Spark.jl: Resilient Distributed Datasets in Julia

Dennis Wilson, Martín Martínez Rivera, Nicole Power, Tim Mickel
`[dennisw, martinmr, npower, tmickel]@mit.edu`

INTRODUCTION

Julia[1] is a new "high-level, high-performance dynamic programming language for technical computing" being developed and used at MIT. The language is competitive in performance to MATLAB, NumPy, Fortran, and other common technical languages, and offers language features far beyond some of its competitors. One such feature is native support of parallelism through remote references and remote calls. In general, a Julia user can use the `@spawn` macro or `remotecall` function to operate an expression on remote workers, and they can retrieve the results of this call with the `fetch` function. Other macros, `@parallel`, `@sync`, and `@async`, allow for easier and more complex use of parallelism.

Julia also offers distributed arrays, DArrays, that can be constructed over multiple workers and accessed in parts corresponding to the allocated worker. This functionality is useful for storing large amounts of data, but is limited in scope. Operations on this distributed array type are cumbersome and expect the user to coordinate access to the different sections of the array. In addition, they lack simple fault tolerance, as do the remote reference and remote calls.

Many of the tasks in scientific computing require the processing of large amounts of data and offer good parallel performance. However, writing such applications in Julia results in repetitive code that leaves the programmer to deal with many network and parallelization problems that have nothing to do with the problem at hand and often cause the introduction of bugs that are difficult to debug. We believe Resilient Distributed Datasets, as presented in Spark

[1] http://julialang.org/

[2], address the two main problems with DArrays: functionality and fault tolerance. Spark's main features are functionality over RDDs, fault tolerance through an operation log and checkpoints, high performance by utilizing memory over disk space, interactivity through an interpreter, and user control of persistence and partitioning. Porting Spark over to Julia has the potential to speed up the development of many of the applications targeted by this new and exciting language.

FOCUS

Spark is a system based upon the use of Resilient Distributed Datasets (RDDs) as a distributed memory abstraction. An RDD is a read-only, partitioned collection of records which represents data formed from deterministic operations. RDDs allow for efficient reuse of data in applications such as iterative algorithms which reuse intermediate results in various computations. Fault tolerance is achieved in this system by storing a dataset's lineage: the coarse-grained transformations that created this dataset from data in stable storage. Thus, a lost partition of an RDD can always be recomputed based off of its lineage.

Providing a complete port of the original implementation of Spark is a task that would take much more time than the allocated for this project. The Scala code contains thousands of lines and uses tools and programming capabilities not available in Julia, making the creating of an exact port a task that would need many hacks and unidiomatic code to be completed. Instead, Spark.jl offers a basic subset of the features of the original Spark and focuses on those more frequently used by scientific computations. Spark.jl offers the basic actions and transformations mentioned in the Spark paper. In order to simplify the inner workings of the implementation, Spark.jl only offers support for data that can be represented as key-value pairs, which allows the reuse of the same code for all the operations.

[2] http://css.csail.mit.edu/6.824/2014/papers/zaharia-spark.pdf

Spark.jl offers interactivity through the native Julia REPL. From there, users can import data, operate on rdds, and start the recovery process when a failure is detected. Support for recovery is provided and can support the recovery of single partitions without having to recompute whole RDDs. At this time, Spark.jl does not offer checkpoint support but it would not be hard to add in the near future.

RPCs

The following remote procedure calls facilitate the main functionality of Spark.jl, and are defined mostly as a caller and handler method with the same name but different parameters. An example is IDENTIFY, which the user calls as Spark.identify(master), and which each worker handles as Spark.identify(worker, args). The second argument, an arguments dictionary, allows for the flexible use of JSON as the message passing interface.

The three calls FIND_ACTIVE, RECOVER_PART, and UPDATE_RDD are used in recover. RECOVER_PART applies operations for a single partition, FIND_ACTIVE finds a random active worker to put the partition on, and UPDATE_RDD updates the partitioning plan for the recovered RDD on each worker.

DO_OP calls operations, either transformations or actions, on the worker. It creates the resultant RDD, based on a partitioning plan passed as an argument, and send that RDD with the operation name and arguments to each worker. The workers handle a DO_OP call by finding or creating the relevant local RDD partition and applying the operation specified from the dependency RDD partitions to the new one.

