International Journal of Grid and Distributed Computing
Vol. 10, No. 3 (2017), pp.1-14
http://dx.doi.org/10.14257/ijgdc.2017.10.3.01

A HDFS-Based High Performance Parallel Framework for Extracting Seismic Gathers from Large Scale Seismic Datasets

Chao Li[1], Jiamin Wen[2], Changhai Zhao[2], Jiguo Du[2], Minqiang Shang[2], Zengbo Wang[2], Haihua Yan[1] and Yida Wang[1]

[1]*School of Computer Science and Engineering, Beihang University Beijing, China*
[2]*Research and Development Center, BGP Inc., China National Petroleum Corporation Zhuozhou, Hebei, China*
casesense@163.com

Abstract

In petroleum industry, potential oil reservoirs can be located by analyzing seismic data. Seismic data is composed of a series of fixed-size traces. In recent years, the size of seismic data can achieve hundreds of Terabytes and a seismic dataset of a large 3D survey area typically contains billions of seismic traces. A seismic gather is a collection of traces that share same attributes. During the processing of interactive analysis, seismic data is accessed by gathers. As the traces contained in a gather are usually distributed among the whole dataset, extracting the gather from seismic data will be translated to a series of random read operations from the parallel file system, which significantly reduces the efficiency. In this paper, we propose a HDFS-based framework to improve the efficiency of extracting gathers. First, the traces of a gather are divided into groups based on the principle of data locality so that each node just needs to read data from its own local disks using multiple threads. Second, dynamic load balancing mechanism is implemented for the framework based on the block duplication feature of HDFS to avoid the performance degradation caused by lagging nodes. Last, fault tolerance is supported to ensure the data integrity in case of node failures. Experimental results demonstrate that the framework promises high scalability, well load balancing ability as well as high availability and it performs more than 178 times better than the traditional approach when using 25 nodes.

Keywords: *HDFS, Seismic exploration, Random read optimization*

1. Introduction

Seismic exploration is widely used in petroleum industry to locate the position of new oil reservoirs. It can be divided into three steps: acquisition, processing, and interpretation. In the acquisition step, seismic data is collected from fields. In the processing step, the seismic data will be processed using various seismic data analysis algorithms to translate the unknown geologic structures into visual images. In the interpretation step, the visual images will be analyzed by geophysicists and geologists to judge whether the survey area is a potential oil reservoir. The problem that this paper is trying to solve lies in the second step.

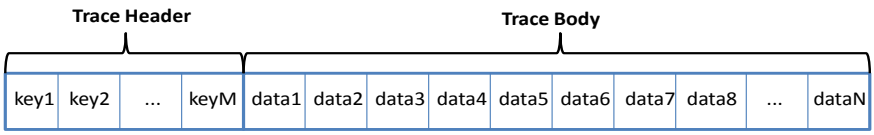


Figure 1. The Structure of a Seismic Trace

ISSN: 2005-4262 IJGDC

Seismic data is composed of a series of fixed-size records and the record is referred as trace in the field of seismic exploration. The structure of one seismic trace is shown in Figure 1. Each trace has two parts: trace header and trace body. Trace header contains the attribute information and trace body contains a vector of seismic data samples which represent the amplitude information of seismic waves. In Figure 1, the length of trace header is M and the number of seismic data samples is N. Although multiple standards exist for organizing seismic data, SEG-Y [1] is the most widely used seismic data format as shown in Figure 2. There are two parts in a SEG-Y seismic data file. The first part is a file header which describes basic information of seismic data and second part contains a sequence of seismic traces.

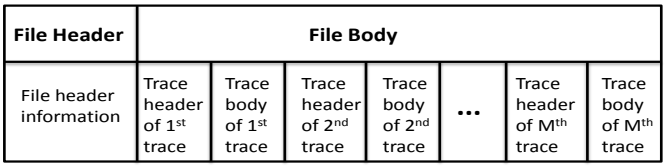


Figure 2. The Format of a SEG-Y Seismic Data File

Seismic data is a typical high-dimensional dataset as it contains rich attribute information. The attributes in trace header can be treated as sorting keys since the seismic data can be sorted by specified attributes. For example, the seismic data can be sorted by key 1 and key 2 or sorted by key 12, key 13, and key 14. A seismic gather is a collection of traces that have common attributes. For example, all the traces contained in a gather may have same key 12, key 13, and key 14. In recent years, the size of seismic data collected from fields can achieve dozens or hundreds of Terabytes with the advance of acquisition technology. A seismic dataset of a large survey area usually contains billions of seismic traces, while a gather usually contains thousands of or tens of thousands of seismic traces. Extracting a gather refers to reading the seismic trace data of each trace in the gather from the seismic dataset and combining the seismic trace data into a whole as shown in Figure 3.

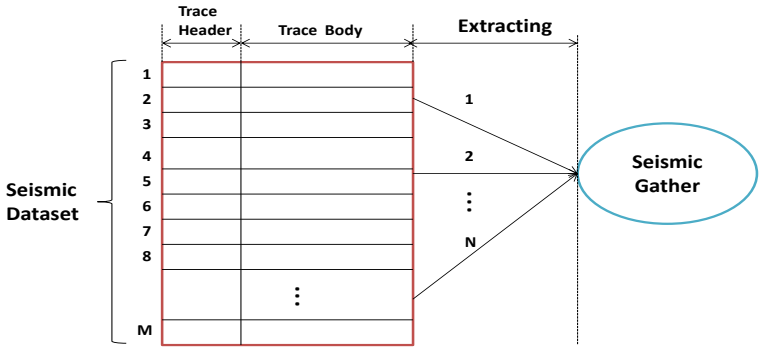


Figure 3. Extracting a Gather from a Seismic Dataset

During the step of processing, seismic data needs to be analyzed by interactive seismic data analysis applications. The applications access the seismic data by seismic gathers. In practice, to save the storage space, there are only one or two duplications sorted by different set of keys for the seismic data. When the existed seismic data on the storage system is not sorted by the specified keys that are used by interactive applications, extracting the gather will be translated to a sequence of random read operations from the parallel file system. This traditional approach of extracting gathers has two disadvantages.

