MicroSpark

Zhan Peng, Bin Lu, Yuefeng Wu

May 2015

1 Introduction

Cluster computing frameworks, like MapReduce, are more and more popular nowadays. User can easily write parallel computations which uses a set of high-level operators, without knowing or worrying any distribution and fault tolerance. However, most of them are inefficient on the operation of reusing intermediate results. Spark suggests an in-memory Resilient Distributed Datasets(RDD) to achieve a great performance. We are going to build a similar and simple MicroSpark, which provides basic RDD transformations/actions and supports closure/code shipping and dynamic worker creation.

2 RDD Implementation

RDD is created by driver then sent to master. The master adds worker information to RDD, and sends it to workers. When workers finish calculating, driver collects data from workers.

2.1 Lineage Graph

Each transformation define a function, then return a new RDD which initialized by current RDD and defined function. In the constructor function of RDD, a pipeline function will be defined to combine the previous transformations and current transformation together.

```
def pipeline\_func(iterator):
    return func(prev.func(iterator))
```

Narrow dependency: Transformations with narrow dependency do calculation by using local data and preserve the previous partition. The following transformations are narrow dependency in MicroSpark: Map, FlatMap and Filter.

Wide dependency: Transformations with wide dependency do calculation according to all partition's data, and repartition during calculation. The following transformations are wide dependency in MicroSpark: GroupByKey, ReduceByKey, Union and Join.

2.2 Repartition in Wide Dependency

1. Get All Keys: Each partition gets all keys from all partitions, then divides keys into. According to worker index, each partition will know which part of key values it need to collect from other partitions.

   Algorithm:

```
for w in workerlist:
    allkeys.extend(w.getAllkeys())
keysIneed = divideKeys(allkeys, len(workerlist), index)
for w in workerlist:
    allkeyValues.extend(w.getAllKeyValues(keysIneed))
```

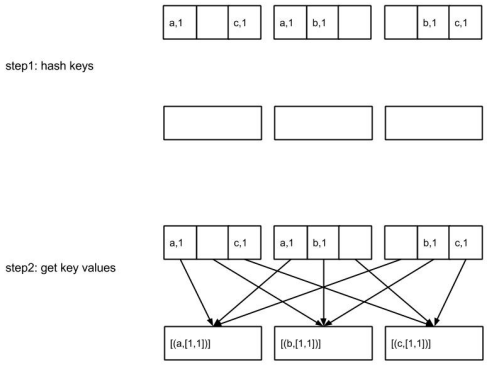

Figure 1: get all keys

2. Get Keys by Hash Function: Each partition divides keys into n buckets. When other partition nodes ask keys, it will send the keys in particular bucket associated with incoming worker index.

   Algorithm:

```
buckets = hashPartition(keys)
for w in workerlist:
    allkeyValues.extend(w.getKeysInBucket(index))
```

Both of 2 algorithms are O(n). In getAllKeys algorithm each partition node need to connect other nodes twice. By using hashPartition, node only connect other nodes once. However, hashPartition algorithm is not always better than Get All Keys. We will discuss it in section 7

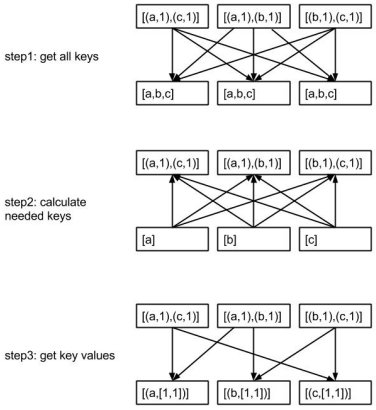

Figure 2: get hashed keys

3 SparkContext

We aim to provide a user friendly design to satisfy user experience. In our design, we create a **Context** class which automatically creates master and workers on different ports and returns a Dummy RDD that contains the master information. User can append RDDs after this Dummy one, and master's information will be passed along with lineage. User can also call collect() on the last RDD which automatically assign and collect work, as demonstrated in Figure 3 and 4.

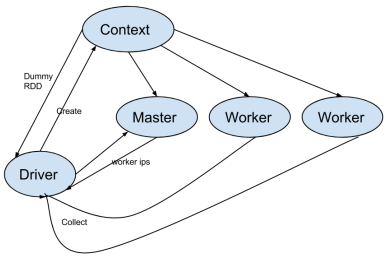

Figure 3: Spark Context

```
C = Context()              //Init context
R = C.init()               //get dummpy RDD
rdd = R.TextFile("file")\
        .flatMap(lambda x: x.split(" "))
                           //transformation
rdd.collect()              //action
```

Figure 4: Driver

4 Fault Tolerance

Fault Tolerance is an important part of RDD implementation. If the failure happens in one worker, the rest of worker should be able to finish the reminding work unless there are no sufficient workers.
In our system, there are two types of workers: **normal** worker and **standby** worker. Normal worker will normally take the work from master, as we mentioned in previous section. Standby worker will do nothing until failure happens: It will be the one to replace the failed normal worker, and do the remaining work. We will demonstrate what happens on master and worker during failure.

