Research on High-performance and Scalable Data Access in Parallel Big Data Computing

Jiangling Yin
University of Central Florida

RESEARCH ON HIGH-PERFORMANCE AND SCALABLE DATA ACCESS IN PARALLEL BIG DATA COMPUTING

by

JIANGLING YIN
M.S. Software Engineering, University of Macau, 2011

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2015

Major Professor: Jun Wang

ABSTRACT

To facilitate big data processing, many dedicated data-intensive storage systems such as Google File System(GFS) [74], Hadoop Distributed File System(HDFS) [20] and Quantcast File System(QFS) [70] have been developed. Currently, the Hadoop Distributed File System(HDFS) [20] is the state-of-art and most popular open-source distributed file system for big data processing [32]. It is widely deployed as the bedrock for many big data processing systems/frameworks, such as the script-based pig system [68], MPI-based parallel programs [40, 6], graph processing systems and scala/java-based Spark frameworks [97]. These systems/applications employ parallel processes/executors to speed up data processing within scale-out clusters.

Job or task schedulers in parallel big data applications such as mpiBLAST [56] and ParaView can maximize the usage of computing resources such as memory and CPU by tracking resource consumption/availability for task assignment. However, since these schedulers do not take the distributed I/O resources and global data distribution into consideration, the data requests from parallel processes/executors in big data processing will unfortunately be served in an imbalanced fashion on the distributed storage servers. These imbalanced access patterns among storage nodes are caused because a). unlike conventional parallel file system using striping policies to evenly distribute data among storage nodes, data-intensive file systems such as HDFS store each data unit, referred to as chunk or block file, with several copies based on a relative random policy, which can result in an uneven data distribution among storage nodes; b). based on the data retrieval policy in HDFS, the more data a storage node contains, the higher the probability that the storage node could be selected to serve the data. Therefore, on the nodes serving multiple chunk files, the data requests from different processes/executors will compete for shared resources such as *hard disk head* and *network bandwidth*. Because of this, the makespan of the entire program could be significantly prolonged and the overall I/O performance will degrade.

The first part of my dissertation seeks to address aspects of these problems by creating an I/O middleware system and designing matching-based algorithms to optimize data access in parallel big data processing. To address the problem of remote data movement, we develop an I/O middleware system, called SLAM [90, 91], which allows MPI-based analysis and visualization programs to benefit from locality read, i.e, each MPI process can access its required data from a local or nearby storage node. This can greatly improve the execution performance by reducing the amount of data movement over network. Furthermore, to address the problem of imbalanced data access, we propose a method called Opass [45], which models the data read requests that are issued by parallel applications to cluster nodes as a graph data structure where edges weights encode the demands of load capacity. We then employ matching-based algorithms to map processes to data to achieve data access in a balanced fashion. The final part of my dissertation focuses on optimizing sub-dataset analyses in parallel big data processing. Our proposed methods can benefit different analysis applications with various computational requirements and the experiments on different cluster testbeds show their applicability and scalability.

I dedicate this to everyone that helped in my development as a PhD student.

ACKNOWLEDGMENTS

First of all I would like express the deepest gratitudes towards my advisor, Dr Jun Wang. His persistent guidance, encouragement, and enthusiasm in research always inspired me. I would also want to thank my dissertation committee members, Dr. Yier Jin, Dr. Mingjie Lin, Dr. Mingjie Lin and Dr.Chung-Ching Wang, for spending their time to view the manuscript and provide valuable comments. My gratitude also goes to my friends and family who were always supportive and stood by me through the years.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

The advances in sensing, networking and storage technologies have led to the generation and collection of data at extremely high rates and volumes. For instance, the collective amount of genomic information is rapidly expanding from terabytes to petabytes [4] and doubles every 12 months [18, 80, 69]. Petascale simulations in cosmology [41] and climate(UV-CDAT) [86] compute at resolutions ranging into the billions of cells and create terabytes or even exabytes of data for visualization and analysis. Also, large corporations, such as Google, Amazon and Facebook produce and collect terabytes of data with respect to click stream or logs in only a few hours [92, 58].

These large-scale scientific and social data require applications to spend more time than ever on analyzing and visualizing them. Parallel computing techniques [49, 11] speed up the performance of analysis applications by exploiting the inherent parallelism in data mining/rendering algorithms [23]. Commonly used parallel strategies [23] in data analysis include *independent parallelism*, *task parallelism*, and *single program, multiple data (SPMD) parallelism*, which allow a set of processes to execute in parallel algorithms on partitions of the dataset. In this dissertation, we refer to each operator on data partitions as *a data processing task* and *parallel data analysis* are referred to as the set of parallel processes running simultaneously to perform the tasks.

Conventionally, in parallel big data computing, multiple processes running on different compute nodes share a global parallel file system. Once a data processing task is scheduled to a process, the data will be transferred from the shared file system to the compute processes. Parallel file systems such as PVFS [73] stripe a dataset into many equal pieces and spread them across the storage as shown in Figure 1.1. While such an equal-stripping strategy can allow the storage nodes to serve data requests in a relatively balanced fashion, the process of decoupling parallel storage with computation involves a great deal of data movement over the network. Therefore, the large amounts of data movement over the shared network could incur an extra overhead during parallel execution in today's big data era, especially for the iterative data analysis, which involves moving data from storage to processes repeatedly.

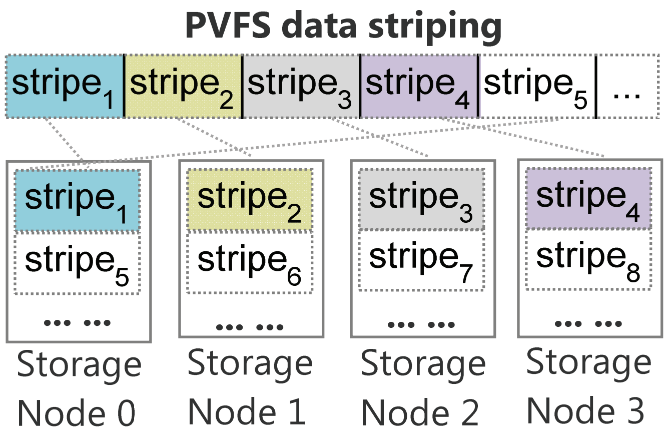


Figure 1.1: File-*striping* data layout in parallel file systems (PVFS).

Distributed file systems [37, 21], such as GFS, HDFS, QFS or Ceph, could be directly deployed on the disks of cluster nodes [38] to reduce data movement. When storing a dataset, distributed file systems will usually divide the data into smaller *chunk/block files* and randomly distribute them with several identical copies (for the sake of reliability). When retrieving data from HDFS, a client process will first attempt to read the data from the disk that it is running on, referred to as *data locality access*. If the required data is not on the local disk, the process will then read from another node that contains the required data. Such a architecture of co-locating computation & stoarge exhibits the benefit of local data access in big data processing.

Job or task schedulers in parallel big data applications such as mpiBLAST [56] and ParaView can maximize the usage of computing resources such as memory and CPU by tracking resource consumption/availability for task assignment. However, since these schedulers do not take the distributed I/O resources and global data distribution into consideration, the data requests from parallel processes/executors in big data processing will unfortunately be served in an imbalanced fashion on the distributed storage servers. These imbalanced access patterns among storage nodes are caused because a). unlike conventional parallel file system using striping policies to evenly distribute data among storage nodes, data-intensive file systems such as HDFS store each data unit, with several copies based on a relative random policy, which can result in an uneven data distribution among storage nodes; b). based on the data retrieval policy in HDFS, the more data a storage node contains, the higher the probability that the storage node could be selected to serve the data. Therefore, on the nodes serving multiple chunk files, the data requests from different processes/executors will compete for shared resources such as *hard disk head* and *network bandwidth*. Because of this, the makespan of the entire program could be significantly prolonged and the overall I/O performance will degrade.

In general, parallel data analyses impose several challenges on today's big data processing system and demands additional functionality as discussed below.

- Typical parallel applications such as mpiBLAST run on a HPC system or a cluster with many parallel processes, which execute the same computational algorithm but process different portions of the datasets. The parallel programming model for these applications is the MPI programming model, in which the shared dataset is stored in a network-accessible storage system like NFS, PVFS [73], or Lustre [75], and transferred to a parallel MPI process during execution. This programming model is well-known as the compute-centric model. Thus, the fundamental challenge of running parallel application on HDFS includes the implementation of the co-located compute and storage properties of MPI-based programs, which usually do *not* take the physical location of data into account during task assignment; and the incompatibility of conventional I/O interfaces, such as *MPI_File_read()*, and HDFS I/O, such as *hdfsRead()*.

- Data requests from script-based or MPI-based applications such as Paraview [12] usually assign data processing tasks to parallel processes during initialization. These processes can simultaneously issue a large number of data read requests to file systems [78, 23, 98]. With the data stored in HDFS, processes will read their data from their local disk if the required data is on the corresponding disk, or from another remote nodes that contain the required data. However, the data distribution/accesses strategies in HDFS could cause parallel data requests to be served in an imbalanced fashion and thus degrade the I/O performance. In general, the main challenge associated with imbalance issues is finding an assignment of processes to tasks, such that the maximum amount of data can be accessed in a balanced fashion, while also adhering to the constraints of data distribution in HDFS and the load balance requirements of each parallel process/executor. This is complicated by the fact that, *a*) Parallel processes/executors usually need to be assigned an equal number of tasks so as to maximize the utilization of resources. *b*) Data in HDFS is not evenly distributed on the cluster nodes, which implies that some processes have more local data than others. *c*) If an application were to require multiple data inputs, the inputs needed by a task may be stored on multiple nodes.

- Sub-dataset analyses are the process of analyzing specific sub-datasets, e.g, events or topics, to ensure system security and gain business intelligence [57]. A large dataset may contain millions or billions of sub-datasets such as advertisement clicks or event-based log data [9, 5, 33]. The content of a single sub-dataset can be stored in different data partitions, e.g, HDFS blocks, and each block usually contains many sub-datasets. Since sub-dataset analysis programs do not have the knowledge of sub-datasets distribution over HDFS blocks, i.e. the data size of the sub-dataset contained by each block, the sub-dataset filtration or sampling in big data processing will unfortunately result in an *imbalanced* execution in parallel sub-dataset analysis. One main challenge to solve this problem is that collecting and storing the meta-data pertaining to the distribution of millions or billions of sub-datasets could incur a substantial cost in memory and CPU cycles.

In order to address the above challenges, we firstly propose a scalable locality-aware middleware (SLAM), which allows scientific analysis applications to benefit from data-locality exploitation with the use of HDFS, while also maintaining the flexibility and efficiency of the MPI programming model. SLAM aims to enable parallel processes to achieve high I/O performance in the environment of data-intensive computing and it consists of three components: (1) a data-centric scheduler (DC-scheduler), which transforms a compute-centric mapping into a data-centric one so that a computational process always accesses data from a local or nearby computation node, (2) a data location-aware monitor to support the DC-scheduler by obtaining the physical data distribution in the underlying file system, and (3) a virtual I/O translation layer to enable computational processes to execute conventional I/O operations on distributed file systems. SLAM can benefit not only parallel programs that call our DC-scheduler to optimize data access during development, but also existing programs in which the original process-to-file assignments could be intercepted and re-assigned so as to achieve maximum efficiency on a parallel system.

Secondly, we present a systematic analysis for parallel data imbalanced read on distribution file systems. Based on the analysis, we propose novel matching-based algorithms for parallel big data processing, which assist parallel applications with the assignment of data tasks such that the distributed I/O resources can be maximally utilized as well as to balance the distribution of covariates in datasets and application demands. To achieve this, we retrieve the data layout information for data analysis programs from the underlying distributed file system and model the assignment of processes/executors to data as a one-to-many matching in a Bipartite Matching Graph. We then compute a solution for the matching graph that enables parallel data requests to be served on HDFS in a balanced fashion.

Thirdly, we propose a novel method to optimize sub-dataset analysis over distributed storage systems referred to as DataNet. DataNet aims to achieve distribution-aware and workload-balanced computing and consists of the following three parts: (1) we propose an efficient algorithm with linear complexity to obtain the meta-data of sub-dataset distributions, (2) we design an elastic storage structure called ElasticMap based on the HashMap and BloomFilter techniques to store the meta-data and (3) we employ a distribution-aware algorithm for sub-dataset applications to achieve a workload-balance in parallel execution. Our proposed method can benefit different sub-dataset analyses with various computational requirements.

CHAPTER 2: BACKGROUND

In this chapter, we present the workflow of existing parallel applications in scientific analysis and visualization. Specifically, we use two well-known applications to demonstrate how parallel processes access their needed data for analysis and visualization. We also discuss how parallel applications can access data from distributed file systems, e.g, HDFS. We then briefly describe challenges associated with parallel data access on the file systems.

2.1 Scientific Analysis and Visualization Applications

In computational biology, genomic sequencing tools are used to compare given query sequences against database to characterize new sequences and study their effects. There are many different alignment algorithms in this field, such as Needleman-Wunsch [66], FASTA [71], and BLAST [13]. Among them, the BLAST family of algorithms is the most widely used in the study of biological and biomedical research. It compares a query sequence with database sequences via a two-phased heuristic-based alignment algorithm. At present, BLAST is a standard defined by the National Center for Biotechnology Information (NCBI).

mpiBLAST [31] is a parallel implementation of NCBI BLAST. As shown in Figure 2.1, mpiBLAST organizes all parallel processes into one master process and many worker processes. Before performing an actual search, the raw sequence database is formatted into many fragments and stored in a shared network file system with the support of MPI or POSIX I/O operations. mpiBLAST follows a compute-centric model and moves the requested database fragments to the corresponding compute processes. By default, the master process accepts gene sequence search jobs from clients and generates task assignments according to the database fragments, and mpi-

BLAST workers load database fragments from a globally accessible file system over a network and perform the BLAST task according to the master scheduling. To search through a large database, the I/O cost, which takes place before the real BLAST execution, takes a significant amount of time.

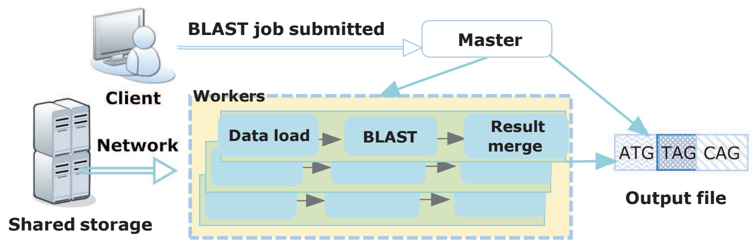


Figure 2.1: The default mpiBLAST framework: mpiBLAST workers load database fragments from a globally accessible file system over a network and perform BLAST task according to the master scheduling.

ParaView [12] is an open-source, multi-platform application for the visualization and analysis of scientific datasets. ParaView has three main logical components: data server, render server, and client. The data server reads in files from shared storage and processes data through the pipeline to the render server, which renders this processed data to present the results to the client. The data server can exploit data parallelism by partitioning the data and assigning each data server process a part of the dataset to analyze. By splitting the data, ParaView is able to run data processing tasks in parallel. Figure 2.2 demonstrates an example of parallel visualization for a Protein Dataset.

Current MPI based visualization applications adopt a compute-centric scheduling in which each data server process is assigned tasks according to their MPI ranks. Once a data processing task is scheduled to a data server process, the data will be transferred from a shared storage system to the compute node. Since parallel file systems such as PVFS or Lustre, are usually deployed on storage nodes and data server processes are deployed on compute nodes, this compute-centric model involves a significant amount of data movement for big data problems and becomes a major stumbling block to high performance and scalability.

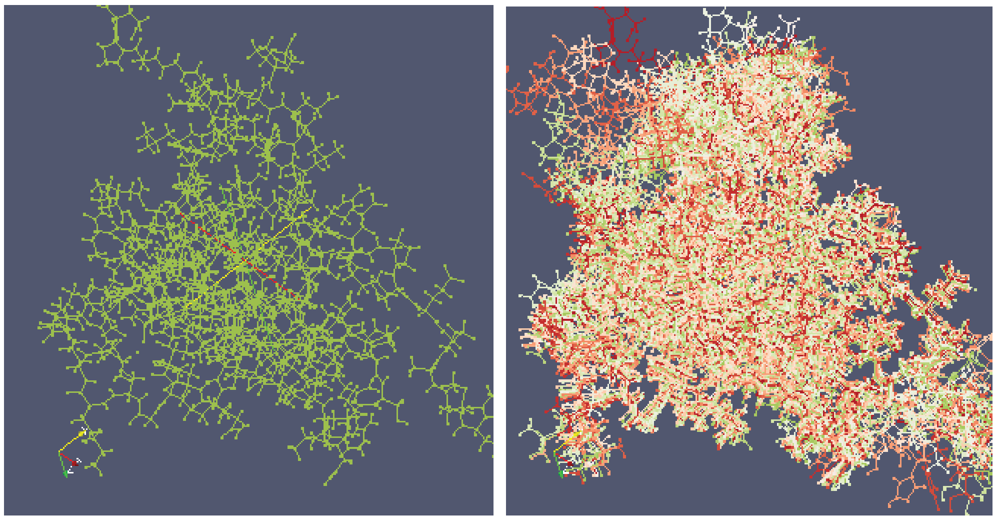


Figure 2.2: A protein dataset is partitioned across multiple parallel processes; the left figure is the sub dataset rendering picture, while the right one is the composite picture of a whole dataset.

2.2 The Hadoop File System and the I/O Interfaces

The Hadoop Distributed File System (HDFS) is an open source implementation of the Google File System (GFS), specifically for the use of MapReduce style workloads. The idea behind the MapReduce framework is that it is faster and more efficient to send the compute executables to the stored data to be processed in-situ rather than to pull the data needed from storage. While HDFS can allow analysis programs to benefit from data locality computation, there are several limitations to running MPI-based analysis applications on HDFS. Firstly, current MPI-based parallel applications are mainly developed with the MPI model, which employs either MPI-I/O or POSIX-I/O to run on a network file system or a network-attached parallel file system. However, HDFS has its own I/O interfaces, which are different from traditional MPI-I/O and POSIX-I/O. Moreover, MPI-based parallel applications usually produce distributed results and employ "concurrent write" methods to output results, while HDFS only supports “append” write.

2.3 Problems of Runing Parallel Applications On HDFS

Parallel applications such Paraview [12] usually assign data processing tasks to parallel processes during initialization. These processes can simultaneously issue a large number of data read requests to file systems due to the synchronization requirements of parallel processes [78, 23]. Compared to distributed computing applications such as MapReduce programs, parallel applications generally require more precise synchronization and thus a greater number of burst read requests[78, 12, 23, 98]. We specifically discuss two parallel data access challenges in parallel data analysis.

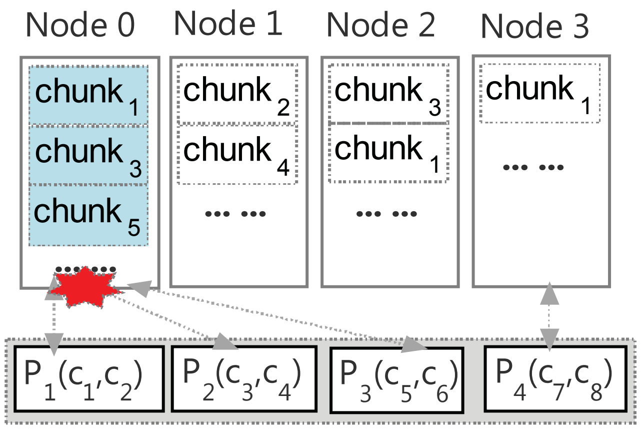


Figure 2.3: The contention of parallel data requests on replica-based data storage.

Parallel Single-Data Access: Most applications based on *SPMD or independent parallelism* employ static data assignment methods, which partition input data into independent operators/tasks, with each process working on different data partitions. A typical example is Paraview [12]. Paraview employs data servers to read files from storage. To process a large dataset, the data servers, running in parallel, read a meta-file, which lists a series of data files. Then, each data server will compute their own part of the data assignment according to the number of data files, number of server parallel processes, and their own process rank. For instance, the indices of files assigned to a process i are in the interval:

$$\left[i \times \frac{\#\ of\ files}{\#\ of\ process}, (i+1) \times \frac{\#\ of\ files}{\#\ of\ process}\right)$$

The processes read the data in parallel and process data through the pipeline to be rendered. With the data stored in HDFS, a process will read the data from its local disk if its required data is on that disk, or from another remote node that contains the required data. Unfortunately, such a read strategy in HDFS in combination with data assignment methods from applications can cause some cluster nodes to serve more data requests than others. For the example shown in Figure 2.3, three processes can read three data chunks from Node 0 and no process will read data from Node 1, resulting in a lower parallelism utilization of cluster nodes/disks.

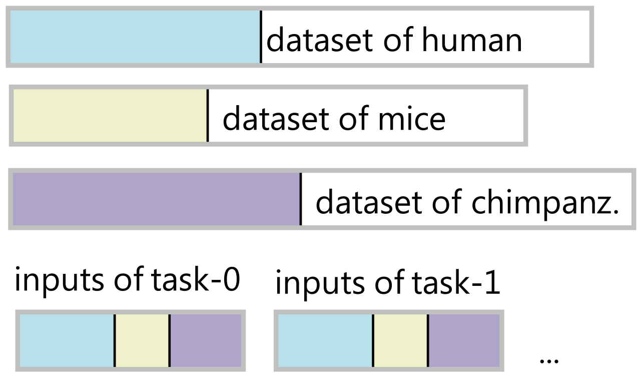


Figure 2.4: An example of parallel tasks with multiple parallel data inputs.

