Mr. HS, a Haskell MapReduce framework

Ye Ji

May 5, 2015

Abstract

The original MapReduce [1] framework has some limitations and drawbacks: (1) computation is restricted to two phases; (2) computation is not type safe. In this paper, I present a more generalized MapReduce framework implemented in Haskell that addresses the two issues.

1 Introduction

The traditional MapReduce framework uses a master-slave architecture and divides a distributed computation into two main phases (*map* and *reduce*) with an interlude phase:

1. **Map** phase, where each slave execute the same function on different data:

$$map : (k_1, v_1) \to [(k_2, v_2)]$$

2. **Interlude** phase, where results from map phase are sorted then partitioned according to k_2:

$$interlude : [(k_2, v_2)] \to [(k_2, [v_2])]$$

3. **Reduce** phase, where each slave executes the following function on a partition of the sorted results

$$reduce : (k_2, [v_2]) \to [v_2]$$

The programmer can provide these functions together with the inputs and the number of mappers and reducers to the framework, and not worry about the intricacies of actually distributing the data and computation.

One might wonder, why only two main phases? Some of the computation might only need the map and the interlude, where results from map are sorted, e.g. the distributed sort example described in the original paper. Some computation might only need the map phase, where data is simply transformed from one format into another. Some computation might need more phases than just map and reduce.

Also, it is not hard to spot that the composition of ($interlude \circ map$) function and *reduce* function can be generalized into one type of function:

$$process : (k_1, v_1) \to [(k_2, [v_2])]$$

This suggests that we can generalize the MapReduce framework into arbitrary number of phases.

Also, the original framework passes strings as inputs and outputs and relies on the user code to decode the strings into appropriate types. This probably makes the framework more flexible. But it is hard to enforce type safety between the output of the map function and the input of the reduce function.

It is desirable that we can catch as many programming errors, such as type errors, as we can during the compile time rather than run time, since the computation can be potentially distributed to and run on thousands of machines.

This paper is to present a framework, Mr. HS, that addresses the two issues. Section 2 describes the computation model and implementation of Mr. HS. Section 3 describes a classic type-safe MapReduce framework that is built on top of Mr. HS.

2 Multi-stage MapReduce

Mr. HS uses the master-worker model. Master is in charge of coordination while workers do the actual computation. Although in my implmentation, the master and worker are both in Haskell, the master can work with worker servers implemented in any language using any framework, as long as the worker implements the appropriate communication protocol.

2.1 Programming model

The computation of Mr. HS is divided into arbitrary number of stages specified by the application programmer. The programmer supplies the function used for each stage and number of partitions of each stage to the framework, the framework will then return two programs (a Haskell `IO` action), one is the master program, the other is the worker program.

The function used for each stage on a single worker are of the following type:

$$process : [String] \to [String]$$

At stage n, a worker takes a list of strings, one from each worker of Stage $(n-1)$, and outputs a list of strings, one for each worker of Stage $(n+1)$.

At the end of each stage, the output of each worker is aggregated and ready to be fed into the next stage. So the entire stage is:

$$stage : [[String]] \to [[String]].$$

Now we can easily compose computation by chaining arbitrary number of stages:

$$computation = stage_0 \circ stage_1 \circ \cdots \circ stage_n.$$

The meaning of the input and output type are for the application programmer to define. `String` can be parsed at runtime to be a file name of a file that is stored in the ambient distributed file system; or it could be keys of a distributed database.

For the sake of generality, I took a simplistic approach designing the programming model. Richer semantics can be added to the framework by easily adding a layer between the framework and the app. E.g. I implemented the classic MapReduce framework on top of Mr. HS by restricting number of stages to be 2 and adding sorting between the first and second stage.

2.2 Master-Worker model

In Mr. HS, the master server coordinates the whole computation. It serves as the synchronization between stages: it tells each worker server which stage function to perform on what inputs; it then collects and aggregates the results from each worker, and starts the next stage.

The types of inputs and outputs are mentioned in the previous section: lists of strings. Since they are to be passed over the network, it is advisable not to put actual data in the input and ouput. Instead we should use "pointers" to data, such as file names.

To perform a MapReduce computation using Mr. HS, the programmer starts the master server, supplying it with the number of stages, number of partitions of each stage, and the initial batch of inputs.

