SparkDriverless and MicroSpark

Wanzhang Sheng
Hao Chen
Minglu Ma
University of San Francisco
May 19, 2015

Abstract

SparkDriverless is a totally decentralized system inspired by Spark, but with a totally different approach for distributed computation. It removes the Driver node from Spark to achieve single-point-failure-free. Every worker in the cluster is exactly the same. Even the client has similar mechanism with workers. Nodes communicate with each other by broadcasting messages. This allows any node, even the client, fails at any time, without break the computation process. And also it has a much simpler model compared to original Spark.

MicroSpark is a subset of the original Spark. It allows users to use the original Spark's API to compute a certain job in a cluster. It is based on Resilient Distributed Datasets model and achieves fault-tolerant.

Keywords: Spark, distributed programming, decentralization

1 Introduction

Spark Driver-less is a new model of distributed computation system based on Resilient Distributed Datasets (RDD)[5] model. There is no driver in the cluster, instead, only workers and clients. They communicate with each other by broadcast. In programming language, actor model describes a cooperation model that each object runs on its own thread, and communicate by sending message. Spark Driver-less uses similar idea that let each worker or client runs on its own node, and communicate by broadcast messages.

Hao Chen and Minglu Ma finished MicroSpark, which is a subset of the original Spark. It will be discussed in Section 7. Wanzhang Sheng finished SparkDriverless, which is the new approach of Spark model. Some basic terms is listed in Section 3. Detailed implementation and work flow is described in Section 4. Some analyses is discussed in Section 5. And Section 6 discusses some further works.

2 Background

The original Spark model is a centralized system. Client send job to the driver, and driver will split into tasks and send to multiple workers. When worker finishes the task, it will send it back to driver. If any worker is failed, the driver will notice it and react to the failure, such at recompute on other workers. Driver is efficient on computation, but also introduces a disadvantage to the cluster, single point failure. It's hard and slow to recover the driver node, and the system becomes unavailable during this time.

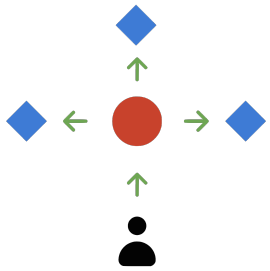

Figure 1. Original Spark Model

The basic idea of Spark Driverless is by removing the Driver entirely and communicate by sending message, we can achieve single-point-failure-free.

3 Terms

Broadcaster
: Broadcaster broadcast multiple services inside the local network. By listening to a certain port, it could respond to Discoverer's searching query, and return the services list.

Discoverer
: Discoverer will keep sending query to certain port to

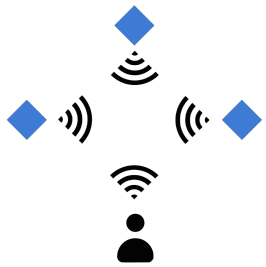

Figure 2. Spark Driverless Model

ask for a certain type of services, and maintain a local list of services.

Service

Service describes a service with properties. It is defined by *type*, *name*, service source *IP*, and *port*. All services use a universal unique ID (UUID) as its name.

RDD

RDD is defined as the same with the original paper. However, in Spark Driver-less, RDD is only used to record operation lineage to generate Partition lineage. It only exists in Client and never be sent to any Worker. Every RDD has a *parent* RDD except data source RDDs, and a *get* function for Partition computation.

Partition

Partition is a slice of the RDD, which will be sent to a single worker to compute. It has a *part_id* for the index in the RDD, a *uuid* generated from the hash of its bounded function, a *parent_list* of dependent Partitions list, and a *func* for computation.

It is also a Service represents a computed partition, with a type of *partition* and its uuid as the name. The computed data is called Result.

Job

Job is a Service related to a partition to be computed. It has a type of *job*, and use the partition's uuid as its name.

Worker

Worker is a computation process on an arbitrary node. It has a Job Discoverer, Job Broadcaster, Partition Discoverer, and Partition Broadcaster.

It is also a Service with a random uuid as its name. But it is only used by Client to optimize *num_of_partitions*.

Client

Client is a interactive console for user to compute in the cluster. All computation is lazy. It creates RDD lineage and transform it into Partition lineage. Broadcast target partitions as a Job. It keeps discovering Workers in the cluster, and uses the number of workers as *num_of_partitions* when it is creating Partition lineage.

Worker and Client's broadcasters and discoveres are shown in Figure 3.

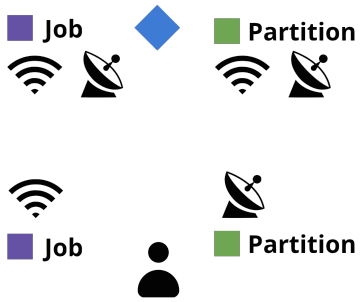

Figure 3. Broadcasters and Discoverers

4 Implementation

4.1 Client

Client accepts user's input. It's a lazy interactive console. User would define their RDD lineage. But until any action is called, there is no real computation at all. It will record down the parent RDD and the function or other parameters related to the RDDs. If it's a wide dependency, it will first create a Repartition RDD between itself and its parent.