Parallel Multi-Data Access: In certain situations, a single task could have multiple datasets as input e.g. when the data are categorized into different subsets, such as with the gene datasets of species [42]. For instance, to compare the genome sequences of humans, mice and chimpanzees, a single task needs to read three inputs, as shown in Figure 2.4. These inputs may be stored on different cluster nodes and, without consideration of data distribution and access policy of HDFS, their data requests could cause some storage nodes to suffer a contention, thus degrading the execution performance.

CHAPTER 3: SCALABLE LOCALITY-AWARE MIDDLEWARE

As data repositories expand exponentially with time and scientific applications become ever more data intensive as well as computationally intensive, a new problem arises in regards to the transmission and analysis of data in a computationally efficient manner. Programs running on large-scale clusters in parallel suffer from potentially long I/O latency resulting from non-negligible data movement, especially in commodity clusters with Gigabit Ethernet. As we discussed, scientific analysis applications could significantly benefit from local data access in a distributed fashion with the use of hadoop file system.

In this chapter, we propose a middleware called "SLAM", which allow scientific analysis programs to benefit from data locality exploitation in HDFS, while also exploiting the flexibility and efficiency of the MPI programming model. Since the data are often distributed in advance within HDFS, the default task assignment, without considering data distribution, may not allow parallel processes to fully benefit the local data access. Thus, we need to intercept the original task scheduling and re-assign the tasks to parallel process so as to achieve the maximum efficiency of a parallel system, including a high degree of data locality and load balance. Also, we need to solve the I/O incompatibility issue, such that the data stored in the HDFS can be accessed through conventional parallel I/O methods, e.g, MPI-I/O or POSIX I/O.

SLAM implements a fragment location monitor, which collects an unassigned fragment list at all participating nodes. To achieve this, the monitor needs to make connections to the HDFS NameNode using libHDFS, and request chunk location information by asking NameNode (specified by a file name, offset within the file, and length of the request). The NameNode replies with a list of the host DataNodes where the requested chunks are physically stored. Based on this locality information, our proposed scheduler will make informed decisions as to which node will be chosen to execute a computation task in order to take advantage of data locality. This could realize local data access and avoids data movement in the network.

Specifically, the SLAM framework for parallel BLAST consists of three major components, a translation I/O layer called SLAM-I/O, a data centric load-balanced scheduler called a DC-scheduler and a fragments location monitor, as illustrated in Figure 3.1. Specifically, the Hadoop Distributed File System (HDFS) is chosen as the underlying storage. SLAM-I/O is implemented as an non-intrusive software component added to existing application codes, such that many successful performance tuned parallel algorithms and high performance noncontiguous I/O optimization methods [56, 78, 95] can be directly inherited in SLAM. The DC-scheduler determines which specific data fragment is assigned to each node to process. It aims to minimize the number of fragments pulled over the network. DC-scheduler is incorporated into the runtime of parallel BLAST applications. The fragment location monitor will then be invoked by the DC-scheduler to report the database fragments locations.

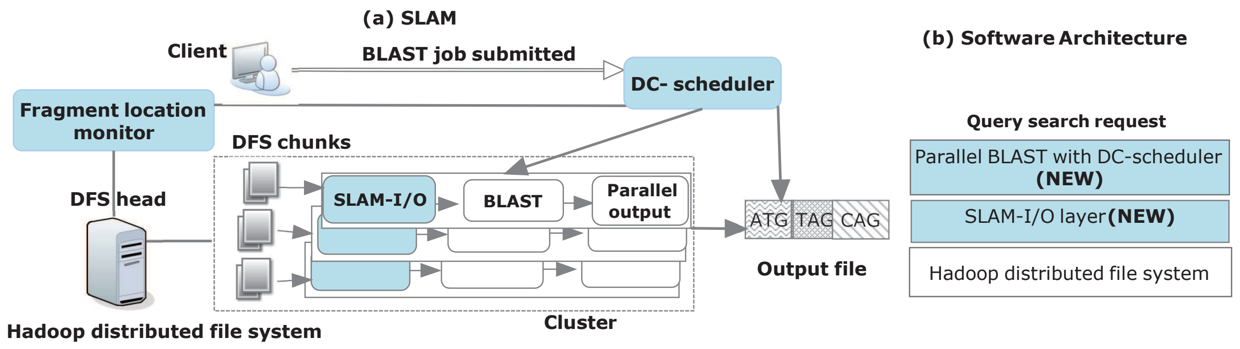


Figure 3.1: Proposed SLAM for parallel BLAST. (a) The DC-scheduler employs a "Fragment Location Monitor" to snoop the fragments location and dispatches unassigned fragments to computation processes such that each process could read the fragments locally, *i.e.*, reading chunks in HDFS via SLAM-I/O. (b) The SLAM software architecture. Two new modules are used to assist parallel BLAST in accessing the distributed file system and intelligently read fragments with awareness of data locality.

By tracking the location information, the DC-scheduler schedules computation tasks at the appropriate compute nodes, namely, moves computation to data. Through SLAM-I/O, MPI processes can directly access fragments treated as chunks in HDFS from the local hard drive, which is part of the entire HDFS storage.

3.1 SLAM-I/O: A Translation Layer

Current scientific parallel applications are mainly developed with the MPI model, which employs either MPI-I/O or POSIX-I/O to run on a network file system or a network-attached parallel file system. SLAM uses HDFS to replace these file systems, and therefore entails handling the I/O compatibility issues between MPI-based programs and HDFS.

More specifically, scientific parallel applications access files through MPI-I/O or POSIX-I/O interfaces, which are supported by local UNIX file systems or parallel file systems. These I/O methods are different from the I/O operations in HDFS. For example, HDFS uses a client-server model, in which servers manage metadata while clients request data from servers. These compatibility issues make scientific parallel applications unable to run on HDFS.

To solve the problem, we implement a translation layer—SLAM-I/O to handle the incompatible I/O semantics. The basic idea is to transparently transform high-level I/O operations of parallel applications to standard HDFS I/O calls. We elaborate how SLAM-I/O works as follows. SLAM-I/O first connects to the HDFS server using hdfsConnect() and mounts HDFS as a local directory at the corresponding compute node. Hence each cluster node works as one client to HDFS. Any I/O operations of parallel applications that work in the mounted directory are intercepted by the layer and redirected to HDFS. Finally, the corresponding hdfs I/O calls are triggered to execute specific I/O functions *e.g.* open /read /write /close.

Handling concurrent write is another challenge in the development of SLAM. Parallel applications usually produce distributed results and may allow every engaged process write to disjoint ranges in a shared file. For instance, mpiBLAST takes advantage of Independent/Collective I/O to optimize the searched output. The *WorkerCollective* output strategy introduced by Lin *et. al.* [56] realizes a concurrent write semantic, which can be interpreted as "multiple processes write to a single file at the same time". These concurrent write schemes often work well with parallel file systems or network file systems. However, HDFS only supports appended write, and most importantly, only one process is allowed to open the file for writing at a time (otherwise an open error will occur). To resolve this incompatible I/O semantics issue, we revise "concurrent write to one file" to "concurrent write to multiple files". We allow every process output their results and the write ranges independently into a physical file in HDFS. Logically, all output files produced for a data processing job are allocated in the same directory. The overall results are retrieved by joining all physical files under the same directory.

In our experimental evaluation, we prototyped SLAM-I/O using FUSE [3], a framework for running stackable file systems in a non-privileged mode. An I/O call from an application to the Hadoop file system is illustrated in Figure 3.2. The Hadoop file system is mounted on all participating cluster nodes through the SLAM-I/O layer. The I/O operations of mpiBLAST are passed through a virtual file system (VFS), taken over by SLAM-I/O through FUSE and then forwarded to HDFS. HDFS is responsible for the actual data storage management. In regards to concurrent write, SLAM-I/O automatically inserts a subpath using the same name as the output filename and appends its process ID at the end of the file name. For instance, if a process with id 30 writes into $/hdfs/foo/searchNTresult$, the actual output file is $/hdfs/foo/serachNTresult/searchNTresult30$.

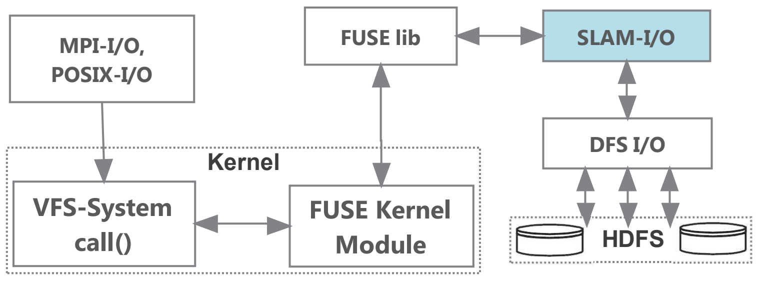


Figure 3.2: The I/O call in our prototype. A FUSE kernel module redirects file system calls from parallel I/O to SLAM-I/O and SLAM-I/O wrappers HDFS clients and translates the I/o call to DFS I/O.

3.2 A Data Centric Scheduler

As discussed, the key to realizing scalability and high-performance in big data scientific applications is to achieve data locality and load balance. However, there exists several heterogeneity issues that could potentially result in load imbalance. For instance, in parallel gene data processing, the global database is formatted into many fragments. The data processing job is divided into a list of tasks corresponding to the database fragments. On the other hand, the HDFS random chunk placement algorithm may distribute database fragments unevenly within the cluster, leaving some nodes with more data than others. In addition, the execution time of a specific BLAST task could vary greatly and is difficult to predict according to the input data size and different computing capacities per node [34, 56].

We implement a fragment location monitor as a background daemon to report updated unassigned fragment statuses to the DC-scheduler. At any point in time, the DC-scheduler always tries to launch a *local task* of the requesting process, that is, a task with its corresponding fragment available on the node issued by the requesting process. There exists a good chance of achieving a high degree of data locality, as each fragment has three physical copies in HDFS, namely, there are three different node candidates available for scheduling.

Upon an incoming data processing job, the DC-scheduler invokes the location monitor to report the physical locations of all target fragments. If a process from a specific node requests a task, the scheduler assigns a task to the process using the following procedure. First, if there exists local tasks on the requesting node, the scheduler will evaluate which local task should be assigned to the requesting process in order to make other parallel processes achieve locality as much as possible (details will be provided later). Second, if there does not exist any local task on the node, the scheduler will assign a task to the requesting process by comparing all unassigned tasks in order to make other parallel processes achieve locality. The node will then pull the corresponding fragment over the network.

Algorithm 3.2.1 Data centric load-balanced Scheduler Algorithm

```
1: Let $F = \{f_1, f_2, ..., f_m\}$ be the set of tasks
2: Let $F_i$ be the set of unassigned local tasks located on node $i$
Steps:
3: Initialize $F$ for a data processing job
4: Invoke $Location\ monitor$ and initialize $F_i$ for each node $i$
5: while $|F| \neq 0$ do
6:     if a worker process on node $i$ requests a task then
7:         if $|F_i| \neq 0$ then
8:             Find $f_x \in F_i$ such that
9:             $x = \underset{x}{\operatorname{argmax}}(\min_{F_k \ni f_x, k \neq i}(|F_k|))$
10:            Assign $f_x$ to the requesting process on node $i$
11:        else
12:            Find $f_x \in F$ such that
13:            $x = \underset{x}{\operatorname{argmax}}(\min_{F_k \ni f_x, k \neq i}(|F_k|))$
14:            Assign $f_x$ to the requesting process on node $i$
15:        end if
16:        Remove $f_x$ from $F$
17:        for all $F_k$ s.t. $f_x \in F_k$ do
18:            Remove $f_x$ from $F_k$
19:        end for
20:    end if
21: end while
```

The scheduler is detailed in Algorithm 1. The input data processing job consists of a list F of individual tasks, each associated with a distinct fragment. While the tasks list, F, is not empty, parallel processes report to the scheduler for assignments. Upon receiving a task request from an process on Node i, the scheduler determines a task for the process as follows:

- 1. If Node i has some local tasks, then the local task x that could make the number of unassigned local tasks on all other nodes as balanced as possible will be assigned to the requesting process. Figure 3.3 illustrates an example to demonstrate how this choice is made. In the example, there are 4 parallel processes running on 4 nodes, where $W1$ requests a task from its unassigned local tasks $F_1 =< f2, f4, f6 >$. For each task f_x in F_1, we compute the minimum number of unassigned tasks among all other nodes containing f_x's associated fragment. For example, the task f_2 is local to F_2 and F_4, so we compute $\min(|F_2|, |F_4|) = 2$. We assign the task with the largest such value to the idle process, which is f_6 in the example. After the assignment, the number of unassigned local tasks for node $W2$, $W3$ and $W4$ become 2, 2, 2, as shown in Figure 3.3 (b).
- 2. If node i does not contain any unassigned local tasks, the scheduler will perform the above calculation for all unassigned tasks in F and assign the task with the largest $\min$ value to the requesting process, which needs to pull data over network.

Since mpiBLAST adopts a master-slave architecture, the DC-scheduler could be directly incorporated into the master process, which performs dynamic scheduling according which nodes are idle at any given time. For such scheduling problems, minimizing the longest execution time is known to be NP-complete [35] when the number of nodes is greater than 2, even for the case that all tasks are executed locally. However we will show that for this case, the execution time of our solution is at most two times that of the optimal solution.

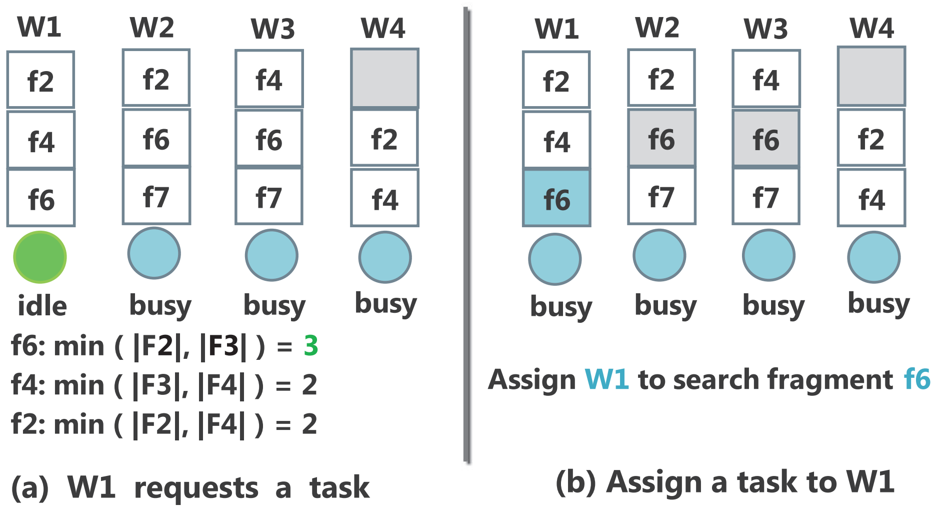


Figure 3.3: A simple example of the DC-scheduler receiving the task request of the process ($W1$). The scheduler finds the unassigned local tasks of $W1$ ($f2$, $f4$ and $f6$ in this example). The task $f6$ will be assigned to $W1$ since the minimum unassigned task value is 3 on $W2$ and $W3$, which also has $f6$ as a local task. After assigning $f6$ to $W1$, the number of unassigned local tasks of $W1$–4 is 2.

Suppose there are $m = |F|$ tasks with actual execution times of $t_1, t_2, ..., t_m$ on $n = |W|$ nodes. We use T^* to denote the maximum execution time of a node in the optimal solution. Notice that T^* cannot be smaller than the maximum execution time of a single task. This observation gives us a lowerbound for T^*:

$$T^* \geq \max_{1 \leq k \leq m} (t_k) \tag{3.1}$$

Let T be the maximum execution time of a node in a solution given by our algorithm. Without loss of generality, we assume that the last completed task f_n is scheduled on node n. Let s_{f_n} denote the start time of task f_n on node n, so $T = s_{f_n} + t_n$.

All nodes should be busy until at least time s_{f_n}; otherwise, according to our algorithm, the task f_n

will be assigned to some nodes earlier. Therefore we have $T^* \geq s_{f_n}$. Because $T^* \geq \max_{1 \leq k \leq m}(t_k)$, we have $T^* \geq t_n$. This gives us the desired approximation bound:

$$T = s_{f_n} + t_n \leq 2T^* \tag{3.2}$$

The scheduling problem is even much harder when we take the location variable into consideration. However, we will conduct real experiments to examine its locality and parallel execution in Section 4.

3.3 ParaView with SLAM

As we discussed, ParaView could suffer from non-negligible data movement and network contention, resulting in serious performance degradation. As with parallel BLAST, ParaView could benefit from local data access in a distributed fashion. Allowing ParaView to achieve data locality computation requires the replacement of the data partition scheduling in the ParaView reader modules on each data server processes, which ingest data from files according to the task assignment.

Currently, there are a large number of data readers with support for various scientific file formats. Specifically, examples of parallel data readers [79] are Partitioned Legacy VTK Reader, Multi-Block Data Reader, Hierarchical Box Data reader, Partitioned Polydata Reader, etc. To process a data set, the data servers, running in parallel, will call the reader to read a meta-file, which points to a series of data files. Then each data server will compute their own part of the data assignment according to the number of data files, number of parallel servers, and the their own server rank. Data servers will read the data in parallel from the shared storage and then filter/render.

In order to achieve locality computation for ParaView, we need to intercept the default tasks assignment and use our proposed DC-scheduler to assign tasks for each data server at run time. Specifically, the SLAM framework for ParaView also includes three components, the translation layer — SLAM-I/O, the DC-scheduler and the fragments location monitor, as illustrated in Figure 3.4. The DC-scheduler determines which specific data fragment is assigned to each data server process. To get the physical location of the target data sets, the Location Monitor is invoked by the DC-scheduler to report the data fragments locations. Through SLAM-I/O, the data server processes can directly access data, treated as chunks in HDFS, from the local hard drive, which is part of the entire HDFS storage.

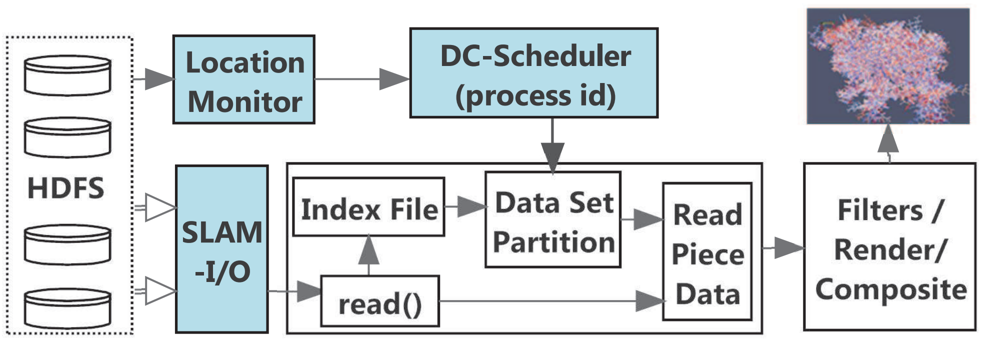


Figure 3.4: Proposed SLAM for ParaView. The DC-scheduler assign data processing tasks to MPI processes such that each MPI process could read the needed data locally, *i.e.*, reading chunks in HDFS via SLAM-I/O.

Our proposed DC-scheduler algorithm in Section 3.3 is very suitable for the applications with dynamic scheduling algorithms, such as mpiBLAST, in which scheduling is determined by which nodes are idle at any given time. However, since the data assignment in ParaView uses a static data partitioning method, the work allocation is determined beforehand; no process works as a central scheduler. For this kind of scheduling, we adopt a round-robin request order for all data server in Step 8 of Algorithm 3.2.1. Until the set F is empty, the the data server process with a specific pid can get all the data pieces assigned to it. Then the data servers will read the data in parallel and then filter/render.

3.4 Specific HDFS Considerations

HDFS employs some default data placement policies. A few considerations should be taken into account when we choose HDFS as the shared storage. First, each individual fragment file size should not exceed the configured chunk size, otherwise the file will be broken up into multiple chunks with each chunk replicated independently of other related chunks. If only a fraction of the specific fragment can be accessed locally, other parts must be pulled over the network. Consequently, the locality benefit is lost. As a result, we should keep the file size of each database fragment smaller than the chunk size when formatting the data set. Second, for parallel BLAST, when applying the database format method, each fragment includes seven related files, six of which are smaller files and one is bigger. The *hadoop Archive* method should be applied to ensure that these seven files are stored together during a formatted execution.

3.5 Experiments and Analysis

3.5.1 Experimental Setup

We conducted comprehensive testing on our proposed middleware SLAM on both Marmot and CASS clusters with different storage systems. *Marmot* is a cluster of PRObE on-site project and housed at CMU in Pittsburgh. The system has 128 nodes / 256 cores and each node in the cluster has dual 1.6GHz AMD Opteron processors, 16GB of memory, Gigabit Ethernet, and a 2TB Western Digital SATA disk drive. *CASS* consists of 46 nodes on two racks, one rack including 15 compute nodes and one head node and the other rack containing 30 compute nodes. Each node is equipped with dual 2.33GHz Xeon Dual Core processors, 4GB of memory, Gigabit Ethernet and a 500GB SATA hard drive.

