MicroSpark: A Spark Subset and Extension Research

Pan Wang, Guoliang Ye, Jie Gao

University of San Francisco

Abstract

MicroSpark is a subset of Spark, which supports following applications: Word Count, Pagerank and Interactive queries on log files. Our MicroSpark implementation include Resilient Distributed DataSets(RDDs), dynamic creation of workers, driver and worker support, interactive usage, closure and code shipping support and fault tolerance of workers. Besides, we evaluate the performance of Word Count application for various files and analyze it.

I. Introduction

Scalable, parallel processing framework such as MapReduce has been widely adopted for large-scale data analytics. We have implemented MicroSpark based on Spark to retrieve the same benefits of performance. Unlike MapReduce, Spark is about performing sophisticated analytics at lightning fast speed. According to stats on Apache.org, Spark can run program up to 100 times faster than Hadoop MapReduce in memory, or 10 times faster on disk[1] MicroSpark is designed to support in-memory processing, and keeping everything in memory supports iterative computations at fast speed while MapReduce does not support.

In this paper, we implement resilient distributed datasets(RDDs) which enables data reuse in a broad range of applications. RDDs are parallel data structures and fault-tolerant which let users persist intermediate results in memory.

In contrast to these systems, RDDs provide an interface based on coarse-grained transformations(textfile, map, flatmap, reduceByKey, groupByKey, filter and join) that apply the same operation to many data items. This provides fault tolerance by building a RDD lineage rather than the actual data. If a partition of an RDD is lost, the RDD lineage has enough information to recompute just that partition without costly replication. We implement three applications: Word Count, Page Rank and Query on Log Files based on RDDs.

This paper begins with an overview design of MicroSpark. We then discuss the implementation of MicroSpark which consists of Driver, Worker and RDDs. We then discuss the extension research to optimize file split and return result of each partition of one worker to driver. Finally, we discuss the performance by experiment of small and large data sets.

II. Data Structure

We have kept two main data structures for bookkeeping in driver class: rddLineageList(list) and workers(list).

- Each worker is a dictionary, which saves the information about tasks(partitions) and information of each worker(ip, port, status).
- Each partition is a dictionary which presents the information about each partition(unique name, startoffset, endoffset, status)
- RddLineageList contains lineage of RDDs.

We keep tracking these important data for job scheduling, closure and code shipping, and fault tolerance.

III. Design & Implements

I. Driver

I.1 Job execution

Job execution flows show as figure 1. In our spark, driver and master are set at one node. Driver generates RDD lineage based on client input. When client call those functions that will call Action RDD, driver will do the following steps. Step 1, generate partition information. Step 2, assign partitions to alive workers. Step 3, sent RDD lineage to alive workers. Step 4, call workers start working. Worker returns results to driver, driver do actions (saveAsTextFile, collect, or count).

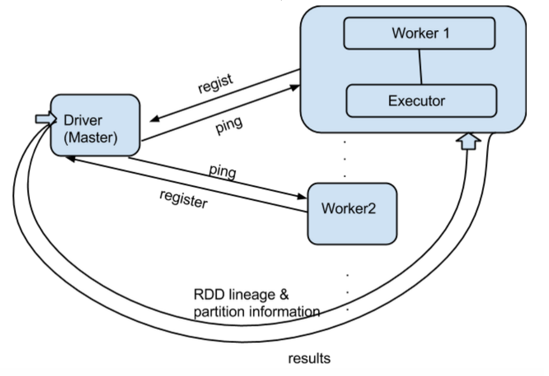


Figure 1. spark execute flows

I.2 Fault Tolerance

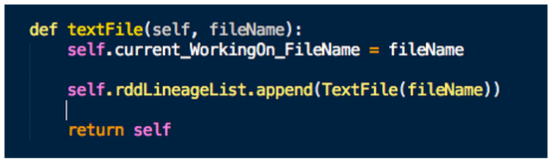
If there is fault happened on worker node. The cluster will start fault tolerance mechanism. First, in our spark implement, driver keeps track of workers, driver ping with workers every 'timeout' seconds, if there is zerorpc exceptions, driver will think the exception worker already done. After driver detect one worker done, it will remove worker from worker list at once.

