ParSA: High-throughput Scientific-data Analysis Framework with Distributed File System

Tao Zhang, Wei Xue
Department of Computer Science and Technology,
Tsinghua University,
Beijing，China

Xiang Zheng Sun
Intel Corporation
Beijing, China

Abstract—**Scientific-data analysis and visualization has become a key component in nowadays large-scale simulations. Due to the rapidly increasing data volume and awkward I/O pattern among high-structured files, known methods/tools cannot scale well and usually lead to poor performance over traditional architectures. In this paper, we propose a new framework --- ParSA for high-throughput and scalable scientific analysis, cooperating with distributed file system. ParSA present the optimization strategies oriented for physical disk to maximize distributed I/O property of distributed file system as well as to maximize overlapping the data reading, processing and transferring during computation. Besides, ParSA provides an easy-use programming framework, with which programmers only need to focus on the implementation of logical operations. We utilize ParSA to accelerate well-known analysis tools for climate models on Hadoop Distributed File System (HDFS). Experimental results demonstrate the high efficiency and scalability of ParSA. (*Abstract*)**

Keywords—Data intensive computing; Scientific data Analysis; Distributed file system

I. INTRODUCTION

In some scientific applications, huge amounts of data are produced. For Example, in a high resolution ocean model(0.1 degrees), 100 years of monthly output is about 48TB with one single 3D variable has 1.3GB size. And the data volume still increases as the scientific results are more precise. Then the scientific-data analysis becomes data-intensive.

Furthermore, almost all of scientific-data are stored in high-structured files. Numerous standard file formats are introduced, like Network Common Data Format (NetCDF) format [1] in climate system model applications.

In traditional scientific-data analysis, large-scale scientific-data are stored in RAID-5/6 or parallel file system. And the analysis methods/tools are always centralized approaches, like NCO [2] and NCL [3] for processing NetCDF files. The scalability is very poor and analysis is very time-consuming.

Inspired by big data solution in Internet Port Data Center (IPDC), numerous frameworks have been developed. MapReduce [4] is a program framework, provides automatic parallel mechanism and build-in fault-tolerance. It would be efficient if the MapReduce is used to data intensive application. However, this solution with MapReduce requires that data first be transformed into a text-based format.

SciHadoop [5] is a Hadoop plugin allowing scientists to specify logical queries over array-based data models. It executes queries as map/reduce programs defined over the logical data model. It shows remarkable improvements for holistic functions of NetCDF data sets for the following optimization goals: reduce total data transfers, reduce remote reads and reduce unnecessary reads. SciHadoop using java language leads to the compatibility problem to the existing climate data analysis tools, which is written by c-shell scripts, NCL and NCO commands.

In this paper, we propose a new framework --- Parallel Scientific-data Analysis (ParSA). We utilize the distributed I/O property with Hadoop Distributed File System (HDFS) [6] to improved data reading throughput. What is more, ParSA optimizes the data layout schedule stored in the distributed file system to overlap the data reading, processing and transferring. Besides, it provides parallel NCO operations, cooperating with HDFS, making it easy to use the efficient tool, without changing a lot for current climate analysis package.

II. HDFS AND SCIENTIFIC-DATA ANALYSIS

In this section, we will present property of distributed file system HDFS and scientific-data analysis. And discuss about the probability of analysis transportation onto HDFS.

A. HDFS

HDFS is an open source project, driven by Google File System (GFS) [7]. As a distributed, scalable and portable file system, HDFS is inherent for large-scale data-intensive process.

In HDFS, there are two types of node --- Namenode and Datanode. Namenode maintains file system tree and metadata for all files or directories stored in HDFS. And Datanodes are where the data are actually stored. Besides, users complete all HDFS operates through Clientnode, as shown in Figure 1.

When we store a file into HDFS, the file will be split into file blocks as the storage unit of HDFS. For achieving fault-tolerance, HDFS stores three replicas, by default, for each file blocks in different datanodes. Therefore, even individual node halts down, all data, which are stored in the halted node, can be accessed from other replicas.

In each Datanode, we can mount several hard disks and Datanode manages those hard disks by itself. By default, Datanode will store each block into the hard disks in a round-robin way.

For example, when store file “File 1” into HDFS via Clinetnode, Namenode will add “File 1” into the file system tree. Then Clientnode begin write the content into HDFS. Once Clientnode detects that current writing size exceeds the block size, 64MB by default, it will ask Namenode for a new block. After receiving the request from Clientnode, Namenode will assign a new block with unique block number. Simultaneously, Namenode need recode mapping-relation between “File 1” and the block number. Since three replicas are used in HDFS for fault tolerance, Namenode will select three Datanodes, form a pipe-line and return to Clinenode. Only all data of one block are successfully written into each Datanode in the pipe-line, this write operation can be successful. In Datanode, each block is stored as a separated file. And Datanode should choose a hard disk for the block. Due to the reason that last block “block k” is store in disk 2, the new one will locate at disk 0 according to the round-robin rule, as shown in Figure 1.

