Pre-Stack Kirchhoff Time Migration on Hadoop and Spark

Chen Yang, Jie Tang[1(✉)], Heng Gao, and Gangshan Wu

State Key Laboratory for Novel Software Technology
Department of Computer Science and Technology
Nanjing University, Nanjing, China, 210046

Abstract. Pre-stack Kirchhoff time migration (PKTM) is one of the most widely used migration algorithms in seismic imaging area. However, PKTM takes considerable time due to its high computational cost, which greatly affects the working efficiency of oil industry. Due to its high fault tolerance and scalability, Hadoop has become the most popular platform for big data processing. To overcome the shortcoming too much network traffic and disk I/O in Hadoop, there shows up a new distributed framework—Spark. However the behaviour and performance of those two systems when applied to high performance computing are still under investigation. In this paper, we proposed two parallel algorithms of the plre-stack Kirchhoff time migration based on Hadoop and Sark respectively. Experiments are carried out to compare the performances of them. The results show that both of implementations are efficient and scalable and our PKTM on Spark exhibits better performance than the one on Hadoop.

Keywords: Big Data; Hadoop; Spark; Kirchhoff; MapReduce; RDD.

1 Introduction

Pre-stack Kirchhoff time migration (PKTM)[1] is one of the most popular migration technique in seismic imaging area because of its simplicity, efficiency, feasibility and target-orientated property. However, practical PKTM tasks for large 3D surveys are still computationally intensive and usually running on supercomputers or large PC-cluster systems with high cost for purchasing and maintaining.

Nowadays, cloud computing has received a lot of attention from both research and industry due to the deployment and growth of commercial cloud platforms. Compared to other parallel computing solutions such as MPI or GPU, cloud computing has advantages of automatically handling failures, hiding parallel programming caomplexity as well as better system scalability. Thus more and more geologists turn to finding seismic imaging solutions on Hadoop and Spark.

Hadoop is an Apache open source distributed computing framework for clusters with inexpensive hardware[2] and is widely used for many different classes of

data-intensive applications[3]. The core design of Apache Hadoop[4] is MapReduce and HDFS. MapReduce is a parallel computing framework to run on HDFS, it will abstract the user's program for two processes: Map and Reduce. A key benefit of MapReduce is that it automatically handles failures, hiding the complexity of fault-tolerance from the programmer[5].

Apache Spark, which is developed by UCBerkeley's AMP laboratory, is another fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala and Python, and an optimized engine that supports general execution graphs[6]. Spark is a MapReduce-like cluster computing framework[6–8]. However, unlike Hadoop, Spark enables memory distributed data sets, provides interactive query, optimize iterative workloads. Most importantly, Spark introduces the concept of memory computing(RDD)[3], i.e. data sets can be cached in the memory to shorten the access latency, which is very efficiency for some applications.

In this paper, we propose two parallel algorithms of the pre-stack Kirchhoff time migration based on Hadoop and Sark respectively. Experiments are carried out to compare the performances of them. The results show that both of implementations are efficient and scalable while PKTM on Spark exhibits better performance than the one on Hadoop.

2 Related Work

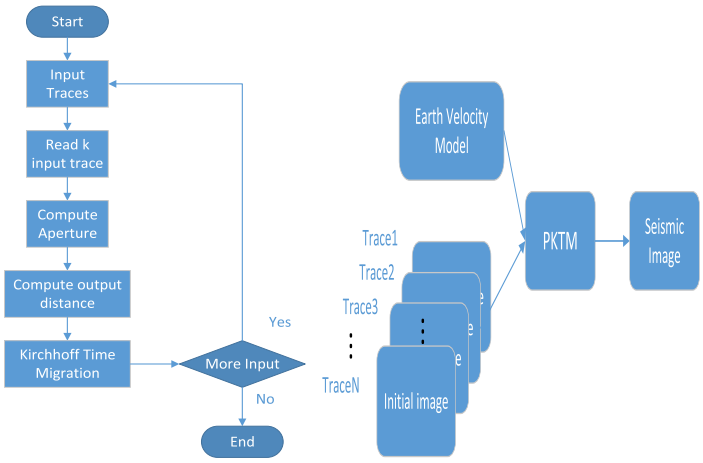
Several works exist in the literature with regards to implementation of Kirchhoff on MapReduce framework. Rizvandi[9] introduces an algorithm of PKTM on MapReduce framework, which splits data into traces, sends traces to Map function, then computes traces, shuffles data to Reduce function. The program use MapReduce parallel framework to realize parallelism of the PKTM, but the problem is that the shuffle data is very huge, and seriously affected the performance of the program.