- **Poor efficiency.** The root cause of poor performance of the traditional approach is that the traces of one gather are read randomly. As the random read operation for each trace is time-consuming, the time cost on extracting the gather from the parallel file system seriously impacts user experience of interactive applications.
- **Strong interference.** In the production environment, there are a large amount of data-intensive seismic applications that need to read seismic data sequentially from the parallel file system. These applications may be scheduled to the same cluster with interactive applications. The random read operations issued by interactive applications may reduce the I/O bandwidth of the data-intensive applications by an order of magnitude [2].

To avoid these disadvantages, we propose a high performance parallel framework for extracting seismic gathers based on HDFS [3]. The strong interference on every other data-intensive applications running on the same cluster can be avoided because HDFS is built on the local disks of all the nodes in the cluster. Meanwhile, some distinct characteristics of HDFS such as multiple block duplications and exposed location of blocks make it possible to optimize the performance of extracting gathers. Specifically, the framework we proposed first divides the traces of a gather into groups on the principle of data locality so that each node can read seismic data from its own local disks to improve the random read bandwidth. Then multiple threads are applied to maximize the aggregated random read bandwidth of local disks. In production clusters, the I/O workload on local disks of each node is varying, we implement a dynamic load balancing mechanism for the framework based on the block duplication feature of HDFS in case that the lagging nodes become the bottleneck of the whole framework. In addition, as node failures have become a normal phenomenon rather than an exception [4] [5]. The framework supports fault tolerance, i.e., even if node failures happen during the process of extracting a gather, the framework can guarantee the data integrity for the target seismic gather. Experimental results show that the framework we proposed promises high scalability, well load balancing ability as well as high availability. When 25 high performance nodes are used, it performs more than 178 times better than the traditional approach.

The rest of this paper is organized as follows. The background of seismic exploration and related work are introduced in Section 2. In Section 3, we describe the design and implementation of our proposed framework in details. Section 4 presents the evaluation results. Section 5 concludes the paper.

2. Background and Related Work

2.1. Background of Seismic Exploration

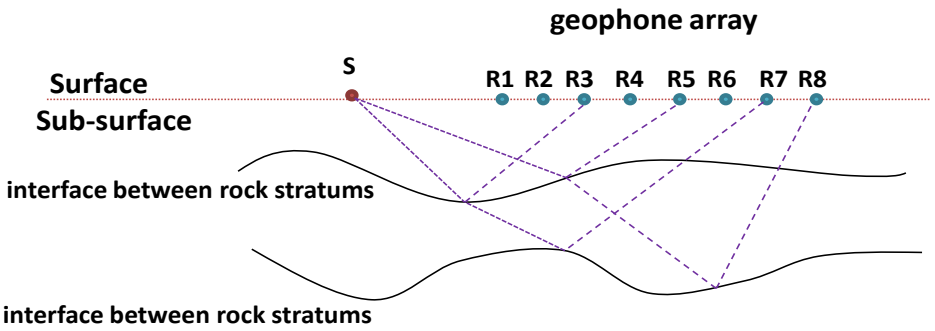


Figure 4. The Principle of Seismic Data Acquisition

Figure 4 illustrates the principle of seismic data acquisition. Seismic waves are generated by an artificial seismic source at shot point S. During the trip of propagating through the subsurface, the waves will be reflected and refracted by the interfaces between rock stratums. Then, the waves will travel back to the surface and be recorded by geophones deployed on the surface (R1-R8). The term trace refers to the seismic waves recorded by one geophone. Each trace is associated with three key position coordinates: the shot point, the receiver point and the middle point of shot point and receiver point. The term offset refers to the distance between the shot point and receiver point. A collection of traces that share a same shot point is called common shot gather. Similarly, a collection of traces that share a same receiver point, a same middle point or a same offset is called common receiver-point gather, common middle-point gather or common offset gather, respectively. The seismic data is usually organized by common shot gather during the acquisition stage. However, during the processing stage, interactive analysis applications may access the seismic data by common receiver-point gather, common middle-point gather or common offset gather, etc. In these cases, the traces of a requested gather need to be extracted from the seismic data.

2.2. A Brief Introduction of HDFS

HDFS, derived from Google's file system GFS [6], is an open source distributed file system. HDFS is also one of core components of Apache Hadoop [7] and the other one is Apache MapReduce [18]. Built on local disks of the cluster, HDFS splits a file into 64MB blocks by default and scatters the blocks among local disks that attached to the computing nodes based on a certain distribution algorithm. Each block has two extra block duplications to guarantee high availability in case of node failures. The run-time system of HDFS including a metadata server and a set of data servers. In HDFS, the metadata server is called NameNode and data server is called DataNode. The data servers can be configured by offering a host file.

Comparing with the traditional parallel file system for clusters such as GPFS [8], HDFS provides the API by which the location of a data block as well as its duplications can be obtained for upper layer applications [9]. Thus, the performance of the applications running on the HDFS can be optimized based on the data location information. For example, by taking advantage of data locality, MapReduce [10] can move computations to data to achieve fast I/O speed and high performance.

2.3. Related Work

Currently, the performance optimization of extracting a record subset from a large scale scientific dataset that is composed of fixed-size records based on HDFS is less researched. To the best of our knowledge, this is the first paper that is trying to optimize the performance of extracting gathers from large scale seismic datasets on HDFS. Random read performance optimization strategies based on data locality of HDFS are introduced in [11]. An approach that aims to improve the query performance towards common query types on HDFS is proposed in [12]. The efficiency improvement of the random read/write on HDFS by Infiniband and SSD are researched in [13].