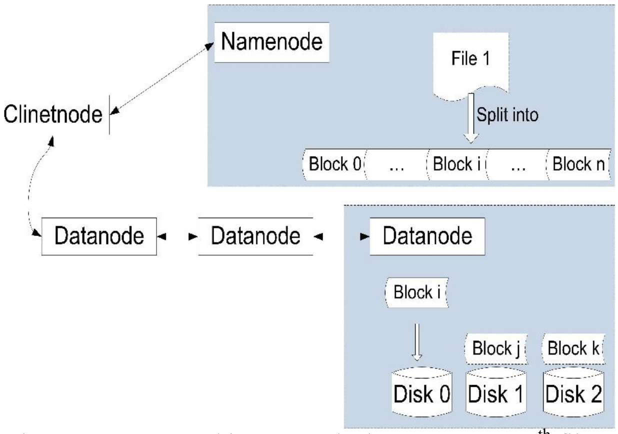


Figure 1. HDFS Architecture. Block ***N*** represents Nth file block of “File 1” on HDFS.

Therefore, we can conclude that files are distributed in two-tiered approach --- among Datanodes in HDFS and among hard disks in one Datanode. When file blocks disperse evenly, the bandwidth of HDFS is the sum up of the bandwidth of all disks theoretically. We can scale out the storage by adding new nodes with more disks.

In order to exploit the distributed I/O property, Mapreduce is introduced as a computation model cooperating with HDFS. In contrast to data migration, Mapreduce can process the data located on local Datanode and reduce network transfer. However, due to the traits of scientific data and the operation pattern, Mapreduce cannot be directly utilized on scientific-data analysis.

B. Scientific-data Analysis

Scientific-data are usually stored in high-structured files and contain logical units. Each logical unit has its own physical meaning. And the size of each logical unit varies a lot in one scientific file. During operation, only several logical units are used in analysis.

In [5], an array data model is given to describe the scientific data. An array query model is defined as the operation pattern.

In climate system model application, the volume of output is usually extremely huge. As mentioned before, the high resolution ocean model has 48TB size data for 100 simulated years, with monthly output. These data are organized by thousands of files with timestamp named in NetCDF format, called history result. Each file contains the same dozens of physical variables. Analysis operations need to process all of the output or part of the history result. These operations include computing average, combining the same variables in different files, producing the remapping file and fast Fourier transform, etc. They require handling all or part of variables in multi-files [8].

C. Problems and Challenges

Problem/Challenge 1: Logical unit size varies a lot from few bytes to gigabytes, which means that one file block may contain hundreds of logical units whereas another file block only store small proportion of one logical unit. Then we cannot use logical unit as the operation unit. Otherwise, the workload of each task will be imbalance. How to define the operation unit considering the storage unit --- file block on HDFS?

Problem/Challenge 2: Analysis may be operated among many files, while usually only data of logical units with same meaning are operated together. If all logical units with same meanings locate on the same Datanode, all data access will proceed locally. However, default block placement policy of HDFS cannot ensure that, which leads huge amount data network transfer. How to schedule tasks among many files to increases the locality and reduce the data network transfer?

Problem/Challenge 3: How to best utilize the disk I/O?

III. PARALLEL SCIENTIFC-DATA ANALYSIS

In order to accelerate the scientific-data analysis and to solve the problems mentioned in previous section, we propose a new framework --- Parallel Scientific-data Analysis (ParSA).

A. Logical Unit Split and Group among File Blocks

No matter how a file is split, all data of the same file block are stored contiguously in a physical location, since file block is the storage unit of HDFS. Thus we can split and group the logical unit according to the file block location to avoid remote data access from other file blocks. When other blocks locate on other Datanodes, data network access should be involved, which will impacts the performance.

The logical unit split and group approach is shown in Figure 2. All logical units located in one block are distributed into one group. If a logical unit spans two or more blocks, this logical unit will split to several parts. And each split part should entirely locate in one file block. For example, LU1 spans three blocks. It will split into three split parts --- LU1 (1), LU1 (2) and LU1 (3). LU1 (1) and LU0 locate in file block 0 and form a logical unit group. LU1 (2) itself is distributed as a group. The remaining LU1 (3), LU2 and LU3 are distributed to the third group. Then we can record the mapping relations between groups of logical unit and file blocks, as shown in Figure 3. Besides, unique identifier is assigned to the split logical unit.

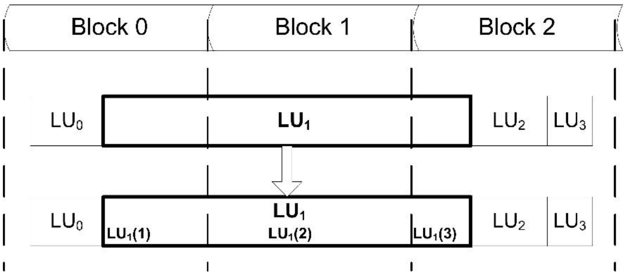


Figure 2. Logical Unit Split. LU$_N$ presents Nth logical unit. LU$_N$ (i) stands for ith split part of LU$_N$