In order to handle the fault tolerance. driver will do to following steps after it detect a done worker. Step 1, remove worker who is done from the self workers list. Step 2, check the removed worker information. In this step, if the removed worker assigned tasks once, driver will copy out the die worker's assigned tasks, and then reassign these tasks to other alive workers. If the removed worker didn't assigned any tasks before, driver just delete it from worker list.

Driver has a mechanism to check whether workers finished their tasks. The mechanism is set as following: In the driver, there is a partition list, which created in the beginning of one task. There is another list called finished partition list. While diver detect that the length of these two list equals, which means that driver receive all results of each partitions, driver will know that all workers finished their works and sent the results to driver already. At this moment, driver will call workers to stop waiting and initialize themselves and prepared for next task.

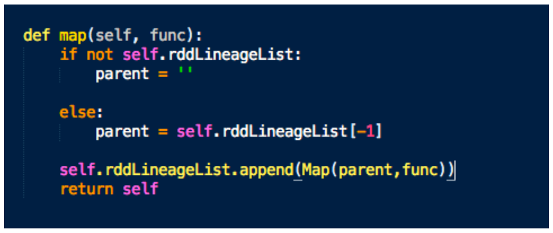
I.3 Functions in Driver

Functions in driver can be divided into two categories. First categories including textFile(), flatMap(), map(), reduceByKey(), filter(), groupByKey, etc. This kind of categories will create regarding rdd and extends the rdd into rdd lineage in current task. Let's make testFile() as example, which show in figure 2. We can see the textFile function in driver appedn RDD TextFile() in rdd lineage. We should notice that RDD TextFile() do not have parent RDD, for those RDD who has parent will be a little different, we can see RDD Map() in figure 3. For those RDD who has parent, they will treat the RDD which infront of it in the rdd lineage as its parent. It's very clear that one RDD will put into rdd lineage as a closure, it includes the input parameters such as filename in textFile(), and the parent information.



```
def textFile(self, fileName):
    self.current_WorkingOn_FileName = fileName

    self.rddLineageList.append(TextFile(fileName))

    return self
```

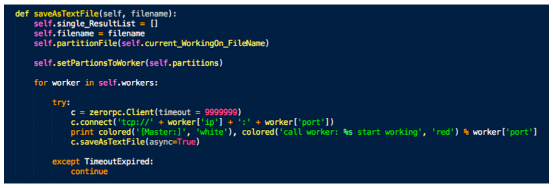
figure 2. textFile() function in driver



```
def map(self, func):
    if not self.rddLineageList:
        parent = ''

    else:
        parent = self.rddLineageList[-1]

    self.rddLineageList.append(Map(parent,func))
    return self
```

figure 3. map() function in driver

The second categories including saveAsTextFile(), collect(), count(), etc. These functions will call workers to start work. We take saveAsTextFile() as example, the saveAsTextFile function showed as figure 4. In this function, it will sent partition informations to regarding workers first, and then sent RDDs lineage of current task to workers, at the end, it will call worker start working. We don't need to consider about what happened if the worker done while driver sent informations to them, because driver has controller function to track workers situation. Controller ensures that all workers in worker list are alive.



```
def saveAsTextFile(self, filename):
    self.single_ResultList = []
    self.filename = filename
    self.partitionFile(self.current_WorkingOn_FileName)

    self.setPartionsToWorker(self.partitions)

    for worker in self.workers:

        try:
            c = zerorpc.Client(timeout = 9999999)
            c.connect('tcp://' + worker['ip'] + ':' + worker['port'])
            print colored('[Master:]', 'white'), colored('call worker: %s start working', 'red') % worker['port']
            c.saveAsTextFile(async=True)

        except TimeoutExpired:
            continue
```

figure 4. saveAsTextFile() function in driver

II. Worker

II.1 Job execution

In our spark, worker receives RDDs lineage and partition information first. After that, workers will wait until driver call them to start working. Then worker will follow the RDDs lineage executes for each partition.

II.2 Fault Tolerance

In the implement, worker has a partition list to store partition informations. Each partition contains partition name, start offset, end offset. Workers delete the partition information after it already calculate results on this partition. One of the interesting thing is that worker will wait at this point in case of there are any other new partitions assigned to them which because of there are some other nodes failed.