To facilitate the random read of records on HDFS, various distributed databases built on HDFS such as Hive [14], Spark SQL [15], Impala [16], and HBase [17] are proposed. However, these distributed databases are not suitable for the problem of extracting seismic gathers. The reasons are listed as follows.

- The data stored on these distributed databases, normally from Internet industry, can be optimized by treating memory as a cache to improve the random read performance. The traces of one seismic gather are usually distributed among the

whole dataset whose size can achieve dozens or hundreds of Terabytes. The efficiency improvement by treating the memory as a cache is limited.

- These distributed databases focus on efficiency improvement of performing complex SQL queries such as join on the non-structural data, while extracting a seismic gather from seismic data will not involve complex SQL queries.
- When extracting a seismic gather, each trace can be acquired by reading the seismic data file directly based on the location of the trace. The distributed databases add an extra layer between the upper layer applications and HDFS. The extra layer will reduce the efficiency of acquiring seismic data from HDFS.

Thus, distributed databases are too heavy for extracting seismic gathers. We need a light-weight framework that can extract the seismic gather fast from seismic data for the upper layer applications, especially for the time-sensitive interactive applications.

3. Design & Implementation

3.1. The Architecture of The Framework

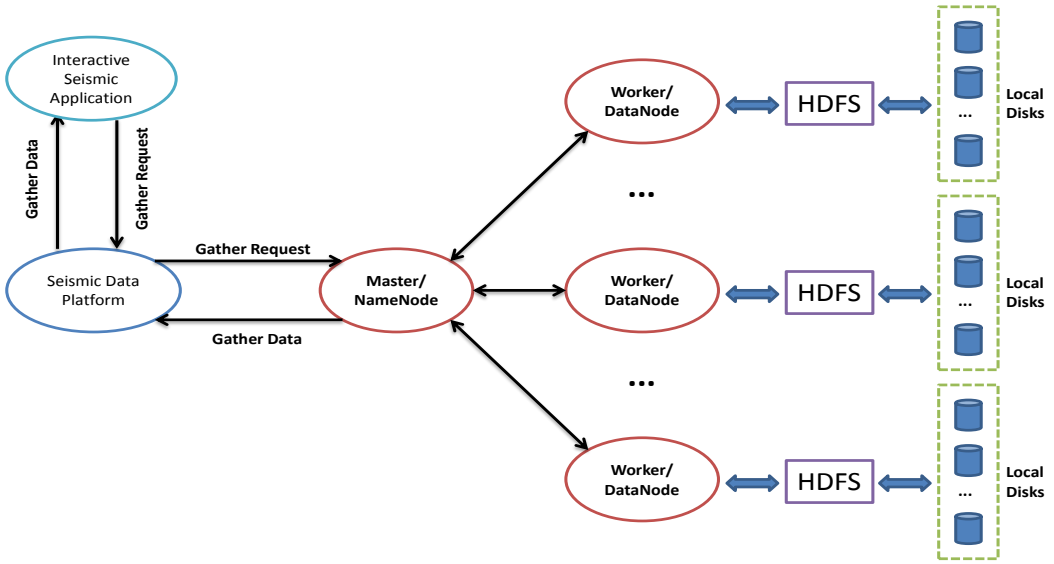


Figure 5. The Architecture of the Framework

Figure 5 shows the architecture of the framework for extracting gathers. The framework is a run-time system using a master/worker architecture. After launching the framework, a daemon will be started on the NameNode as the master and each DataNode will start a daemon as the worker. When the seismic data platform receives the request for a seismic gather from one interactive seismic application, it first checks whether there is a seismic dataset that is sorted by the same set of specified keys as the requested gather. If the seismic data is not available, it turns to the framework by sending the gather request to master. Note that each trace of a seismic dataset is numbered in sequence by the seismic data platform. The gather request is a series of trace numbers. The framework reads the seismic data according to the trace number by which the location of each trace in the seismic file can be calculated. After master receives the gather request, it will divides the traces of the gather into groups so that different nodes can read data for different groups in parallel. During this process, numerous questions are raised, such as how to divide the gather, balance the load, deal with the node failures, etc. The details of how we solving these problems are described in the next sub-sections.

3.2. Two-level Parallelism Based on Data Locality

A two-level parallelism is exploited to improve the efficiency of extracting gathers as shown in Figure 6. At the cluster level, a pool of nodes read different group of seismic traces in parallel. And at the node level, a pool of threads read different seismic traces in parallel. A task-based parallel pattern is adopted to coordinate the communications between master and workers. Specifically, each worker continually requests a task from master and each task contains several trace numbers. After a worker gets a task successfully from master, it will put the task into the task pool whose size equals to the number of threads. Each thread first fetches a task from the task pool, then it will read the seismic data based on the trace numbers of the task. After all the trace data for the task is acquired, it sends the data back to master. Until all the tasks for the gather is finished and the data of each task is sent back, master will return the integral seismic data of the gather to the seismic data platform. Apart from the two-level parallelism, task requesting, task executing and sending data back to master are all performing in the style of pipeline to minimize the overhead caused by the communication between master and workers.