Another kind of parallel PKTM uses GPUs to achieve parallelism. Shi[1] presents a PKTM algorithm on GPGPU[10], it uses multi-GPUs to compute the data and runs much faster than CPU implementation. However, the memory on GPUs are not large enough. Therefore, when the data gets bigger and bigger and cannot be hold in GPU memory, the data transfer between RAM and GPU memory will be the bottleneck of the program. Li[11] puts forward another P-KTM on GPUs. It's 20 times faster than a pure CPU execution, still the data transfer between RAM and GPU memory as well as loop control are overloaded. Generally, Running PKTM on GPUs will have the problems of data transfer and synchronize between GPUs and CPU.

Gao[12] presents a solution utilizing the combination of the GPUs and MapReduce. It can greatly accelerate the execution time. However, it just abstract the Map function to a GPU implementation. If the Map function needs to change, the whole GPU codes must be modified, which is not flexible. So far, there is no Kirchhoff works on Spark.

3 Algorithms

Kirchhoff migration uses the Huygens-Fresnel principle to collapse all possible contributions to an image sample. Wherever this sum causes constructive interference, the image is heavily marked, remaining blank or loosely marked on destructive interference parts. A contribution to an image sample T(z,x,y,o) is generated by an input sample S(t,x,y,o) whenever the measured signal travel time t matches computed travel time from source to (z,x,y,o) subsurface point and back to receiver. The set of all possible input traces (x,y,o) that may contribute to an output trace (x,y,o) lie within an ellipsis of axis ax and ay (apertures) centered at (x,y,o). Input traces are filtered to attenuate spatial aliasing. Sample amplitudes are corrected to account for energy spreading during propagation. PKTM algorithm and program data flow structure shown in Fig.1.



(a) PKTM algorithm (b) RKTM flowchart

Fig. 1: PKTM flowchart

The schematic of the seismic imaging shown in Fig.2.

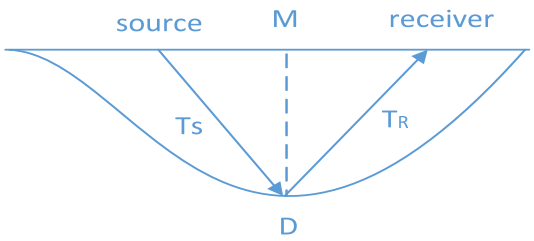


Fig. 2: Seismic imaging

The pseudocode of the Kirchhoff algorithm shown as below:

Algorithm 1 Kirchhoff Algorithm

```
1: procedure KIRCHHOFF(inputtraces)
2:     for all input traces do
3:         read input trace
4:         filter input trace
5:         for all output traces within aperture do
6:             for all output trace contributed samples do
7:                 compute travel time
8:                 compute amplitude correction
9:                 select input sample and filter
10:                accumulate input sample contribution into output sample
11:            end for
12:        end for
13:    end for
14:    dump output volume
15: end procedure
```

3.1 PKTM on Hadoop

We propose an algorithm to carry on PKTM on Hadoop framwork. The steps of the algorithm are as follows:

