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

Tao Zhang

Tsinghua

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 serial methods/tools cannot scale well and usually lead to poor performance over traditional architectures. In this paper, we propose a new framework: ParSA (parallel scientific-data analysis) for high-throughput and scalable scientific analysis, 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 the similar interfaces as the NCO (netCDF Operator), which is used in most of climate data diagnostic package, making it easy to port this framework. We utilize ParSA to accelerate well-known analysis methods for climate models on Hadoop Distributed File System (HDFS). Experimental results demonstrate the high efficiency and scalability of ParSA.

Keywords:
Data intensive, Scientific data analysis, Distributed file system

1. Introduction

In most of modern scitific applications, huge amounts of data are produced. Large-scale simulations, such as climate modeling, high-energy physics simulation and genome mapping, generate hundreds of terabytes data volumes(Tevfik, 2009; Hey, 2003). Additionally, it still increases as the high resolution model developping. As a consequence, analysis of scientific-data is data-intensive.

In fact, almost all of scientific-data are stored in high-structured files, some of which provide parallel I/O interface, such as Network Common Data Format version 4(NetCDF4), Hierarchical Data Format 5(HDF5) (hdfs) and ADIOS BP data format (bp) (bp), and some of which only support serial I/O interface, like Network Common Data Format version 3(NetCDF3). All of these are self-describing, machine-independent data format.

In scientific-data analysis, large-scale scientific-data are stored in RAID-5/6 or parallel file system. Yet the analysis methods/ tools are always centralized approaches, such as NCO and NCL, which are the most used in climate applications for prossing NetCDF files, leading to very poor scalability and time-consuming performance.

Inspired by big data solution in Internet Port Data Center (IPDC), numerous frameworks with distributed strategy have been developed. MapReduce is a program framework for processing and generating large data sets, provideing automatic parallel mechanism and build-in fault-tolerance on a cluster. However, this solution with MapReduce requires that data first be transformed into a text-based format. SciHadoop 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. Nevertheless, 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. The SWAMP project [9] has provided the parallel NCO operations, but the reading performance is still bottleneck.

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) 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.

2. HDFS and scientific-data analysis

In this section, we will present the property of distributed file system HDFS, relicas and scheduler, which can be taken advantage of to optimize distributed I/O performance. We also present the character of scientific-data analysis, and discuss about the probability of analysis transportaion onto HDFS.

2.1. HDFS

HDFS is an open source project, driven by Google File System (GFS). 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. When a file are stored into HDFS, it 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. All of I/O operates can be mantipulaed through Clientnode.

Each Datanode can mount several hard disks and it manages these hard disks by itself. By default, Datanode will store each block into the hard disks in a round-robin way. For example, when a file "a" will be stored into HDFS via Clientnode, Namenode will add "a" into the file system tree. Then Clientnode begin writing 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 with unique block number. Simultaneously, Namenode need recode mapping relation between "a" and block numbers. Since three replicas are used in HDFS for fault tolerance, Namenode will select three Datanodes to store a block in the file "a". In Datanode, HDFS should choose a hard disk to each relica of a block. As shown in Figure 1, the last block "k" is stored in disk "2", then the new one "j" will locate at disk "0" according to the round-robin rule.

Blocks of files are distributed in the two-tiered storage achitechture, Datanode and Datanode's disk. It will make full use of the collective bandwidth of HDFS if each relca of blocks is appropriately choosen and scheduled to reduce remoting reading among Datanodes.

MapReduce is usually introduced as a computaion model cooperating with HDFS. It can process the data with good locality. However, due to the traits of scientific data and the operation pattern, MapReduce cannot be directly utilized on scientific-data analysis with perfect performance.

2.2. Scientific-data Analysis

Scientific-data are usually stored in high-structured files, a kinds of spectial binary format, which can not be read directly by MapReduce. Each file contain multi logical units, and each unit are corresponding to its own phsical meaning. The size of each logical varies a lot in one file and only same logical units can be manipulated in most of analysis operations

In this paper, climte model data is used, the total size of which is extremely huge. The high resolution ocean model, a sub-component model in climate model, has 48TB 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 the whole or part of the history results. These operations include computing average, combining the same variables in different files, producing the remapping file and fast Fourier transform (FFT), etc. They require to handle whole or part of variables in multi-files.

2.3. Problems and Chanllenges

Problem/Challenge 1: How to define the operation unit considering the storage unit Logical unit size varies a lot from a few bytes to gigabytes. Then it can not be used as the operation unit. Otherwise, the workflow of each task will be imbalance.

Problem/Challenge 2: How to schedule tasks among many files to increases the locality and reduce the data network transfer? 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, as the above methtioned.

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

3. Parallel scientific-data analysis

In order to solve the problems mentioned in previous section to take full use of distributed I/O performance, we propose a new framework - Parallel Scientific-data Analysis (ParSA).

3.1. Logical Unit Split and Group among File Blocks

No matter how a file is split, all data in one block are stored contiguously in a physical location, since file block is the storage unit of HDFS. Thus the block of HDFS can be used as the basic operation unit to keep continuity in disk and improve I/O performance by proper scheduling to reduce network read. In a block, we can group the small size of logical units or split the big size one.