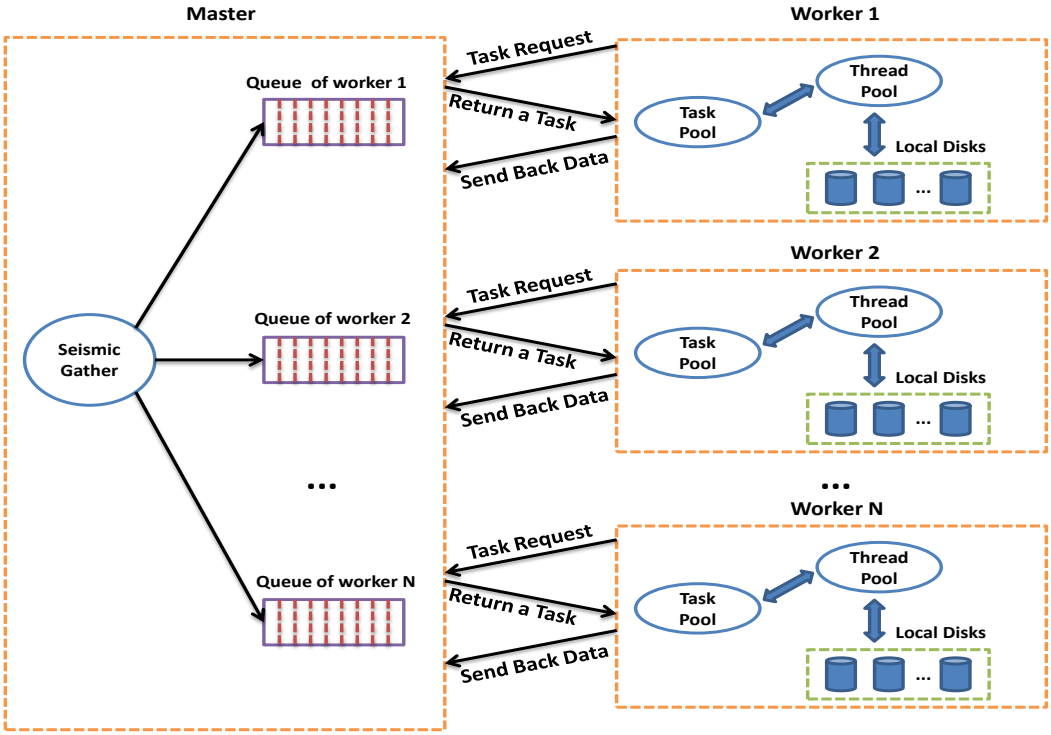


Figure 6. Two-Level Parallelism Based on Data Locality

In HDFS, if the data that the client is trying to read resides on the same node with the client, then the data can be read more fast than the cases in which the data needs to be read from remote nodes [11]. To fully take advantage of the data locality, the traces of one gather is divided into groups according to the trace data distribution among the nodes. By this strategy, each node just needs to read seismic trace data from its own local disks to achieve maximum random read bandwidth. Since the size of each block is 64MB by default and the size of each seismic trace is fixed, so the block number can be calculated based on the trace number. HDFS provides the API by which a host array can be acquired according to the block number. Each host in the host array has a duplication of the block. Normally, the host array for one block has three different hosts as the number of block duplications is 3 by default in HDFS. In our implementation, each trace number of the gather will be placed to one of the three groups that are corresponding to the three hosts

randomly. For a large seismic dataset whose size achieves dozens or hundreds of Terabytes, querying the host array for each block to build the block-host mapping information is time-consuming. In our implementation, the block-host mapping information will be compressed and stored independently on the parallel file system after the first time it is built. The compressed block-host mapping information can be read more efficiently from the parallel file system if the block-host mapping needs to be re-built.

In the framework, each group is represented by a queue as shown in Figure 6. When a worker requests a task from master, the master will pop several trace numbers for the task from the queue that is corresponding to the worker. By this means, no seismic trace needs to be read from a remote DataNode. Note that only a small part of the trace numbers will be packed into the task for two reasons: 1) a finer-grain size of trace numbers for a task can produce more tasks which will improve the degree of parallelism for multiple threads; 2) It can facilitate the implementation of I/O load balancing among different nodes.

3.3. Load Balancing Mechanism

In practice, the framework may encounter various load balancing problems caused by different reasons. First, when the size of a seismic dataset is small, the distribution of the blocks is highly uneven among the DataNodes. This results in that the node which has much more blocks than others will cost more time on reading the seismic data. Second, even if the distribution of traces among different nodes is even, the trace data distribution on the local disks of one node may be strongly uneven comparing with other nodes. That could lead to the node becoming the lagging node which reduces the performance of the whole framework. Last but not least important, on a production cluster, the I/O workload on the local disks of each node is different and keeps changing dynamically caused by the interferences imposed by other applications that is running on the same cluster, the node which has heavy I/O workload still can become the lagging node. Thus, a dynamic load balancing mechanism is highly necessary for the framework to reduce the efficiency loss.

Algorithm 1: Dynamic Load Balancing Algorithm

maxQueueSize is the maximum size of all queues
topWorkerID is the worker ID corresponding to the queue which has the maximum size
avgQueueSize is the average size of all queues
taskGrain is the number of trace numbers contained in a task
01: if(***maxQueueSize*** < ***taskGrain***) return;
02: ***differNum*** = ***maxQueueSize*** - ***avgQueueSize***
03: if(***differNum*** < ***taskGrain***) return;
04: while(***differNum*** > 0):
05: pop a trace number from the queue of worker ***topWorkerID***
06: get another two workers (denoted by ***worker1*** and ***worker2***) who have a replication of the trace
07: ***size1*** = the size of the trace queue of ***worker1***
08: ***size2*** = the size of the trace queue of ***worker2***;
09: ***targetWorkerID*** = ***size1*** < ***size2*** ? ***worker1*** : ***worker2***;
10: push the trace into the queue of worker ***targetWorkerID***
11: ***differNum***--

In our implementation, the load balancing problem is solved by taking advantage of the fact that each block has two extra duplications in HDFS. The dynamic balancing algorithm is detailed by Algorithm 1. The core idea of the algorithm is to transfer the trace numbers from the queue which has the maximum remaining trace numbers to the queues that have less trace numbers frequently. To guarantee the data locality of transferred trace numbers, for each trace that needs to be transferred, the algorithm first gets the other two

hosts whose attached local disks have a replication of the trace data. Between the two hosts, the transferred trace number will be push to the queue whose size is less than the other one. The number of trace numbers that needs to be transferred is determined by the difference between of the maximum queue size and the average queue size. To avoid system jittering, if the maximum queue size is less than the task grain or the number of trace numbers that needs to be transferred is less than the task grain, the transfer operation will not be launched. Whenever a worker requests a task from master, master will first invoke the load balancing algorithm to balance the queues, then it will pop several trace numbers from corresponding queue of the worker. Thus, the load balance algorithm can be invoked frequently to ensure the framework can adapt to the dynamic and changing I/O workload environment on a production cluster.