In fault tolerance, worker will receive new assigned partitions. The new received assigned partitions will calculated following the RDDs lineage.

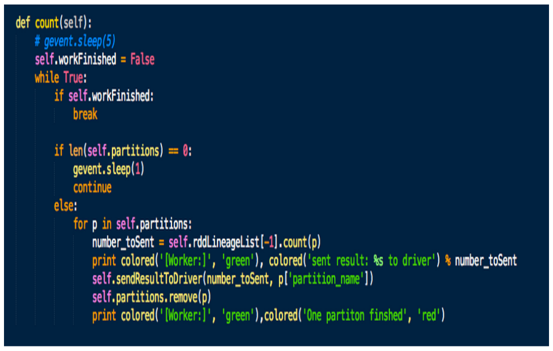
Worker keeps waiting until received stop notice from driver. If worker receive stop command from driver, it will stop waiting and initialized itself what is in order to do the next task.

II.3 Functions in Worker

Besides the basic functions such as ping() and controller() which used to meet normal worker working needs. Functions in workers are used to execute real calculation. We use count() function as example (figure 5).

The worker working mechanism is that after worker receive RDDs lineage and partitions list. It will wait driver call them start working. After driver call worker's functions such as count() function by using zeroRPC. Worker will calculate each partitions following the same RDDs lineage and return the result to driver by sendResultToDriver() function in worker. Worker pop the partition which it already finished and wait in the for loop until driver calls its stop() function. Worker will Initialize itself after driver called its stop().

What should be mention is in our spark implement, worker sent result of one partition as soon as it finished calculation of that partition. Why worker did not send result to driver wait until it finish all partitions assigned to it is because we set a optimizer mechanism in out spark, this optimizer will help to short times while there is fault tolerant needed. We will talk this in the following parts.



```
def count(self):
    # gevent.sleep(5)
    self.workFinished = False
    while True:
        if self.workFinished:
            break

        if len(self.partitions) == 0:
            gevent.sleep(1)
            continue
        else:
            for p in self.partitions:
                number_toSent = self.rddLineageList[-1].count(p)
                print colored('[Worker:]', 'green'), colored('sent result: %s to driver') % number_toSent
                self.sendResultToDriver(number_toSent, p['partition_name'])
                self.partitions.remove(p)
                print colored('[Worker:]', 'green'),colored('One partiton finshed', 'red')
```

figure 5. count() function in worker

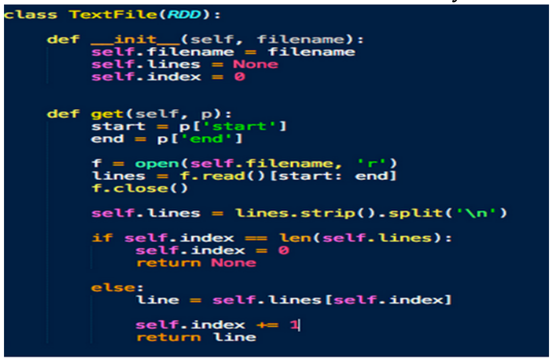
III. RDD

III.1 RDD generation

RDD created in two ways:

- Created from the file system (or other storage systems) inputs.
- Generate child RDD form parents RDD.

Let's look at a way to generate RDD from Hadoop File System, such as: val file = spark.textFile('myfile'), where 'myfile' is the name of input file. When calculating on the testFile RDD, the process of read data is as figure1. Because in out spark implement, the whole files are stored in each nodes. Thus workers just need to read data in their assigned partitions, what showed in figure 6 , p is the parameters refers to partition informations, which decided by driver.

