Implementing Apache Spark in Haskell

Yogesh Sajanikar

March 17, 2016 (CS240H)

Abstract

This paper presents hspark, a Haskell library inspired from Apache Spark. **Hspark** implements a framework to enable running a distributed map-reduce job over a set of *nodes*. **Hspark** also presents a extendible DSL to specify a job by dividing it into multiple stages. **Hspark** translates the DSL into a set of distributed *processes* with the help of *cloud Haskell* libraries.

1 Overview

1.1 Apache Spark

Apache spark is a very popular and fast cluster computing framework. It is reported to give significant performance benefits[1] above Hadoop. The *jobs* are specified in terms of RDD [1] (Resilient Distributed Data) in Spark. Each RDD does an atomic mapping or reduction step. When executed, an RDD along with its dependent RDDs are split into partitions. This step creates a DAG (Directed Acyclic Graph) between an RDD and its dependent RDDs. This DAG is then scheduled to run over a set of distributed nodes. The backend for execution can be either Hadoop or Mesos cluster. Use of in-memory blocks, and strategy to efficiently localize the data gives Spark a better performance.

1.2 Hspark

Hspark implements a simple and extensible DSL to specify a job. Hspark takes a configuration of cluster, and translates the job at runtime into a set of distributed tasks using distributed-process library of cloud haskell.

[1] `http://spark.apache.org/`

2 Hspark components

Hspark has three components

- *Context* - Context provides a information about cluster.
- *RDD DSL* - Provides a way of expressing *hspark* job.
- *Execution* - A backend integrated with RDD and context that executes RDD with its dependencies.

2.1 Context

A context specifies the environment and configuration of the cluster. The cluster consists of set of *nodes*. Each node works as a logical unit capable of running come computations. The nodes are separated from each other through a transport layer.

The essential components of the context are

1. **Master Node** - A master node triggers the job by distributing it on slave nodes in the cluster. After the job has finished, it also collects the data from all the nodes.
2. **Slave Node** - A slave node is a worker node. A master node spawns computations on slave node(s).

2.2 RDD

RDD is implemented as a type class. The type that is produced my an RDD must be *serializable* so that it can be transported over the wire to another node.

```
newtype Blocks a = Blocks { _blocks :: M.Map Int ProcessId }

class Serializable b => RDD a b where

    flow :: Context -> a b -> Process (Blocks b)
```

An RDD implements a method *flow* that uses a context, and triggers a process that returns *Blocks*. A process in *cloud haskell* is a lightweight action container. Each *block* is a *process id* of a process in a cluster.

A process being implemented asynchronously, *flow* can immediately return. The downstream application (or an RDD) must send a **Fetch** query to the process in a block to retrieve the data.

Each chunk of data for *Block b* is a list *[b]*.

```
-- pid is a process id contained in a block
-- Send a message to that PID, and wait for it.
do
  sendFetch dict pid (Fetch thispid)
  receiveWait [ matchSeed dict $ \xs -> return xs ]
```

2.2.1 Closure

Distributed-process (and hence *hspark*) heavily rely on closure, and *StaticPointer* extension provided by GHC > 7.10.x. A static pointer is implemented as a fingerprint of a closed expression that can be valid across machines, and can be dereferenced later on a different machine. [2]

An RDD accepts closure built around static values using composition, so that they can be serialized across nodes. Polymorphic types are serialized through rank1dynamic library, by building a remote table for methods.

Hspark currently implements following RDD.

2.2.2 SeedRDD - Populating the data

Seed RDD simply splits up the data and populates it across all partitions, or given number of nodes.

```
seedRDD :: Context
        -> Maybe Int   -- ^ Number of partitions
        -> Static (SerializableDict [a])
        -> Closure [a] -- ^ Input data
        -> SeedRDD a
```

2.2.3 MapRDD/MapRDDIO - Mapping with a function

A *MapRDD* is takes a parent RDD, and a function *(b -> c)* that maps RDD of type *a* to RDD of type *b*

```
-- | Create map RDD from a function closure and base RDD
mapRDD :: (RDD a b, Serializable c) =>
          Context -- ^ Context
       -> a b -- ^ Parent RDD
```

```haskell
        -> Static (SerializableDict [c])
        -> Closure (b -> c)
           -- ^ Transformation
        -> MapRDD a b c
           -- ^ Map representing transformation (b -> c)
```

A *MapRDDIO* is similar to *MapRDD* except that it takes an IO action *(b -> IO c)*.