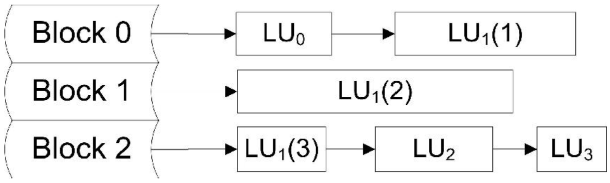


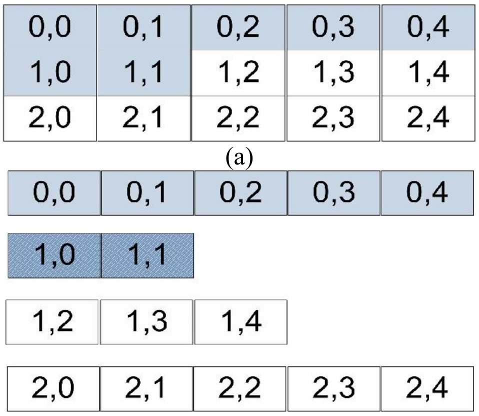
Figure 3. Mapping Relations between Logical Unit Groups and File Blocks

When we use the split logical unit as the operation unit, less or no data need accessing from other blocks.

For some scientific-data, it may not be so simple to directly split the logical unit into pieces. Let’s take splitting the variable in NetCDF as an example.

Variable in NetCDF can be viewed as multi-dimension array. An array section is defined as a slab specified by two vectors --- index vector and count vector [1]. The index vector indicates the start offset in each dimension. And the count vector restricts the number of values along each dimension, in order. The lengths of both index vector and count vector are the numbers of dimensions of the variable. For instance, the variable given in Figure 4 (a) has two-dimension. If we access the whole variable, as a special slab, index vector is (0, 0) and count vector is (3, 5).

In hypothesis, this variable locates on two file blocks when we store this NetCDF onto HDFS. And variable should be split between value (1, 1) and value (1, 2). The variable cannot be split into two pieces, like values from (0, 0) to (1, 1) as a piece. The reason is that we cannot use an index vector and count vector to represent this piece. The number of two rows is not equal, 5 and 2 respectively. In this situation, we should split the variable into four pieces, as Figure 4 (b) shows.



0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

(a)

0,0	0,1	0,2	0,3	0,4

1,0	1,1

1,2	1,3	1,4

2,0	2,1	2,2	2,3	2,4

(b)

Figure 4. Variable Split in NetCDF. (a) Variable to be split. (b) The split result of variable given in (a). The index vectors of four split slab are (0, 0), (1, 0), (1, 2) and (2, 0) respectively. And the counts vectors are (1, 5), (1, 2), (1, 3) and (1, 5) respectively.

After splitting logical units according to file block in HDFS, operations on logical units, grouped in one file block, are constrained within file block. Therefore, we can also view file block as the operation unit. The problem 1 is resolved.

B. Scheduler

When we regard the file block as the operation unit, the next step is to determine which Datanode to process the give file block, which is the task of scheduler.

As we discussion in section 2.3, operations may involve more than one files. Since quite a few operations are only executed on logical units with the same meaning and the size of the logical unit must be equal among different files, we should keep the blocks with the same block number in the same Datanode as many as possible, in principle. Therefore, logical units of larger proportion of files can be processed in the local Datanode, reducing the data network transfer.

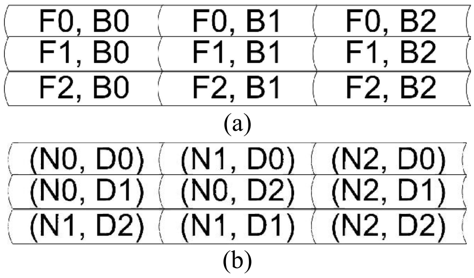
At first, we need gather all information about file block distribution. In this paper, we define the tuple (Datanode number, hard disk number) as the location of a file block. Then we can calculate which Datanode contains the maximum number of given file block.

F0,B0: (N0, D0), (N1, D2), (N5, D1)
F1,B0: (N0, D1), (N2, D0), (N3, D1)
F2,B0: (N1, D2), (N6, D0), (N7, D1)

For the three files listed in Figure 5 (a), each file has three blocks. As discussion in section 2.1, file block has three replicas by default. Assume we get the following block location information for the first file block of all three files (using same abbreviation in Figure 5).

For the nine replicas for B0 of F1, F2 and F3, Datanode N0 covers 2 replicas, N1 contains 2 replicas and other Datanode only contain 1 for each. Datnode N0 and N1 contain the same number of replicas. While we choose Datanode N0 to process the B0 block of F1 and F2, instead of N1 to process B0 of F0 and F2. The reason is that B0 of F0 and F2 locates on the same hard disk, which is more likely to impart the performance (explained in section workflow of ParSA). When we decide which Datanode to process B0 of F2, although the number of replicas in Datanode N1, N6 and N7 is equal, we should select N1. When disk 0 of N0 is broken, we can access the replica of F0, B0 in Datanode N1. And the scheduler result is given in Figure 5 (b).