In both clusters, MPICH [1.4.1] is installed as parallel programming framework on all compute nodes running CENTOS55-64 with kernel 2.6. We chose Hadoop 0.20.203 as the distributed file system, which is configured as follows: one node for the NameNode/JobTracker, one node for the secondary NameNode, and other nodes as the DataNode/TaskTracker. In addition, we chose two conventional file systems as our baseline file systems for a comprehensive test. We run experiments on NFS as the developers of mpiBLAST use NFS as shared storage [56]. We also installed PVFS2 version [2.8.2] with default setting on the cluster nodes: one node as the metadata server for PVFS2, and other nodes as the I/O servers (similar to HDFS).

3.5.2 Evaluating Parallel BLAST with SLAM

To make a fair comparison with the open source parallel BLAST, we deploy mpiBLAST [1.6.0] on all nodes in the clusters that support the techniques of query prefetch and computation & I/O coordination methods that would coordinate dynamic load balancing of computation and high performance noncontiguous I/O. Equipped with our SLAM-I/O layer at each cluster node, HDFS can be mounted as a local directory and used as shared storage for parallel BLAST. The BLAST itself can run on HDFS without recompilation. We implement the fragment location monitor and the DC-scheduler and incorporate both modules into the mpiBLAST master scheduler to exploit data locality as shown in Figure 3.1. When running parallel BLAST, we let the scheduler process run on the node on which the NameNode is configured, and parallel processes run on the DataNodes for the sake of locality.

We select the nucleotide sequence database *nt* as our experimental database. The *nt* database contains the GenBank, EMB L, D, and PDB sequences. At the time when we performed our experiments, the *nt* database contained 17,611,492 sequences with a total raw size of about 45 GB. The input queries to search against the *nt* database are randomly chosen from *nt* and revised, which guarantees that we find some close matches in the database.

When running mpiBLAST on the cluster with directly attached disks, users usually run *fastasplitn* and *formatdb* once and reuse the formatted database fragments. To deliver database fragments, we use a dynamic copying method such that the node will copy and cache a data fragment only when a search task to the fragment is scheduled on the node. These cached fragments are reused for subsequent sequences searches. mpiBLAST is configured with two conventional file systems—NFS and PVFS2 and both work as baselines. SLAM employs HDFS as a distributed storage. Therefore, there is no need for gathering a fragment over network from multi data nodes as PVFS does, and we do not cache fragments in local disks either.

We studied how SLAM could improve the performance for parallel BLAST. We scaled up the number of data nodes in the cluster and compared the performance with three host file system configurations: NFS, PVFS2 and HDFS, respectively. For clarity, we labeled them as NFS-based, PVFS-based and SLAM-based BLAST. During the experiments, we mount NFS, HDFS and PVFS2 as the local file systems at each node if a BLAST process is running on that node. We used the same input query in all running cases and fix the query size to 50 KB with 100 sequences, which generated a consistent output result of around 5.8 MB. The *nt* database was partitioned into 200 fragments.

3.5.2.1 Results from an Marmot Cluster

The experimental results collected from Marmot are illustrated in Figures 3.5, 3.6, 3.7 and 3.8.

When running parallel BLAST on a 108-node configuration system, we found the total program execution time with NFS-based, PVFS-based and SLAM-based BLAST to be 589.4, 379.7 and 240.1 seconds, respectively. We calculate the performance gain with Equation 3.3, where $T_{\text{SLAM-based}}$ denotes the overall execution time of parallel BLAST based on SLAM and $T_{\text{NFS/PVFS-based}}$ is the overall execution time of mpiBLAST based on NFS or PVFS.

$$improvement = 1 - \frac{T_{\text{SLAM-based}}}{T_{\text{NFS/PVFS-based}}}. \tag{3.3}$$

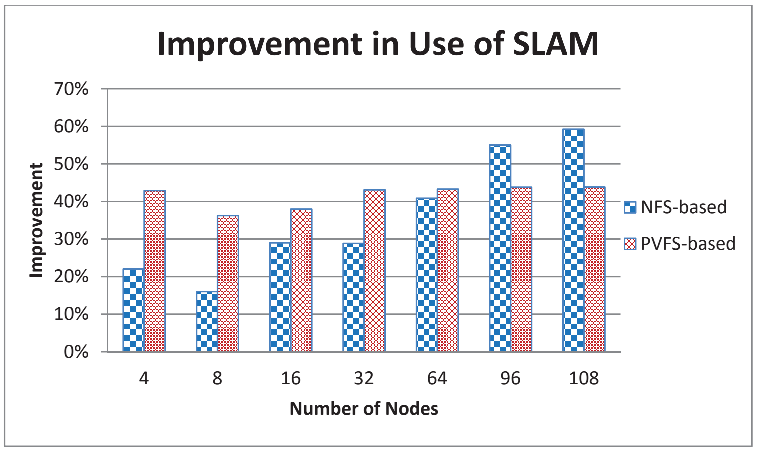


Figure 3.5: The performance gain of mpiBLAST execution time when searching the *nt* database in use of SLAM as compared to NFS-based and PVFS-based.

As seen from Figure 3.5, we conclude that SLAM-based BLAST could reduce overall execution latency by 15% to 30% for small-sized clusters with less than 32 nodes as compared to NFS-based BLAST. Given an increasing cluster size, SLAM reduces overall execution time by a greater percentage, reaching 60% for a 108-node cluster setting. This indicates that the NFS-based setting is not scaling well. In comparison to PVFS-based BLAST, SLAM runs consistently faster by about 40% for all cluster settings.

To test scalability we collected results of aggregated I/O bandwidth for an increasing number of nodes as illustrated in Figure 3.6. We find that in SLAM the I/O bandwidth greatly increases as the number of nodes increases, proving it to be a scalable system. However, the NFS and PVFS based BLAST schemes have a considerably lower overall bandwidth, and as the number of nodes in the other file systems increases, they do not achieve the same bandwidth increase from more nodes. This indicates a large data movement overhead exists in NFS and PVFS that hinder their scalability.

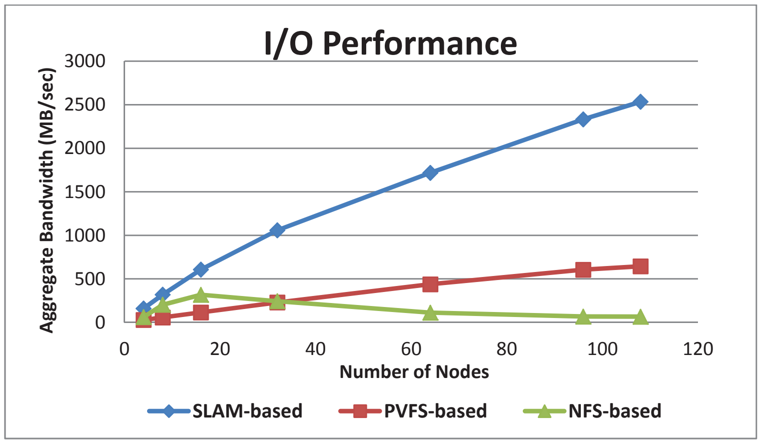


Figure 3.6: The input bandwidth comparison of NFS-based, PVFS-based and SLAM-based BLAST scheme. The key observation is that SLAM scales linearly well for search workloads.

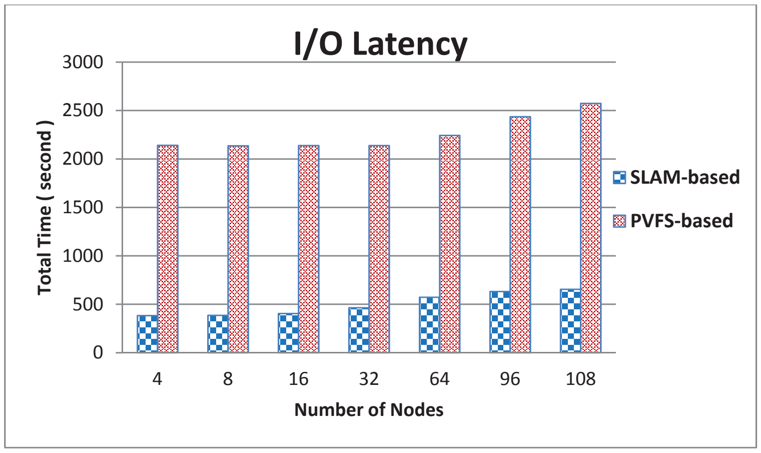


Figure 3.7: The I/O latency comparison of PVFS-based and SLAM-based BLAST schemes on the *nt* database with an increasing number of nodes.

To gain some insight on the time costs of data preparation, data input time, referred as I/O latency, was measured for an increasing number of nodes on both SLAM and PVFS based BLAST, as illustrated in Figure 3.7. We find that the total I/O latency of PVFS based BLAST is close to 2,000 seconds for clusters of 32 nodes and increases thereafter. On the other hand, SLAM achieves a much lower I/O latency than PVFS with latency times being a quarter of that of PVFS for clusters up to 64 nodes, and a fifth for larger networks. Figure 3.8 shows the particular case of I/O latency times on a 64 node cluster using SLAM, PVFS, and NFS. Three latency figures are presented for each file system: maximum node latency, minimum node latency, and average latency. SLAM excels in having low latency times in all three tests while maintaining a small difference in I/O time between the fastest and slowest node, and ultimately achieves the lowest latency times of all three systems. NFS and PVFS suffer from an imbalance in node latency and on average are considerably slower than SLAM.

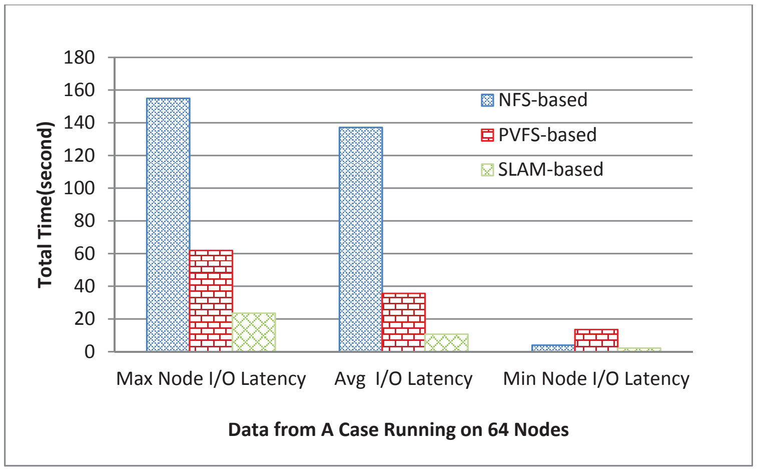


Figure 3.8: The max and min node I/O time comparison of NFS-based, PVFS-based and SLAM-based BLAST on the *nt* with varying number of nodes.

3.5.2.2 Results from a CASS Cluster

For a comprehensive testing, we performed similar experiments at an on-site CASS cluster. We distinguish the average actual BLAST times from I/O latency to gain some insights about scalability.

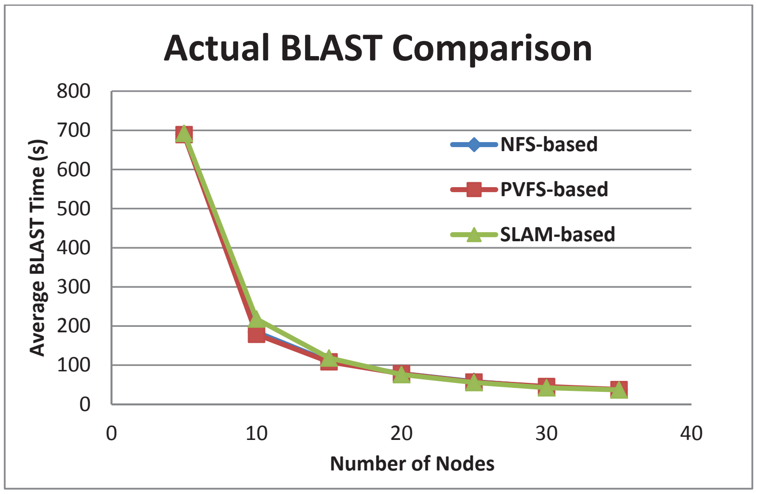


Figure 3.9: The actual BLAST time comparison of NFS-based, PVFS-based and SLAM-based BLAST programs on the *nt* database with different number of nodes.

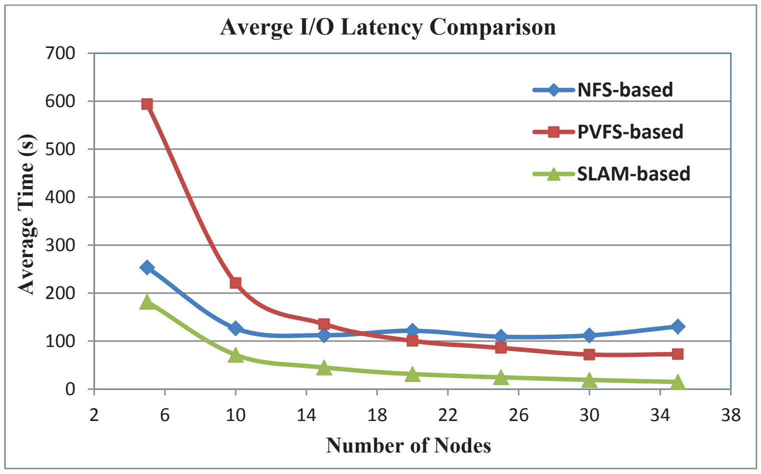


Figure 3.10: The average I/O time of NFS-based, PVFS-based and SLAM-based BLAST on the *nt* database with different number of nodes.

Figure 3.9 illustrates the average actual BLAST computation times (excluding I/O latency) in an increasing cluster size. We find that the average actual BLAST time in Figure 3.9 decreases sharply as the number of nodes grow. The three systems that we tested obtained comparable BLAST performances. This supports our conjecture as SLAM only targets I/O rather than real BLAST computation. Different file system configurations—NFS, PVFS, and HDFS account for the differences among three BLAST programs. Figure 3.10 illustrates the I/O phase of the BLAST workflow. In NSF and PVFS baselines, the average I/O cost remains consistent, at around 100 seconds, after cluster size exceeds 15. In contrast, SLAM adopts a scalable HDFS solution, which realizes a decreasing I/O latency along with an increasing number of nodes.

The priority of our DC-scheduler is to achieve data-task locality while adhering to load balance constraints. To explore the effectiveness of the DC-scheduler,(*i.e.*, to what extent search processes are scheduled to access fragments from local disks), Figure 3.11 illustrates one snapshot of the fragments searched on each node and the fragments access by the network. We specifically ran experiments five times to check how much data is moved through the network in a 30-node setting, and track down a total number of fragments 150, 180, 200, 210, 240 respectively. As seen from the Figure 3.11, most nodes search a comparable number of fragments locally. More than 95% of the data fragments are read from local storage.

We also run mpiBLAST on HDFS using only our I/O translation layer (without the locality scheduler) and found that the performance is only slightly better than that of PVFS-based BLAST. This is because BLAST processes need to read data remotely without the coordination of data locality scheduler. We will show the detail comparison in the next Subsection.

3.5.2.3 Comparing with Hadoop-based BLAST

We only show a simple comparison with Hadoop-based Blast from Marmot, as such a comparison may be unfair since the efficiency, while being the design goal of MPI, is not the key feature of the MapReduce programming model.

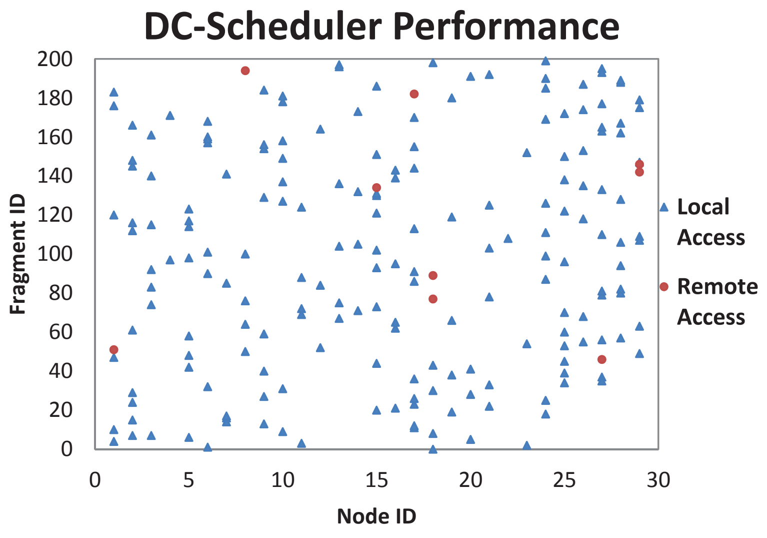


Figure 3.11: Illustration of which data fragments are accessed locally on which node and involved in the search process. The blue triangles represent the data fragments accessed locally during the search, while the red dots represent the fragments accessed remotely.

We chose Hadoop-Blast [7] as the Hadoop-based approach. The database for both programs are ‘nt’ and the input query is same. With a 25-nodes setting on marmot, SLAM-based BLAST takes 568.664 seconds while Hadoop-Blast takes 1427.389 seconds. We run the tests several times, and the SLAM-based BLAST is always more efficient than Hadoop-based BLAST. The reasons could be, 1) the task assignment of Hadoop-Blast relies on the Hadoop Scheduler, which is built on the heartbeat mechanism, 2) the advantages of I/O optimization based on MPI are not adapted by Hadoop-Blast, and 3) the difference in efficiency of Java and C/C++ implementation [81].

3.5.3 Efficiency of SLAM-I/O Layer and HDFS

In the SLAM-based BLAST framework, a translation layer—SLAM-I/O is developed to allow parallel I/O to execute on distributed file systems. In our prototype, we chose FUSE mount to transparently relink these I/O operations to the HDFS I/O interface. Thus, there is a need for evaluating the incurred overhead of a FUSE-based implementation.

SLAM-I/O is built through a Virtual File System (VFS). The I/O call needs to go through the kernel of the client operating system. For instance, to read an index file *nt.mbf* in HDFS, mpiBLAST issues an open() call first through the VFS's generic system call (sys-open()). Next, the call is translated to hdfsOpenFile(). Finally, the open operation will take effect on HDFS. We conduct experiments to quantify how much overhead the translation layer running for parallel BLAST would incur.

We run the search programs and measured the time it takes to open the 200 formatted fragments. We did two kind of tests. The first directly uses the HDFS library while the other uses the default POSIX I/O, running HDFS file open through our SLAM-I/O layer. For each opened file, we read the first 100 bytes and then close the file. We repeated the experiment several times. We found that the average total time through SLAM-I/O is around 15 seconds. The time through direct HDFS I/O was actually 25 seconds. This may result from the overhead of connecting and disconnecting with hdfsConnect() independently for each file. For the second experiment, we ran a BLAST process on multiple nodes through SLAM-I/O. The average time to open a file in HDFS is around 0.075 seconds, which is negligible compared with the overall data input time and BLAST time. In all, a FUSE based implementation does not introduce non-negligible overhead. Sometimes, SLAM-I/O actually performs better than the libhdfs based hard coding solution.

In the default mpiBLAST, each worker maintains a fragmentation list to track the fragments on its local disk and transfers the metadata information to the master scheduler via message passing. The master uses a fragment distribution class to audit scheduling. In SLAM, the NameNode is instead responsible for the metadata management. At the beginning of a computational workflow, a fragment location monitor retrieves the physical location of all fragments by talking to Hadoop's NameNode. We evaluated the HDFS overhead by retrieving the physical location of 200 formatted fragments. The average time is around 1.20 seconds, which accounts for a very small portion of the overall data input time.

3.5.4 Evaluating ParaView with SLAM

To test the performance of ParaView with SLAM, ParaView 3.14 was installed on all nodes in the cluster. To enable off-screen rendering, ParaView made use of the Mesa 3D graphics library version 7.7.1. The DC-scheduler is implemented with VTK MultiBlock datasets reader for data task assignment. A multi-block dataset is a series of sub datasets, together they represent an assembly of parts or a collection of meshes of different types from a coupled simulation [8].

To deal with MultiBlock datasets, a meta-file with extension ".vtm" or ".vtmb" is read as an index file, which points to a series of VTK XML data files constituting the subsets. The series of data files are either PolyData, ImageData, RectilinearGrid, UnstructuredGrid or StructuredGrid with the extension .vtp, .vti, .vtr, .vtu or .vts. Specifically, our scheduler method is implemented in the vtkXMLCompositeDataReader class and called in the function ReadXMLData(), which assigns the data pieces to each data server after processing the meta-file. Through intercepting the original static data assignment into our DC-scheduler, each data server process can receive the proper task assignment with it's associated data locally accessible. The data server will then be able to perform the data processing tasks according to the data assignment.

