John Burke
CS 240H
6/12/13

A Haskell Implementation of MapReduce

Abstract

The MapReduce programming model, introduced in 2004 in a paper by Google researchers Jeff Dean and Sanjay Ghemawat [1], has become a highly influential model for cluster-based computing on large sets a data, a field that has been blowing up in recent years as companies amass more and more data and find more and more ways to use it. Google has not released their own implementation into the wild, but the open source Apache Hadoop implementation has become extremely popular, and many other services, such as Hive and Pig, have been built on top of it. However, as any Hadoop user can attest, the experience of writing code for Hadoop suffers from complex APIs and a large, internecine hierarchy of classes, due mostly to the fact that it is written in Java. In this paper, I present an implementation written in Haskell, in the hopes that the many wonderful features distinguishing Haskell from Java, and the fact that MapReduce's inherently functional nature maps better to a functional language, can create a better experience for programmers who wish to use MapReduce.

Introduction: MapReduce

The MapReduce model takes its name from two of the main canonical functions of functional programming: map, which applies a function to each element of a collection, and reduce, also known as fold, which uses a given function to combine a collection into a single result. A MapReduce execution is analogous to the application of these two functions: first, a "mapper" function is applied to all of the pieces of some input file; then, a "reducer" function is applied to the results of the mapper phase. Each mapper outputs results in the form of (key, value) pairs, and each reducer receives a single key and a collection of all of the values that were emitted by mappers together with that key.

The difference between MapReduce and simply applying maps and folds in a similar fashion is that MapReduce distributes these computations across a cluster of machines. MapReduce runs on top of a distributed file system so that the nodes can easily pass data around. The underlying MapReduce framework, mostly handled by a central master node, takes care of starting up jobs at nodes, coordinating between them, handling node failures, and combining the outputs of the mappers into the form expected by the reducers.

Introduction: Cloud Haskell

In order to implement this in Haskell, I needed some kind of platform for distributing computations across nodes in a cluster. Writing my own would probably have taken more than the time I had in and of itself, so I went searching for distributed programming solutions already made for Haskell. Unfortunately, it turns out that Haskell as it currently stands is not particularly well suited to distributed programming, and there are few extant solutions. Of the systems I found, nearly all required their own runtime,

only a couple seemed to have been touched in the last 4 years, and all were poorly documented. The system that I finally settled on was the least bad of these options: Cloud Haskell.

Cloud Haskell is a distributed programming library first described in a 2011 paper by Simon Peyton-Jones, Andrew Black and Jeff Epstein. Unlike Eden or GdH (Glasgow distributed Haskell), Cloud Haskell runs on the standard GHC implementation of Haskell, which was what initially drove me to choose it. The distributed programming model implemented by Cloud Haskell is inspired by that of Erlang, in which processes share nothing and communicate only through message passing. That sounds simple enough, but the tricky part comes when you start to consider the details: how does one start up a process on another machine and get it to run some function in Haskell? It turns out that in order to do so, one needs to be able to serialize function closures and send them across the network, and GHC intentionally does not support the serialization of functions in the interest of preserving type safety. The need for this support is why most other distributed Haskell systems have their own runtime system: they simply add support for serializing functions. The developers of Cloud Haskell found a different way to attack this problem: by making certain functions entirely known statically at compile time, and by assuming that all nodes are running the same executable, they can essentially serialize these functions by simply sending a code pointer along with a representation of the functions environment (basically just an argument and some instructions on how to interpret the values and what types to give them). In order to accomplish this, they use Template Haskell, an extension that adds compile-time metaprogramming, to allow the programmer to generate all the necessary annotations from the names of functions and types they have already defined.

Unfortunately, this approach has some pretty serious limitations. Because they are run at compile time, Template Haskell functions operate on names rather than actual Haskell constructs. This means that if you need some functions passed into your library to be serializable, you cannot hide that fact from your users: they must use the Template Haskell functions provided by Cloud Haskell themselves and make serializable functions that they can then pass to you, as well as making serialization dictionaries that define how to deserialize the types involved. Furthermore, these provided functions are a bit strange and difficult to use, making them even more undesirable for a library writer. But alas, Cloud Haskell appears to be the best option for distributed Haskell programming at the moment, and it has far more examples and tutorials on the web than the other options out there. It is also currently under development, so it will hopefully improve in the near future and maybe even find a way to throw off the shackles of Template Haskell.

Design and Implementation

In designing my implementation of MapReduce, I strove mostly for two things: simplicity and generality. Rather than restricting the user to a single map phase and a single reduce phase, I allow any number of either composed together into a defined order. Mappers and reducers are both the same type: Workers. The difference is in the type of function from which they are constructed, and the fact that a mapper processes data in a streaming fashion as a it comes in, while a reducer must wait until all the data from the preceding stage has arrived so that it can see all of the values associated with a given key. At either end of this chain of workers is a TupleSource and a TupleSink. As the names

imply, the TupleSource reads file data and sends it out to the first Workers in the chain, while the TupleSink receives data from the last Workers and writes it to a file.