In this way, we can increase the access locality during operations on many files, problem 2 is resolved.



F0, B0	F0, B1	F0, B2
F1, B0	F1, B1	F1, B2
F2, B0	F2, B1	F2, B2

(a)

(N0, D0)	(N1, D0)	(N2, D0)
(N0, D1)	(N0, D2)	(N2, D1)
(N1, D2)	(N1, D1)	(N2, D2)

(b)

Figure 5. Task Scheduler of ParSA. (a) Three files stored in HDFS. (b) Static task scheduler result of ParSA for the three files in (a). F***N*** presents the Nth file to be processed; B***N*** presents Nth file block on HDFS; (N***i***, D***j***) stands for corresponding file block in (a) locates on jth disk of ith Datanode.

C. *Workflow of ParSA*

Although we utilize the file location to schedule the tasks among Datanodes, I/O bandwidth of disks may not be efficiently utilized intra-Datanode. For example, the file block B0s of F0 and F1 are assigned to N0 in Figure 5. If we implement the operation from logical level at Datanode N0 as following, although F0, B0 and F1, B0 locate on different disks --- D0 and D1 respectively, the two disks are accessed sequentially instead of parallelly.

```
for each logical-unit LU in B0; do
    for each file Fx whose block B0 locates on N0; do
        read LU from Fx, B0
        operate on LU
    done
done
```

We propose a new workflow to efficiently access the data on HDFS and maximize utilizing all disk bandwidth in one Datanode.

In ParSA, in each Datanode there is a MPI process handling all the communications. The MPI process whose rank is 0 is the major process, while other MPI processes are worker processes. And major process gathers all the intermediate result of worker processes. In each Datanode, we use Pthread to implement the detail operations, constituting of three major components: reading-thread, processing-thread, and receiving-thread.

Reading-thread Due to the reason that data processing speed of CPU is much higher compared to disk I/O bandwidth, if more than one thread dedicatedly access one disk, the performance cannot be higher than only one thread. Thus we invoke one thread for each disk. We define this type of thread as reading-thread and use Treadi to present ith reading-thread. Treadi accesses all the file blocks located on ith disk and the disk number is recorded in the task scheduler result as part of location information (shown in Figure 5 (b)). For each block, Treadi reads values of split logical units, shown in Figure 3. Then put the logical unit into a logical unit queue. The logical unit contains values as well as index information, such as file index and the value range. Furthermore, we should mark this logical unit as LOCAL_FLAG, extinguishing with receiving data. The workflow of reading-thread is given below.

```
function reading-thread
inputs:
datanode_number: the Datanode number where the reading-thread locates.
disk_number: the disk number to be accessed
scheduler_result: task scheduler result, given in Figure 5(b).
LU_groups_map: mapping relations between logical unit groups and file blocks in Figure 3.

begin
    for each tuple (node_no, disk_no) in scheduler_result
        if node_no = datanode_number and disk_no = disk_number
            get the file-index and block-index according to the index of tuple, shown in Figure 5(a).

            for each lu in LU_groups_map[block_index]
                read values of lu of file_index file
                store file-index and values into lu
                mark lu as LOCAL_TAG
                insert (lu) into data queue
                notify processing-thread
            end for
        end if
    end for
    send ending-flag to processing-thread
end
```

Processing-thread When reading-threads inert logical units into queue, they will wake processing-thread up. The processing-thread takes in charge of invoking user-defined function ***local_data_process*** to process those logical units. Users only focus on implementing operations on logical unit. Since each Datanode only covers part of entire dataset, when processing-thread completes all logical units with the same meaning in this node, it sends the intermediate results to major process. Therefore, the processing-thread in major process needs to accumulate the intermediate results into the final results. The accumulating operation is defined in user-defined ***recv_data_process*** function.

```
function processing-thread
inputs:
num_tasks: total number of ending-flag to wait
intermediate_results: store the intermediate results

cur_num_tasks = 0
while cur_num_tasks < num_tasks
    wait until receiving notification
    if receive ending-flag
        cur_num_tasks++
    end if

    fetch the logical unit queue
    for each lu in logical unit queue
        get interm_res according to lu identifier from
        intermediate_results

        switch lu's flag
        case LOCAL_FLAG:
            call local_data_process (lu, interm_res)
        case RECV_FLAG
            call local_data_process (lu, interm_res)
        end switch

        if not in major process
            if all logical unit with same identifier are
            processed in this Datanode
                store interm_res into lu
                mark lu as RECV_TAG
                send (lu) to major process
            endif
        end if
    end for
end while
```

Receiving-thread Receiving-thread only locates in major process. After receiving logical units for worker processes, it inserts the unit, with RECV_FLAG flag, into queue, shared with reading-thread. And notify processing-thread to fetch the data.

ParSA is the framework for scientific-data access on HDFS and lets users to implement specific operations. Because ParSA is implemented using MPI, user can utilize performance features of MPI. Except the local_data_process and recv_data_process, users also need provide functions to allocate the memory for final result and intermediate results and serialization of intermediate results.

In order to relieve the network pressure, all data are compressed before sending.