The worker servers can start joining the computation once the master server starts and they can join at any point of the computation regardless of how many stages have already been performed. This enables us to dynamically resize the computation power. The worker servers first register with the master by requesting a websocket connection. The master will accept the request and exchange messages with the worker over the connection thereafter. After the acceptance of the websocket request, the master puts the newly joined worker into a pool of "idle" workers. When executing a stage, the master spawns one "process" thread for each data partition for that stage. The "process" thread then fetches one of the idle workers from the pool and sends it a message that encodes which function to execute, what the input strings are, and number of partitions for the next stage. Upon receiving the message, the worker parses it and starts the corresponding computation. The message is in JSON format, e.g.

```
{
  "wcReducerCount":5,
  "wcWorkerID":6,
  "wcInput":["hello","world"],
  "wcStageID":3
}
```

This means that the worker should execute the function of Stage 3 with the input "hello" and "world"; the worker should also partition its output into 5 slices.

After it finishes the computation, the worker reports the output back to the master in a similar format. The "process" thread on the master then puts the worker back into the idle worker pool so that the worker can be used for a new computation issued by other "process" threads. If, for some reason, the worker server failed and cannot report back the results, the "process" thread just closes the websocket connection and fetches another worker server from the

worker pool and restarts the computation. The current implementation can not recover from master's failure. If this happens, the whole computation has to restart from Stage One.

After all the stages are performed, the result of it, a list of lists of strings, is presented to the application programmer.

3 Type safe MapReduce

In order to address the type safety issue, I built on top of Mr. HS a classic two-phase MapReduce framework that is type safe. The main idea is that the application programmer has to provide three functions (`MapperFun`, `Hash`, `ReducerFun`) that have matching types, in order to form the correct type, `MRFun`, to be passed into our library function `mapreduceWith` and get the master and worker `IO` actions (some details of the code are omitted):

```
type MapperFun k v =
      String             -- ^ file name
    -> ByteString        -- ^ contents of the file
    -> [(k, v)]          -- ^ output

--| hashing function for partition
type HashFun k = k -> Int

type ReducerFun k v =
       [(k, [v])]
    -> ByteString       -- ^ output

type MRFun k v =
  (MapperFun k v, HashFun k, ReducerFun k v)

--| takes a trio and number of partitions
--returns two io actions, master and worker
mapreduceWith :: MRFun k v -> Int -> (IO (), IO ())
```

The framework will take care of encoding, writing to disk, and decoding during the interlude. This way, we ensure that the mapper will produce outputs whose type matches the type of the reducer's input.

Here is an example of word count implemented using this framework:

```
{-# LANGUAGE OverloadedStrings #-}
import Network.MapReduce.Classic
import System.Environment
import Control.Monad
import Control.Concurrent
import qualified Data.ByteString.Lazy.Char8 as BL
import Data.ByteString.Builder
import Data.Monoid

mapper :: MapperFun BL.ByteString Int
mapper f v = map (\x -> (x, 1)) (BL.words v)
```

```
reducer :: ReducerFun BL.ByteString Int
reducer = BL.unlines .
      map (\(k, l) -> toLazyByteString $
           lazyByteString k
        <> char8 ' '
        <> intDec (length l))

main :: IO ()
main = do
  s <- getArgs
  let addr = "127.0.0.1"
      port = 8080
      (master, worker) = mapreduceWith
        (mapper, const 1, reducer) s 1 addr port
  replicateM_ (length s) (forkIO worker)
      >> master
      >>= BL.readFile . head >>= BL.putStrLn
```

This is the whole program with nothing omitted. We can see that the framework enables application programmer to write distributed programs with very few lines of code. I tested it on my Macbook Pro (Intel i7, two 3GHz cores, 256 KB L2 Cache, 4 MB L3 Cache). By using four worker thread, it took about 4.5 seconds to process four documents, each of which contains about 1.6 million characters. For comparison, a sequential version of the algorithm in Haskell needs about the same amount of time; a sequential version of the algorithm in C++ needs 2.7 seconds. But the mapreduce version does much more I/O (writes and reads four intermediate files).

It is quite straightforward to implement a type safe two-stage MapReduce in Haskell. However, it seems impossible [2] to implement a type safe framework for computations that have arbitrary number of stages.

4 Future work

The framework that I presented is just a prototype. Many things need to be improved before it is "production" ready.

1. Implement worker library in a more "user" friendly language, e.g. python
2. Provide interfaces to popular distributed file systems, e.g. HDFS, S3, etc.
3. Improve performance of the type-safe MapReduce, which currently uses in-memory sort

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[2] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.