Users of the library can create mappers, reducers, sinks and sources by using the provided functions: makeMapper, makeReducer, makeSink and makeSource. These functions take a function for processing or reading/writing the data as an argument and produce another function in which all of the inter-process communication is handled and the passed-in function is called as appropriate. Due to the Template Haskell basis of Cloud Haskell, users must call these functions within a named top-level function, and then use the mkClosure and remotable Template Haskell utilities provided by Cloud Haskell to make them serializable, as well as defining a static serialization dictionary for the types involved. They can then compose together their mappers and reducers into a WorkFlow using the WorkFlow and SingleWorker constructors as defined in the source.

Once they have these serializable functions, users can create a MapReduceJob, consisting of a path to a local data file, a TupleSource, a TupleSink, a WorkFlow, and a static serialization dictionary for the output type of the end of the WorkFlow. A configuration file is used to define the number and locations of nodes in the network. The MapReduceJob and configuration file can then be passed to runMapReduce along with the remote table (a template-generated data structure matching names to code locations and static serialization dictionaries), and runMapReduce will launch the master node, fire up each Worker process on each worker node, and then start the computation.

Example: WordCount

As an example of how to use the API, here is the classic MapReduce example: counting word frequencies (imports omitted for brevity, actual source attached).

```
wordCountReader ::([OutPort String String] -> (ReceivePort String
      -> Process ()))
wordCountReader = makeSource (\h -> do
  contents <- hGetContents h
  hPutStrLn stderr "Read from file"
  return $ map (\w -> ("from file", w)) $ words contents )

wordCountWriter :: (FilePath -> InPort String Int -> Process ())
wordCountWriter = makeSink (\pair h -> hPutStr h $ show pair)

countMap :: ([(OutPort String Int)] -> InPort String String
      -> Process ())
countMap = makeMapper (\_ word -> (word, 1))

countReduce :: ([(OutPort String Int)] -> InPort String Int
      -> Process ())
countReduce = makeReducer (\word countList -> [(word, (length
      countList))])

sdictStrStr :: SerializableDict (Maybe (String, String))
sdictStrStr = SerializableDict

sdictStrInt :: SerializableDict (Maybe (String, Int))
```

```
sdictStrInt = SerializableDict

remotable ['wordCountReader, 'wordCountWriter, 'countMap,
       'countReduce, 'sdictStrStr, 'sdictStrInt]

countMapper = Worker ($(mkClosure 'countMap))
                                ($(mkStatic 'sdictStrStr))

countReducer = Worker ($(mkClosure 'countReduce))
                                ($(mkStatic 'sdictStrInt))

wordCountMR :: FilePath -> MapReduceJob String String String Int
wordCountMR = \f -> MapReduce f
                                                ($(mkClosure 'wordCountReader))
                                                ($(mkClosure 'wordCountWriter))
                                (WorkFlow (SingleWorker countMapper) countReducer)
                                                ($(mkStatic 'sdictStrInt))

rtable :: RemoteTable
rtable = MR.__remoteTable $ initRemoteTable

main = do
  args <- getArgs
  case args of
    "master" : confFile : inputFile : [] -> runMapReduce confFile rtable $
wordCountMR inputFile
    "slave" : host : port : [] -> runWorker host port rtable
```

Conclusions and Future Work

As you can surely see, this API is not quite as clean or simple as most would desire, including myself. As mentioned above, this is due mostly to the limitations of using Template Haskell. Furthermore, understanding and working with the template-based features of Cloud Haskell proved to be a trying experience, necessitating a couple of rewrites (and leaving me without even compiling code at the time when I had to present my project).

However, there may be another way. Very recently (too late for another bottom-up rewrite) I realized that, for the specific case of implementing a MapReduce system in which all of the nodes perform all of the functions (ie all are both mappers and reducers), function serialization is not necessary. Instead, given that we are already assuming all nodes are running the same executable, one could design such a system by simply having the nodes synchronize with each other over the network and pass data back and forth as they do already. This approach removes the need for any Template Haskell nastiness and opens the way to a much more elegant API in which the user can simply pass in functions and data without worrying about implementation details. This would allow far more flexibility as well, and make it possible for the functions run on worker nodes to contain monadic computations. It's possible that the system could even be implemented as a Monad or a Monad Transformer itself, where all of the details of network communication would be hidden away within the bind and return functions. Such a system would be

much cooler and more Haskell-flavored than what I have presented here, and I plan to try to develop it this summer. If I manage to get it done, the results will appear in the following github repository, so stay tuned if you are interested: https://github.com/jcburke14/MapReduceHS.

References

1. Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters.* Google, Inc. Appeared in OSDI 2004.
2. Jeff Epstein, Andrew Black, and Simon Peyton-Jones. *Towards Haskell in the Cloud*. Haskell '11. September 22, 2011.