The workflow of ParSA is shown in Figure 6. We use ParSA to analyze the science-data in Figure 2. In Figure 6, major process locates in Datanode 0. And we only present the worker process on Datanode 2. Each reading thread reads all logical units of the file blocks. After Tread0 in Datanode 2 reads the logical LU1 (3) of file F0, it reads LU2 of file F0, inserts the values into queue and notifies processing-thread. The processing-thread processes all the values in queue sequentially. When processing-thread detects that LU1 (3) are all processed in Datanode 2, it will send the intermediate result to major process. When Receiving-thread in major process gets the intermediate result, it also inserts it into the logical unit queue. Because the logical unit queue is the shared resource, all threads should access exclusively.

From the workflow of ParSA, we can conclude that all disks keep being accessed until all located data are accessed. Simultaneously, all data are processed by a stand-alone thread. Therefore, ParSA can efficiently utilize the bandwidth of each disk. The problem 3 is resolved.

ParSA can achieve load-balance based on current data layout information. If the data layout is unbalance in itself, the performance is also impacted. In the next section, we will discuss how to further improve the data layout.

IV. DATA LAYOUT OPTIMIZATION

In section 3.2, the scheduler of ParSA utilizes the location information of file blocks, both Datanode and disk location, to assigns tasks. In principle, we should keep the file blocks with same block number in the same Datanode as many as possible for the operations involving multi-files. More important, we should keep the load balance among Datanodes. While the default file block layout strategy of HDFS cannot guarantees this.

In HDFS, if Clinenode is one of the Datanodes, the Datanode for first replica of each file block will be stored in Clinenode. And Datanodes for other replicas are chosen randomly in the same rack, which is different from the first replica. If the Clinenode is not Datanode, the fisrt replica will be randomly stored in a Datanode.

Although the scheduler can assign tasks based on current file block locations, we can control the file block location according to the principle for task assignment in order to achieve load-balance. Since scheduler use two tired location information --- Datanode level and disk level, we optimize the layout also in these two tired.

A. Data Layout inter-Datanodes

When store scientific-data, we should put the tightly coupled data in one Datanode to avoid data network transfer and achieve locally process. Simultaneously, we also need distribute the file block equally among Datanodes.