When user called a action, Client will firstly create the Partition lineage based on the given RDD lineage as Figure 4. Any Partition will only have the dependency links to the partitions it needs in the parent RDD. If it's a narrow dependency, it will only have one dependency partition. And if it's a wide dependency, it will depend on multiple ones. By limiting the reference inside Partition object, it becomes easy to dump the partition object by CloudPickle[1] without introduce any unnecessary object, which will reduce data transferring and dump failure.

Secondly, for every partition in the final target RDD, the client will create a job for it. And broadcast them by partitions' uuid. Only these partitions will be created as a job, because these partitions are the ones that the user really care about, not those intermediate results.

Finally, it will keep discovering for the finished Partitions of these jobs. And fetch them back to local when find them available in the cluster. Merge the results of those partitions and return final result to the user.

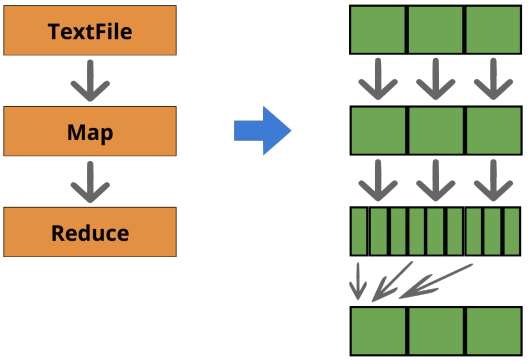

Figure 4. RDD Lineage to Partition Lineage

4.2 Worker

When the Job Discoverer of a Worker finds a job broadcasted in the cluster, it will connect back to the source and retrieve the corresponding Partition object back, and add it to the Worker's Jobs queue.

Worker has a thread that keep trying to take the Partition object out of the Jobs queue. After successfully taken one out, it will call the Partition object's function. For the narrow dependency transform, like Map and Filter, it will directly call the parent partition's function. For the wide dependency transform, like Repartition, more works will be done.

First, it will check all the dependency partitions in its own Partition Discoverer, and save the results to the Partition objects. If that partition is local, the discoverer will simply return the partition's result. Otherwise, it will connect to the remote worker, fetch the result back, and cache it locally for the future needs.

Second, the worker will find all the missing parent partitions that can't be found in the cluster. If any parent partition is missing, it will wrap them as a Job and broadcast it. Then suspend the current job by appending it to the end of the jobs queue. If all parent partitions are done, the worker will compute the current job, since all the dependencies are local now. And put it into Partition Broadcaster to announce that the partition is finished in the cluster.

4.3 Broadcaster

Broadcast system is based on UDP broadcast. One side of broadcasters and discoverers will keep sending packages starts with a magic prefix string, to a predefined multi-cast address. The other side will keep receiving packages from that address and filter packages with the magic prefix string.

A python wrapper called MinusConf[4] is used with gevent[2] monkey-patching.

5 Analyses

5.1 Accomplishment

Since the RDD lineage is a directed acyclic graph, the partition lineage should also be a directed acyclic graph, even with wide dependencies. This means, for any given partition in the partition lineage, there is always a topological sort for all its dependencies to be executed sequentially. This guarantees that any given partition could be done eventually.

5.2 UUID

The key point of the whole system is the UUID of partitions. It is generated by first dumping the partition's function into string by CloudPickle and then hash the string into an id. The id is considered as a universal unique, but it's not guaranteed. However, it's very rare to have a conflict. Consider the bits of the hash is l, then the total hash space has $m = 2^l$ numbers. And random pick n ids from it. Then it's a pigeonhole problem[3] with m and n. The likelihood of having a conflict is:

$$P(x) = 1 - \frac{(m)_n}{m^n} = 1 - \prod_{k=1}^{n} \frac{m-k+1}{m}$$

If given $l = 64$ and $n = 10^5$, which means 64 bits hash function and 10^5 partitions at the same time. Then the likelihood of having a conflict is:

$$\begin{aligned}
1 - P(x) &= \prod_{k=1}^{n} \frac{m-k+1}{m} \\
\ln(1-P(x)) &= \ln(\prod_{k=1}^{n} \frac{m-k+1}{m}) \\
&= \sum_{k=1}^{n} (\ln(m-k+1) - \ln(m)) \\
&= \sum_{k=1}^{10^5} (\ln(2^{64}-k+1) - \ln(2^{64})) \\
&= -1.39698 \times 10^{-9} \\
P(x) &= 1 - e^{-1.39698 \times 10^{-9}} \\
&= 2.71048 \times 10^{-10}
\end{aligned}$$

The code to calculate is in Appendix A.1.

Moreover, the bits of the hash could be arbitrary large to reach the desire level.

5.3 Cache

Anywhere that accesses the result of each partition, will cache theses results. This could be done because of the na-

ture of RDD. RDD is a immutable and stateless data structure. Which means that the result of a partition is independent of when and where is computed, even with repeatedly computations. They are cached in:

1. RDD's *collect* method.
2. Client's Partition Discoverer.
3. Worker's Partition Discoverer.
4. Worker's Partition Broadcaster.