1. Acquiring cluster environment variables: As we all know, Hadoop framework is closely related to machine configuration. So in the first step, some system variables are detected and stored, including number of nodes n, memory of each node M_1, M_2... M_n, number of CPUs *cpus*, number of cores per CPU *cores*, threads per core *threads*.
2. Inputting data: In Hadoop, each input file block corresponds to a mapper. Also according to Hadoop[4] document, a node performs well when the number of mappers running on it are between 10 and 100. Therefore, in order to control the number of mappers, we override the *FileInputFormat* class, which is in charge for how to logically split the input files. Each split will produce a mapper. Hadoop usually splits a file into default size. The size of the split is key to system efficiency. Hence ,we need to re-split the whole file into splits with proper size. Firstly, we read in default splits lengths of the whole file S_1, S_2, S_3... S_n. Based on these lengths, we can calculate the total size of the input file as $\sum_{i=1}^{n} S_i$. The number of reduce tasks r_n is set by the user. Then to limit the number of mappers between 10∼100, the size of a split is computed as follows:

$$f_{split} = \frac{(\sum_{i=1}^{n} S_i)}{k * min\left((cpus * cores + r_n), \left(\frac{\sum_{i=1}^{n} M_i}{M}\right)\right)}$$

where $k \in [1, \infty)$ is a controllable parameter that can be modified. f_{split} must be greater than 0. M is the memory size of a mapper. Finally we set the *mapreduce.input.fileinputformat.split.maxsize* as f_{split} which decides the split size. Through input split processing, we read in the samples of an input trace, producing $< key, value >$ pairs. *key* represents the offset of input trace in the input files, *value* represents the coordinate points of each input trace. One pair may combine many input traces. These pairs will be submitted to the mapper for execution.

3. Mapping: Since the input files are logically divided into E splits, each split data is submitted to a mapper, these mappers will be computing in parallel. Each mapper computes a lot of input traces, and each input trace will produce many output traces, so in order to achieve parallelism within the mapper, we use the multi-threads to process input traces. Each thread shares a data structure HashMap which is used to save the output trace with the same output key, so that the same output trace will be merge locally which greatly decrease the load of the data transmission. We detect the *Hyper-Threading* mode of the cluster system. If it is on, we set the number of threads of a mapper to $2 * (threads - 2)$, else we set it to $(threads - 2)$. Another strategy to decrease the load of data transmission is to write map values to the HDFS files, only send $< key, filename\#offset >$ pairs.

4. Combining: In this phase, we aggregate the output pairs which are generated by mappers with the same key on the same node and will reduce the network traffic across nodes, thus improve the efficiency of the program. Because in the same mapper and in the same node, there will be many output traces with the same key. This also largely decrease the load of data transmission.

5. Partitioning: The output $< key, value >$ pairs are mapped to reduce nodes with the key. Ensure that all keys are sorted in each reduce node according to its reduce task attempt id. The formula is as follows:

$$f_{hash} = [\frac{key}{\left(\frac{key_{max}}{reduce_{id}}\right)}]$$

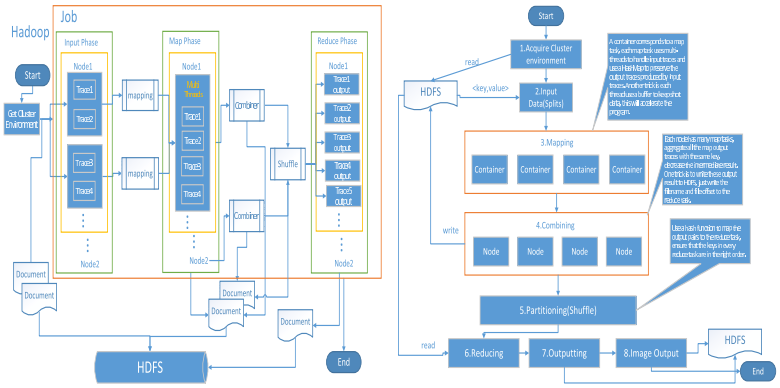
In this way, the final large image data sort time will be reduced. We only need to sort the keys in each reduce task in parallel and this further reduces the application execution time.

6. Reducing: According to the output $< key, filename\#offset >$ pairs which are sent by map tasks, we read HDFS files according to *filename, offset* and aggregate them with the same key comes from different nodes. All the reduce tasks run in parallel. Besides, the number of the reduce tasks is set by the user, user can adjust this number to get the best performance.

7. Outputting: Each reduce task produces an output $< key, value >$ pairs, we sort these keys in each reduce task, then write them to a binary file, the file name contains the minimum key in each reduce task. All the sort operations are run in parallel.

8. Image Output: According to the reduce output files, we sort these file names with the minimum key, this only take a little time, then write them into only one image binary file sequentially. This image file is the finally imaging file to show to the professionals.

