workers, thereby addressing the issue of stragglers that take unexpectedly long to finish their tasks (3.6). This simple backup task optimization led to an observed runtime improvement of 44% (5.4), which implies that MapReduce functions impressively well in the presence of unreliable workers. This is further supported by the very limited observed performance degradation when worker processes were deliberately killed (5.5).

1.3 Scalability

The master provides a centralized means of communicating between machines and storing the status of each task and worker (3.2). Although this architecture allows worker failures to be addressed quickly, which is a significant efficiency benefit in Google’s computing environment consisting of clusters of many unreliable machines (3), it also introduces a potential performance bottleneck. This emphasizes the scope problem of MapReduce: not only must the program be expressible in map-reduce form, but the constituent Map and Reduce tasks must be numerous enough to justify parallelization across many machines but no so numerous that the master cannot effectively communicate with all the workers. The paper avoids a potentially illuminating comparison between the performance of a parallelized MapReduce program and a sequential implementation (5), which raises the question of how frequently MapReduce is actually applicable. However, MapReduce’s impressive fault tolerance and subdivision flexibility allow it to scale to large tasks within its narrow band of use cases.

1.4 Efficiency

Although the implementation of MapReduce handles the low-level details of parallelizing the task efficiency, the user exercises significant control over the realized runtime by setting the parameters for number of subdivisions of the Map and Reduce spaces (3.5). MapReduce provides a default partition function of these two spaces to tolerate users who wish to be fully abstracted from the parallelization. Simultaneously, MapReduce supports custom partition functions to allow users to optimize this aspect of the load balancing for their specific use cases (4.1). Hence MapReduce is able to avoid a tradeoff between ease of use and efficiency by handling all aspects of task granularity that the user does not explicitly control.
1.5 Conclusion

Overall, MapReduce accomplishes its goal of providing a way for users without knowledge of distributed systems to write parallel applications. By expressing previously-implemented distributed code for document indexing in MapReduce, the authors were able to reduce the number of lines of code from 3800 to 700 (6.1), which highlights the simplicity of using MapReduce. This simplicity is its greatest strength and is the reason why MapReduce fills a gap in the area of parallelized application automation. Like with UNIX, MapReduce’s simplicity and ease of use have led to its widespread implementation across a variety of programming languages. Today, this proliferation is seen in Apache Hadoop, an open-source implementation of the MapReduce framework.

1.6 Works Cited


Word Count: 878