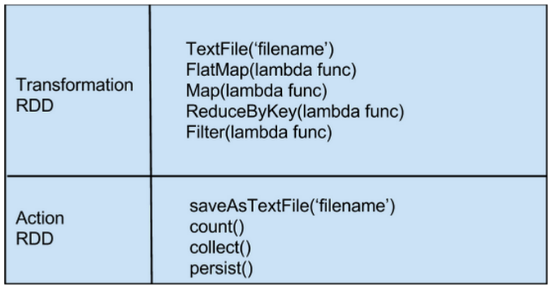


```
class TextFile(RDD):

    def __init__(self, filename):
        self.filename = filename
        self.lines = None
        self.index = 0


    def get(self, p):
        start = p['start']
        end = p['end']

        f = open(self.filename, 'r')
        lines = f.read()[start: end]
        f.close()

        self.lines = lines.strip().split('\n')

        if self.index == len(self.lines):
            self.index = 0
            return None

        else:
            line = self.lines[self.index]

            self.index += 1
            return line
```

figure 6. reading data in textFile rdd

III.2 Action & Transformation

In our design, RDD can calculated in two ways, Transformation and Action. All implemented RDD in our spark shows in figure 7. For transformation RDDs, (such as TextFile(), FlatMap(), Map(), ReduceByKey(), Filter(), etc.). Transformation RDDs are lazy, in another word, one RDD transforms to another RDD will not execute immediately, Spark just records these kind of RDDs until there is the Action RDD. Operations need to wait Action rdd then it truly start the calculation, which following the RDD lineage, actions will return results or the RDD datas to the storage system.



Transformation RDD	TextFile('filename')
	FlatMap(lambda func)
	Map(lambda func)
	ReduceByKey(lambda func)
	Filter(lambda func)

Action RDD	saveAsTextFile('filename')
	count()
	collect()
	persist()

figure 7. Implemented RDDs

IV. Extension

I. Split input file

In our implementation, the MicroSpark splits the file based on the default chunk size. If the input file is a large file, the total number of partitions will much larger than the number of workers. Evenly partitions will be assigned to each worker. If the input file is a small file, it is unnecessary to use all workers sometime.

In our spark implement, we set the default split size 2000 bytes, which is about 40-60 lines per split. We have test many split size and find that this size partition can be finished in a efficient way. For the partition assign mechanism, we use the pop function in list. Firstly, we get the number of partitions that needed to assign to one worker. Secondly, pop the number of partitions to worker from the first one in worker list. This mechanism ensure that besides the last assigned worker, the former ones can have a balance number of partitions, which also help to make spark more efficient.

II. Immediate return

For the improvement of fault tolerance, each worker will return the result of current partition to the driver immediately when it done. With this method, if the worker fail in the next partition, the driver only need to reassign partitions, which haven't been done by the failed worker, to other alive workers for further computations. So the driver doesn't need to reassign all partitions of failed worker.

V. Performance

We evaluate MicroSpark and RDDs with the experiment on WordCount application. We illustrate the result of the experimentation in table 1.

Table 1

File Size	1 Worker	3 Workers	Sequential
70K	0.1s	0.16s	0.015s
1.4M	4.33s	2.59s	1.66s
2.8M	8.27s	3.89s	2.45s
14M	14.51s	7.35s	16.48s

Based on the result of the experiments, we can find out that our MicroSpark works worse on the cluster than the sequential execution on local machine for small files, but it works better for large files. There are various reasons can cause this phenomenon, such as remote protocol calls between servers, code shipping and data transitions in the system may result in the delays in the system, which has an important impact on the performance of the application.

VI. Conclusion

We have implemented MicroSpark, a Spark like framework to reuse working set of data across multiple parallel operations. We have implemented resilient distributed datasets(RDDs), a fault-tolerant abstraction for sharing data across applications in cluster based on architectures presented there[2]. We have implemented three applications: Interactive queries on log files, Pagerank and Word Counter.

For optimization design, we let each worker handle several partitions and return the result of each partition to master when it is done. What we can optimize in the future is try to find out the best number of partitions returned to master each time to reduce network overload and efficiently achieve fault tolerance at the same time.

We had challenges when we implemented RDD lineage and we use get() function to iteratively get data from parent RDD. We had another challenge trying to implement wide dependency in MicroSpark, and we ended implementing wide dependency in Queries on log files: each worker will filter logs starts with "error" and return to driver, and driver will collect all logs starts with "error".

References

[1] Cluster Computing with Working Sets] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, University of California, Berkeley

[2] Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing]. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica, University of California, Berkeley