2.2.4 ReduceRDD - Reducing with a combining function and a partition

A *ReduceRDD* works a parent RDD that produces key value pair *(k,v)*. Hence *ReduceRDD* and its RDD *instance* are designed as,

```haskell
data ReduceRDD a k v b

-- | Constraint parent to produce a key-value pair.
instance (Ord k, Serializable k, Serializable v, RDD a (k,v))
    => RDD (ReduceRDD a k v) (k,v) where

reduceRDD :: (RDD a (k,v), Ord k, Serializable k, Serializable v) =>
             Context
          -> a (k,v) -- ^ Base RDD
          -> Static (OrdDict k)
             -- ^ Key must be orderable
          -> Static (SerializableDict [(k,v)] )
          -> Closure (v -> v -> v)
             -- ^ Combining values for a key
          -> Closure (k -> Int)
             -- ^ Choosing a partition for a key
          -> ReduceRDD a k v (k,v)
```

Reducing a data with a combining function is done in two stages [3] :

- **Stage 1: Local Reduction** The data is locally reduced using combining function. Local reduction results in a reducing serialization overhead over the network.
- **Stage 2: Shuffled Reduction** Each process is mapped to a partition number. The partition number is sent to the processes producing *Stage 1*. Each *Stage 1* process responds by delivering only those keys which belong to a given partition.

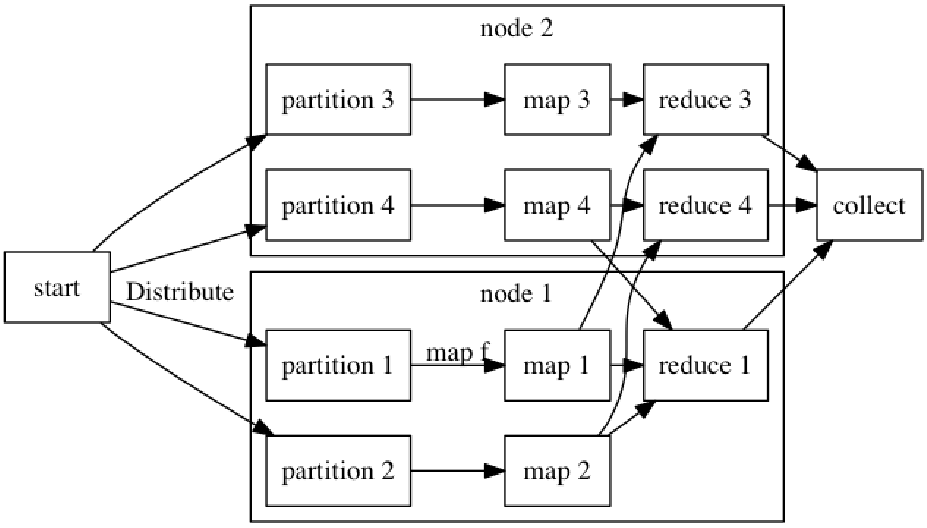
  Stage 2 further does the reduction using combining function.

2.3 Execution Strategy

Hspark implements following strategy to allocate partitions to node, and do further processing.

- **Partitioning Data** - Each partition of data is assigned to a node in the cluster. If number of partitions are larger than the number of worker nodes, the nodes are wrapped over.
- **Mapping Jobs Allocation** - The mapping jobs is done on the same node where its parent block is present.
- **Reduction Job** - The number of partitions in the reduction are kept same as the parent RDD.
- **Storage** - The processes are also responsible for the storing the results of the computation.

The execution plans for a simple seed-map-reduce job looks like following.



3 Limitations and Future Scope

- Does not handle exceptions well. Hence, **hspark** is yet to achieve the *resiliency*.
- It should be possible to implement a execution strategy driven by context, where a failed process can be restarted in case of a network failure.

- When the mapping processes share the same node, the data is still serialized (not reused). It may be possible to model it through share *MVar* in such a way that the proceses working on the same node can resolve directly to the data.
- Processes are spawned on demand without any monitoring. Monitors should be added to detect failures, and propagate.
- The closures are used to spawn processes. And hence, the task allocation has to be done by RDD itself. Instead, it is proposed that RDD should evaluate to a DAG of closures (rather than a blocks of processes).

  Each graph node in the closure DAG would represent a process that can be spawned on any of the node in the cluster. This will put *Context* in the total control, and also will give an ability to restore a node by looking at a lineage of any graph node and re-processing the closure.

These points should be considered only when the library has stabilized.

- Benchmarking on the known data and against *Apache Spark*.
- Using different backends for *distributed-process*

4 Sample Code

Sample **hspark** code is provided here.

```
do
  sc <- createContextFrom remoteTable master slaves
  -- Create RDD with 2 partitions
  let partitions = Just 2
      dt = [1..100]
      -- Seed the data with
      seed = seedRDD sc partitions dict ($(mkClosure 'input) dt)
      -- Map the data
      maps = mapRDD sc seed dict square
      -- Reduce with a combiner
      reduce = reduceRDD sc maps odict dict combiner partitioner

  -- Compute, will trigger seed, maps, reduce
  result <- collect sc reduce
```

5 Source Repository

The repository is maintained at git-hub at `https://github.com/yogeshsajanikar/hspark`. Any suggestions and contributions are always welcome.

References

[1] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. 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*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[2] Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. Towards haskell in the cloud. *SIGPLAN Not.*, 46(12):118–129, September 2011.

[3] Ralf Lämmel. Google's MapReduce Programming Model – Revisited. *Science of Computer Programming*, 2008.