CS240H Final Project

Arpad Kovacs SUNet ID: akovacs

March 18, 2016

For my CS240H final project, I implemented the MapReduce abstraction in Haskell, and demonstrated an example wordcount application which runs on a single-node or a distributed cluster.

1 MapReduce Background

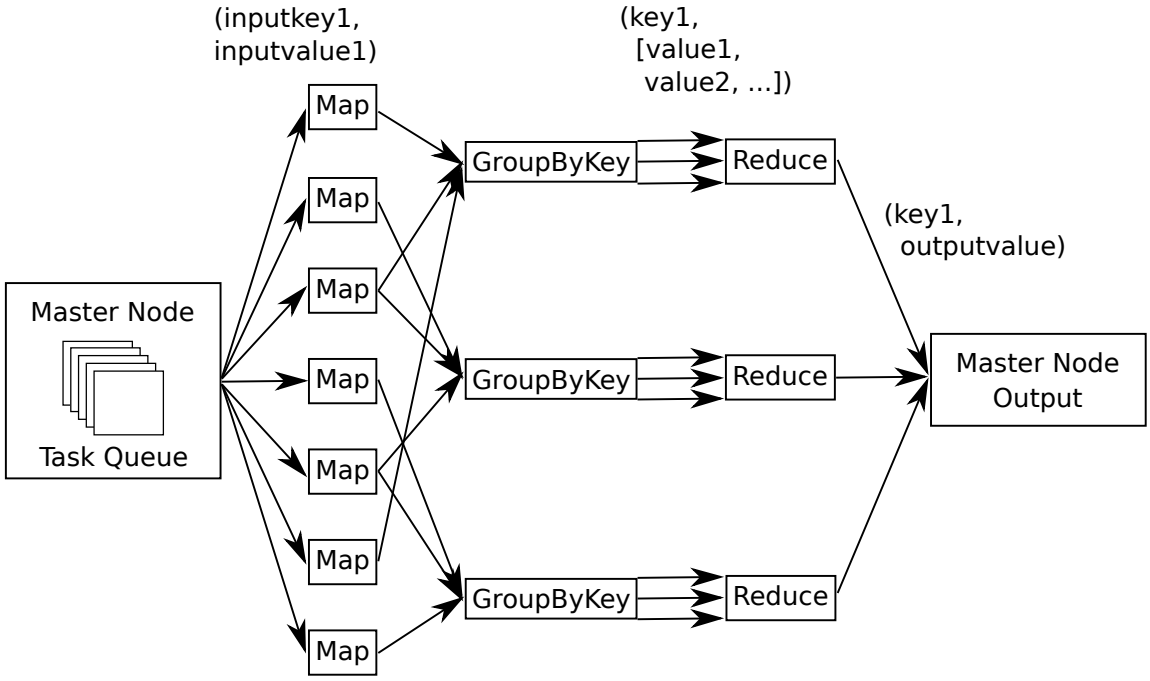
MapReduce is a parallel, distributed programming model introduced by Jeff Dean and Sanjay Ghemawat from Google [1]. The abstraction is characterized by its namesake functions: **Map** processes an input key-value tuple, and outputs intermediate key-value tuples.

```
type Mapper inputKey inputValue intermediateKey intermediateValue =
(inputKey, inputValue) -> [(intermediateKey, intermediateValue)]
```

Reduce takes an intermedate key and list of intermediate values, and folds those values into single output value.

```
type Reducer reduceKey reduceValue =
reduceKey -> [reduceValue] -> reduceValue
```

While seemingly simple and limited, this abstraction enables the construction of scalable and fault-tolerant systems for processing large quantities of data in parallel. Specifically, because the map operations are independent, they can be executed in parallel on a shared-nothing architecture. Likewise, when the reduce operation is associative, or if keys are grouped to the same reducer, then reduce operations can also be parallelized. The diagram below shows how the mapper and reducer can be composed into a MapReduce system.



Although the conceptual underpinnings of MapReduce are simple and elegant, the available implementations at the time of this writing are quite complex or inaccessible. Google's original implementation remains proprietary, and therefore cannot be studied beyond the details published in the previously mentioned 2004 OSDI paper. An open-source implementation in Java named Hadoop has become popular in industry, but suffers from excessive boilerplate configuration and setup code. Consequently, Haskell, which is a statically-typed pure functional programming language, could provide the foundation for a more elegant and type-safe, yet still expressive MapReduce implementation.