5.4 Data Expiration

Because the UUID of the partition only relies on the function closure not the data source, it will always the same even the data is changed. Spark Driverless shares the same assumption that the data won't be changed during computation. However, there are several ways to expire the partitions:

1. Restart all the workers. This is the brute force way to do it. It's very cheap to restart workers, since they are stateless and no need to configure. This is the easiest way to guarantee expiring those partitions.
2. Add a time-stamp to the function closure. So that the UUID will be changed. This also guarantees expiring partitions.
3. Broadcast to tell workers to expire a certain partition. However, this can't guarantee to expire without any further mechanism.

5.5 Lazy

There is not any computation until any worker or client requires it. The work flow and data flow is not planned ahead but determined just in time.

5.6 Decentralization

Spark Driverless is totally decentralized. Decentralization means no single point failure. And all nodes dynamically find each others in runtime. So that the cluster becomes easy to maintain, since no configuration is required at all.

5.7 Fault Tolerant

There is no code directly related to fault tolerant. Fault tolerant comes from the nature of the combination of stateless dataset, lazy evaluation, and decentralization. There is no difference between a normal evaluation and a fault recovery.

6 Future Work

Python has no real parallel thread system. Currently Spark Driverless uses gevent for concurrency. But it get stuck in the gevent thread of taking job out of the queue, rather than jobs discovering. It is not sure whether it is due to gevent or minusconf. Further research is required to achieve better performance.

7 MicroSpark

7.1 Components

Driver Driver will visit the lineage of the RDD and split the lineage into different stages. Each stage will be convert to a partition operation object and add in to the stage list. During the execution phrase driver will send the stage object to the workers.

Worker Worker will management the real data partition, do the transformation and cache the intermedia data. When the worker receive the stage object it will do the transformation to the relevant RDD partition and generate new RDD partitions.

MacroSparkShell An interactive shell which can submit and run the code directly on the cluster.

Client Client is the user defined application.

7.2 Cluster Setup

The driver will ssh to different machines and launch different worker processes on it. We can also add new workers through the MacroSparkShell during the execution time. Each time the new workers was added the driver will reset the worker list.

7.3 Failover

The design of the failover if pretty simple. When a worker failure happened. Drive will reset the cluster and recalculate the RDD using the RDD lineage.

7.4 Stage Split

We will split the lineage based on the repartition as showed in Figure 5. After the stage split each stage will be a narrow dependency stage except the repartition stage. So we can pipeline the execution for each stage.

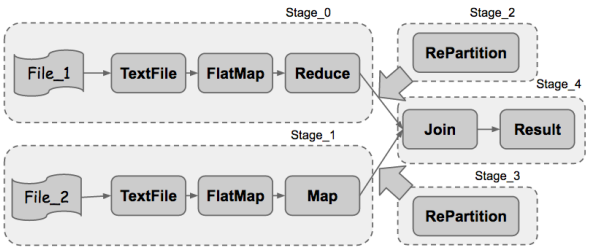

Figure 5. Stage Split

7.5 Execution

The real execution process is to transform from one RDD to another and then execute the next transformation. This is fine when we can put all the data into the memory, but when we have a data set that too large to fit into the memory we may meet some problems. We pipeline the execution in each stage so we can put part of our data in memory and do transformation iterations as much as possible.

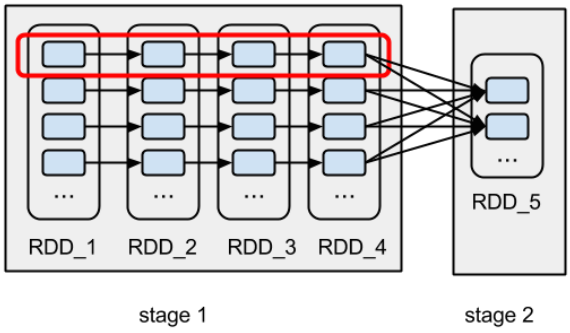

Figure 6. Execution Flow

As we can see in Figure 6, we will do all the transformations in the red block for the data in the memory at once, so we dont need to reload the data when it need to do the transformation from RDD_2 to RDD_3 or from RDD_3 to RDD_4.

A Appendix

A.1 UUID Analysis Python Code

```
m = 2^64
n = 10^5
a = Sum[Log[m-k+1], {k,1,n}]-n*Log[m]
N[a]
N[1-Exp[a]]
```

References

[1] Cloudpipe/cloudpickle. https://github.com/cloudpipe/cloudpickle, 2015. [Online; accessed 18 May 2015].

[2] gevent. http://www.gevent.org/, 2015. [Online; accessed 18 May 2015].

[3] Pigeonhole principle. http://en.wikipedia.org/wiki/Pigeonhole_principle#Generalizations_of_the_pigeonhole_principle, 2015. [Online; accessed 18 May 2015].

[4] Ranmocy/minusconf. https://github.com/ranmocy/minusconf, 2015. [Online; accessed 18 May 2015].

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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*, pages 2–2. USENIX Association, 2012.