The determination of task grain is a trade-off problem. A small task grain allows the load balancing algorithm to adjust more precisely. However, if the task grain is too small, the communications between master and workers will increase because more tasks and more data sending back requests. The increased communications may lead to master node becoming the bottleneck of the framework. We determine this parameter by a series of experiments.

3.4. Fault Tolerance

In recent years, the MTBF(Mean Time Between Failures) of a cluster is decreasing with the increasing scale of the cluster. As the framework needs to provide the service of extracting gathers for upper layer applications sustainably, the daemons of the framework need to reside on the nodes of HDFS for a long time. Thus, the ability to tolerate node failures is necessary for the framework.

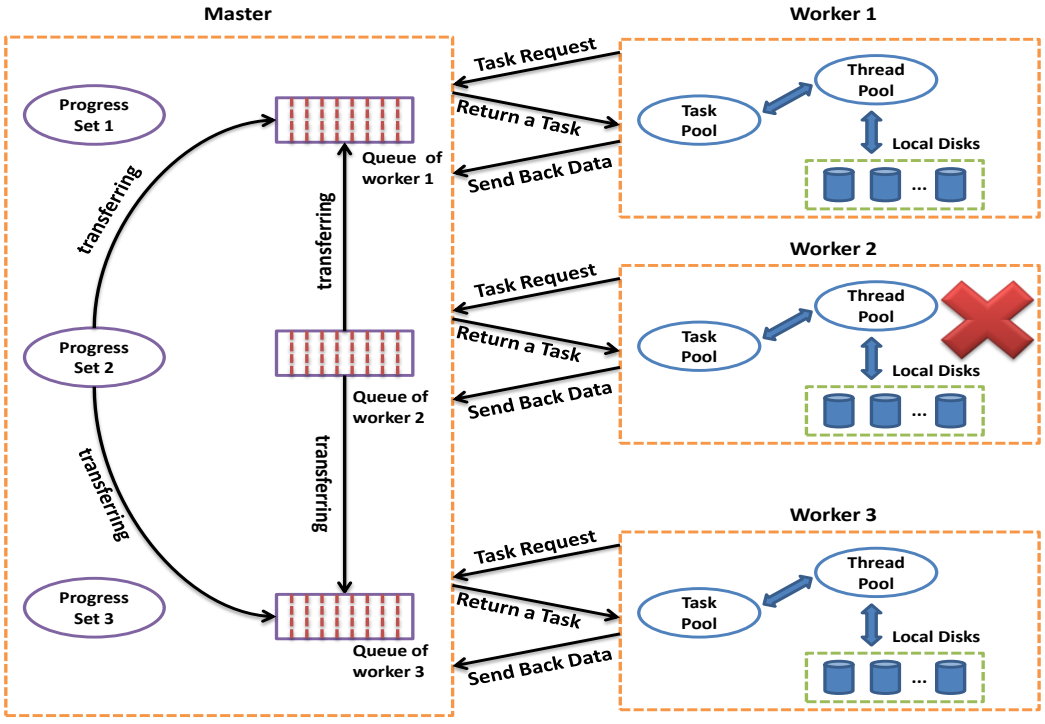


Figure 7. Fault Tolerance Design of the Framework

The fault tolerance design of the framework is shown in Figure 7. In the framework, master sends heartbeat information to workers periodically and expects responses from workers. If no response is received from a worker, master marks the worker as a failed node. An extra group will be set up for each worker on master node, called progress set.

The set contains the trace numbers that has been scheduled out to the worker but the trace data of the trace number has not been received from the worker. When a node failure happens, as the fault detection takes a certain time to perceive the failed node, normally, the trace numbers in the queue of failed node will be transferred to other queues by the load balancing mechanism. Once the master is aware of the failed node, trace numbers in the queue and the progress set of the failed node will be transferred to the queues of healthy workers based on the principle of data locality as shown in Figure 7 where there are three workers and the second worker goes down.

After a node failure is confirmed by master, master will remove the queue of the failed node. As HDFS will recover the lost block duplications of the failed node on other healthy nodes, the block-host mapping information stored on the parallel file system needs to be updated. The framework will update the information at a certain time after a node fails. And the time interval can be configured in the framework. Fault tolerance also affects the load balancing mechanism. Because of the failed nodes, when transferring a trace number, there may be only one or zero extra queue for the trace number. The trace number will be placed to the queue directly if there is only one available queue or stay in the original queue if no queue is available. The framework may fail if there are more than three nodes fail in a short time. In that case, a error report will be returned to the seismic data platform.

4. Evaluation and Analysis

All experiments are performed on a cluster called DN10 which contains 25 computing nodes. Each node is equipped with a 128GB DRAM and two 2.6GHz, 10 cores, 64-bit Intel Xeon(R) E5-2669 CPUs. Thus, each node has 20 physical cores in total. The operating system of each node is Red Hat Enterprise Linux Server release 6.6 (Santiago). The parallel file system deployed on DN10 is GPFS which has a total storage space of 512TB. DN10 is configured as a Hadoop cluster in which the first node is NameNode and the rest 24 nodes are DataNodes. Each DataNode has 3 SATA HDDs and the storage space of each HDD is 4TB. The framework is written in C++ and the C library for HDFS we used in the framework is libhdfs3 [19].