For our test dataset we use the data of a Macromolecular structure that was obtained from a Protein Data Bank [10] containing a repository of atomic coordinates and other information describing proteins and other important biological macromolecules. The processed output of these protein datasets are polygonal images, and ParaView is used to process and display such structures from the datasets. Through ParaView, scientists can also compare different biological datasets by measuring the distances and bond angles of protein structures to identify unique features. In our test, we take each data set as a time step and convert it to a subset of ParaView's MultiBlock file with extension ".vtu". Due to the need to download multiple data sets to the test system, we duplicate some datasets with some revisions and save them as new datasets.

It should be noted that the data was written in "binary" mode to allow for the smallest possible amount of time to be spent on parsing the data by the ParaView readers. Additionally, for each rendering, 96 subsets from 960 data sets were selected. As a result, our test set was approximately 40 GB in total size and 3.8 GB per rendering step.

3.5.4.1 Performance Improvement with the Use of SLAM

Performance is characterized by two aspects of the ParaView experiment: the overall execution time of the ParaView rendering and the data read time per I/O operation during program execution.

Figure 3.12 illustrates the execution time of a ParaView analysis for an increasing number of nodes with PVFS, HDFS file systems. With a small cluster of 16 nodes, the total time of the ParaView experiment did not greatly differ between the three methods, though there was some advantage to the SLAM based ParaView, which executed in 300 seconds. A 32 node cluster displayed the same attributes with ParaView executing in 200 seconds. At 64 nodes however, the SLAM based ParaView shows it's strength in large clusters seeing a large reduction in total time when compared with the PVFS and HDFS filesystems being nearly 100 seconds quicker in execution for a total execution time of 110 seconds. In a 96 node cluster, the difference between SLAM and the other filesystems is lessened. However, a great improvement is observed with SLAM based ParaView, which executes in 70 seconds, a reduction of almost 2X over PVFS and HDFS.

Figure 3.12 visualizes the time per process of a ParaView simulation on a 96 node cluster in marmot. With PVFS, data read times are consistently slow due to it's network loaded datasets and exhibit frequent bursts in read times, indicating a bottleneck due to read requests. PVFS achieves a 9.82 second average with a standard deviation of 0.669. HDFS Paraview shows the benefit of strictly implementing a distributed file system without a locality algorithm or fragmentation tracker. HDFS ParaView, in certain instances, is able to achieve very low read times with the fastest time being 2.63 seconds, however these instances of quick access are negated by the times in which data is not locally available and must be fetched over the network. Overall HDFS achieves an average readtime of 5.65 seconds with a standard deviation of 1.339. SLAM-based ParaView consistently achieves low read times with only a few outliers in which an I/O operation readtime was longer than usual. SLAM achieves an average readtime of 3.17 seconds with a standard deviation of 0.316. Overall, SLAM is able to take better advantage of large clusters by consistently making data locally available to processes through it's DC scheduler and fragmentation monitor whereas PVFS and HDFS are constantly hindered by delays in dataset movement.

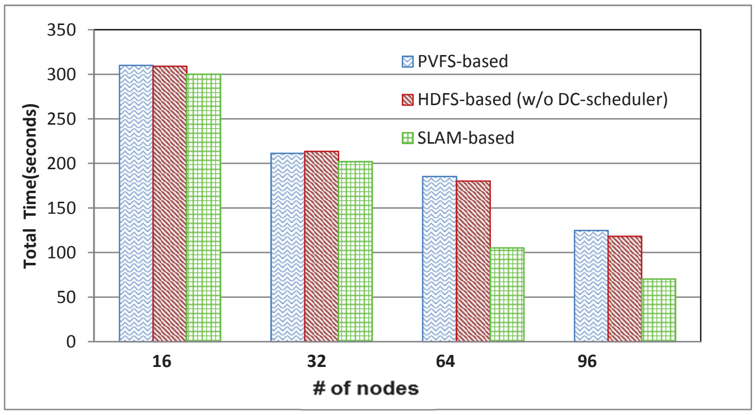


Figure 3.12: The execution time of PVFS-based, HDFS-based and SLAM-based ParaView with different number of nodes.

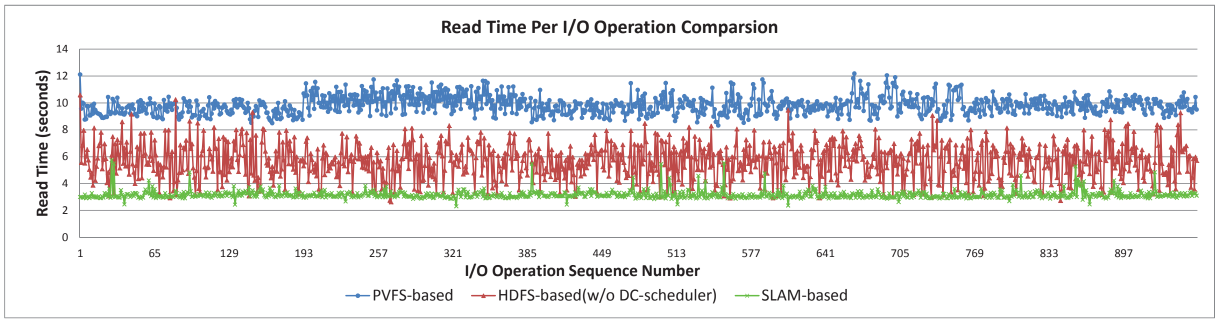


Figure 3.13: Trace of time taken for each call to vtkFileSeriesReader with/out SLAM support ParaView. Compared to PVFS-based, the number of spikes in read time are diminished and there is a smaller deviation around the trend line when computation is kept predominantly to nodes containing local copies of the needed data.

3.5.4.2 Experiments with Lustre and Discussion

Lustre is a popular parallel cluster file system known for powering seven of the ten largest HPC clusters worldwide [75]. Thus, we deploy Lustre as a storage system for comprehensive testing.

We set up an experimental architecture similar to HPC systems: a dedicated shared storage of a fixed number of I/O nodes and a variable of clients to access the storage. In practical use, it is not likely that Lustre would be co-located on the compute nodes within a cluster since Lustre is highly dependent on hardware. For example, in the experiments performed on Marmot, if one I/O node is disabled the storage system becomes inaccessible or very slow. In comparison, an HDFS DataNode failure will not affect storage performance and will in fact be completely transparent to the user.

16 nodes were selected as dedicated storage nodes. PVFS2 version [2.8.2] and Lustre version [1.8.1] were installed with default settings with one node acting as the Metadata Server and the other nodes acting as Object Storage Servers or I/O servers. For HDFS, we always co-located the storage and compute nodes. We use the Macromolecular datasets as well.

Using PVFS, Lustre, HDFS and SLAM, we first run ParaView with 16 data server processes as client processes and then increase the processes. A comparison of their performance is shown in Table 3.1. Each I/O operation performs operations on data about 60 Mb in size. The processing time is then collected from the vtkFileSeriesReader. Lustre performs very well in the experiment compared to PVFS and HDFS(without DC-scheduler), however, as with PVFS, it fails to scale after a certain number of client processes is reached, indicating a peak in bandwidth. SLAM however, is the best performer in the experiment, with an average of less than four seconds for all tested client processes.

Table 3.1: Average read time per I/O operation (s)

# of Client process	16	64	116	152
PVFS-based	6.17	11.42	20.38	–
Lustre-based	2.98	4.82	6.15	8.55
HDFS-based (w/o Scheduler)	4.34	6.64	6.94	–
SLAM-based	3.039	3.47	3.91	–

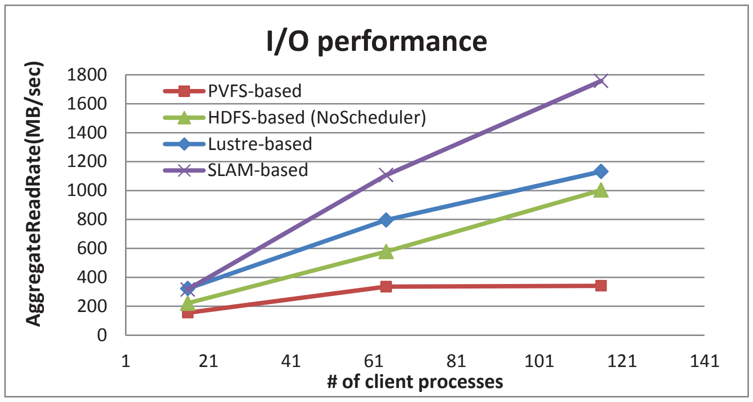


Figure 3.14: Read bandwidth comparison of Lustre, PVFS, HDFS (without scheduler) and SLAM based ParaView.

A read performance comparison is illustrated in Figure 3.14 which illustrates that PVFS and Lustre reach their bandwidth peak at 64 and 116 client processes respectively, after which there will be almost no gain in read performance with any increase in client processes. In fact, it can be derived from Table 3.1 that with 152 client processes, the bandwidth of Lustre is even slightly less than that of 116 client processes. This shows that dedicated storages will reach a bandwidth bottle neck with an increasing number of client processes.

3.6 Related Works

There are many methods that are used in parallel BLAST. ScalaBLAST [67] is a highly efficient parallel BLAST, which organizes processors into equally sized groups and assigns a subset of input queries to each group. It can use both distributed memory and shared memory architectures to load the target database. Lin *et. al.* [55] developed another efficient data access method to deal with data initial preparation and result merging in memory. MR-MPI-BLAST [81] is a parallel BLAST application, which employs MapReduce-MPI library developed by Sandia Lab. These parallel applications can benefit from the flexibility and efficiency of the HPC programming model while still following a compute-centric model.

AzureBlast [60] is a case study of developing scientific applications such as BLAST on the cloud. CloudBLAST [61] adopts a MapReduce paradigm to parallelize genome index and search tools and manage their executions in the cloud. Both AzureBlast and CloudBLAST only implement query segmentation but exclude database segmentation. Hadoop-BLAST [7] and bCloudBLAST [63] present a MapReduce-parallel implementation for BLAST but don't adopt existing advanced techniques like collective I/O as well as computation and I/O coordination. Our SLAM is orthogonal to these techniques, as it allows parallel BLAST applications to benefit from data locality exploitation in HDFS and exploit the flexibility and efficiency of the MPI programming model.

There exists methods that are used to improve the parallel I/O performance. iBridge [99] uses solid state drives to serve request fragments and bridge the performance gap between serving fragments and serving large sub-requests. TCIO [95] is a user-level library and allows program developers to incorporate the collective I/O optimization into their applications. C. Sigovan *et. al.* [77] presents a visual analysis method for I/O trace data that takes into account the fact that HPC I/O systems can be represented as networks. R. Prabhakar *et. al.* [72] propose a disk-cache and parallelism aware I/O scheduling to improve storage system performance. The FlexIO [101] is a middleware that offers simple abstractions and diverse data movement methods to couple simulation with analytics. Sun *et. al.* [93] propose a data replication scheme(PDLA) to improve the performance of parallel I/O systems. Janine *et. al.* [17] developed a platform that realizes efficient data movement between in-situ and in-transit computations that perform on large-scale scientific simulations. Haim Avron *et. al.* [16] develop an algorithm that uses a memory management scheme and adaptive task parallelism to reduce the data-movement costs. In contrast to these methods, our SLAM uses an I/O middleware to allow parallel applications to achieve scalable data access with an underlying distributed file system.

The data locality provided by a data-intensive distributed file system is a desirable feature to improve I/O performance. This is especially important when dealing with the ever-increasing amount of data in parallel computing. VisIO [64] obtains a linear scalability of I/O bandwidth for ultra-scale visualization by exploiting the data locality in HDFS. The VisIO implementation calls the HDFS I/O library directly from the application programs, which is an intrusive scheme and requires significant hard coding. Mesos [44] is a platform for sharing commodity clusters between multiple diverse cluster computing frameworks. Mesos shares resources in a fine-grained manner, allowing frameworks to achieve data locality by taking turns reading data stored on each machine. The aforementioned data movement solutions work in different contexts from SLAM.

CHAPTER 4: PARALLEL DATA ACCESS OPTIMIZATION

In this chapter, we will present a complete analysis for parallel data imbalanced read on distribution file systems. And then we propose novel matching-based algorithms for optimizing parallel read access, referred to as Opass.

4.1 Problems and Theroitical Analysis

In this section, we will formally discuss the severity of the problem for parallel read access on distributed file systems.

4.1.1 Remote Access Pattern Analysis

Assume a set of parallel processes are launched on an m-node cluster with an r-way replication storage architecture to analyze a dataset consisting of n chunks and they are randomly assigned to processes, the probability of reading a chunk locally is r/m (r out of m cluster nodes has the copy). Let X be the random variable denoting the number of files read locally, X has a Binomial Distribution and its *cumulative distribution function* is

$$P(X \leq k) = \sum_{i=0}^{k} \binom{n}{i} \left(\frac{r}{m}\right)^{i} \left(1 - \frac{r}{m}\right)^{n-i}$$

By default, r is equal to 3 in HDFS. Given a $32G$ dataset consisting of 512 chunks, in Figure 4.1, we plot the cumulative distribution function of X for $k = 0, 1, 2, ..., 20$ with cluster sizes of 64,

128, 256 and 512. The probability of reading more than 5 chunks locally is

$$P(X > 5)|_{m=64} = 1 - P(X \leq 5)|_{m=64} = 81.09\%,$$

$$P(X > 5)|_{m=128} = 1 - P(X \leq 5)|_{m=128} = 21.43\%,$$

$$P(X > 5)|_{m=256} = 1 - P(X \leq 5)|_{m=256} = 1.64\%,$$

$$P(X > 5)|_{m=512} = 1 - P(X \leq 5)|_{m=512} = 0.46\%$$

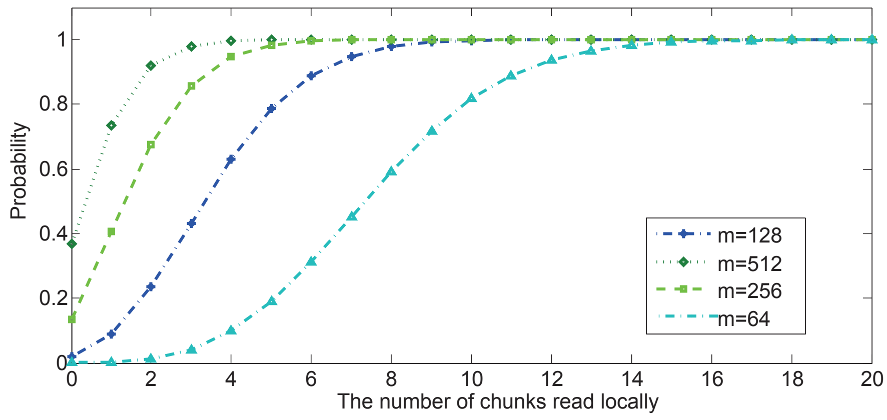


Figure 4.1: CDF of the number of chunks read locally. The cluster size, m, changes from 64 to 512.

We see that the probability of reading data locally exponentially decreases as the size of the cluster increases. Furthermore, with a cluster size $m = 128$, the probability of reading more than 9 chunks locally is about 2%. This implies that almost all data will be accessed remotely in a large cluster.

4.1.2 Imbalanced Access Pattern Analysis

When a chunk must be accessed remotely, the cluster node to serve the data request is chosen from the nodes which contain the required chunk. We assume that these nodes have an equal probability of being chosen to serve the data request. We will show how this policy will result in an imbalance of read access patterns. For a given storage node n_j, let Z be the random variable denoting the number of chunks served by n_j and Y be the number of chunks on n_j. By default, data are randomly distributed within HDFS, so the probability of the node n_j containing a certain chunk is r/m. Thus, the probability that n_j contains exactly a chunks is

$$P(Y = a) = \binom{n}{a} \left(\frac{r}{m}\right)^a \left(1 - \frac{r}{m}\right)^{n-a}$$

Based on the observation in *Remote Access Pattern Analysis*, we can assume that almost all data requests served by n_j are remote requests. For any chunk on n_j, the probability of the process requesting that chunk being served by n_j is $1/r$. Given that n_j contains exactly a chunks, the conditional probability $P(Z \leq k|Y = a)$ is a Binomial cumulative distribution function, and according to the Law of Total Probability, the probability that n_j will serve at most k chunks is

$$\begin{aligned} P(Z \leq k) &= \sum_{a=0}^{n} P(Z \leq k|Y = a)\, P(Y = a) \\ &= \sum_{a=0}^{n} \left(\sum_{i=0}^{k} \binom{a}{i} \left(\frac{1}{r}\right)^i \left(1 - \frac{1}{r}\right)^{a-i} \right) P(Y = a) \end{aligned}$$

Given $r = 3, n = 512,$ and $m = 128$, the expected number of nodes serving at most 1 chunk is $512 \times P(Z \leq 1) = 11$ while the expected number of nodes serving more than 8 chunks is $512 \times (1 - P(z \leq 8)) = 6$, which implies that some storage nodes will serve more than $8X$ the number of chunk requests as others. On the nodes serving 8 chunks, the read requests from different processes will compete for the *hard disk head* and *network bandwidth*, while the nodes serving 1 chunk will be idle while waiting for synchronization. The *imbalanced* data access patterns will result in inefficiency in parallel use of storage nodes/disks and hence a low I/O performance.

4.2 Methodology and Design of Opass

In this section, we first retrieve the data distribution information from the underlying distributed file system and build the processes to data matching with locality relationship in Section 4.2.1. We then find a matching from processes to data through matching based algorithms with the constraints of locality and load balance. Finally, we apply Opass to dynamic parallel data access in heterogeneous environments in Section 4.2.4.

4.2.1 Encoding Processes to Data Matching

Based on the data read policy in HDFS and our analysis in Section 4.1, we can allow parallel data read requests to be served in a balanced way through maximizing the degree of data locality access. To achieve this, we retrieve data distribution information from storage and build the locality relationship between processes and chunk files, where the chunk files will be associated with data processing operators/tasks according to different parallel applications, as discussed in Section 4.2.2 and 4.2.3. The locality relationship is represented as a *Bipartite Matching Graph* $G = (P, F, E)$, where $P = \{p_0, p_1, ..., p_n\}$ and $F = \{f_0, f_1, ..., f_m\}$ are the vertices representing the processes and chunk files respectively and $E \subset P \times F$ is the set of *edges* between P and F. Each edge connecting some $p_i \in P$ and some $f_j \in F$ is configured with a *capacity* equal to the amount of data associated with f_j that can be accessed locally by p_i. If there is no data associated with f_j that is *co-located* with p_i, no edge will be added between the two vertices.

There may be several processes co-located with a chunk since the data set has several copies stored on different cluster nodes. We show a bipartite matching example in Figure 6.3. The vertices on the bottom represent processes, while those on the top represent chunk files. Each edge indicates that a chunk file and a process are co-located on a cluster node. To achieve a high degree of data locality access, we need to find a *one-to-many* matching that contains the largest number of edges. We define a matching in which all of the needed data are assigned to co-located processes as a *full matching*.

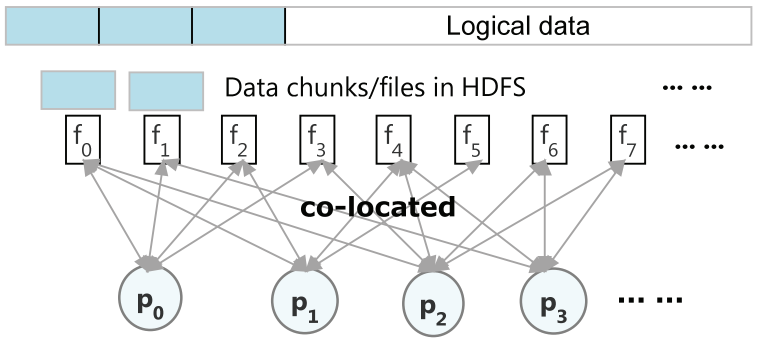

Logical data
Data chunks/files in HDFS
f_0 f_1 f_2 f_3 f_4 f_5 f_6 f_7
co-located
p_0 p_1 p_2 p_3

Figure 4.2: A bipartite matching example of processes and chunk files. The edges represent that the files are located on the same cluster node with the processes.

4.2.2 Optimization of Parallel Single-Data Access

In general, the overall execution time for parallel data analysis will be decided by the longest running process. As mentioned in Section 2.3, applications such as Paraview use a static data assignment method to assign processes with an equal amount of data files so as to adhere to load balancing considerations. Also, each data processing task takes only one data input. We refer to this as *Single-data Access*. We can encode this type of matching problem as a flow network as shown in Figure 4.3.

Assume that we have m parallel processes to process n chunk files, each will be processed only once. First, two vertices, $s, t \notin P \bigcup F$, are added to the process to file matching graph. Then, m edges, each connecting s and a vertex in P are added with equal capacity $TotalSize/m$, where $TotalSize$ is the net size of all data to be processed. Next, edges are added between processes and tasks, each with capacity of the file size. Finally, n edges, each connecting a vertex in F with t, are added, each has a capacity equal to the size of the file that it connects to t. With data stored in HDFS, the file size is equal to or smaller than the setting of the chunk size (by default 64M).