2 Single Node MapReduce Implementation

I started by implementing MapReduce on a single-machine. My implementation takes as input a single Data.Map containing the input keys and values to be processed. Given the earlier Mapper and Reducer definitions, I defined the single-node MapReduce as:

```
-- Single-node mapReduce implementation
mapReduce :: (Ord reduceKey)
          => Mapper inputKey inputValue reduceKey reduceValue
          -> Reducer reduceKey reduceValue
          -> Map.Map inputKey inputValue
          -> Map.Map reduceKey reduceValue
mapReduce mapper reducer = reduce reducer . groupByKey . asList . mapInput mapper
  where mapInput mapper = concatMap mapper . Map.toList
        asList keyValuePairs = [(key, [value]) | (key, value) <- keyValuePairs]
        groupByKey = Map.fromListWith (++)
```

```
reduce reducer = Map.mapWithKey reducer
```

This initial single-node implementation is quite concise, yet expresses all the salient characteristics of the Map-Reduce paradigm. Each mapper operates on tuples, therefore we transform the input Data.Map into lists of tuples, and apply the mapper function to each tuple. We take the resulting lists, and combine them back into a Data.Map, using the concatenation operator ++ to merge lists of values that map to the same key. This operation is analogous to the shuffle/groupBy step of MapReduce, which ensures that all values for a reduce key map to the same reducer. The mapReduce function signature ensures that the output types of the mapper (reduceKey and reduceValue) align with the input types of the reducer at compile-time. The reduceKey must be constrained by the type Ord to provide an ordering of the reduce keys, since Data.Map is implemented internally as a self-balancing binary tree.

Given this framework, implementation of the wordcount application is fairly straightforward. Initialization code in main loads an input data corpus of 42 works by Shakespeare from the filesystem into a Data.Map which has document identifiers as keys, and each document's contents as ASCII bytestring values. Since filesystem access can be stateful, this is performed in the IO monad, however the subsequent mapper and reducer computations are pure.

The wordCount mapper operation named **countWords** takes inputs tuples of (docid, docContents) and splits the document contents on whitespace, emitting individual words each with a count of 1 as tuples.

```
-- Mapper takes input of (docid, docContents);
-- outputs list of (word, count=1) tuples
countWords :: (Docid, B.ByteString) -> [(Term, Count)]
countWords (fileIndex, fileContents) =
  map (\word -> (word, 1)) (B.words fileContents)
```

The MapReduce framework's groupByKey proceeds to group all tuples with the same key word together, emitting reduce groups with the word as the key and a list of the counts (1 for each occurrence of the word in a document) as the value. Finally, the **sumCounts** reducer sums each list of counts, outputting the total number of occurrences of the word across all documents in the corpus.

```
-- Reducer takes key=word and list of counts as input;
-- outputs total count of that word
sumCounts :: Term -> [Count] -> Count
sumCounts _ counts = sum counts
```

The final result is a Data.Map with each word occurring in the corpus as the keys, and the number of appearances of that word as the value.

3 Distributed MapReduce Implementation

3.1 Distributed Framework

Next, I added the ability to run MapReduce on a distributed cluster. I initially attempted to roll my own distributed runtime, but soon ran into challenges in implementing node and service discovery, message-passing, as well as synchronization. I subsequently looked into several preexisting distributed runtimes including Glasgow Distributed Haskell [4] and Eden [3] but found many of them

to be outdated, or lacking clear examples or documentation. I finally settled on the Distributed-Process library, an evolution of Cloud Haskell introduced by Jeff Epstein, Andrew Black, and Simon Peyton-Jones [2]. Cloud Haskell was inspired by Erlang-style actors, which do not have shared memory. Instead, actors (named Processes in Cloud Haskell) communicate explicitly with each other through messages. The messages are sent asynchonously, and Cloud Haskell provides no guarantees about the order in which messages are received.