The length of the trace header and trace body of the seismic dataset used for tests is 60 and 6000, respectively. The size of each trace is 24240 in bytes. To avoid measuring errors caused by memory caching, we generate 1TB seismic data for each node. For example, if the number of nodes is 24, the size of seismic data is 24TB; And the size will be 4TB if the node number is 4. The seismic data is stored in a set of seismic files and the size of each file is less than 30GB in our system. Each seismic gather contains 10000 traces. To ensure that the traces are distributed among the whole dataset evenly, the trace numbers are produced randomly. The running time for each test job is 30 minutes. During the 30 minutes, the test job will request gather continuously, then the average time and bandwidth of extracting gathers are calculated. For comparison, we generate 24TB seismic data on GPFS, using the traditional approach of extracting gathers, the average bandwidth is 0.457MB/s and the average time cost on extracting a seismic gather is 505.844 seconds.

4.1. Multiple-Thread Scalability

In this experiment, we aims to observe the relationship between the performance of the framework and multiple threads. The task grain is set to 20 and the load balancing mechanism is not enabled in this experiment. The number of threads for each node ranges from 1 to 20. The maximum thread numbers is restricted to 20 since the maximum physical cores of a node is 20 and excessive threads (more than 20) will impose a large system load and interfere other applications running on the same cluster. The results of the multiple-thread scalability is demonstrated in Figure 8. Apparently, with a increasing

number of threads, the time cost on extracting a seismic gather is decreasing. When the number of threads achieves 20, the time cost on reading a gather is reduced to 3.166 seconds and the bandwidth achieves 73.021MB/s. At the mean time, the framework attains a substantial speedup of 158 times comparing with the traditional approach. The performance improvement is much more significant when thread number ranges from 1 to 5 than from 5 to 20 since each node has only 3 HDDs.

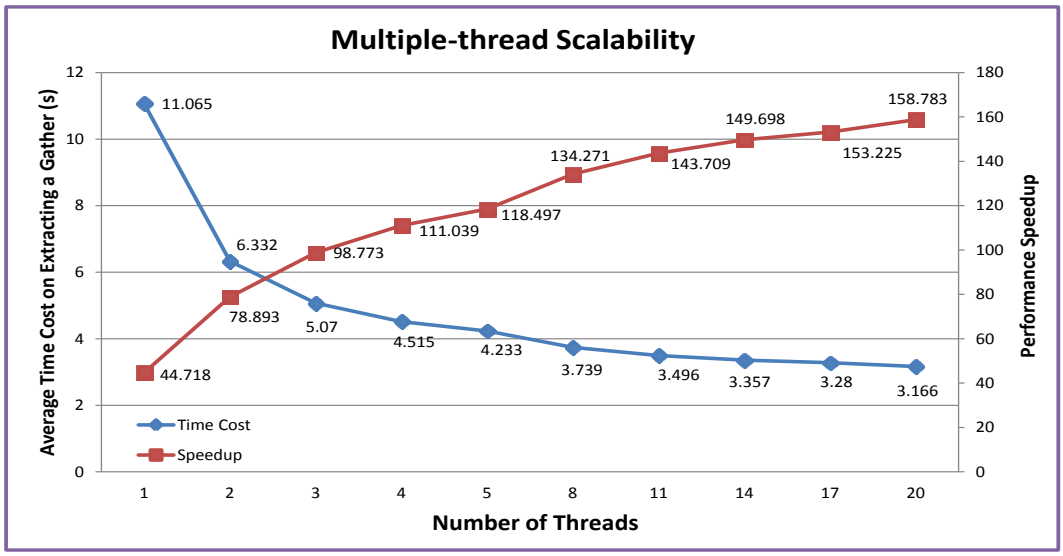


Figure 8. Experimental Results of Multiple-Thread Scalability

4.2. Determination of Task Grain

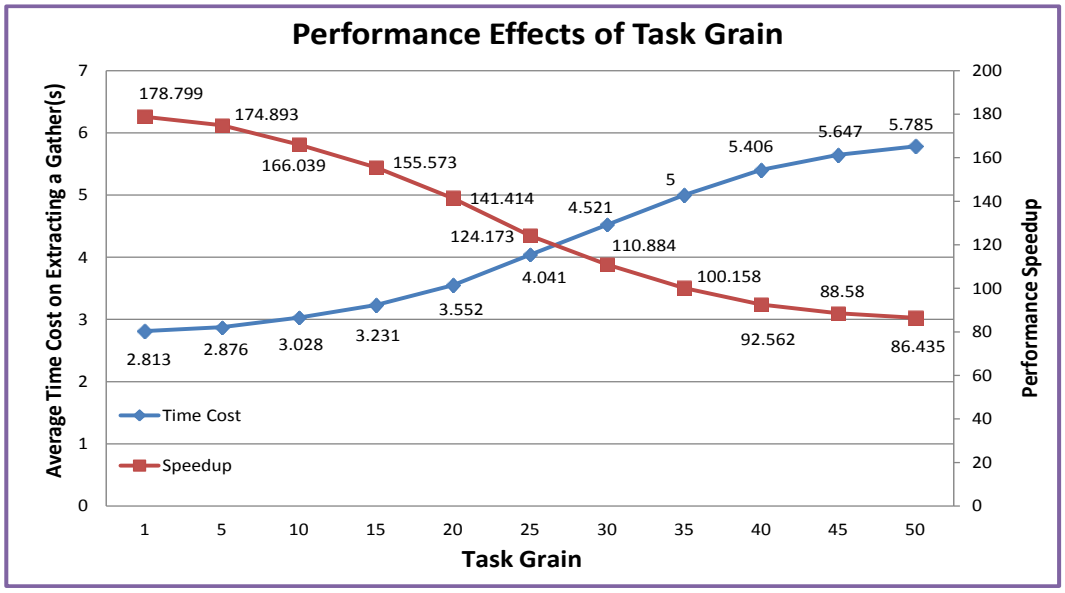


Figure 9. Experimental Results of Performance Effects of Task Grain

This experiment is trying to find the optimal value for the parameter of task grain by observing the relationship between the performance of the framework and the parameter of task grain. The number of threads for each node is set to 20. The task grain ranges from 50 to 1. The results of this experiment is shown in Figure 9. We can observe that the time cost on extracting a seismic gather is decreasing with a decreasing value for task grain. When the task grain is set to 1, the time is reduced to 2.813 seconds and maximum