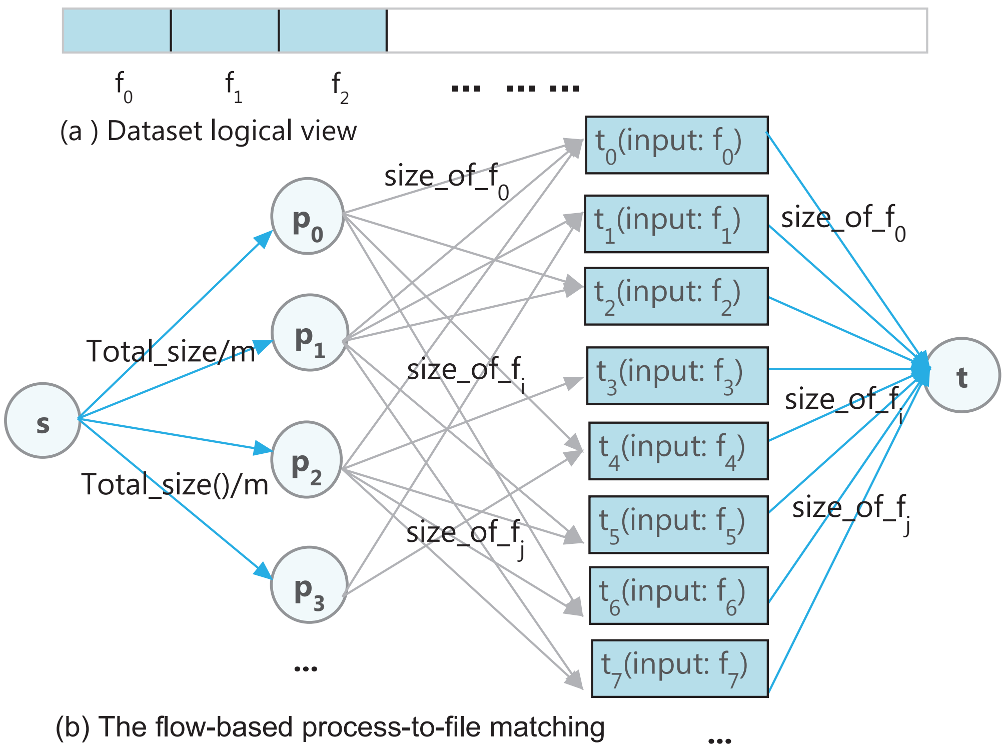


Figure 4.3: The matching-based process-to-file configuration for Single-Data Access in equal-data assignment. Each data processing task has only one input.

In order to achieve a maximum amount of local data reads, we employ the standard max-flow algorithm, Ford-Fulkerson [29], to compute the largest flow from s to t. The algorithm will iterate many times. In each iteration it increases the number of tasks/files assigned to processes. With the use of *flow-augmenting paths* [29], if a task t has been assigned to process i, but the overall size of the graph's maximum matching could be increased by matching t with another process j, the assignment of t to i will be canceled and t is reassigned to j. Such a *cancellation* policy enables the assignments of processes on tasks to be optimal. The formal proof can be found in [29]. The complexity of our implementation of task assignment is $O(nE)$, where n is the number of files and

E is the number of edges in Figure 6.3.

We briefly discuss the maximum matching achieved through the Ford-Fulkerson algorithm. A *maximum matching* is defined as a matching of processes and files with the greatest possible number of edges satisfying the flow capacity constraint. In an ideal situation in which data is evenly distributed within the cluster nodes, a full-matching is achieved. However, in HDFS, there are cases that can cause the data distribution to be unbalanced. For instance, node addition or removal could cause an unbalanced redistribution of data. Because of this, the maximum matching achieved through the flow-based method may be not a full matching, which implies that some processes are assigned less than $TotalSize/m$ of data. To rectify this, we randomly assign unmatched tasks to each such process until all processes are matched to $TotalSize/m$ of data.

4.2.3 Optimization of Parallel Multi-data Access

For a single task with multiple data inputs as shown in Figure 2.4, the inputs may be placed on multiple nodes, which implies that some of the data associated with a given task may be local to the process assigned to that task and some may be remote. Because of this, tasks with multiple inputs will complicate the matching of processes to data. Such a matching problem is related to the stable marriage problem, which however only deals with one-to-one matching [29]. In this section, we propose a novel matching-based algorithm for this type of parallel data access.

Our algorithm aims to associate each process with data processing tasks such that a large amount of data can be read locally. To achieve this, we use the matching information obtained in Section 4.2.1, as shown in Figure 6.3 to find co-located data between tasks and parallel processes. Figure 4.4(a) shows a table that records the tasks, processes, and the size of the data that is co-located between them. We then assign each processes with the equal number of tasks for parallel execution. Based on the co-located data information, we assign tasks to processes such that a task with a large amount data co-located with a process will have a high assignment priority to that process. For instance, task t_4 has the highest priority to be assigned to process P_0 because there is 40 MB of data associated with t_4 that can be accessed locally by P_0. We also allow a task to cancel its current assignment and be reassigned to a new process if that task is associated with more data co-located with the new process than it's current process. Figure 4.4(b) shows a re-assignment event happening on task t_5. t_5 is already assigned to p_2, however when p_3 begins to choose its first task, we find that t_5 and p_3 should a better matching as it has a larger matching value, and we cancel the assignment for p_2 on t_5 and reassign t_5 to p_3.

(a). Matching table based on locality

	p_0	p_1	p_2	p_3
t_0	30 MB	30 MB	10 MB	20 MB
t_1	10 MB	30 MB	30 MB	10 MB
t_2	20 MB	20 MB	30 MB	10 MB
t_3	20 MB	20 MB	20 MB	20 MB
t_4	40 MB	0	20 MB	20 MB
t_5	10 MB	0	40 MB	50 MB
t_6	10 MB	40 MB	10 MB	20 MB
t_7	0	40 MB	40 MB	0

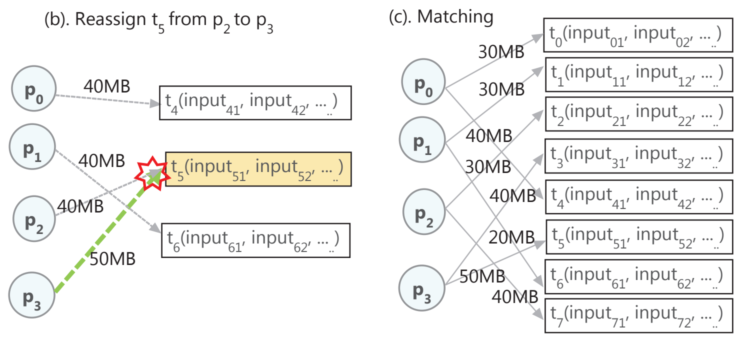

(b). Reassign t_5 from p_2 to p_3
p_0
p_1
p_2
p_3
40MB
40MB
40MB
50MB
t_4(input$_{41}$, input$_{42}$, ...)
t_5(input$_{51}$, input$_{52}$, ...)
t_6(input$_{61}$, input$_{62}$, ...)
(c). Matching
30MB
30MB
40MB
30MB
40MB
20MB
50MB
40MB
t_0(input$_{01}$, input$_{02}$, ...)
t_1(input$_{11}$, input$_{12}$, ...)
t_2(input$_{21}$, input$_{22}$, ...)
t_3(input$_{31}$, input$_{32}$, ...)
t_4(input$_{41}$, input$_{42}$, ...)
t_5(input$_{51}$, input$_{52}$, ...)
t_6(input$_{61}$, input$_{62}$, ...)
t_7(input$_{71}$, input$_{72}$, ...)

Figure 4.4: The process-to-data matching example for Multi-data Assignment.

The method is detailed in Algorithm 6.3.1. Assume that we have a set of n tasks $T = \{t_0, t_1, ..., t_{n-1}\}$ and a set of m parallel processes $P = \{p_0, p_1, ..., p_{m-1}\}$. Each process will be assigned n/m tasks for parallel execution. First, through the matching information in Figure 6.3, we can obtain the local data $d(p_i)$ for each parallel process i, as well as the data $d(t_j)$ needed by each task j. The output of Algorithm 6.3.1 consists of the tasks assigned to each process i, denoted as $T(p_i)$. Similar to the stable marriage problem, our algorithm achieves the optimal matching value from the perspective of each process.

To begin with, we compute the amount of co-located data associated with each task and each process and encode these values as the matching values between them. Then, if there exists a process p_k that is assigned less than n/m tasks, we will find a task t_x with the highest matching value to p_k which has not yet been considered as an assignment by p_k. If t_x has not been assigned to any other process, we assign t_x to process p_k. However, if t_x is already assigned to some other process p_l, we compare the matching values between t_x and p_l and between t_x and p_k. If a greater matching value can be achieved by assigning t_x to p_k, we will reassign t_x to p_k. Finally, we mark t_x as a task that has already been considered by p_k as an assignment. In the worst case, a process could consider all of the tasks as its assignment, thus the complexity of our algorithm is $O(m \cdot n)$, where m is number of processes and n is the number of tasks.

Algorithm 4.2.1 Matching-based Algorithm for Tasks with Multi-data Inputs

```
 1: Let $d(P) = \{d(p_0), d(p_1), ..., d(p_{m-1})\}$ be the set of local data associated with each process.
 2: Let $T = \{t_0, t_1, ..., t_{n-1}\}$ be the set of data operators/tasks.
 3: Let $d(T) = \{d(t_0), d(t_1), ..., d(t_{n-1})\}$ be the set of data associated with each data operator/task.

 4: Let $T(p_i)$ be the set of tasks assigned to process $i$.
Steps:
 5: $m_i^j = |d(p_i) \bigcap d(t_j)|$ // the matching size of co-located data for process $i$ and task $j$
 6: while $\exists p_k : |T(P_x)| < n/m$ do
 7:    Find $t_x$ whom $p_k$ has not yet considered as assignment and $x = \max_x(m_k^x)$
 8:    if $t_x$ has not been assigned then
 9:       Assign $t_x$ to process $k$
10:    else    // $t_x$ is already assigned to $p_l$
11:       if $m_l^x < m_k^x$ then
12:          Add $t_x$ to $T(p_k)$
13:          Remove $t_x$ from $T(p_l)$
14:       end if
15:    end if
16:    Record $t_x$ has been considered as assignment to $P_k$
17: end while
```

4.2.4 Opass for Dynamic Parallel Data Access

For irregular computation patterns such as gene comparison, the execution times of data processing tasks could vary greatly and are difficult to predict according to the input data [56]. To address this problem, applications such as mpiBLAST [56] usually combine task parallelism and SPMD parallelism by adopting a *master process* that controls and assigns tasks to *slave processes*, which will run in parallel to execute data analysis. This can allow for a better load balance in the heterogeneous computing environment. However, since the task assignments made by master processes do not consider the data distribution in the underlying storage, data requests from different processes could also encounter a contention on some storage nodes. In this seciton, we will demonstrate how to adopt our proposed methods to enable the parallel application with dynamic data assignment to benefit from locality access.

Before actual execution, we assume that each process will process the same amount of data. The scheduler process employs our matching based algorithms to compute an assignment $A*$ for each slave process, and will assign tasks to slave processes during execution using the assignment $A*$ as a guideline. The main steps involved are as follows.

1. Before execution, the scheduler process calls our matching based algorithm to obtain a list of task assignments for each slave process i, denoted as L_i.
2. When process i is idle and L_i is not empty, the scheduler process removes a task from the list L_i and assigns that task to the process i.
3. When a process i is idle and the list L_i is empty, we pick a task t_x from L_k, where L_k is the longest remaining list and the task t_x in L_k has the largest co-located data size with process i. We assign task t_x to process i and remove t_x from L_k.

4.3 Experimental Evaluation

We have conducted a comprehensive testing of Opass on both benchmark applications and well-known parallel applications on Marmot. *Marmot* is a cluster of the PRObE on-site project [38] that is housed at CMU in Pittsburgh. The system has 128 nodes / 256 cores and each node in the cluster has dual 1.6GHz AMD Opteron processors, 16GB of memory, Gigabit Ethernet, and a 2TB Western Digital SATA disk drive. For our experiments, all nodes are connected to the same switch.

On Marmot, MPICH [1.4.1] is installed as parallel programming framework on all compute nodes running CentOS55-64 with kernel 2.6. The Hadoop distributed file system (HDFS) is configured as follows: one node for the NameNode/JobTracker, one node for the secondary NameNode, and other cluster nodes as the DataNode/TaskTracker. HDFS is configured as normal with 3-way replication and the size of a chunk file is set as 64 MB. When reading data, the client will attempt to read from a local disk. If the required data is not on a local disk, the client will read data from another node that is chosen at random.

4.3.1 Opass Evaluation

4.3.1.1 Evaluating Opass on Parallel Single-Data Access

To test Opass on applications that implement the equal data assignment method, we instruct parallel processes to read a dataset from HDFS via two methods. The first method, in which the data assignment of each process is mainly decided by its process rank, is used by ParaView. The second method is our proposed method: Opass. Our test dataset contains approximately ten chunk files for every process. Note that this is an arbitrary ratio that could be changed without affecting the performance of Opass. We record the I/O time taken to read each chunk file and we show the comparison of three metrics, the *average* I/O time taken to read all chunk files, the *maximum* I/O time and the *minimum* I/O time in Figure 4.5, 4.6 and 4.7.

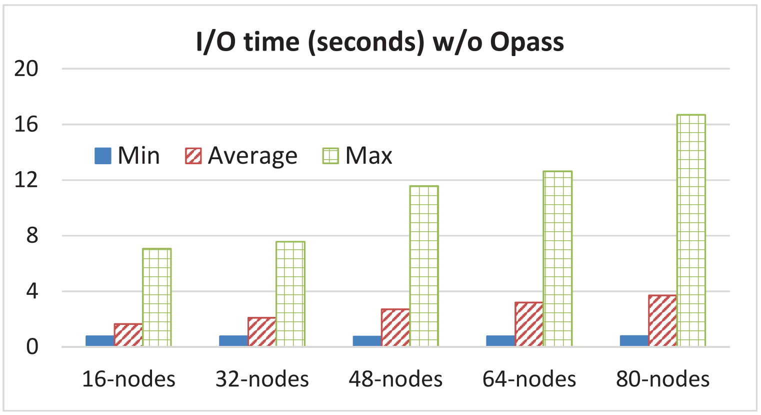


Figure 4.5: Access data from HDFS w/o Opass.

As we can see in Figure 4.5, without the implementation of Opass, the I/O time becomes more varied as the cluster size increases. For instance, on a 16-node cluster, the maximum I/O time to read a chunk file is 9X that of the minimum. However, on an 80-node cluster, this value becomes 21X. Moreover, the maximum I/O time increases dramatically while the minimum I/O time remains relatively constant. This is not desirable for parallel programs, since the longest operation will prolong the overall execution. With the use of Opass, as shown in Figure 4.6, the I/O performance remains relatively constant as the cluster size scales, with an average I/O time of around 0.9 seconds. We attribute this improvement to the fact that without the use of Opass, more than 90% of the data are accessed remotely. The detailed analysis will be presented along with Figure 4.8and 4.10.

To gain further insight into the I/O time distribution, we plot the I/O time taken to read every chunk on a 64-node cluster, which contains 640 chunks and the size of each chunk is approximately 64 MB. The execution results are shown in Figure 4.7. The figure shows that without the use of Opass, the I/O time increases dramatically after the initiation of the execution. This is due to the fact that as the application runs, an increasing number of data requests are issued from parallel processes to storage, which causes contention on disks and the network transfer on some storage nodes. In contrast, with the use of Opass, the I/O time during the entire execution is approximately one to two seconds. In all, the average I/O operation time with the use of Opass is a quarter of that without Opass.

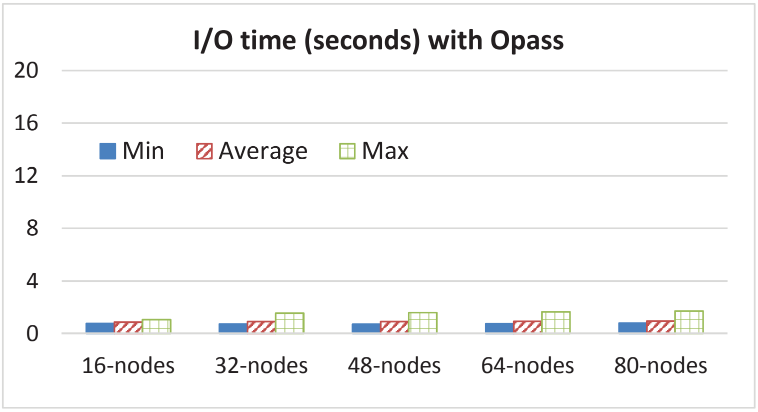


Figure 4.6: Access data from HDFS with Opass.

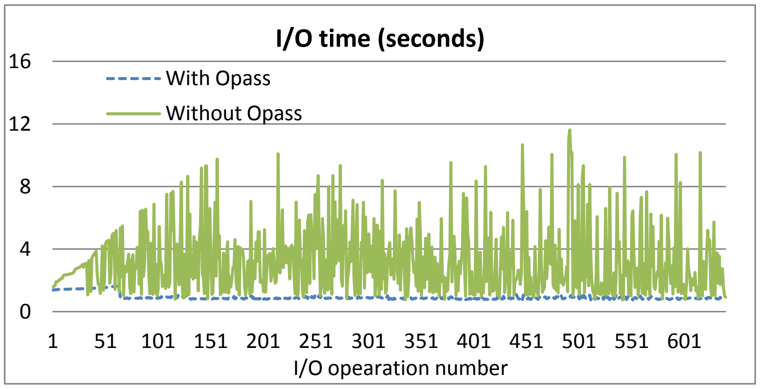


Figure 4.7: I/O time on a 64-node cluster.

To study the balance of data access between cluster nodes, we implement a monitor to record the amount of data served by each storage node. We show the comparison of three metrics in

Figure 4.8and 4.9: the *average* amount of data served by each node as well as the *maximum* and *minimum* amount of data served by a node. As we can see from Figure 4.8, the imbalance of data access becomes more serious as the size of the cluster increases with the comparison to that of using Opass as shown in Figure 4.9. For instance, on an 80-node cluster, the maximum amount of data served by a node is 1500 MB while the minimum is 64 MB.

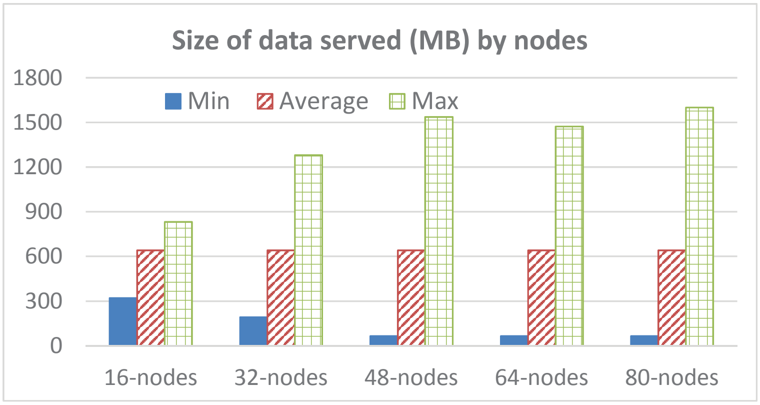


Figure 4.8: Access patterns on HDFS w/o Opass.

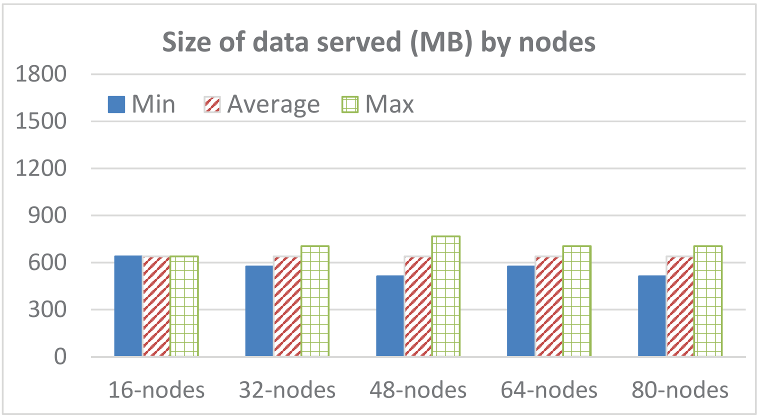


Figure 4.9: Access patterns on HDFS with Opass.

We also plot the amount of data served by each node on a 64 node cluster in Figure 4.10. We find that the amount of data served per node varies greatly with the use of the static assignment. Some nodes, such as node-44, serve more than 1400 MB of data while some node serves 64 MB. Such an imbalance will cause the disk head to become a bottleneck, and thus the I/O read time could increase, as shown in Figure 4.7. This also confirms our analysis in Section 4.1. In contrast, with the use of Opass, every storage node serves approximately 640 MB.

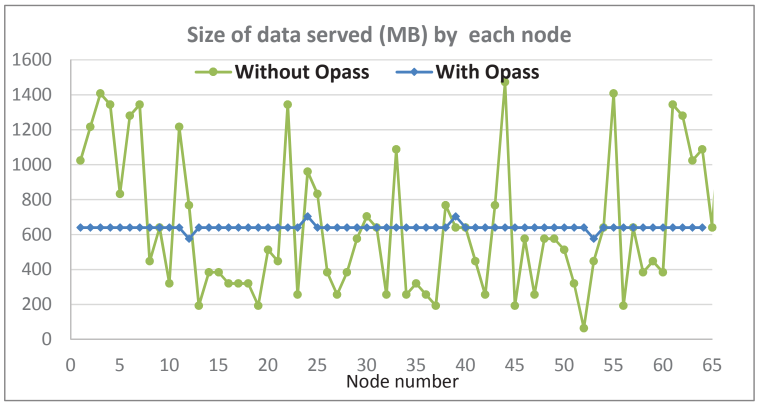


Figure 4.10: Access patterns On a 64-node cluster.