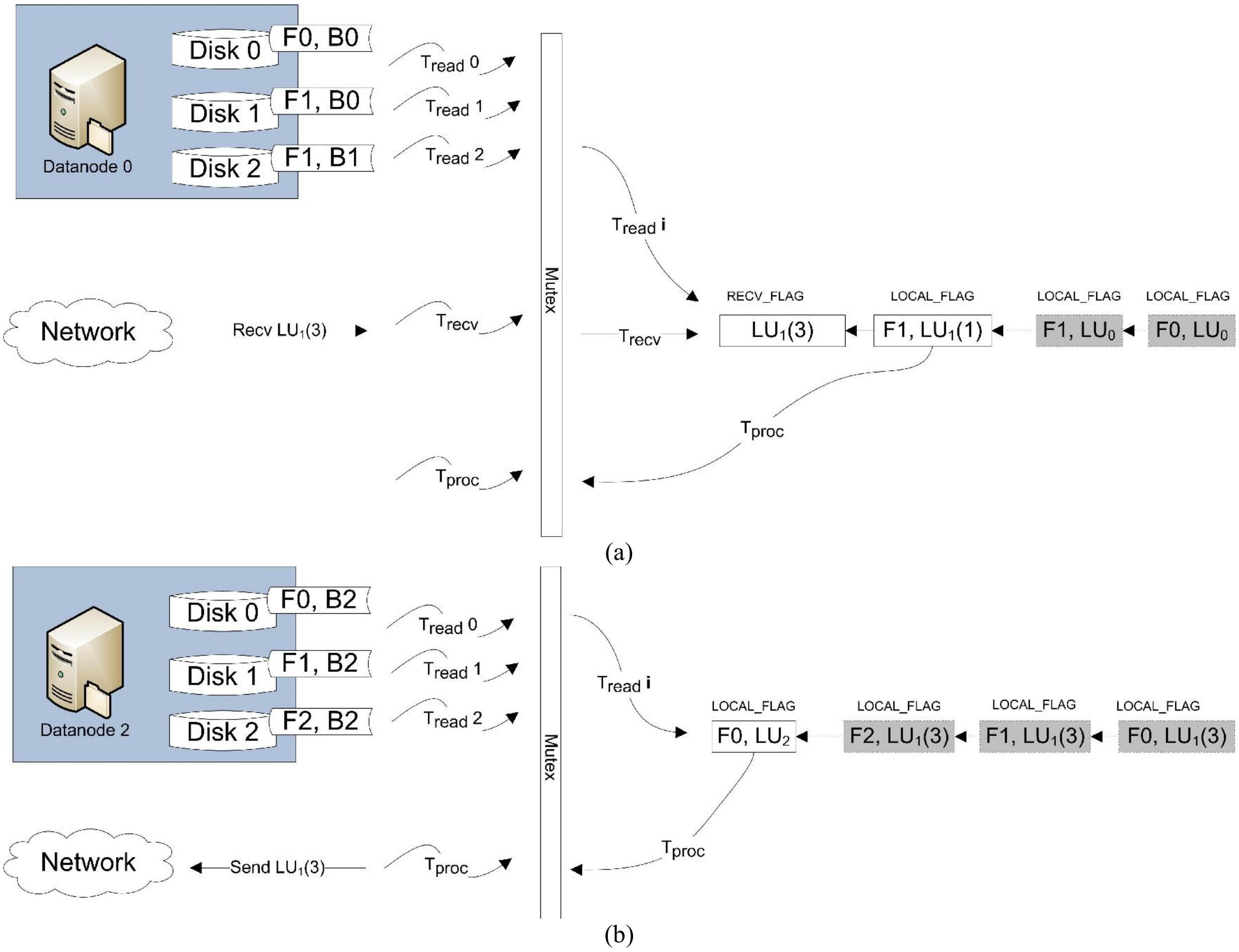


Figure 6. Workflow of ParSA. (a) Major process; (b) Worker process. The logical units with gray background stand for already-processed logical unit.

Since operation patterns vary among different scientific-data analysis, we modifies the layout strategy and expose an interface for users. Then user can determine where the block should locate, but only for the first replica. By default, HDFS randomly choose a Datanode for the second replica. We increase the probability for these Datanode, on which few file blocks locate on. And the fewer file blocks a Datanode has, the more probably it will be chosen. As for the third replica, it complies with the HDFS default strategy.

B. Data Layout intra-Datanode

Even file blocks are distributed evenly among Datanodes, the throughput of hadoop cluster may not reach the best, as discussed in section 3.3. The work flow of ParSA invokes a thread for a disk to access all the data needed in the operation. If all the useful data locate on one disk, it will waste the bandwidth of other disks. Therefore we also need distribute data evenly among disks intra-Datanode. Although HDFS default disk (volume in HDFS) selection is a round-robin way and each disk contains the same number of file blocks, it may not be suitable for scientific-data analysis, because each disk may not contains the same number of file block with the same meanings. When Datanode is chosen randomly for storing file block, the file block number is also random for a Datanoded. In scientific-data analysis, we usually focus on some specific logical units. The performance will drop dramatically if all logical units locate on a small amount of disks.

Our solution is also using round-robin method for file blocks with same meanings, not for all received file blocks, as shown in **Figure 7**. If there is no file block with same meaning stored in Datanode before, the round-robin number of HDFS will be used.

We should note that all the data layout optimizations, both inter-Datanode and intra-Datanode, are hints for HDFS and used as an auxiliary strategy. If few Datanodes fail to work, ParSA can efficiently process analysis based on current data layout.

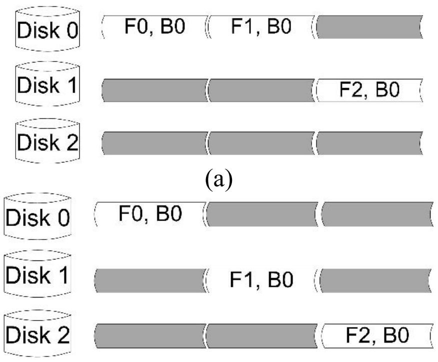


Figure 7. Round-robin Method for File Block with Same Meaning. (a) Block layout unbalance for B0. There are two B0s locate on Disk 0 while Disk 2 contains no B0. (b) Using the improved strategy. B0s are stored in a round-robin way and locate evenly among disks.

V. DATA LAYOUT OPTIMIZATION

In order to demonstrate the efficiency of ParSA, we use climate data analysis of a typical Climate System Model (CSM) simulation application.

The major operation is calculating average using ncea command of NCO package, which is the basic operation for other analysis methods.

A. Experimental Setting

There are two small hadoop clusters with different configuration, as shown in Table 1 and Table 2. And cluster 2 is mainly used for analyzing the scalability along with different disks.

Table 1. System Configuration of hadoop cluster 1

# nodes	16
# disk per node	2 * 1TB
OS	RedHat 6.2
MPI	Intel® MPI Library 3.2
Compiler	Intel® Composer XE 2011
NetWork	***GigE Ethernet***
CPU	Intel® Xeon® Processor E5620 @2.4GHz
Memory	48GB
Hadoop	Apache Hadoop 1.0.3

Table 2. System Configuration of hadoop cluster 2

# nodes	6
# disk per node	5 * 500GB
OS	CentOS 6.2
MPI	Intel® MPI Library 4.1
Compiler	Intel® Composer XE 2011
NetWork	***Dual GigE Ethernet***
CPU	Intel® Xeon® Processor E5645 @2.4GHz
Memory	48GB
Hadoop	Apache Hadoop 1.0.3

To evaluate the scalability of ParSA, we try to increase number of disks per node and increasing number of Datanodes. Since number of nodes of cluster 1 is larger than cluster2, we use cluster 1 for increasing number of Datanodes, while each node has more disks in cluster 2, we use cluster 2 for increasing number of disks per node.