4.1 Master

1. RDD Meta Backup
   The "serialized" RDD that master received from Driver, contains all the meta data and functions for computation. It also has the smallest size in its life circle: no computation results at all. We will make a backup of this initial RDD before sending it to other workers, in order to reuse it when failure happen.
   An alternative way is to retrieve RDD(strip out computation data) from other workers. It will cost more overhead waiting time than previous solution. RDD has a **WorkerList**: a hashmap that has [ip+port] as key, and worker index as value. It can not only be used for RDD computation(e.g. index for splitting file or hash bucket), but also for failure handling.

2. Failure Detection The master periodically pings on both normal and standby workers. If lost connection on standby worker: remove it from standby workerlist directly. No other things need to do in this case, for standby worker is doing nothing and only master has the connection to it. If lost connection on normal worker, it will proceed the next step.

3. Worker Replacement, see Figure 5.

   (a) Try select a new worker from standby list
   - If there is no standby worker available, set every worker's status to threatening. Master will notify Driver the work with N partition(N is the number of normal workers at the beginning) cannot be finished.
   - If there exists at least one, pop it out as NewWorker.

   (b) Pop the lost worker from worker list, record its worker index as WorkerIndex

   (c) Put (NewWorker, WorkerIndex) on WorkerList

4. WorkerList Broadcast
   Broadcast the new WorkerList over all workers in normal worker list, updating their WorkerList copy

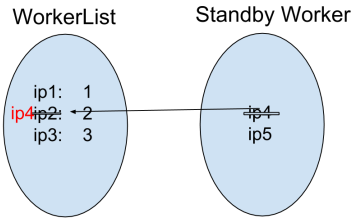

Figure 5: Worker Replacement

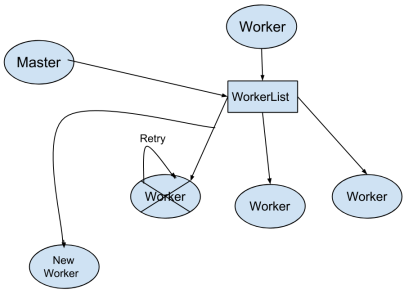

Figure 6: Worker Retry

4.2 Worker

When worker makes remote procedure call, it will make call based on a worker index. When it fails to make a call on a worker index, it will periodically retry the connection, as shown in 6. Eventually, it will get connection on this worker index, since the failed worker will be replaced by a new worker by master with the same worker index.

5 Application

We make three applications to test our MicroSpark: WordCount, PageRank and Interactive Log Mining.

5.1 WordCount

It reads given text file and counts how often words occur. Results are directly dumping into the console.

5.2 PageRank

Our PageRank iteratively updates a rank for each link by adding up contributions from other links that link to it. On each iteration, each link sends a contribution of its rank and number of neighbors to its neighbors. After that, it updates its rank to the sum of the contributions received.

5.3 Interactive Log Mining

A console interactive application will give user more flexibility to mine the data from console. Thus, we create an interactive application by using python code interactively, which will preload all the dependencies and listen to user's keyboard, taking it as input and execute.

6 Extension

While we implement the MicroSpark, one of the hardest part we have is the partitions handling when we meet the wide dependency. When wide dependency happens, all partitions of child RDD need to contact with all partitions of parent RDD to retrieve related data. For example, in groupByKey transformation, each partition of child RDD will deal with a part of keys from maybe all partitions of parent RDD.

1. Each partition must get an all unique keys list from parent RDD partitions
2. Each partition needs to get its own target keys by its own partition index. After getting its own target key list, each partition will contact the parent partition who stores the target key-values pair and retrieves all key-values
3. Each partition sends the final key-values result back to the driver and the driver will merge all results and respond to client.

As we talk above, we use the getAllKeys method to deal with the partitions handling problem. It is a pretty simple implementation but may not have a well performance. After doing some research work on Spark source code, we find that the Spark uses a hash partition algorithm to deal with this problem. Then, we decides to implement the same way as original Spark and do some performance comparison between getAllKeys and hashPartition methods.
Contrast with the hash function in Spark, we use the python built-in hash function from hashlib to convert the keys as an integer hash values. Then, we build a bucket for all hash values. To get the index of each hash value in the bucket, we use the hash value to module the number of partitions. After done the module operation for all hash values, all keys will be divided into number of partitions parts into the bucket. Then, each partition will retrieve its own key-value pairs by pass its partition index to the data source partitions.
In the first version implementation, each partition performs a getAllKeys or hashPartition once some retrieve key-value pairs request comes in. Consider of the performance, we decide to let each partition do a one-time partition at the beginning of the groupByKey execution. Then it only need to respond to each retrieve request determined by the pass-in partition index. After this optimization, out partition performance achieve a significant improvement.
As extension, first we do a comparison for the original partition with the revised

partition implementation. Then, we focus on the performance comparisons between getAllKeys and hashPartition. In the following section, we will introduce the results for the comparison and explain our compare mechanisms.