bandwidth (82.168MB/s) is achieved. The framework attains a speedup of 178 times comparing with the traditional approach. Although master has to suffer more pressures caused by the decreasing task grain which results in more tasks, the pressure can still be sustained. Thus, the aggregated random read bandwidth can be improved further as the load balancing algorithm can perform more precisely.

4.3. Load Balancing

Table 1. Experimental Results of Load Balancing Mechanism

Dataset	Extra I/O Workload	Number of Selected Nodes	Performance without Load Balancing (MB/s)	Performance with Load Balancing (MB/s)	Performance Improvement (%)
Small	No	0	51.713	260.305	403.36
Large	No	0	76.87	82.168	6.89
Large	Yes	1	67.891	81.017	19.33
Large	Yes	2	65.297	82.257	25.97
Large	Yes	3	71.774	82.183	14.50

In this experiment, the performance improvement of load balancing mechanism is observed. The thread number for each node is 20 and the task grain is set to 1. There are two group of seismic data, i.e., a large seismic dataset and a small seismic dataset. The large dataset is 24TB. The small dataset contains 1200000 traces and the size is 27GB. The experimental results of load balancing mechanism is displayed in Table 1. When the load balancing mechanism is enabled, the performance improves 403% and 6% for the small dataset and large dataset, respectively. Since the size of the small dataset is less than the limited size of seismic data file, the whole dataset is stored in only one file. When the file is written to HDFS by one node, HDFS will always write the first block of three duplications into the local node. Thus, the traces are distributed highly uneven. The data is distributed relatively even as there are enough files for the large dataset and these files are written to HDFS by all nodes in parallel, so the improvement for the small dataset is far more significant than the large dataset. The bandwidth of the small seismic dataset can achieve 260.305MB/s because the data is cached by the memory of computing nodes.

Extra I/O workload is introduced on several selected nodes for the large dataset. Specifically, a 512GB file is generated on each HDD of each selected node. For each HDD, there will be a thread which reads data randomly and continuously from the 512GB file. In the case of extra I/O workload, the performance improves 19.33%, 25.97%, 14.50% by the load balancing mechanism when the number of selected nodes is 1, 2, 3 as shown by the third row to the fifth row in Table 1.

4.4. Scalability on the Cluster

The goal of this experiment is to observe the scalability of the framework when using different number of nodes. The thread number for each node is 20 and task grain is set to 1 in this experiment. The load balancing mechanism is enabled for all the tests. The number of nodes ranges from 4 to 24 with a increment of 4. The experimental results of scalability on cluster DN10 is shown in Figure 10. The ideal performance is calculated proportionally based on the bandwidth observed using 4 nodes. From Figure 10, we can concludes that the framework achieves high scalability because the line of actual performance is very close to the line of ideal performance.

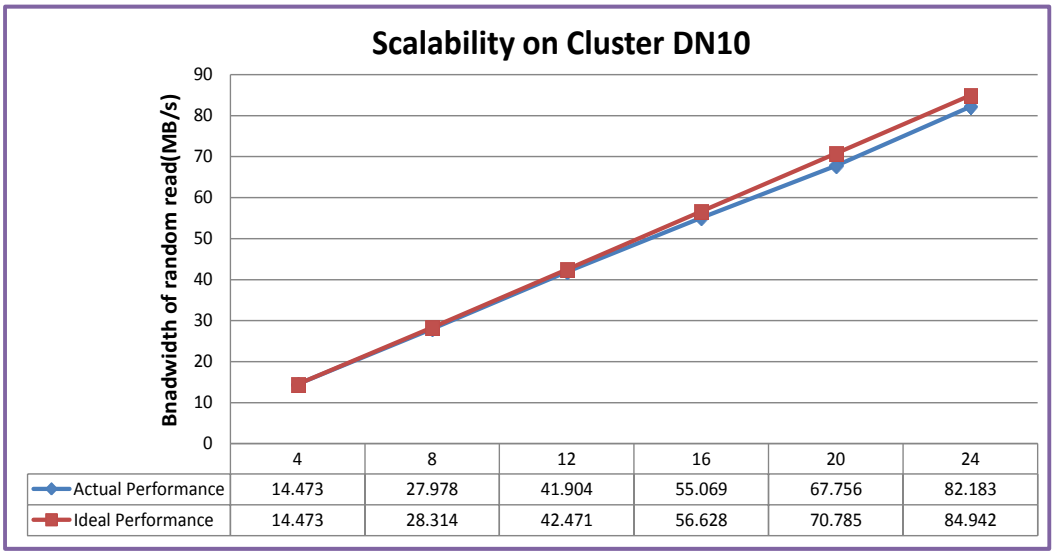


Figure 10. Experimental Results of Scalability on Cluster DN10

4.5. Fault Tolerance

This experiment aims to verify the ability of the framework to tolerate node failures. The thread number for each node is 20 and the task grain is set to 1. The number of DataNodes is 4 and the size of seismic data is 4TB. The node failures are injected when the framework is extracting a gather. Table 2 shows the experimental results of fault tolerance. The number of injected node failures is 1 and 2. For each case, the performance before failures, during failures, and after failures are displayed. After a gather is requested, one or two workers of the framework will be killed to simulate the node failures. We can observe that the framework can acquire integral seismic data for the gather even if node failures happen. The performance during failures drops significantly since it takes several seconds for master to detect the failed nodes. The performance of the framework after failures drops proportionally as expected.