One of the requirements for a distributed MapReduce implementation is the ability to serialize function closures, and send them to a remote node for execution. In my earlier homegrown distributed implementation, I attempted to accomplish this by taking the string representation of the mapper and reducer functions, sending them over the wire, and then compiling them into functions on-the-fly using an embedded Haskell interpreter. Unfortunately, this approach results in the loss of compile-time type safety, which was one of the main benefits of writing code in Haskell. Since GHC does not natively support serializing functions, the alternative runtimes implement custom compilers or special middleware to support this functionality. Glasgow Distributed Haskell retrofit the runtime with a shared distributed heap and distributed MVars for explicit synchronization and and communication between remote I/O threads. Eden augments the GHC compiler with distributed programming primitives, and runs the system on top of MPI middleware to achieve distributed message passing.

In contrast, the Distributed-Process library solves the problem using a different approach which does not require compiler or runtime customizations. Since all nodes are assumed to run the same codebase, we can register code pointers to functions to be called on remote nodes in a "remote table". To invoke a function on a remote node, the distributed-process system creates a closure, consisting of the code pointer and an environment representing the inputs to the function. I selected Distributed-Process as the foundation of my distributed MapReduce because this seemed like the cleanest approach for remote function invocation. However, the actual implementation utilizes Template Haskell and multiple wrapper functions due to Distributed-Process limitations, so the final codebase turned out to be considerably messier than the single-node implementation (and was also significantly harder to debug).

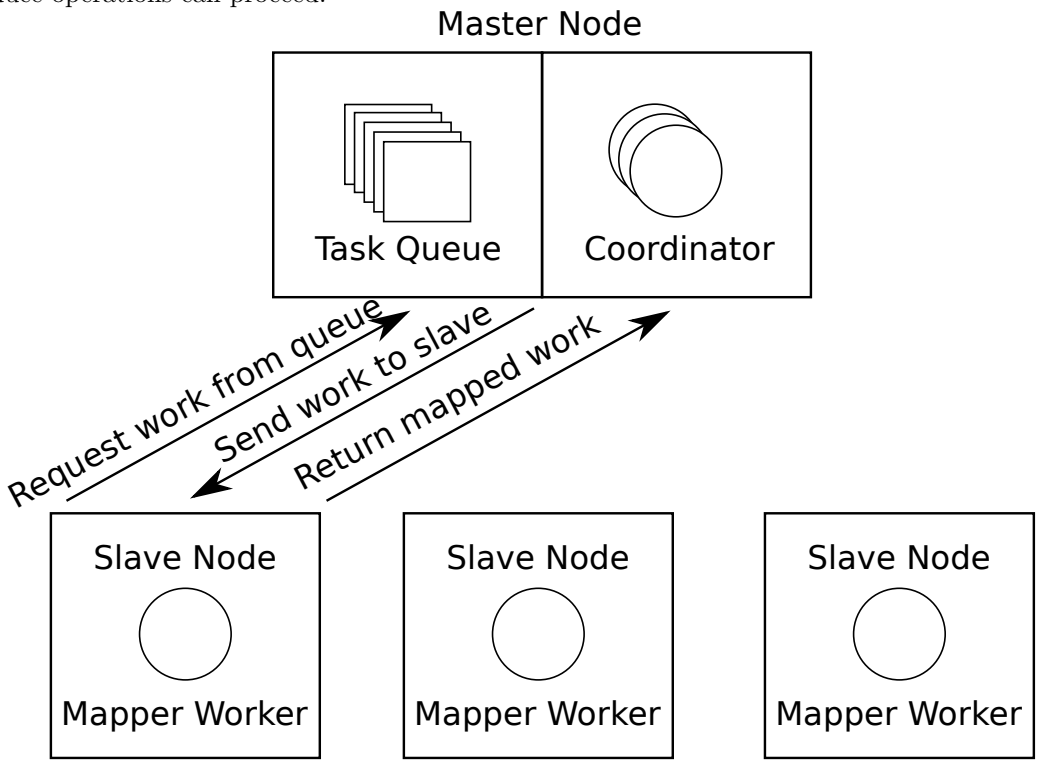
3.2 Network Topology and System Operation

My network topology consists of a single master node which controls multiple slave nodes. I opted to use the SimpleLocalNet topology, which uses TCP for messages and provides a simple node discovery mechanism using UDP multicast, since it seemed to be the best-supported and simplest Distributed-Process network topology.

