Resilient Distributed Datasets in Python

6.824 Final Project - Spring 2013

Eben Freeman & Jonas Helfer

{helfer,emfree}@mit.edu

May 10, 2013

Abstract

Distributed computing framewokrs such as MapReduce are well-suited to certain tasks, but poorly suited to others. For example, iterative computations such as PageRank or many machine learning algorithms require running multiple MapReduce jobs, resulting in poor performance. The concept of resilient distributed datasets addresses these limitations by providing a way to persist data in memory across computations. We implemented a simple version of RDDs in Python and successfully tested it on the PageRank algorithm.

Introduction

Running multiple MapReduce jobs requires reading to and writing from disk in between each job. This limits performance, and makes live interaction with a dataset unnecessarily cumbersome. RDDs address both these limitations by caching data in memory. A resilient distributed dataset (RDD) is a distributed key-value set together with a computation history that specifies how it was constructed.[2] Transformations such as map, reduce, filter, or join can be applied to yield a new RDD. Users create RDDs by reading initial data from disk and specifying an appropriate series of transformations. These can be evaluated lazily, with a computation only executing when the program needs to yield data to the user or write back to disk. Users can specify the partitioning of data across machines, or use a reasonable default hash. RDD data is held in memory whenever possible, but can be spilled to disk or replicated across machines as needed. In the event of worker failure, lost data from an RDD can efficiently be reconstructed by using the RDD's lineage to recompute the missing partitions.

We use a delay scheduling algorithm to schedule computations across worker machines.[1] Delay scheduling gives preference to workers that already have the requisite data in-memory, but provides flexibility to mitigate the effect of stragglers.

Implementation

We implemented our system in Python. Our system is lightweight and standalone, and does not require any particular additional software such as HDFS. We first considered using Go to leverage our existing codebase, but we decided to use Python, because it makes serializing functions, a crucial aspect of RDDs, a lot easier.

Our system consists of one master and a variable number of workers that can be added to and removed from the master's pool of workers. Workers can be added and removed at any time, even in the middle of a computation. Thus our implementation is also tolerant to worker failures.

Workers are RPC servers that spawn a new thread for every request and can thus execute several requests concurrently. While the master will try to only send one request at a time to any worker, the workers need to be able to send requests for data to each other which need to be processed concurrently.

The scheduler runs on the master. The scheduler can be invoked from a program running on the master or interactively by the user. If the user passes an RDD to the scheduler, the scheduler will try to add it to the dispatcher queue. If the RDD has no parents or if the RDD's parents have already been computed, the partitions of the RDD are added to the dispatcher queue. The jobs in the queue are assigned to workers according to the delay scheduling scheme [1], which balances data locality and worker load. It is highly preferable to schedule a task on the worker that has the input data in memory because fetching it from another worker will take time and network bandwidth.

In principle, multiple users could interact concurrently with the same scheduler without any problem, as all the state is stored in the RDDs.

Fault tolerance

As mentioned before, our system treats worker failures the same way as planned removals and is able to adapt on the fly. We simulate fail-stop failures by simply removing workers from the pool while they are doing computations. The RPC call on the master will then time out, and the scheduler will find a new worker to execute the task. If a worker fails to fetch data from another worker, it notifies the master by returning an error. The master then tries to ping the presumably failed server. If the server does not respond for a set amount of time, it is removed from the worker pool and the master schedules the now missing partitions for re-execution.

Our system is also tolerant to network failures, such as arbitrary delays and network failures. Since all computations done on the workers are deterministic, dispatching the same task to two workers or to the same worker twice has no negative consequences, apart from the computation overhead. In a similar fashion, the system deals with stragglers by scheduling partitions to be re-executed if they are not completed within the RPC timeout. Of course the timeout should be chosen in such a way that the workers actually have time to complete their computations.

Since there is only one master server, the system cannot tolerate its failure.

Results

We tested our system on a single dual-core machine. We simulate independent workers by using multiple threads that communicate by RPCs. While that allowed us to debug and test our code for race-conditions or deadlocks, it did not allow us to do a meaningful performance evaluation. However, since our system is intended as a proof-of-concept and in no way optimized for performance, we consider this an acceptable drawback in light of the gain of simplicity compared to actually testing on multiple machines.

We were able to implement all the transformations necessary to get a working implementation of PageRank that was tolerant to fail-stops, arbitrary network delays and dropped RPC calls.

Our system can also be used interactively, for example through ipython, which would allow users to analyze large datasets interactively, which would be very useful for big data applications.

Conclusion

We built a simple python implementation of Resilient Distributed Datasets with all the transformations necessary to run the PageRank algorithm, and demonstrated that it works on a small dataset of pages, links and ranks. Our system is not at all optimized for performance, but it has all the parts necessary and should thus be seen as a proof of concept. We had a lot of fun implementing it and overcoming all the challenges along the way. Initially, the biggest challenge was that data sets and transformations require a particular way of thinking about tasks. Apart from that, the most difficult things were keeping track of the locations of partitions and debugging RPC server failures that did not return a useful stack trace.

References

[1] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278. ACM, 2010.

[2] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, page 22, 2012.