The program's running architecture shown in Fig.3.(a) and its flowchart shown in Fig.3.(b). In the flowchart, the steps exact match the steps in the algorithm above.



(a) Running on Hadoop (b) Flowchart of Kirchhoff

Fig. 3: Kirchhoff Implementation on Hadoop

3.2 PKTM on Spark

Spark provides RDD which shields a lot of interactions with the HDFS and achieve better efficiency for application with lots of iterations. Spark develops the ApplicationMaster which acts much more appropriate with Yarn. Hence we develop a new algorithm with Spark system on Yarn framework.

1. Acquiring cluster environment variables: The program on Spark need to read data from HDFS, we just simply read splits from HDFS using *newAPIHadoopFile* with whatever splits, producing $< key, value >$ pairs as records of RDD. RDD can be partitioned according to user's wishes. One partition corresponds to an executor which is similar to mapper. In addition, Spark provides the command—"spark-submit" to submit an application. Users can set the number of task executors N, the memory of each executor and the CPU cores of each executor through the command line. This is very convenient compared to Hadoop. So we just read the environment variables from the command line, which is very convenience. Then the partitions of the

RDD will be set as:

$$f_{pars} = k * min\left(N, \frac{\sum_{i=1}^{n} M_i}{M_{min}}\right)$$

2. Inputting data: Because we run the Spark on Yarn and HDFS framework, we need to read files from HDFS. Spark provides several functions to read input files from HDFS and return them in RDD model. Each RDD contains many records of $< key, value >$ pairs, *key* represents the offset of the input trace and *value* represents the coordinate samples of the corresponding key. One record combines many traces.
Moreover, RDD can persist data in memory after the first calculation, so we persist those RDD records in memory and hard disk(if it's too large). This greatly reduces repeatedly reading from HDFS. Then we partition the RDD with the above formula. These partitions will be calculated in parallel.

3. FlatMapping: In Spark, RDD partitions will be sent to executors. Executor starts to calculate a partition. After it's done, it continues with the remaining partitions. All of the executors calculate in parallel. In this map period, we also use the multi-threads to compute the input traces. Whether a cluster system opens *Hyper-Threading* or not, we just set the threads number as $(threads - 2)$ to ensure a good performance, because Spark uses thread model while Hadoop uses process model. In Hadoop, when map function running to a proper percentage(such as 5%, this can be set by the configuration), reduce tasks will be launched to collect output pairs from mappers. But in Spark, reduce tasks wait until all mappers finish and return with RDD. This feature make the program on Spark more efficient as reduce task do not occupy the resources used by mappers. It's worth to mention that the partitions of RDD do not change until you invoke the *repartition* function.

4. Partitioning: Firstly, we get the total number of output trace *onx*. Then we divide these keys depending on the number of reduce partitions R_n. Smaller keys correspond to the smaller reduce task id. This helps the later sort operation. Each record in RDD applies a mapping operation to choose into which the reduce partition goes. The formula is as follows:

$$f_{partition} = [\frac{key}{\left(\frac{onx}{R_n}\right)}]$$

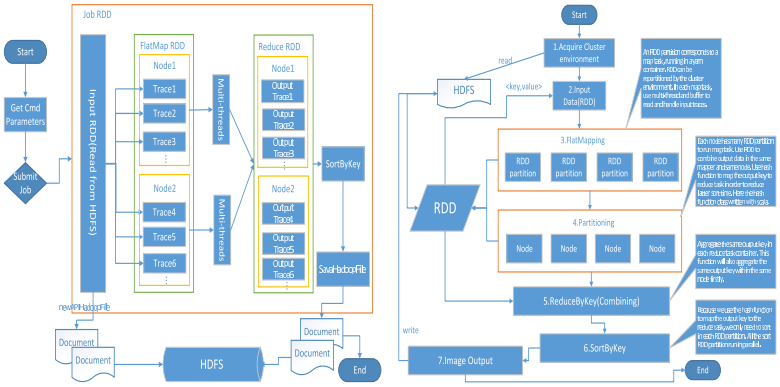
5. ReduceByKey: Spark ensures that the same key pairs will be sent to the same reduce task. According to this feature, the RDD datasets that come from a map operation are sent to the reduce tasks with the same key. So we aggregate the values by the same key in each reduce task. Each reduce task return an RDD partition which will be aggregated into a total RDD to the user, however, its partitions still exist. The important point of the *ReduceByKey* function is that the operation firstly merges map output pairs in the same node, then send pairs to the correspond reduce tasks. This feature greatly reduces the load of network traffic.