As we discussed, ParSA is designed to efficiently access scientific-data and let users define the specific operations. We use the metric throughput to evaluate the performance of ParSA, whose value presents how fast ParSA to process a given volume of scientific data.

$$\text{Throughput} = \frac{\textit{Scientific data volume}}{\textit{processing time}}$$

Obviously, throughputs can be comparison between two different applications, for the operations may be totally different.

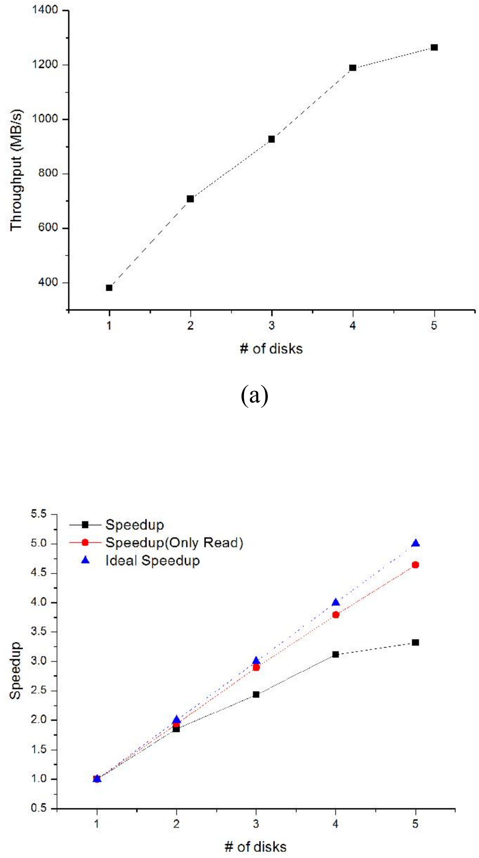
We should note the processing time is total consuming time including data access. The operation in evaluation is calculating the average. In this operation, parts of this program are in a serial manner, such as writing the result file to disk.

In order for a reference for other operations, we also give the data reading time. We only remove the user-defined operations, and the logical units are accessed in original way.

B. Scalability Along with Increasing Number of Disks per Datanode

We use all the six Datanodes of the cluster 2. Every time, we use one more disk to evaluate the performance variance.

The evaluation results are given in **Figure 8**. The throughput increases very fast at the beginning. When number of disks becomes five, the speedup becomes very little. When we evaluate the data access time, we find the scalability can be kept for 5 disks. According to Amdahl's law, when the time parallel part becomes less and the proportion of sequential part becomes larger as well as the performance waste for parallelism, the speedup will increase less.



(a)

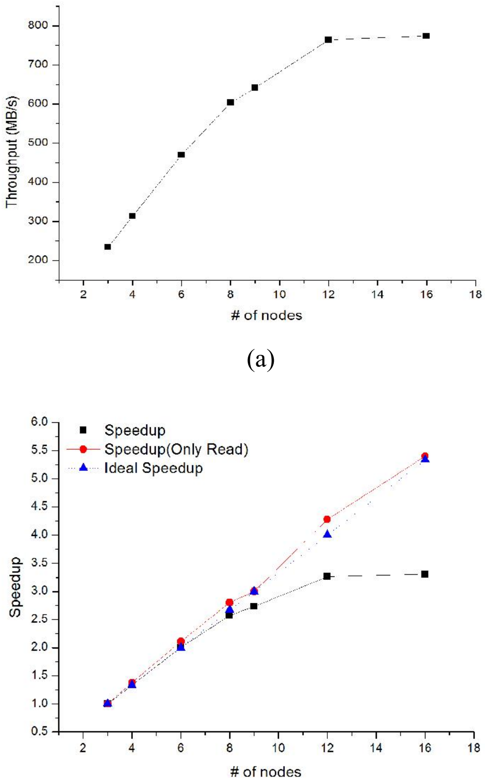
(b)

Figure 8. Scalability of ParSA along with increasing number of disks per Datanode. (a) Throughput; (b) Speedup.

C. *Scalability Along with Increasing Number of Datanodes*

In this evaluation, we use all the two disks per node. The throughput varying curve is shown in 错误!未找到引用源。. The throughput increases almost linearly before using 9 nodes. When number of nodes keeps increasing, not only proportion of sequential part increases, the network pressure becomes very heavy. As the design of ParSA, data processing is overlapped with data accessing. After data are processed, processing-thread sends the immediate result to major process. From the evaluation result, the time no-overlapped occupies almost 38.89% of local processing time on 16 nodes. In contrast, the proportions are 0.02% and 7.96% when number of nodes is 9 and 12 respectively. The reason is that each node only covers parts of the final result, and each node should send the local intermediate results to major process. Moreover, the data volume should be sent in calculating average value is very large. The intermediate result not only contains the value of variable, but also 8-bytes attribute to be sent with each value.

If we only take the access time into account, the speedup is almost linearly, for each node only mounts two disks, referring to **Figure 8** (b). The reason why speedup is higher than ideal speedup is that although we use identical disks for all nodes, while when we evaluate the disk bandwidth for each node, the performance varies about xx%.