4.3.1.2 *Evaluating Opass on Parallel Multi-Data Access*

We conduct experiments to test parallel tasks with multi-data inputs. Each task includes three inputs, one 30 MB data input, one 20 MB input, and one 10 MB input. These three inputs belong to three different data sets. Again, we evaluate two assignment methods, the default assignment method and Opass, on a 64 node cluster containing 640 chunk files. We record the I/O time taken to read each chunk file and plot the data in Figure 4.11. The improvement of Opass in this test is not as great as that in Figure 4.7. This is because each task involves several data inputs that are distributed throughout the cluster; therefore, to execute a task, part of data must be read remotely. In all, the average time cost on each I/O operation is 2 times less than that with the use of the default dynamic assignment method.

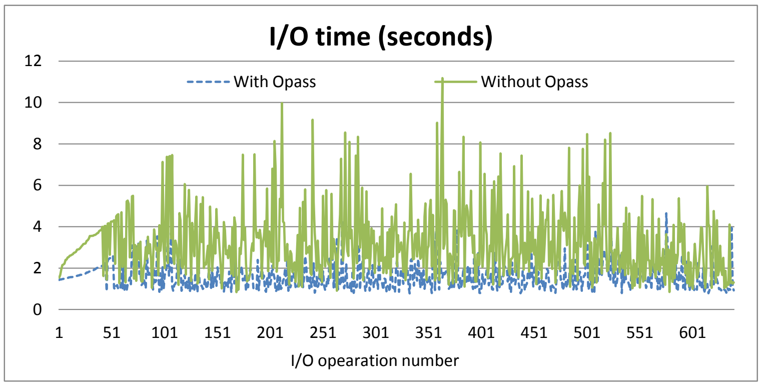


Figure 4.11: I/O times of tasks with multiple inputs on a 64-node cluster.

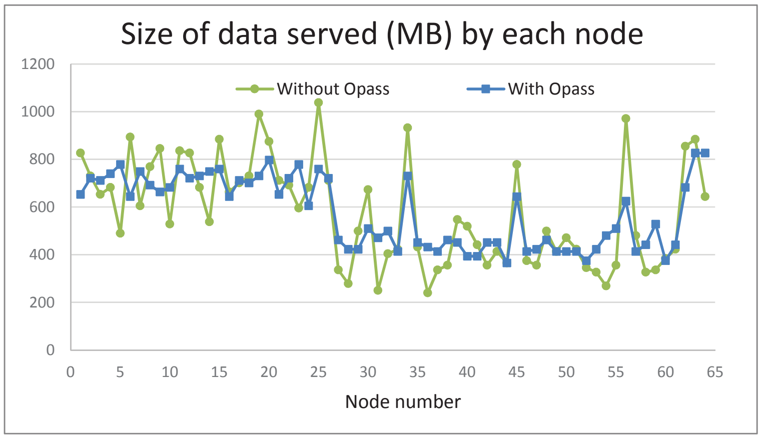


Figure 4.12: Access patterns of multi-inputs on a 64-node cluster.

We also record the the amount of data served by every node on the cluster and plot the results in Figure 4.12. While the balance of data access between nodes is improved with the use of opass, the change is not nearly as dramatic as with the equal data assignment and dynamic data assignment tests. Because the three inputs needed by a task are not always found on that task's local disk, Opass will not be able to optimize all of the data assignments and some processes will read data remotely.

4.3.1.3 Evaluating Opass on Dynamic Parallel Data Access

For the dynamic data access tests, we allow a master process to control the task assignments with an architecture similar to that of mpiBLAST [56]. The master process assigns tasks to slave process, which access data from storage nodes and issue data requests via a random policy to simulate the irregular computation patterns in parallel computing. As with the Equal data assignment test, we evaluate two assignment methods on a 64 node cluster containing 640 chunk files. The first method is the default dynamic data assignment and the second is Opass. The execution results are shown in Figure 4.13. The results obtained from these tests are similar to those of the equal data assignment tests shown in Figure 4.7. For the execution with the use of Opass, the average time on each I/O operation is 2.7 times less than with use of the default dynamic assignment method.

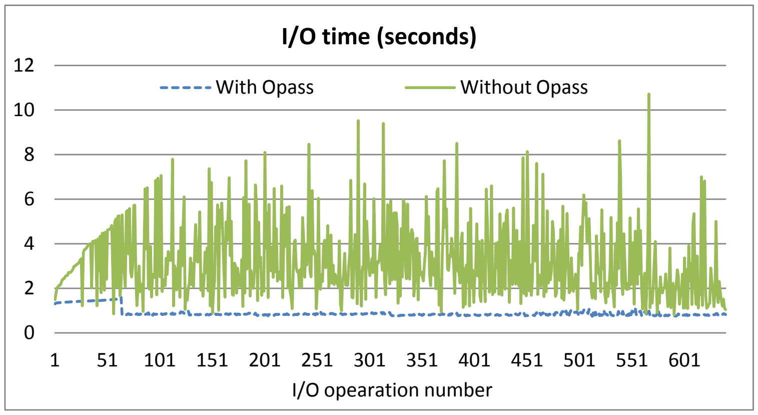


Figure 4.13: I/O times of dynamic data assignment on a 64-node cluster.

4.3.2 Efficiency and Overhead Discussion

4.3.2.1 Efficiency of Opass

With the comparison of Figure 4.7, 4.11, and 4.13, we find that the improvements of I/O time and data access balance vary between the three tests. This is due to the different I/O requirement for different parallelism. For multiple data inputs, parallel processes need to access part of the data remotely. Thus, the I/O performance improvements are not as great. We can conclude that if a data processing task involves too many inputs, our method may not work as well and data reconstruction/redistribution [76] may be needed.

Since Opass does not modify the design of HDFS, HDFS still control how the data requests should be served. Unlike a supercomputer platform, clusters are usually shared by multiple applications. Thus, Opass may not greatly enhance the performance of parallel data requests due to the adjustment of HDFS. However, Opass allows the parallel data requests to be served in an optimized way as long as the cluster nodes have the capability to deliver data in the fashion of locality and balance.

4.3.2.2 Scalability

With our current experimental settings, we found that the overhead created by the matching method was less than 1% of the overhead involved with accessing the whole dataset. However, as the problem size becomes extremely large, the matching method may not be scalable. We leave this problem as a future work, since the scheduling scalability issue is less important compared to the actual data movement. For instance, in our test, reading a single chunk file remotely could take more than 2 seconds, the worst case being 12 seconds. With the use of our method, each file read request could be in finished around 1 second, and thus the overall performance could be greatly improved. Also, many study [46, 96] shows that scheduling scalability is not an critical issue for data-analysis applications.

4.4 Related Works

Distributed file systems are ever-increasing in popularity for data-intensive analysis and they are usually deployed within cluster nodes for the sake of avoiding data movement. HDFS is an open source implementation of the Google File System. Many researches have been proposed to use the Hadoop system for parallel data processing. Garth [30] and Sun [48, 89] propose methods to write parallel data into HDFS and achieve high I/O performance. MRAP [76] is proposed to reconstruct scientific data according to data access patterns to assist data processing using Hadoop system. Sci-Hadoop [22] allows scientists to specify logical queries over array-based data models. VisIO [64] and SLAM [90] obtain high I/O performance for ultra-scale visualization with using HDFS. The aforementioned methods work in different ways than Opass, which systematically studies the problem of parallel data read access on HDFS and solve the remote and imbalanced data read using novel matching based algorithms.

There are scheduling methods and platforms to improve data locality computation. Delay scheduling [96] allows tasks to wait for a small amount of time for achieving locality computation. Quincy [46] is proposed to schedule concurrent distributed jobs with fine-grain resource sharing. Yarn [82], the new generation of Hadoop system, supports both MPI programs and MapReduce workloads. Mesos [44] is a platform for sharing commodity clusters between MPI and Hadoop jobs and guarantees performance isolation for these two parallel execution frameworks. These methods mainly focus on managing or scheduling the distributed cluster resources and our method is orthogonal to them, which allows parallel data read requests to be served in a balanced and locality-aware fashion on HDFS.

CHAPTER 5: DATA LOCALITY APIs AND THE PROBABILITY SCHEDULER

In this chapter, we present a general approach to enable data locality computation for parallel applications and refer to it as DL-MPI. We firstly present a data locality API to allow MPI-based programs to retrieve data distribution information from the underlying distributed file system. The proposed API is designed to be easily integrated into existing MPI-based applications or new MPI programs. In addition, we discuss the problem of data processing task assignment based on the data distribution information among compute nodes and formalize it into an integer programming problem. To balance data locality computation and the parallel execution time in heterogeneous environments, we propose a novel probability scheduler algorithm, which schedules data processing tasks to MPI processes through evaluating the unprocessed local data and the computing ability of each compute node.

5.1 DL-MPI Design and Methodologies

In this section, we will present the design and methodologies of DL-MPI. After giving our design goals and system architecture, we describe an API for MPI-based programs to retrieve the data distribution information among nodes from a distributed file system. We also discuss data resource allocation, which is involved in the assignment of data processing tasks to compute processes.

5.1.1 Design Goals and System Architecture

We aim to provide a generic approach for enabling MPI-based data-intensive applications to achieve data locality computation using a distributed file system. There are two main issues we need to address. 1). The MPI programming model doesn't have an interface that allows MPI programs to retrieve data distribution information from underlying storage. 2). An effective and efficient scheduler to assign data processing tasks to parallel processes in a heterogeneous running environment is needed.

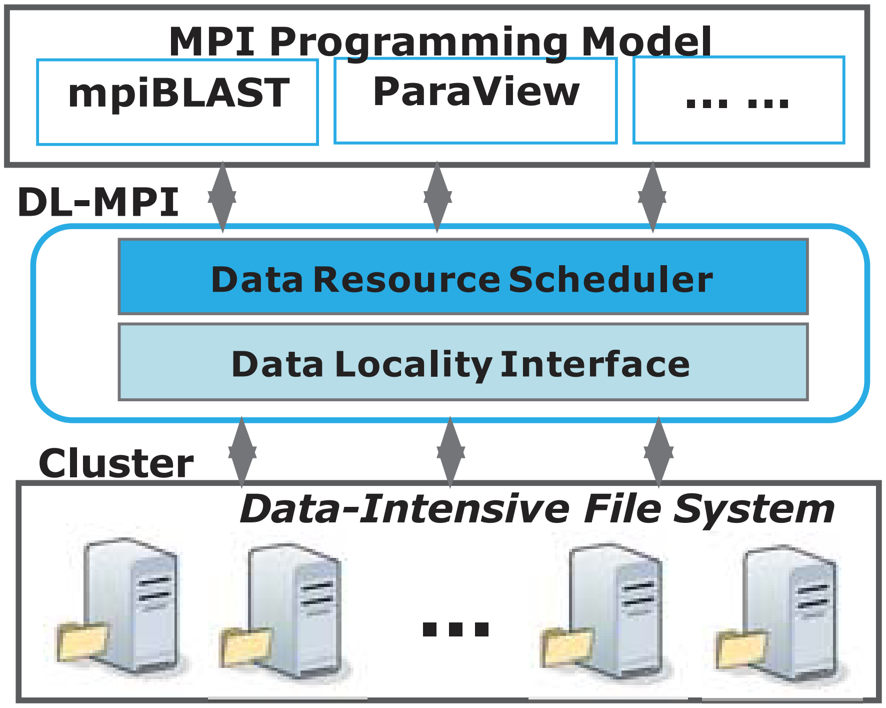


Figure 5.1: Two new modules are used to assist MPI-based programs in accessing the distributed file system and intelligently read data with awareness of data locality.

To address these difficulties, we propose an API, which is convenient for MPI-based programs to retrieve data distribution information from an underlying distributed file system. In addition, a scheduler algorithm based on probability is proposed to determine data to process scheduling. The scheduler aims to balance parallel execution time and data locality computation through evaluating the unprocessed data and data processing speed of each computing node.

The DL-MPI system consists of two important components, a Data Locality Interface and a Data

Resource Scheduler as illustrated in Figure 5.1. The data to be processed is stored, along with several copies, in a distributed file system, which supports scalable data access. Through our DL-MPI system, the MPI-based application on top could do complexity computation and achieve data locality computation with HDFS as the underlying storage system.

5.1.2 Data Locality Interface for MPI-based Programs

In this section, we describe the API for retrieving data distribution information from an underlying distributed file system deployed in a disk-attached cluster.

As we discussed, with traditional MPI-based programming architectures, data locality is not considered and it is not necessary to have a data location querying interface. However, with the co-located storage and analytic processes, data locality should be considered to gain high I/O performance, especially for massively parallel applications with big data as input. Our data locality interface is proposed to enable MPI-based programs to take advantage of locality computation in data-centric architectures. By allowing MPI-based programs to query locality information, the programs can efficiently map compute processes to data.

We show a typical example of partitioning data across involved MPI processes as follows, where each MPI process statically calculates its accessing offset based on the rank.

- MPI_Comm_rank(..., &rank);
- offset = rank * Avg_access;
- MPI_File_open(..., &fh);
- MPI_File_read_at(fh, offset...);

where $rank$ is the process id, Avg_access is access range per process and fh is the file pointer. When such applications are compute-intensive, the performance will not be affected by the data partitioning and assignment. However, when the applications become data-intensive, the data locality is more relevant because more data potentially needs to be transferred to compute nodes. Thus, a better I/O performance could be achieved by knowing the location of data fragments and assigning MPI processes the fragments that are local to them.

We chose to build our API on the Hadoop Distributed File System (HDFS) for this study, which is actually designed for MapReduce programs. The API set for data locality computation is summarized in Table I. To retrieve the local chunks mapping to a process on a specific dataset, a retrieval function (*DLI_map_process_chunks*) is required, which computes the data range of a given dataset that is local to a given process. "Chunk" is the storage unit (64 MB by default) in HDFS. Another basic function is *DLI_map_chunks_process*, which builds the map of all processes to their local chunks on a given dataset. In order to get general distribution of a dataset, these functions include *DLI_get_total_chunks, DLI_get_data_percentage*, which help obtain the number of chunks local to a process and the percentage of a dataset co-located with a given process, respectively. Moreover, the function of *DLI_data_check* is used to check whether a given offset of a dataset is local to a given process and the function of *DLI_map_Dprocess_Dchunks* is to return the location information of all the files under a directory. These functions give client applications the ability to make scheduling decisions based on data locality, which can reduce data movement over the network.

To show how the API is executing for a call from a client application, we take the *DLI_map_process_chunks* function as an example, and demonstrate how it works. For a given file name, the steps involved are: 1) retrieve the file id based on the file name; 2) get the chunk size and calculate the range of each HDFS chunk; 3) for each chunk, retrieve a list of datanodes with that chunk on its local hard disk; 4) store the chunk $index$ for each chunk found in step 3, that is local to the given process; 5) write the chunk-map-process information to the buffer. Currently, we take the data as flat-files and more functions are needed for dealing with high-level data format like NetCDF and HDF5.

Table 5.1: Descriptions of data locality API in DL-MPI

int DLI_map_process_chunks(char, char*, void*)* Retrieves the chunk list of a dataset co-located with a given process The arguments are dataset/file name, process name and return list buffer
int DLI_map_chunks_process(char, char**, void*)* Builds the map of chunks local to a group of compute processes given a dataset The arguments are dataset/file name, processes name list, and return map buffer
int DLI_map_Dprocess_Dchunks(char, char**, void*)* Builds the maps of chunks local to a group of processes given a directory The arguments are directory name, processes name list and return map buffer
int DLI_get_total_chunks(char, char*, int*)* Retrieves the total number of chunks of a dataset local to a given process The arguments are dataset/file name, process name and return number buffer
int DLI_get_data_percentage(char, char*, double*)* Retrieves the percentage of a dataset local to a given process The arguments are dataset/file name, process name and return buffer
int DLI_check_data(char, size_t*, bool*)* Check whether a given offset of a dataset is local to a given process The arguments are dataset/file name, offset and return buffer

5.1.3 Data Resource Allocation

To achieve data locality computation for data-intensive applications, not only the information of data distribution among nodes is required, but also an effective algorithm for mapping data to processes is needed. In this section, we discuss how to efficiently assign data processing tasks to processes.

5.1.3.1 Naive Data Locality Algorithm

To allow a process to achieve data locality computation, we present a naive data locality scheduler in Algorithm 5.1.1. The naive algorithm takes the chunks and nodes distribution as input. The output of the algorithm is the assignment of a data processing task for a process.

The assignment policy is straight-forward: whenever a process requests a new task, we greedily assign the first task which has access to a local chunk. If the process has no unprocessed local chunks, we assign an arbitrary task to that process.

Algorithm 5.1.1 Naive Data Locality Scheduler Algorithm

```
 1: Let $C$ = $\{c_1, c_2, ..., c_n\}$ be the set of participating nodes
 2: Let $D$ = $\{d_1, d_2, ..., d_m\}$ be the set of unprocessed data chunks;
 3: Let $D_i$ be the set of unprocessed data chunks located on node $i$;
Steps:
 4: while $|D| \neq 0$ do
 5:   if an mpi process on node $i$ requests a new task then
 6:     if $|D_i| \neq 0$ then
 7:       Randomly choose a $d_x \in D_i$
 8:       Assign $d_x$ to the process on node $i$
 9:     else
10:       Randomly choose a $d_x \in D$
11:       Assign $d_x$ to the process on node $i$
12:     end if
13:     Remove $d_x$ from $D$
14:     for all $D_k$ s.t. $d_x \in D_k$ do
15:       Remove $d_x$ from $D_k$
16:     end for
17:   end if
18: end while
```

There exist several heterogeneity issues that could potentially result in low execution performance for such a greedy strategy. First, the HDFS random chunk placement algorithm may pre-distribute the target data unevenly within the cluster, leaving some nodes with more local data than others. Second, the execution time of a specific computation task could vary a lot among different nodes, due to the heterogeneous run-time environment, e.g. multiple users and multiple applications. These issues could make the processes in some compute nodes take remote computation and thus have a long I/O wait time.

5.1.3.2 Problem Formalization

As we discussed, in a heterogeneous run-time environment, the execution time on each compute node may vary greatly from one to another. Thus, to achieve high-performance computing, we need to take the parallel execution time into consideration as well. We formally discuss the assignment issue as follows:

Our goal is to assign data processing tasks to parallel MPI processes running on a commodity cluster, such that we can achieve a high degree of data locality computation and minimize the parallel execution time. Let the total number of chunks of a dataset be n, and the total number of nodes be m. To easily discuss our problem, we suppose the process i run on node i. We denote $D = \bigcup_j d_j$, where d_j is the j^{th} chunk, $j \in [1, n]$, and $P = \bigcup_i p_i$ where p_i is the i^{th} process, $i \in [1, m]$. Through the Data Locality Interface, we can identify all the local chunks for the process i. Let

$$L_i = (l_{ik}),\ \text{where } l_{ik} = \begin{cases} 1 & \text{if } d_k \text{ is local to } p_i \\ 0 & \text{otherwise} \end{cases}$$

$$F_i = (f_{ix}),\ \text{where } f_{ix} = \begin{cases} 1 & \text{if } d_x \text{ is processed by } p_i \\ 0 & \text{otherwise} \end{cases}$$

We define the degree of data locality on dataset D being,

$$P(D) = \frac{\sum_{1\leq i\leq m}(L_i^T F_i)}{|D|} \tag{5.1}$$

Suppose the optimal parallel execution time for D being OPT_D and the execution time of process p_i on assigned data F_i being $T(F_i)$. We define the degree of optimization for execution time on dataset D being,

$$P(T) = \frac{OPT_D}{\max\limits_{1\leq i\leq m} T(F_i)} \tag{5.2}$$

The goal of the assignment algorithm is to assign chunks for each process i that maximizes the degree of data locality and the degree of execution time optimization, subject to a constraint that the union of chunks processed by all processes covers the entire D. More formally, we need to solve for F_i in the following optimization problem:

$$\begin{aligned} &\text{maximize } \alpha * P(D) + \beta * P(T) = \\ &\alpha * \frac{\sum_{1\leq i\leq m}(L_i^T F_i)}{|D|} + \beta * \frac{OPT_D}{\max\limits_{1\leq i\leq m} T(F_i)} \end{aligned} \tag{5.3}$$

subject to

$$\sum_{1\leq i\leq m} F_i \geq (1, 1, \cdots, 1) \tag{5.4}$$

Where α and β are the parameters representing weights of the two objectives.

Suppose the process speed of each cluster node could be estimated according to historical execution performance and the time to finish processing the chunks is determined when a F_i is given. The

OPT_D is a constant reference value and given a fixed α and β, e.g. 0.5, the problem can be converted into an integer programming problem and be solved using the CPLEX solver [1].

However, in a heterogeneous environment, the process speed is not fixed but changes with time and we always have a inconsistent evaluated $T(F_i)$ from time to time. To get an effective assignment, we should re-evaluate the running situation every time we assign a task to a process. Thus, a fast and dynamic algorithm is a must.