6. SortByKey: This function sorts the keys in each RDD partition which reduce tasks returned. The sort operation is also executed in parallel among different RDD partitions.

7. Image Output: According to the sorted keys, we write the corresponding values to a binary file to HDFS for permanent preservation.

The program's running architecture shown in Fig.4.(a) and its flow chart shown in Fig.4.(b). The steps of the flow chart corresponds to the steps of the algorithm above.

The program flowchart shown in Fig.4.



(a) Running on Spark (b) Flowchart of Kirchhoff

Fig. 4: Kirchhoff Implementation on Spark

4 Results and analysis

We have run our two implementations on a cluster with six nodes, the details of which are shown in Table.1.

Table 1: Cluster Configuration

Name	CPUs	Cores Per CPU	Thread Per Core	Memory(G)
Master	2	8	4	32
Slave1	2	6	4	32
Slave2	2	6	4	32
Slave3	2	6	4	32
Slave4	2	6	4	32
Slave5	2	6	4	32

4.1 Experiment Configuration

The node "Master" acts as the Hadoop Master node and the remaining nodes act as Hadoop Slave nodes. "Master" is not only the master of the HDFS framework(NameNode), but also the master of the Yarn framework(act as ResourceManager). Five nodes, "Slave1", "Slave2", "Slave3", "Slave4", "Slave5", are used as the DataNode and the NodeManager.

4.2 Data Preparation

We use Sigsbee2 Models to test our proposed PKTM methods on Hadoop and Spark. The model could be downloaded from http://www.reproducibility.org/RSF/book/data/sigsbee/paper html/. PKTM includes the following three main steps: data preprocessing, migration and output. PKTM use two input data file formats including "meta" files and "data" files[13]. The input files in this program contains input trace meta file(shot.meta), input trace seismic source meta file(fsxy.meta), input trace detector location meta file(fgxy.meta), input trace center points meta file(fcxy.meta), velocity meta file(rmsv.meta). Each of the meta file has a corresponding data file(*.data), such as shot.data(about 30GB), fsxy.data, fgxy.data, fcxy.data, rmsv.data.

4.3 Experimental Results

We experiment our program from the following aspects:

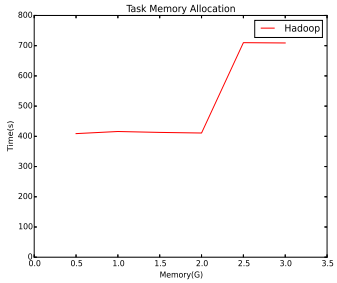
(1). Experiments on Hadoop:

a. We firstly test how the memory of a mapper's container affects the performance of the program. The test results shown in Fig.5.(a). It can be seen that when the consumed memory exceeds over some threshold, the number of mappers reduces accordingly, which causes the execution time getting longer. The reason is that the number of active parallel tasks is constrained by the the total memory of these mappers.

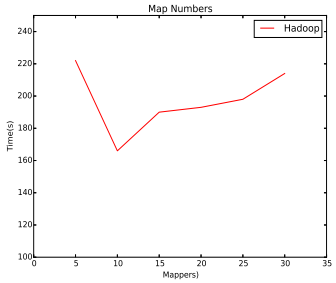
b. Secondly, we test how the number of mappers affects the performance. The test results are shown in Fig.5.(b). In the figure, it can be seen that the executing time is smallest when the mapper's number fits to the cluster's resources. If the number of mappers is small, the capacity of the cluster is not fully utilized, and results in a longer execution time. If the number of mappers is too large, the scheduling time usually will increase, which also results in a longer execution time.