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. For example, LU1 spans three blocks. It will split into three split parts - LU1 (1), LU1 (2) and LU1 (3), unique identifier assigned to each part. Then LU1 (1) and LU0 locate in file block 0 and form a operation unit group. LU1 (2) itself is distributed as a group. The remaining LU1 (3), LU2 and LU3 are distributed to the third group. The mapping relationship between groups and file blocks should be records, as shown in Figure 3.

Variables in NetCDF, used in climate data, can be viewed as multi-dimension array. An array section is defined as a contiguous rectangular block specified by two vectors - index vector and count vector. The index vector indicates the start offset of the element in the corner closest to the origin. The count vector gives the lengths of the edges along each of the variable's dimension.

For instance, the variable given in Figure 4 (a) is a two-dimension slab. If we access the whole variable, index vector is (0, 0) and count vector is (3, 5). However, if this variable locates on two file blocks stored with HDFS, and variable are split between value (1, 1) and value (1, 2), it cannot be split into these two pieces. The reason is that each of part is not a rectangular shape, which can not use an index vector and count vector to represent. In this situation, we should split the variable into four pieces, as Figure 4(b) shows.

3.2. Scheduler

As discussed in section 2.3, quite a few operations are only executed on the same logical units among multi-files. In principle, blocks containing these units should be scheduled in one Datanode as many as possible to increase the acess locality.

At first, information about file distribution should be gathered. As discussion in section 2.1, file block has three replicas by default. In this paper, we define the tuple (Ni, Dj) as the location of a file block, which means Datanode number and hard disk number, respectively. Assume we get the following block location information for the first file block of all three files.

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. Then Datnode N0 and N1 contain the same number of replicas. Therefore, Datanode N0 is chosen 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 in N1 Datanode, which is more likely to impact the performance (explained in section 3.3). Although N1, N6 and N7 all have the B0 relica of F2, N1 is selected. Because when disk 0 of N0 is broken, the relica B0 of F0 can be accessed in N1. Accroding to this principle, other blocks can be scheduled as the Figure 5(b) shown.

3.3. Workflow of ParSA

Although we utilize the file block location in HDFS to schedule which block relica are selected 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. Even though B0 of F0 and B0 of F1 locate on different disks - D0 and D1 respectively, the two disks are accessed sequentially instead of parallelly, if the operations are implement from logical level at Datanode N0 as following:

Algorithm 1 Sequential operation on disk in the same Datanode

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

In this paper, we propose a new workflow to efficiently access the data on HDFS and maximize utilizing all disk bandwidth in one Datanode.

In ParSA, the framework of hybird of MPI and pthread is used. MPI is responsible for parallel and communication among Datanodes. There is one MPI process in each Datanode. Pthead is responsible to implement the detail operations within each Datanode, constituting of three major components: reading-thread, processing-thread and receiving-thread.

Reading-thread If more than one thread dedicatedly access one disk, it cannot get a higher performance than only one thread. Thus we invoke one thread for each disk. This is defined as reading-thread and use $Tread_i$ to present the i_{th} reading-thread. For each block, $Tread_i$ reads the queue of logical units, shown in Figure 3. After reading the logical unit, it is marked as LOCAL_FLAG, distinguishing with receiving data. The workflow of reading-thread is given below:

Algorithm 2 reading-thread

```
inputs:
datanode_number:the Datanode 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.

for each tuple (node_no, disk_no) in scheduler_result do
  if node_no = datanode_number and disk_no = disk_number then
    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] do
      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
```

Processing-thread After reading-threads read logical units into queue, the processing-thread will be waked up. The processing- thread takes in charge of calling user-defined function local_data_process to process these logical units. Users only focus on implementing operations on logical unit. Since each Datanode only covers part of entire dataset, the major process should reduce the intermediate results, computed with each process in each Datanode. This operation is another user-defined recv_data_process function.

Receiving-thread Receiving-thread is a component in major process. It receives logical inits from worker processes, and puts the logicl units into the receiving queue. Then the processing-thread is notified to fetch these data.

The workflow of ParSA is shown in Figure 6. Major process locates in Datanode 0, and a worker process is presented as Datanode2. Each reading thread reads the logical units which is assigned to this thread. After $Tread_0$ in Datanode 2 reading the logical LU1 (3) of F0, it reads LU2 of F0, inserts the values into queue and notifies processing-thread to wake up. 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 results to the major process. Once Receiving-thread in major process gets the intermediate results, it inserts the results into the logical unit queue. Since the logical unit queue is the shared resource, all threads must access exclusively.

ParSA is the framework for scientific-data access on HDFS. Users can only put their focus on logical operations. ParSA is implemented using MPI and Pthread, providing more flexible configration with regard to performance than MapReduce. ParSA can achieve load-balance based on current block relicas layout information. However, if the layout itself is unbalance, the performance is also impacted. In the next section, how to further improve the performance by optimizing layout will be discussed.

4. Block layout optimization

Algorithm 3 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 do
  wait until receiving notification
  if receive ending-flag then
    cur_num_tasks++
    fetch the logical unit queue
    for each lu in logical unit queue do
      get interm_res according to lu identifier from intermediate_results
      if lu's flag == LOCAL_FLAG then
        call local_data_process (lu, interm_res)
      end if
      if lu's flag == RECV_FLAG then
        call recv_data_process (lu, interm_res)
      end if
      if is not major process then
        if all logical unit are processed in this Datanode then
          store interm_res into lu
          mark lu as RECV_TAG
          send (lu) to major process
        end if
      end if
    end for
  end if
end while
```