5.1.3.3 Probability based Data Scheduler

In this section we introduce the probability based data scheduler algorithm that balances data locality computation and the parallel execution time among MPI processes.

The probability scheduler algorithm takes the distribution information of unprocessed data chunks and the data-processing speed of each node as input. The output of the algorithm is the assignment of a data processing task for a process.

The pseudo code of the probability scheduler algorithm is listed in Algorithm 5.1.2. Whenever a process i requests a task, we initially try to launch a task with its requested chunks stored at the node. However, unlike the Naive Data Locality Scheduler Algorithm, which takes all unprocessed chunks with equal probability, we calculate for each candidate chunk d_x the probability to be assigned, based on an estimated execution time of other processes. Due to the replication mechanism of HDFS, for each candidate chunk d_x, we may have other processes j that could take local computation on d_x. We estimate the remaining local execution time of these processes j, as the number of all unprocessed chunks on process j divided by the process speed s_k. We choose the minimum execution time over all these processes j to decide the probability for assigning d_x to process i. This is because, the process with less remaining local execution time will have a larger probability to take remote computation in future. That is,

$$T_{d_x} = \min_{1 \leq k \leq n, d_x \in D_k, k != i} \left(\frac{|D_k|}{s_k}\right) \tag{5.5}$$

The probability of assigning d_x to the requesting process is calculated as the following equation,

$$P(d_x) = \frac{T_{d_x}}{\sum_{X \in D_i} T_X} \tag{5.6}$$

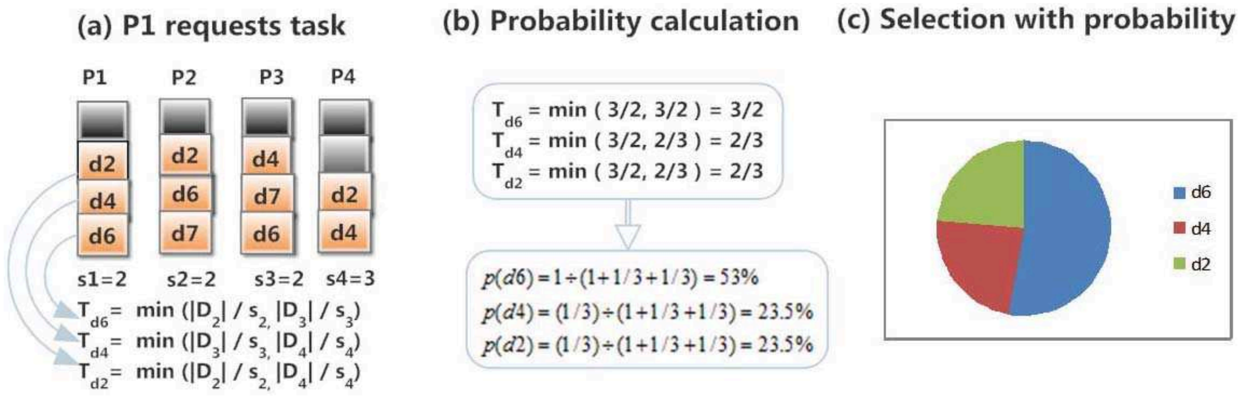
Specifically, upon receiving a task request from a process on node i, the scheduler process determines a task for the process as follows:

- 1. If the node i has some data chunks on its local disk, for each chunk d_x that is on node i disk, the scheduler will calculate its probability of being assigned by estimating the minimum execution time T_{d_x} associated with d_x on other processes. We then assign a chunk to the process based on the probability distribution. An example is shown in Figure 5.2.
- 2. If the node i does not contain any data chunks on its local disk, the scheduler will calculate the probability for all unassigned chunks, and assign the one to the process based on their probability distribution.

The probability assignment algorithm could be implemented in applications with dynamic scheduling algorithms, such as mpiBLAST, in which scheduling is determined by what nodes are idle at any given time. This kind of scheduling adopts a master-slave architecture and the assignment algorithm could be incorporated into master process. The probability assignment algorithm could also be implemented in applications with static scheduling, such as ParaView, which uses static data partitioning so the work allocation can be determined beforehand. For this kind of scheduling, we can assume a round-robin request order for assignment in Step 6 of Algorithm 5.1.2.

Algorithm 5.1.2 Probability based Data Scheduler Algorithm

```
1:  Let $C$ = $\{c_1, c_2, ..., c_n\}$ be the set of participating nodes
2:  Let $S$ = $\{s_1, s_2, ..., s_n\}$ be the data-process speed set of $n$ nodes;
3:  Let $D$ = $\{d_1, d_2, ..., d_m\}$ be the set of unprocessed data chunks;
4:  Let $D_i$ be the set of unprocessed data chunks located on node $i$;
Steps:
5:  while $|D| \neq 0$ do
6:    if an mpi process on node $i$ requests a new task then
7:      Update $s_i$ according to the historical execution performance
8:      if $|D_i| \neq 0$ then
9:        for $d_x \in D_i$ do
10:         $T_{d_x} = \min_{1 \leq k \leq n, d_x \in D_k, k != i} (\frac{|D_k|}{s_k})$
11:       end for
12:       for $d_x \in D_i$ do
13:         $P(d_x) = \frac{T_{d_x}}{\sum_{X \in D_i} T_X}$
14:       end for
15:       Assign $d_x$ to the process on node $i$ with probability $P(d_x)$
16:     else
17:       for $d_x \in D$ do
18:         $T_{d_x} = \min_{1 \leq k \leq n, d_x \in D_k, k != i} (\frac{|D_k|}{s_k})$
19:       end for
20:       for $d_x \in D$ do
21:         $P(d_x) = \frac{T_{d_x}}{\sum_{X \in D_i} T_X}$
22:       end for
23:       Assign $d_x$ to the process on node $i$ with probability $P(d_x)$
24:     end if
25:     Remove $d_x$ from $D$
26:     for all $D_k$ s.t. $d_x \in D_k$ do
27:       Remove $d_x$ from $D_k$
28:     end for
29:   end if
30: end while
```



(a) P1 requests task
P1
P2
P3
P4
d2
d4
d6
d2
d6
d7
d4
d7
d6
d2
d4
s1=2
s2=2
s3=2
s4=3
T_d6 = min (|D2| / s2, |D3| / s3)
T_d4 = min (|D3| / s3, |D4| / s4)
T_d2 = min (|D2| / s2, |D4| / s4)
(b) Probability calculation
T_d6 = min (3/2, 3/2) = 3/2
T_d4 = min (3/2, 2/3) = 2/3
T_d2 = min (3/2, 2/3) = 2/3
p(d6) = 1÷(1+1/3+1/3) = 53%
p(d4) = (1/3)÷(1+1/3+1/3) = 23.5%
p(d2) = (1/3)÷(1+1/3+1/3) = 23.5%
(c) Selection with probability
d6
d4
d2

Figure 5.2: A simple example where the scheduler receives a task request from a process ($P1$). The scheduler finds the available unassigned chunks on $P1$ ($d2$, $d4$ and $d6$ in this example)and calculates the probability to assign for each chunk.

5.2 Experiments and Analysis

5.2.1 Experimental Setup

We conducted comprehensive testing on our proposed DL-MPI at both Marmot and CASS clusters. Marmot is a cluster of the PRObE on-site project [38] that is housed at CMU in Pittsburgh. The system has 128 nodes / 256 cores and each node in the cluster has dual 1.6GHz AMD Opteron processors, 16GB of memory, Gigabit Ethernet, and a 2TB Western Digital SATA disk drive. For our experiments, all nodes are connected to the same switch. CASS consists of 46 nodes on two racks, one rack including 15 compute nodes and one head node and the other rack containing 30 compute nodes, as shown in Table 1.

In both clusters, MPICH $[1.4.1]$ is installed as parallel programming framework. We installed PVFS2 version $[2.8.2]$ on the cluster nodes: one node as the metadata server for PVFS2, and other nodes as the I/O servers. We installed Hadoop $0.20.203$ as the distributed file system, which is configured as follows: one node for the NameNode/JobTracker, one node for the secondary

Table 5.2: CASS cluster configuration

15 Compute Nodes and 1 Head Node	
Make& Model	Dell PowerEdge 1950
CPU	2 Intel Xeon 5140, Dual Core, 2.33 GHz
RAM	4.0 GB DDR2, PC2-5300, 667 MHz
Internal HD	2 SATA 500GB (7200 RPM) or 2 SAS 147GB (15K RPM)
Network Connection	Intel Pro/1000 NIC
Operating System	Rocks 5.0 (Cent OS 5.1), Kernel:2.6.18-53.1.14.e15
30 Compute Nodes	
Make& Model	Sun V20z
CPU	2x AMD Opteron 242 @ 1.6 GHz
RAM	2GB - registered DDR1/333 SDRAM
Internal HD	1x 146GB Ultra320 SCSI HD
Network Connection	1x 10/100/1000 Ethernet connection
Operating System	Rocks 5.0 (Cent OS 5.1), Kernel:2.6.18-53.1.14.e15
Cluster Network	
Switch Make & Model	Nortel Nortel BayStack 5510-48T Gigabit Switch

NameNode, and other nodes as the DataNode/TaskTracker. When running MPI processes, we let the scheduler process run on the node where NameNode is configured, and data processing processes run on DataNodes for the sake of data locality.

5.2.2 Evaluating DL-MPI

In this section, we measure the performance of our DL-MPI using a benchmark application developed with MPI and the proposed API. The benchmark application uses a master-slave implementation of the described scheduling algorithms. Specifically, a single MPI process is developed to dynamically assign data processing tasks to slave processes which execute the assigned tasks. In our benchmark program, we analyze genomic datasets of varying sizes. Since we are more concerned with the I/O performance of DL-MPI, our test programs read all gene data into memory for sequence length analysis and exchange messages between MPI processes.

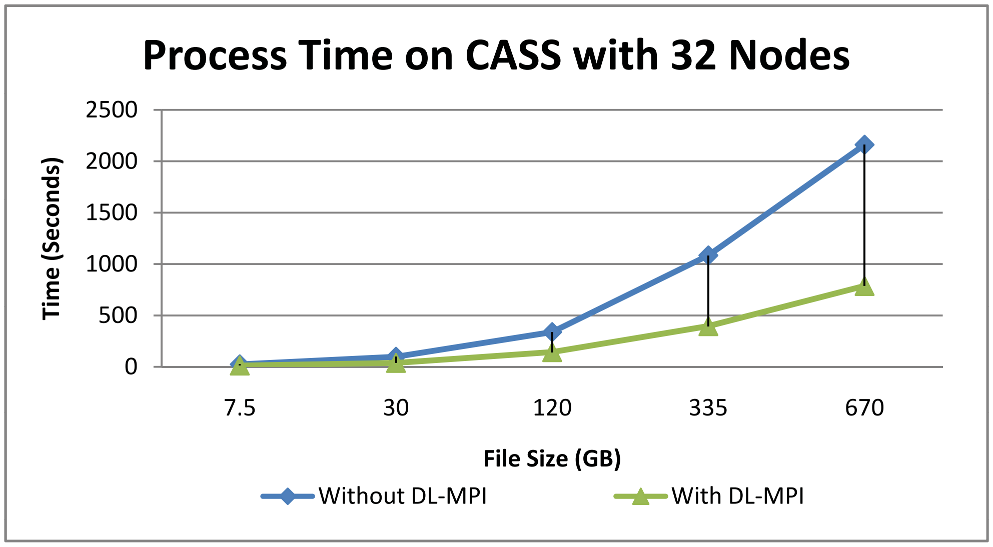


Figure 5.3: Process time Comparison of our benchmark program with and without DL-MPI on CASS using variable file sizes. We see much faster process times for DL-MPI than without DL-MPI.

We firstly compare the process time of our benchmark program using DL-MPI probability based scheduling, referred to as ”With DL-MPI”, and non locality scheduling, referred to as ”Without DL-MPI”, on CASS for variable file sizes. We use 32 nodes for these experiments and show the performance comparison in Figure 5.3. From the figure, we find that the benchmark program using DL-MPI consistently obtains much lower process times than without using DL-MPI. With increased size of input data, the running time increases quickly for the benchmark program without using DL-MPI.

We also compare the bandwidth performance through varying the number of nodes in the Marmot cluster. We measure the bandwidth for processing a 1 TB file and show the bandwidth comparison in Figure 5.4. We find a massive improvement for the benchmark program using DL-MPI as the number of nodes increases, resulting in approximately a four fold increase when the number of nodes reaches around 100. We also see the bandwidth of the benchmark program using PVFS begin to level off at around 60 nodes while the test with DL-MPI scales much better.

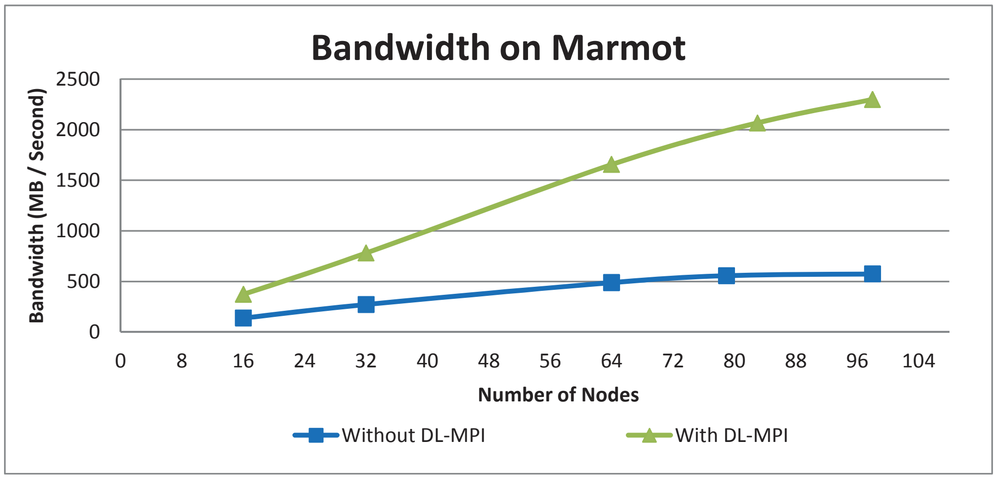


Figure 5.4: Bandwidth comparison of our benchmark program on PVFS and HDFS using DL-MPI on Marmot using variable number of nodes. The bandwidth for DL-MPI scales much better with an increased number of nodes than without DL-MPI

To show the effect of data locality scheduling on bandwidth, we also compare three data processing task assignment algorithms: an Arbitrary Scheduler, a Naive Data Locality Scheduler and a Probability based Data Scheduler. The Arbitrary Scheduler simply schedules the next data chunk, in a queue of unprocessed data chunks, to a slave upon receipt of an idle message. The Naive Data

Locality Scheduler and Probability based Data Scheduler are described in Section 3.3.

We use HDFS as the storage for the three schedulers and vary the file sizes on CASS. We use 32 nodes for these tests and the result is shown in Figure 5.5. We find that the bandwidth obtained by Probability scheduling algorithm outperforms a factor of 2.5 than the Arbitrary scheduling. In addition, we find that the Probability scheduler can always achieve higher bandwidth than the Naive scheduler. This indicates that the Probability scheduler can achieve higher degree of data locality computation. The bandwidth obtained by both the Probability scheduler and the Naive scheduler has a high increasing rate when the file size is smaller than 125 GB. As the size of the file grows, the Probability scheduler obtains nearly constant bandwidth and Naive scheduler has a slightly increasing bandwidth. This is because more local data is available to the parallel processes as file size increases.

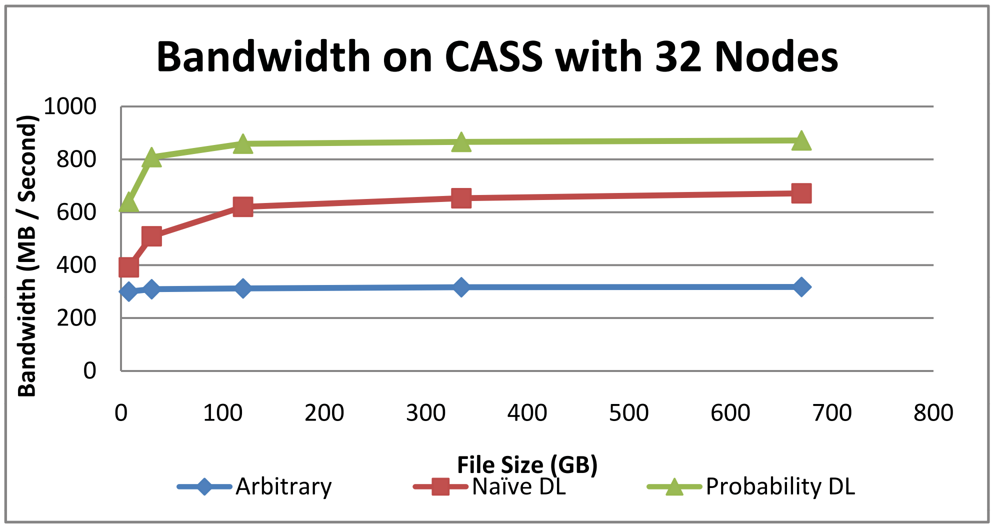


Figure 5.5: Bandwidth comparison of our benchmark program using several schedulers on CASS with 32 nodes and varying file size. We see nearly constant bandwidth for all scheduling algorithms, but much higher performance of DL-MPI over arbitrary scheduling.

5.3 Related Works

With the explosive growth of data, numerous research works are presented to solve the data movement or data management. Janine *et. al.* [17] developed a platform which realizes efficient data movement between in-situ and in-transit computations performed on large-scale scientific simulation data. Wu *et. al.* developed scalable performance data extraction techniques, and combined customization of metric selection with on-the-fly analysis to reduce the data volume in root cause analysis [87]. The ISABELA-QA project compressed the data and then created indexing of the compressed data for query-driven analytics [15]. Haim Avron *et. al.* [16] analyzed the data-movement costs and memory versus parallelism trade-offs in a shared memory parallel out-of-core linear solver for sparse symmetric systems. They developed an algorithm that uses a memory management scheme and adaptive task parallelism to reduce the data-movement costs. To support runs of large-scale parallel applications in which a shared file system abstraction is used, Zhang *et. al.* [100] developed a scalable MTC data management system to efficiently handle data movement. In summary, these approaches alleviate the network transfer overhead for the specific applications. However, they cannot completely eliminate the overhead as our proposed solution.

SciMATE [84] is a framework that is developed to improve the I/O performance by allowing scientific data in different formats to be processed with a MapReduce like API. Kshitij *et. al.* [62] developed a new plugin for HDF5 using PLFS to convert the single-file layout into a data layout that is optimized for the underlying file system. Jun *et. al.* [43] demonstrated how patterns of I/O within scientific applications can significantly impact the effectiveness of the underlying storage systems and utilized the identifying patterns to improve the performance of the I/O stack and mitigate the I/O bottleneck. These methods improve the I/O performance though using data access regularity. While our DL-MPI improves the I/O performance by capitalizing on data locality computation.

The data locality provided by a data-intensive distributed file system is a desirable feature to improve I/O performance. This is especially important when dealing with the ever-increasing amount of data in parallel computing. Mesos [44] is a platform for sharing commodity clusters between multiple diverse cluster computing frameworks. Mesos shares resources in a fine-grained manner, allowing frameworks to achieve data locality by taking turns reading data stored on each machine. CloudBLAST [61] adopts a MapReduce paradigm to parallelize gnome index and search tools and manage their executions in the cloud. VisIO [64] obtains a linear scalability of I/O bandwidth for ultra-scale visualization by exploiting data locality of HDFS. VisIO implementation calls the HDFS I/O library directly from the application programs, which is an intrusive scheme and requires significant hard coding effort. The aforementioned data movement solutions work in different contexts from DL-MPI.

CHAPTER 6: SUB-DATASET ANALYSIS OPTIMIZATION IN HADOOP CLUSTERS

In this chapter, we study the problem of sub-dataset analysis over distributed file systems, e.g, the Hadoop file system. Our experiments show that the sub-datasets' distribution over HDFS blocks can often cause the corresponding analysis to suffer from a seriously imbalanced parallel execution. This is because the locality of individual sub-datasets is *hidden* by the Hadoop file system and the content clustering of sub-datasets results in some computational nodes carrying out much more workload than others. We conduct a comprehensive analysis on how the imbalanced computing patterns occur and their sensitivity to the size of a cluster. We then propose a novel method to optimize sub-dataset analysis over distributed storage systems referred to as DataNet. DataNet aims to achieve distribution-aware and workload-balanced computing and consists of the following three parts. Firstly, we propose an efficient algorithm with linear complexity to obtain the meta-data of sub-dataset distributions. Secondly, we design an elastic storage structure called ElasticMap based on the HashMap and BloomFilter techniques to store the meta-data. Thirdly, we employ a distribution-aware algorithm for sub-dataset applications to achieve a workload-balance in parallel execution.

6.1 Content Clustering and Sub-datasets Imbalanced Computing

6.1.1 Sub-Datasets Analysis and Content Clustering

Collecting and analyzing log or event data is important for gaining business intelligence and ensuring system security. For example, the well-known distributed log collection system Flume [2] can directly save log data into a Hadoop File System for distributed analysis. Log or event-based datasets are usually lists of records, each consisting of several fields such as source/user id, log time, destination, etc. To discover knowledge, these data need to be further filtered for individual analysis. Specifically, the sub-dataset $S(e)$ related to a specific event or topic analysis e could be represented as follows,

$$S(e) = \{r | related(r, e), r \in R\} \tag{6.1}$$

where R is the collection of all log records.