c. Finally, in Hadoop, the cluster will start Reduce tasks to receive map output files even when some Mappers still not finished. In this situation, the started Reduce tasks occupy the resources of memory and CPUs which affects other mappers' execution. Therefore, we try to test how reduce task numbers affects the performance. The results are shown

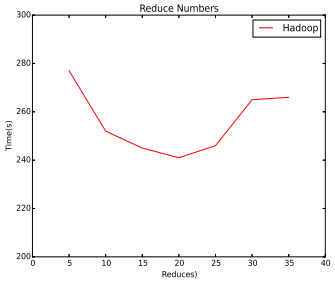
in Fig.5.(c). In the figure, it can be seen that when the numbers of reduce tasks match the number and output pairs of mappers, the best performance will be achieved.



(a) Map Task Memory



(b) Mappers' numbers



(c) Reduces' numbers

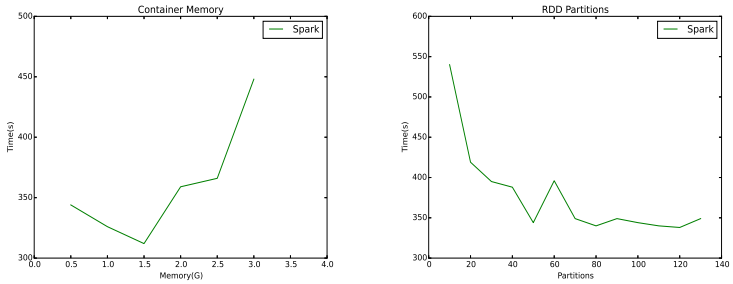
Fig. 5: Kirchhoff Experiment on Hadoop

(2). Experiments on Spark: in Spark, the contribution of two factors, memory of each execution(container) and number of RDD partitions, are investigated.

a. We adjust the configuration of the container's memory and evaluate the performance of the system. The results are shown in Fig.6.(a). Like Hadoop, if the memory of a container exceeds to a proper value, the execution time grows longer. Because the large memory affects the parallel number of tasks. When the number of parallel tasks is small, the execute time all the tasks also increases.

b. In this part, we test the RDD partitions about the input traces, hope to find out the best partitions. The test results are shown in Fig.6.(b). The figure indicates that when the number of RDD partitions is not enough to the cluster, it takes a longer time, but when partitions grows, it tends to a less and stable execution time. The reason of this phenomenon is that when RDD partition is more than the cluster resources can have,

it makes the cluster run busy to handle RDDs with the same amount of time.



(a) Container Memory (b) RDD partition

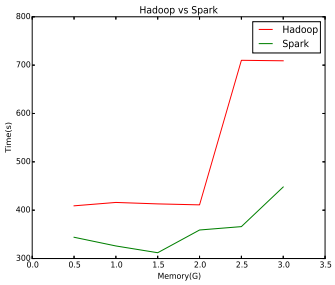
Fig. 6: Kirchhoff Experiment on Spark

(3). Experiments of comparison between Hadoop and Spark: We mainly from three aspects to compare the Hadoop algorithm and Spark algorithm.

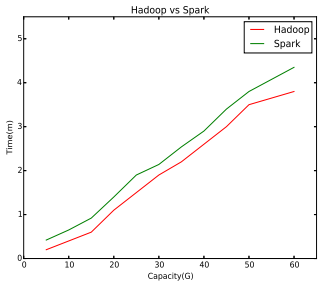
a. Firstly, we compare the yarn container memory between Hadoop and Spark. On Yarn, the tasks of Hadoop or Spark jobs are started in container, one task correspond to a container. The comparison figure is shown in Fig.7.(a). As shown in the figure, we know that with the same memory of container, Spark shows a better performance, Because Spark use the RDDs to read input traces and persists data in memory, the computation of RDDs is in memory, so it runs fast.

b. Secondly, we compare the reading and writing time of Hadoop and Spark I/O capacity. The results are shown in Fig.7.(b). We can see that when we access the same capacity of I/O data, the Spark algorithm runs faster than the Hadoop's. During the same period, Spark's I/O capacity is larger than Hadoop's I/O capacity.