GET_RDD is a Worker → Master call that fetches the RDD meta information for a given RDD ID. This is used if a worker is required to operate on an existent RDD that it doesn't know about, beyond the ID.

SEND_KEY starts an RPC call to another worker with a single (key, value) pair and a RDD and partition reference. The recipient worker handles this with RECV_KEY, which integrates the (key, value) pair into their local RDD and partition. Note that this is the only case where the actual (key, value) pairs are communicated over the network, as opposed to the small information stored in the RDD objects.

OPERATIONS

Operations are split into two categories: transformations, which modify RDDs, and actions, which don't. The transformations we implemented are MAP, FILTER, GROUP_BY_KEY, JOIN, and PARTITION_BY. We did not implement UNION, COGROUP, CROSSPRODUCT, FLAT_MAP as they apply to RDDs that are not based on (key, value) pairs. SAMPLE, SORT, REDUCE_BY_KEY, and MAP_VALUES are redundant or simple to recreate using the implemented functions, so we left these as possible future work.

We implemented the COUNT, COLLECT, and LOOKUP actions. COUNT returns the length of the RDD, COLLECT returns a copy of the RDD to the master, and LOOKUP takes a key as an argument and returns the list of values mapped to that key in the RDD. We did not implement REDUCE, or SAVE. REDUCE can be performed locally from a collect, and a local SAVE to disk, enabling snapshots, is a feature for future work, but is not necessary in the core operations of Spark.

RECOVERY

Recovery occurs on a partition base which saves computational power and space compared to recovering full RDDs. If an operation fails due to a network or disk error, the operation returns false. The user can then call Spark.recover on the last known RDD, usually the

RDD they had intended to perform the failed operation on. The recovery function checks the worker.active flag in each partition for that RDD and begins a recursive recovery method for each lost partition.

This call checks the RDDs which the lost partitions depended on and determines whether or not they are complete, ie there are no inactive worker in their partitions list. In this way, network faults where a worker appears inactive for a short time will be accounted for in the recover method, as well as changing network topologies.

Once the dependencies for a partition are secured, the operation chain which will result in the desired partition is begun. To reduce redundancy, the operations are only computed on the relevant partition. For narrow operations, this is done simply by calling the operation only on the relevant worker. For wide operations, dependent on partition_by, the resultant partition id is passed as an argument, and only keys allocated to that partition are sent. In addition, for all operations, the RDD partition lists on the workers are updated to exclude the inactive worker.

TESTING

We emulated workers with threads independently running Julia, listening on ports specified in a configuration file. As this file lists the worker IP and port, it is applicable to a networked configuration as well as a local one. The input files for the RDD are plaintext, but a user could easily specify an HDFS file for reading using the HDFS.jl[3] package. A port was also opened on the local machine as the master RPC handler. The start of a set of known workers and the creation of a master object, which opens the master listening port, is all the setup required to start performing Spark operations, and therefore to start testing.

[3] https://github.com/tanmaykm/HDFS.jl

We first tested all operations, asserting that the three workers were able to input and manipulate a known input file simply containing the numbers 1-10. For map and filter, we defined deterministic map and filter functions and collected the results of these operations. To test recovery, we set a single worker's active flag to false, which would be the result of a network or disk error on the worker.

FUTURE WORK

The current state of our project presents a basic working version of Spark with native and interactive Julia support. However, the time constraints prevented the project from supporting some of the more complex and rarely used functionality of the original Spark. For example, a range partitioner and the transformations dependant on it (i.e. sort) could be added in the future to support ordered datasets. Even if more time could be allocated for this project, Julia is an experimental language and it may be advisable to wait for a stable version of the language to build a production version of Spark because the evolving nature of Julia could affect negatively the development of Spark.

For now, our version of Spark offers the user better support for data-heavy parallel computations than the native @spawn macro because it abstracts away from the user all the complexities and quirks of the language from file input to the distribution of work among the workers and recovery of lost data.

The possibility of turning this class project into a more mature and production ready library certainly exists because Julia was designed with scientific computations in mind and merging the nice features it provides with the fault-tolerance, speed, and ease of Spark could streamline many common operations of scientific computing. Spark.jl is open source under the MIT license and can be found at https://github.com/d9w/Spark.jl.