7 Results

In order to improve the accuracy of the comparison, we run hundreds of tests for sequential mode, local mode and cluster mode. For sequential mode and local mode, we run the word-count program in a four cores laptop. For cluster mode, we run the word-count program in the bass nodes. We use one master and three workers, includes a standby worker for both local mode and cluster mode. Our test files' sizes are from 21 KB until 63.4 MB. We generate five charts for the final results, the X-axis is the file size in MB and the Y-axis is the program execution time in Second.
The Figure 8 indicates that the performance results between our original partition implementation and the revised version. For the original version, if we run a file that large than 2 MB, our program will run out of time and crash because it takes too long time. But for the optimized version, it takes almost no time compared with the original partitions.

Since the code execution time in local is much different with the execution

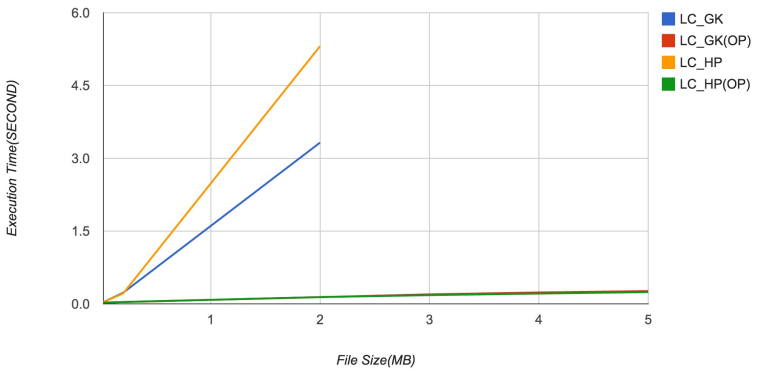

Figure 7: Performance chart for original and optimized partitions

time in cluster, we generate two charts, the one is with cluster execution times and the other one is without them. The Figure 10 is the data for the file size from 21 KB to 1.1 MB. This chart easily shows that the performances in local and sequential modes are much better than the cluster mode for small input files. The Figure 12 is the data for the file size from 2.1 MB to 63.4 MB. For the hashPartition of cluster mode, it could run at most 31.7 MB file and then run out of time. For the getAllKeys of cluster mode, it could run at most 42.3

MB file. From these two figures, we conclude that the performance of cluster mode is much worse than the cluster and sequential modes both for small files and large files. The reason is the network transmission is extremely expensive in our cluster.

In Figure 9, the performance of getAllKeys in local mode is a little better

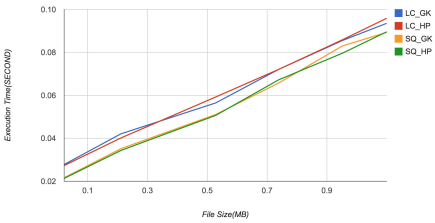

Figure 8: Performance chart for small files without cluster mode

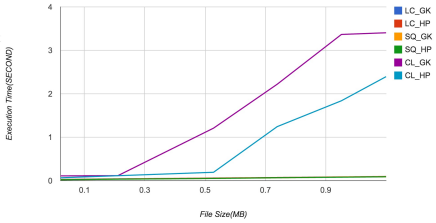

Figure 9: Performance chart for small files with cluster mode

than hashPartition and the performance of getAllKeys in sequential mode is a little worse than hashPartition for small files. In Figure 11, the performance of getAllKeys in both local and sequential modes are obviously better than hash-Partition for large files.

There are two reasons to explain why getAllKeys' performance is better than

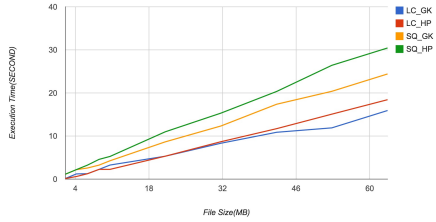

Figure 10: Performance chart for large files without cluster mode

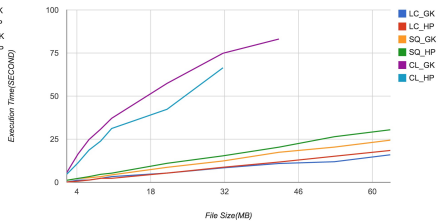

Figure 11: Performance chart for large files with cluster mode

hashPartition in local and sequential modes. The first reason is the module operations in hashPartition is extremely expensive. The second reason is the hashPartition is not balanced while the partition of getAllKeys is absolutely balanced. But in the cluster mode, since the hashPartition only need one round network transmission, its performance is a little better than the getAllKeys.

8 Conclusion

The resilient distributed datasets (RDDs) we presented is an an efficient and fault-tolerant data structure to share data via cluster. It supports the code shipping from client side and allows dynamic adding workers.It can also implement

a wide range of parallel applications, such as WordCount, PageRank and Log Mining.

References

[1] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica, *Spark: Cluster Computing with Working Sets*, 2010.

[2] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica, *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*, 2012.