We initialize the system by starting up remote nodes in "slave" mode. The slaves are initially idle and await further instructions from the master coordinator. We begin the MapReduce computation by starting up a master coordination process. The master coordinator first spawns a local process known as the "workQueue" which slaves can query for work, then reads all input data (a corpus of 42 works from Shakespeare for the wordCount task) from the local filesystem, and adds work units with the file index as the key and file contents as the value to the workQueue. The master proceeds by taking inventory of all available slaves using the SimpleLocalNet UDP multicast discovery mechanism, and initializes a mapperWorker process on each slave which continuously pulls the next item from the workQueue, applies the map operation to it, and returns the mapped result to the master process until the workQueue is exhausted.

Since the dedicated workQueue process is responsible for dispatching work units to mapper-

Workers on remote slaves, after initialization the master coordinator's only role is to collect the intermediate results (output of the map step) from the mapperWorkers. The master coordinator accumulates the intermediate results in memory until all work units have been processed, at which point it proceeds to the shuffle phase which groups values by key. By waiting to collect all intermediate results, the master node essentially creates a synchronization barrier between the map phase and subsequent shuffle and reduce phases. This simplifies the implementation, and ensures that the reducer successfully processes all values for a given key, however it also incurs a performance hit since the master must wait for any straggler mapperWorkers to complete before the shuffle and reduce operations can proceed.



In my initial implementation, the master node would spawn a separate slave process for each work unit. While this resulted in a high level of concurrency, it also exhausted resource limitations (specifically memory) when the work units were large. Furthermore, the master is required to push the tasks to the slaves upfront; if one of the slaves is slow or dies, then the work allocated to it may not be completed.

To rectify this problem, I restructured my code so that only a single worker runs on a slave at a time. The work units live in the separate workQueue process on the master node. The worker on each slave node requests one work unit at a time from the master's workQueue, executes the mapper on the data, and pushes the result back to the master. If the slave does not return a completed work unit back before a timeout, the work unit is put back in the workQueue, and any late responses from the worker are discarded. Since slaves pull work units on-demand, this achieves a rudimentary form of load-balancing, where idle slaves can request additional work until the work queue is exhausted. The drawback of this approach is that there is slightly lower utilization of the

slaves, since after completing a mapper computation, the slave must wait for the workQueue to send the next work unit to be processed.

4 Limitations and Future Work

In my current distributed map-reduce implementation, all input data lives on the master node, and is forwarded to the slaves when they request work units from the workQueue. This is not very efficient, as it turns the serialization and transfer of the data through the network into the main bottleneck. I contemplated building a distributed filesystem to solve this, however implementing replication, data locality, synchronization, and error-handling would be sufficient scope for another final project.

There is also room for improvement in the master-slave network topology. Since the SimpleLocalNet results in a fully-connected grid of nodes, the scalability of the system is limited. Observing that mapper nodes only communicate with the master node, a star or tree topology with the master at the center/root may result in lower networking overhead since mapper nodes would not need keep track of each other.

The current system is resilient to mapper failures; any "dead" mappers will timeout and fail to pull results from the workQueue or push results to the master. As long as the master and a single slave node remain operational, the slave's mapperWorker will continue pulling uncompleted work units from the master's workQueue and push completed map units to the master, so the system will continue making forward progress. However, the master node itself is a single point of failure in the system, since it is responsible for dispatching work units, collecting intermediate results, and performing the final shuffle/group/reduce operations. Adding leader-election capabilities or building a peer-to-peer MapReduce framework would be an interesting area of future work, although maintaining distributed state across the network and handling partitions of the network safely could be quite challenging.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. 1

[2] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM. 3.1

[3] Rita Loogen. Eden - parallel functional programming with haskell. In *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*, CEFP'11, pages 142–206, Berlin, Heidelberg, 2012. Springer-Verlag. 3.1

[4] R.F. Pointon, P.W. Trinder, and H.-W. Loidl. *Implementation of Functional Languages: 12th International Workshop, IFL 2000 Aachen, Germany, September 4–7, 2000 Selected Papers*, chapter The Design and Implementation of Glasgow Distributed Haskell, pages 53–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. 3.1