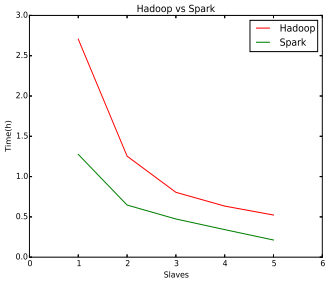
c. Lastly, we compare the running time of the Kirchhoff application on Hadoop and Spark framework. The results are shown in Fig.7.(c). We can see that with the RDD mechanism and the optimum of the algorithm, PKTM on SparK achieve better efficiency than that on Hadoop.



(a) Container Memory Comparison



(b) I/O with Time



(c) Running Time Comparison

Fig. 7: Kirchhoff Hadoop vs Spark

5 Conclusion

In this paper, we proposed two parallel algorithms of pre-stack Kirchhoff time migration based on Hadoop and Sark respectively. The results show that both of the implementations are efficient and scalable. And PKTM on Spark exhibits better performance than the one on Hadoop.

The future work includes how to improve the data transferring speed in both Hadoop and Spark, since the efficiency of these 2 programs are closely related to the data preparing speed. If the machine has a high throughput of the network and the high-speed hard disk I/O, the program will run faster. In Hadoop, we can apply RDMA(Remote Direct Memory Access) in HDFS through Infiniband[14]. This will greatly promote the acceleration of HDFS read and write time. We hope we can also apply Infiniband Network Interface Card to accelerate the network transfer in Spark.

Acknowledgments. We would like to thank the anonymous reviewers for helping us refine this paper. Their constructive comments and suggestions are very helpful. This paper is partly funded by National Science and Technology Major Project of the Ministry of Science and Technology of China under grant 2011ZX05035-004-004HZ. The corresponding author of this paper is Tang Jie.

References

1. Shi X, Wang X, Zhao C, et al.: Practical pre-stack kirchhoff time migration of seismic processing on general purpose gpu[C]//2009 World Congress on Computer Science and Information Engineering. IEEE, 2009: 461-465.
2. ChengXueQi, JinXiaoLong, WangYuanZhuo, etc.: Summary of the Big Data systems and analysis techniques[J] in Chinese. Journal of Software, 2014, 25(9).
3. Gu L, Li H.: Memory or time: Performance evaluation for iterative operation on hadoop and spark[C]//High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on. IEEE, 2013: 721-727.
4. Apache Hadoop. [Online]. Available: http://hadoop.apache.org
5. Zaharia M, Konwinski A, Joseph A D, et al.: Improving MapReduce Performance in Heterogeneous Environments[C]//OSDI. 2008, 8(4): 7.
6. Apache Spark. [Online]. Available: http://spark.apache.org
7. Zaharia M, Chowdhury M, Franklin M J, et al.: Spark: cluster computing with working sets[C]//Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 2010, 10: 10.
8. Zaharia M, Chowdhury M, Das T, et al.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing[C]//Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012: 2-2.
9. Rizvandi N B, Boloori A J, Kamyabpour N, et al.: MapReduce implementation of prestack Kirchhoff time migration (PKTM) on seismic data[C]//Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2011 12th International Conference on. IEEE, 2011: 86-91.
10. De Verdiere G C.: Introduction to GPGPU, a hardware and software background[J]. Comptes Rendus Mcanique, 2011, 339(2): 78-89.
11. Panetta J, Teixeira T, de Souza Filho P R P, et al.: Accelerating Kirchhoff migration by CPU and GPU cooperation[C]//Computer Architecture and High Performance Computing, 2009. SBAC-PAD'09. 21st International Symposium on. IEEE, 2009: 26-32.
12. Gao H, Tang J, Wu G.: A MapReduce Computing Framework Based on GPU Cluster[C]//High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on. IEEE, 2013: 1902-1907.
13. WangGang, TangJie, WuGangShan.: GPU-based Cluster Framework[J] in Chinese. Computer Science and Development, 2014, 24(1): 9-13.
14. Lu X, Islam N S, Wasi-ur-Rahman M, et al.: High-performance design of Hadoop RPC with RDMA over InfiniBand[C]//Parallel Processing (ICPP), 2013 42nd International Conference on. IEEE, 2013: 641-650.