Table 2. Experimental Results of Fault Tolerance

Number of Node Failures	Performance Before Failures (MB/s)	Performance During Failures (MB/s)	Performance After Failures (MB/s)
1	14.473	8.22	10.954
2	14.473	8.154	7.333

5. Conclusion

In this paper, we propose a HDFS-based framework that can extract seismic gathers fast from large scale and high-dimensional seismic data for interactive seismic applications on high performance clusters. Traces are read in parallel by different nodes at cluster level and different threads at node level. The data locality principle guarantees the maximum random read bandwidth and dynamic load balancing mechanism avoids the lagging node becoming the bottleneck of the framework. In summary, the framework we proposed has achieved four goals listed as follows.

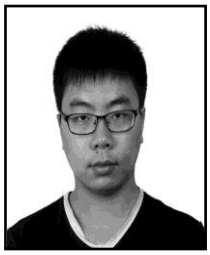
- **High performance.** In a cluster consisted by 25 computing nodes with 3 local disks attached to each node, the framework performs 178 times better than the traditional approach.

- **Load balancing.** The dynamic load balancing mechanism makes the framework achieve 403% better for a small dataset and 25.97% better for a large dataset, respectively.
- **High scalability.** As observed in the experiments, when the number of DataNodes increases, the performance of our framework increases proportionally. Predictably, if more nodes are added, the framework can provide higher random read bandwidth.
- **Fault tolerance.** As the framework needs to offer a sustainable service of extracting gathers for upper layer interactive applications, fault tolerance is a necessity. Our framework can guarantee data integrity of one seismic gather even if multiple node failures happen.

References

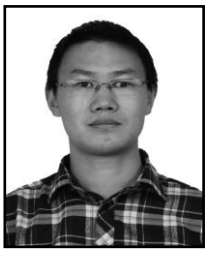
[1] SEG Technical Standards. http://www.seg.org/seg, **(2016)**.
[2] J. Y. Shin, M. Balakrishnan, T. Marian and H. Weatherspoon, "Gecko: contention-oblivious disk arrays for cloud storage", Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST), **(2014)**; San Jose, California.
[3] K. Shvachko, H. Kuang, S. Radia and R. Chansler, "The hadoop distributed file system", Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST), **(2010)**; Lake Tahoe, Nevada, USA.
[4] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer and F. Cappello, "Modeling and tolerating heterogeneous failures in large parallel systems", Proceedings of the 25th International Conference for High Performance Computing, Networking, Storage and Analysis (SC), **(2011)**; Seattle, Washington, USA.
[5] B. Schroeder and G. A. Gibson, "Understanding failures in petascale computers", Journal of Physics: Conference Series, vol. 78, no. 1, **(2007)**, pp. 12-22.
[6] S. Ghemawat, H. Gobioff and S. T. Leung, "The Google file system", ACM SIGOPS operating systems review , vol. 37, no. 5, **(2003)**, pp. 29-43.
[7] T. White, "Hadoop: The definitive guide", O'Reilly Media, Inc., Sebastopol, **(2012)**.
[8] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters", Proceedings the 1st Conference on File and Storage Technologies (FAST), **(2002)**; Monterey, California. R. Ananthanarayanan, K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah and R. Tewari, "Cloud analytics: Do we really need to reinvent the storage stack?", Proceedings of the 1st USENIX Workshop on Hot Topics in Cloud Computing (HotCloud), **(2009)**; San Diego, California, USA.
[9] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters", Communications of the ACM - 50th anniversary issue: 1958 - 2008, vol. 51, no. 1, **(2008)**, pp.107-113.
[10] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang and R. Schmidt, "Apache Hadoop goes realtime at Facebook", Proceedings of the 2011 International Conference on Management of data (SIGMOD), **(2011)**; Athens, Greece.
[11] H. Liao, J. Han and J. Fang, "Multi-dimensional index on hadoop distributed file system", Proceedings of the 5th IEEE International Conference on Networking, Architecture, and Storage (NAS), **(2010)**; Macau SAR, China.
[12] S. Sur, H. Wang, J. Huang, X. Ouyang and D. K. Panda, "Can high-performance interconnects benefit hadoop distributed file system", In Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds (MASVDC), **(2013)**; Atlanta, Georgia.
[13] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony and R. Murthy, "Hive: a warehousing solution over a map-reduce framework", Proceedings of the VLDB Endowment, **(2009)**; Lyon, France.
[14] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley and M. Zaharia, "Spark sql: Relational data processing in spark", Proceedings of the International Conference on Management of Data (SIGMOD), **(2015)**; Melbourne , Australia.
[15] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi and I. Joshi, "Impala: A Modern, Open-Source SQL Engine for Hadoop", Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR), **(2015)**; Asilomar, California , USA.
[16] L. George, "HBase: the definitive guide", O'Reilly Media, Inc., Sebastopol, **(2011)**.
[17] M. Bhandarkar, "MapReduce programming with apache Hadoop", Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS), **(2010)**; Atlanta, Georgia, USA.
[18] libhdfs3. https://github.com/PivotalRD/libhdfs3, **(2016)**.

Authors



Chao Li, born in 1990, is currently working toward the Ph.D. degree in Beihang University. His main research interests include high performance computing, seismic data software platform, and performance optimization of seismic imaging algorithms.

Wen Jiamin, born in 1972, is a professor in GeoPhysical Technique Research Center, BGP, CNPC. His main research interests include high performance computing and geophysical software.



Changhai Zhao, born in 1979, obtained his Ph.D. from Beihang University. His main research interests include high performance computing and seismic imaging.