(a)

(b)

Figure 9. Scalability of ParSA along with increasing number of Datanodes. (a) Throughput; (b) Speedup.

D. *Throughput Comparison with Different Task Assignment strategies*

ParSA splits and groups the logical unit to make the file block as both storage unit and operation unit. And the scheduler distributes the tasks using file block location and considering the operations among different files. Intra-Datanode, the disk oriented workflow maximizes the utilization of all disks.

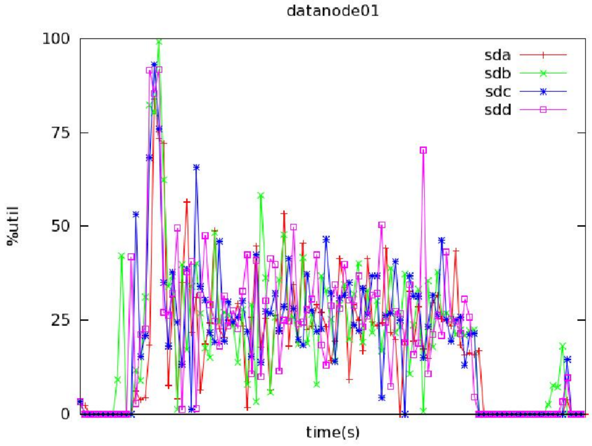
In our development, we also try other task assignment strategies among threads intra-node, including the following two strategies.

1. Strategy of NCO. This strategy directly uses logical unit as the operation unit and distributes logical units among different Datanodes. Since the size of logical unit varies among a lot, threads may access the few disks for processing large amount of small size of logical units. The utilization of disk bandwidth may be poor. In our evaluation, the size of 116 variables in NEtCDF varies from 8 bytes to 40 MB and file size is 309 MB.

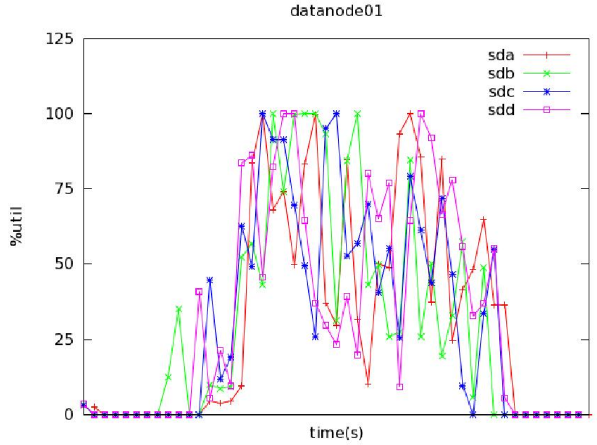
2. Improved strategy of NCO. In the improved strategy, we also use logical unit as the operation unit, but when we assign tasks considering the block location of logical unit. All logical units locate in the file block location are viewed as an operation unit. While we don't split the logical unit when it locates on more than one file block. In this situation, when threads access logical unit from other file blocks, many multiple threads may access the same disk, which will impact the bandwidth utilization. Otherwise, the disks can be accessed parallelly.

In Figure 10, we present the comparison result of disk utilization for the three strategies to calculate the average. We can find bandwidth of each disks can be efficiently utilized.

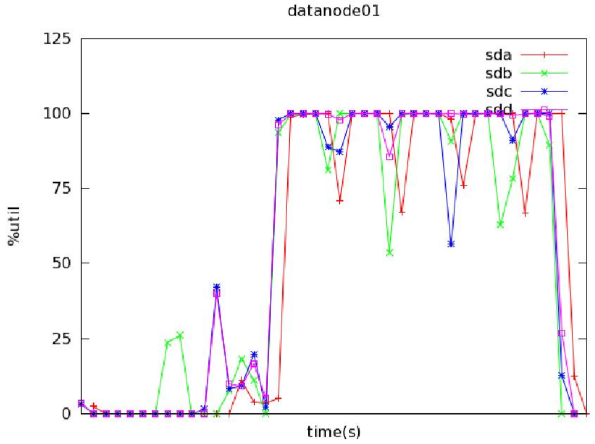
When use throughput to evaluate the speedup the comparison, ParSA outperforms improved and original NCO strategy by a factor of 1.14X and 3.34X respectively.



(a)



(b)



(c)

Figure 10. Comparison of Disk-utilization with Different Task Assignment Strategies. (a) Strategy of NCO. (b) Improved strategy of NCO. (c) Strategy of ParSA

VI. CONCLUSION AND FUTURE WORK

A high-throughput and scalable scientific framework combined with HDFS distributed file system and NCO analysis tool has been designed, implemented and evaluated. Maximize distributed I/O performance has been gained to overlap the data reading, processing and transferring. Instead of Hadoop framework, we use multi-process with MPI to parallel the NCO basic operations, without leading to compatible problem. In future work, we plan on testing the framework on some complex climate data analysis, such as FFT, principal component analysis and Wavelet transform. We also intent to reduce the bottleneck of root process of this framework, resulting from data transferring. Also, the scheme of data compress and decompress will be added.