Researches [65, 14, 53, 50, 27, 47] have shown that sub-datasets pertaining to related topics or features will often be clustered together in most large datasets, e.g, the majority of logs for a popular movie would be concentrated around the time of its release. Also, photos or videos recently uploaded to Facebook [65] are often retrieved/commented on at a much higher rate than older ones. In a social network such as LinkedIn or Twitter, users are prompted to group themselves with others sharing similar skills or interests[14]. Moreover, in graph processing, graph partitioning technologies tend to place highly connected nodes in a single partition and the nodes containing relatively few edges in separate partitions in order to reduce communication between partitions [53, 50, 27].

However, in parallel data analysis applications such as MapReduce, scheduling tasks based on block granularity [85] without the consideration of the sub-datasets' distribution does not result in an optimal scheduling for parallel execution. In the next section, we will present an analysis of the imbalanced workload for sub-dataset processing in parallel execution.

6.1.2 Probability Analysis of Imbalanced Workload

Assume a set of parallel processes/executors are launched on an m-node cluster to analyze a specific sub-dataset S, which is distributed among n block files. Due to content clustering, blocks will contain different amounts of data from each sub-dataset. In most situations, the majority of a given sub-dataset is contained in only a few blocks, while other blocks may contain little data related to the sub-dataset. We can model such a distribution using a Gamma distribution, which is widely used to model physical quantities [52, 39] that take positive values such as information or message distribution over time. In our analysis, we let the amount of data contained by each block, X, follow a Gamma distribution $X \sim \Gamma(k, \theta)$, and assume that each X for different blocks is independent. To theoretically discuss the issue of workload imbalance, we suppose that each cluster node performs the analysis on an equal amount of n/m randomly chosen blocks. By taking the summation of the independent random variables X for each block, we obtain the amount of workload processed on a cluster node, Z, which has the distribution $Z \sim \Gamma(\frac{nk}{m}, \theta)$ and its density function is

$$f(z; \frac{nk}{m}, \theta) = \frac{1}{\Gamma(\frac{nk}{m})\theta^{\frac{nk}{m}}} z^{\frac{nk}{m}-1} e^{-\frac{z}{\theta}} \tag{6.2}$$

The ideal case is that each cluster node processes the same amount of workload, that is, the expected value $E(Z) = \frac{nk\theta}{m}$. However, with a different number of cluster nodes and data blocks, we could have an imbalanced workload distribution, that is, some cluster nodes process significantly more workload than the average while other cluster nodes process much less than the average. To study this imbalance issue, we compute the cumulative probability of the workload performed by a cluster node as follows.

$$P(Z < w) = \int_{-\infty}^{w} f_Z(t)dt \tag{6.3}$$

And the probability of a workload greater than w on the node is

$$P(Z > w) = 1 - \int_{-\infty}^{w} f_Z(t)dt \tag{6.4}$$

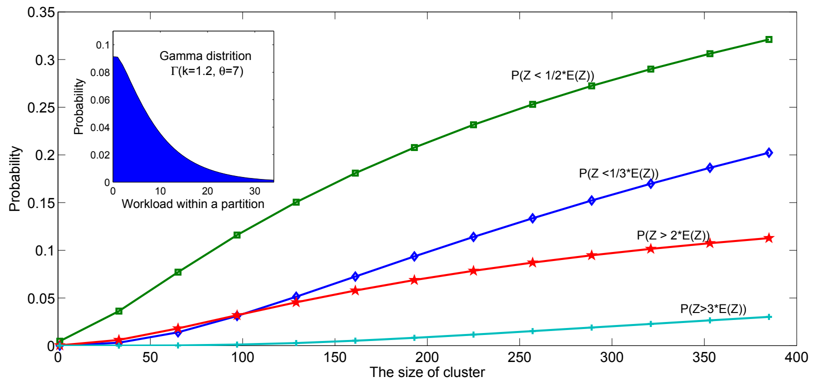


Figure 6.1: As the size of the cluster increases, more and more cluster nodes tend to have an imbalanced workload.

In general, the probability of the workload size on a cluster node being an extreme value will increase as m increases. For instance, given the value of $k = 1.2, \theta = 7$ and $n = 512$, we can observe from Figure 6.1 that $P(Z < 1/3 * E(Z))$, $P(Z < 1/2 * E(Z))$, $P(Z > 2 * E(Z))$ and $P(Z > 3 * E(Z))$ will increase with the growth of the cluster size. This implies that a larger number of cluster nodes will result in a higher chance of an imbalanced workload.

The expected number of nodes that will have a workload of at most w is $m * P(Z < w)$ while the expected number of nodes that will have a workload of size greater than w is $m - m * P(z < w)$. Based on the example in Figure 6.1 and given a cluster size of 128, the expected numbers of nodes that will have a workload of less than $1/2 * E(Z)$ and $1/3 * E(Z)$ are 3.9 and 1.5 respectively; and the expected number of nodes that will have a workload greater than $2 * E(Z)$ is 4.0. This implies that some nodes will have a workload 4 to 6 times greater than others. On the nodes with larger workloads, a longer execution time is needed to finish the tasks while the nodes with less workload will be idle for a long time before performing the next phase of execution. Experiments in Section 6.4 verify the theoretical analysis here that the *imbalanced* distribution could result in an inefficient parallel use of cluster resources and hence a low execution performance.

6.2 Sub-dataset Meta-data Management

The fundamental challenge of DataNet is to create a compact meta-data storage to store the sub-dataset distributions such that blocks with more data from a given sub-dataset will have a higher priority to be considered for workload-balanced computing in comparison with other blocks with less data. In this section, we will present the corresponding solutions for this challenge.

6.2.1 ElasticMap: A Meta-data Storage of Sub-datasets Over HDFS Blocks

In order to obtain a sub-dataset distribution, DataNet maintains the size of data related to each sub-dataset over block files, that is $|b_i \bigcap s_j|, i = 1, ..., n, j = 1, ..., m$, where b_i is the set of data records on the ith block and s_j is one sub-dataset contained by b_i. A simple method of recording this information is to use a table such as a hash map to store the pair $\langle id, quantity \rangle$, which represents the id of a sub-dataset, e.g, source id or event name, and the relative size of data associated with the sub-dataset that resides on a block file. We show an example in Table 6.1, which records the number of reviews corresponding to different movies within a block file.

Table 6.1: The size information of movies within a block file

id	movie_1	moive_2	...	movie_m
# of reviews	3578	3038	...	1

The above method has a memory cost of order $O(n * m)$, where the n is the number of blocks and m is the number of sub-datasets. Since the size information will be stored along with the master node and will be used by the task scheduler to achieve a balanced computational workload as shown in Section 6.3.2. In the case where the number of sub-datasets is large, the meta-data could incur a high memory cost. Therefore, in our implementation, we develop a data structure called $ElasticMap$, which consists of a hash map together with a bloom filter to store the size information of sub-datasets. In comparison to a hash map, a Bloom Filter uses a bitmap to represent the existence of sub-datasets in a block. Bloom Filters are well-known for space-efficient storage. For example, under a typical configuration, storing a sub-dataset's information over a single block in a HashMap will cost 85 bits while using a bloom filter will cost 10 bits.

Because of content clustering, a small number of sub-datasets could dominate the content of a block file while a large number of sub-datasets could have a small amount of data contained by the block. As the block containing a small amount of a sub-dataset will have a negligible impact on the workload-balance in sub-dataset analysis, a bloom filter is sufficient to provide the information for sub-dataset's tasks assignment. Thus, we design the $ElasticMap$ to store the information of dominant sub-datasets in a hash map and store that of non-dominant sub-datasets in a bloom filter. Such a design is very flexible, as we can store all the meta-data into the hash map when the memory is large enough and store most of the information into the bloom filter when the memory is limited.

Let n be the number of block files in a dataset. We maintain an ElasticMap array to record the sub-datasets' distribution information over n blocks. The array has n pointers, each pointing to the meta-data over a block file. Figure 6.2 shows an example of the data structure, where id is the id of a sub-dataset. By querying this structure, we can obtain the distribution of a sub-dataset over all block files.

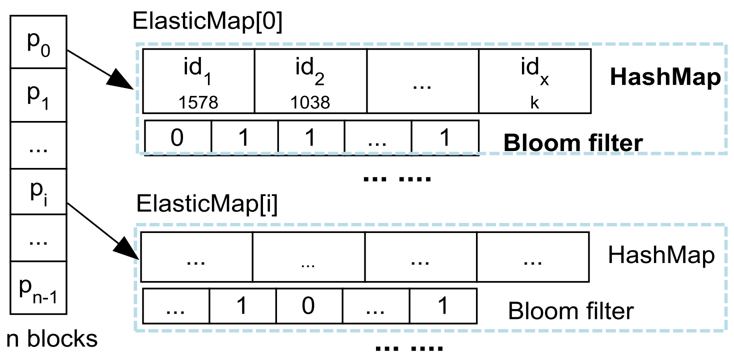


Figure 6.2: The DataNet meta-data structure over n block files.

According to the analysis of a bloom filter [19], if a bloom filter aims to represent sub-datasets with false positive probability ϵ, the average memory usage per sub-dataset can be expressed as $-\frac{ln(\epsilon)}{ln^2(2)}$. To evaluate how much memory space is needed to store an $ElasticMap$, we assume that there are m sub-datasets contained by a block, in which α percent of the sub-datasets will be stored in the hash map and the others are put into the bloom filter. Assume each record in the hash map uses a k-bit representation with the load factor of δ, which is a measure of how full the hash table is allowed to get, the memory cost of the $ElasticMap$ on one block is given in Equation 6.5.

$$Cost(memory) = \frac{m * (1 - \alpha) * ln(\epsilon)}{ln^2(2)} + \frac{m * \alpha * k}{\delta} \tag{6.5}$$

6.2.2 ElasticMap Constructions

The design of the $ElasticMap$ needs an efficient method to decide which sub-datasets should be stored into the hash map and which should be stored into the bloom filter. An intuitive method to achieve this is to sort the sub-datasets based on how much of their data is contained by the block file and then store the sub-datasets with larger size values into the hash map and others into the bloom filter. Unfortunately, such a sorting method in the big data era is not efficient, as the time complexity is $O(m \cdot \log m)$, where m is the number of sub-datasets in the block file. In this section, we discuss how to efficiently separate the sub-datasets without sorting.

In order to obtain the size information of sub-datasets over a block b_i, we define a series of size intervals or buckets, and distribute the sub-datasets s_j into the corresponding buckets according to the size $|b_i \cap s_j|$ via a single scan of the block. We maintain a variable S_j for each sub-dataset s_j to compute $|b_i \cap s_j|$. Before scanning, the variable S_j is set to 0. When a data record belonging to sub-dataset s_j is encountered, we increase the variable S_j and adjust the sub-dataset's bucket accordingly. Due to content clustering, the buckets corresponding to larger data sizes will contain a smaller number of sub-datasets. Consequently, we could use non-uniform buckets where larger data sizes have sparser intervals. One instance is the following series of buckets based on fibonacci sequence,

(0,1kb), [1kb,2kb), [2kb, 3kb), [3kb, 5kb), [5kb, 8kb), [8kb, 13kb), [13kb,21kb), [34kb, ∞).

After the scanning is complete, we will have the number of sub-datasets over each bucket. Then, we can decide which sub-datasets should be put into the hash map with the memory consideration based on Equation 6.5. Since a block file will be dominated by a small number of sub-datasets and will contain a small amount of data from many other sub-datasets, it is sufficient for us to distinguish dominant sub-datasets using a small number of buckets. For instance, to deal with a block file of $64MB$, one appropriate upper-bound size is $32kb$, since there will at most $64M/32k = 2048$ sub-datasets in the highest bucket, and we may put all of them into the hash map with a small memory cost of around $16kb$. On the other hand, one appropriate lower-bound of the size is $1kb$, since the sub-datasets smaller than $1k$ have little impact on the workload-balance and thus we can put them into a bloom filter. Therefore, tens of buckets could be sufficient to separate the dominant sub-datasets within the block file.

In fact, our algorithm for dominant sub-dataset separation is based on Bucket/Count sorting [29]. However, we are not actually sorting these sub-datasets, we only need to know the statistic value on different buckets to identify the dominant datasets and put them into the hash map. The time complexity of our algorithm is $O(m)$, where m is number of sub-datasets contained by a block. To deal with n blocks, the time complexity is $O(m * n)$, which means only a single scan of the raw data is needed for the meta-data construction.

6.3 Sub-dataset Distribution-aware Computing

With the use of ElasticMap, we could identify the imbalanced distribution of sub-datasets before launching the actual analysis tasks. In this section, we will present a distribution-aware method for sub-dataset analysis applications to achieve balanced computing.

6.3.1 Sub-dataset Distribution In Hadoop Clusters

Based on the block-locality driven scheduling in Hadoop systems and our analysis in Section 6.1, we can optimize data processing with the knowledge of sub-dataset distribution over HDFS blocks. To achieve this, we build the distribution relationship between cluster nodes and block files, where a block file is mapped to three cluster nodes and different block files contain different amounts of each sub-dataset. The sub-dataset distribution could be retrieved from the ElasticMap during task scheduling.

The distribution relationship with respect to a sub-dataset s is represented as a *Bipartite Graph* $G = (CN, B, E)$, where $CN = \{cn_0, cn_1, ..., cn_n\}$ and $B = \{b_0, b_1, ..., b_m\}$ are the vertices representing the cluster nodes and HDFS block files respectively and $E \subset CN \times B$ is the set of *edges* between CN and B. There exists an edge connecting a computation node $cn_i \in CN$ and a block $b_j \in B$ if and only if b_j is placed on cn_i. There may be several edges connected to a block b_j since a block has several copies stored on different cluster nodes. Each edge is configured with a *weight* equal to $|b_j \cap s|$, the size of the sub-dataset s contained by the block file b_j, which can be obtained through the $ElasticMap$ array.

We show a bipartite graph example in Figure 6.3. The vertices at the bottom represent cluster nodes, while those at the top represent block files. Each edge indicates that a block file b_j is located on a cluster node cn_i with a weight $|b_j \cap s|$. Based on this mapping information, we can optimize sub-dataset assignments according to different computation requirements such as workload balance among cluster nodes or reducing the data transferred for data aggregation.

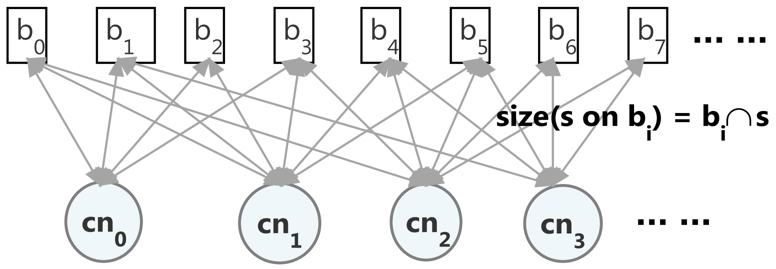


Figure 6.3: An example of a bipartite graph representing cluster nodes and block files. The edges represent the co-location of block files and cluster nodes, and the weight values of edges adjacent to node b_j is size of the sub-dataset s contained by block file j.

6.3.2 *Sub-dataset Distribution-aware Scheduling*

To balance the workload among cluster nodes, we first compute the total size of a sub-dataset as follows,

$$Z = (\sum_{b_j \in \tau_1} |s \cap b_j| + \delta * |\tau_2|) \tag{6.6}$$

where s is a given sub-dataset, τ_1 is the set of blocks that have the size information of s in the hash map, τ_2 is the set of blocks that have the size information of s in the bloom filter, and δ, the smallest size value of $|s \cap b_j|$, is the approximate data size per block that belongs to sub-datasets stored in the bloom filter. According to the computing capability of computational nodes, we can calculate the amount of sub-datasets to be assigned to each node.

We present a distribution-aware algorithm to balance the workload among computation nodes as shown in Algorithm 6.3.1. The algorithm aims to allow each computation node to have an equal amount of workload to be processed. The tasks are assigned with two considerations, the first is to assign local blocks to the requesting computation node i (line 8) while the second is to compare the current workload on node i with the average amount of workload (line 10, 14).

Algorithm 6.3.1 Distribution-aware Algorithm for Balanced Computing over a Sub-dataset s

```
1: Let $d_i$ be the set of blocks adjacent to cluster node $cn_i$ in the bipartite graph $G$.
2: Let $T$ = $\{t_0, t_1, ..., t_{n-1}\}$ be the set of tasks corresponding to the $n$ blocks.
3: Let $|b_j \bigcap s|$ be the size of sub-dataset $s$ in block $j$.
4: Let $W_i$ be the current workload on cluster node $cn_i$.
Steps:
5: Compute the average workload $\overline{W} = (\sum_{b_j \in \tau_1} |s \cap b_j| + \delta * |\tau_2|)/m$, where $m$ is the total
   number of cluster nodes
6: while $|T| \neq 0$ do
7:    if a worker process on $cn_i$ requests a task then
8:       if $|d_i| \neq 0$ then
9:          Find $b_x \in d_i$ such that
10:         $x = \underset{x}{\text{argmin}} |W_i + b_x \bigcap s - \overline{W}|$
11:         Assign $t_x$ to the requesting process on node $i$
12:      else
13:         Find $t_x \in T$ such that
14:         $x = \underset{x}{\text{argmin}} |W_i + b_x \bigcap s - \overline{W}|$
15:         Assign $t_x$ to the requesting process on node $i$
16:      end if
17:      Remove $t_x$ from $T$
18:      for all $cn_k$ adjacent to $b_x$ in $G$ do
19:         Remove the edge $(cn_k, b_x)$ from $G$
20:      end for
21:   end if
22: end while
```

In general, for applications with heavy computational requirements at the map phase, such as similarity computation, Algorithm 6.3.1 is useful to balance the parallel execution time. In a homogeneous execution environment, we can actually compute an optimized task assignment through the Ford-Fulkerson method [29]. For applications with aggregation requirements, the output may need to be transferred over the network and finally written into HDFS with several files. For these applications, in which the amount of output could be determined by the size of the input sub-dataset, ElasticMap can also be used to minimize the data transferred with the knowledge of sub-dataset distributions.

6.4 Experimental Results and Evaluations

We conduct comprehensive experiments on *Marmot* to show the benefits of DataNet in parallel big data computing with the MapReduce programming model. *Marmot* is a cluster of the PRObE on-site project [38] that is housed at CMU in Pittsburgh. The system has 128 nodes / 256 cores and each node in the cluster has dual 1.6GHz AMD Opteron processors, 16GB of memory, Gigabit Ethernet, and a 2TB Western Digital SATA disk drive. For our experiments, all nodes are connected to the same switch. The Hadoop system is configured as follows: one node is designated to be the NameNode/JobTracker, one node is the secondary NameNode, and other cluster nodes are the DataNodes/TaskTrackers. HDFS is configured with 3-way replication and the size of a chunk file is set to 64 MB.

6.4.1 DataNet Evaluation

To test DataNet for sub-dataset analysis under Hadoop frameworks, we first launch map tasks to filter out our target sub-dataset and store them locally on the cluster nodes. Then, we run various analysis jobs with different computation patterns to process the filtered sub-dataset and compare their performance. We employ two methods for the map task assignments. The first method (without DataNet) is the default block locality-driven scheduling used by the Hadoop system [85]. The second method (with DataNet) is our proposed distribution-aware method introduced in Section 6.3.2. For the meta-data stored in ElasticMap, we set the value of α in Equation 6.5 to 0.3. We will specifically discuss the performance of ElasticMap as α changes in Section 6.4.2.

We implement the following analysis jobs with the MapReduce interface.

- *Moving Average*: analyzing data points by creating a series of averages over intervals of the full dataset. Moving Average is often implemented in the analysis of trend changes and can smooth out short-term fluctuations to highlight longer-term cycles.

- *Top K Search*: finding K sequences with the most similarity to a given sequence. This algorithm needs heavy computation due to the similarity comparison between sequences.
- *Word Count*: reading the sub-dataset and counting how often words occur. Word Count is one of the representative MapReduce benchmark applications.
- *Aggregate Word Histogram*: computing the histogram of the words in the input sub-dataset. This is a fundamental plug-in operation in the MapReduce framework.

In our experiments, we mainly use a dataset consisting of movie ratings and reviews stored in chronological order in HDFS. The dataset is based on the distribution of the movie names, ratings and categories of "MovieLens" [33]. The text reviews are randomly generated and we also randomly duplicate some ratings for large-scale tests. The total number of block files is 256.

6.4.1.1 Overall Comparison

The overall execution times for the four analysis jobs from 32 cluster nodes are shown in Figure 6.4. As we can see, in all cases, the parallel execution time with the use of DataNet is smaller than that without DataNet. This can be explained by the fact that, without the use of DataNet, certain nodes will have a heavier workload than others, resulting in longer execution times and degrading the overall performance. Besides, we find that DataNet can achieve greater improvements for computationally intensive applications such as TopK Search in comparison to MovingAverage. In all, with DataNet, the improvements of MovingAverage, WordCount, Histogram and TopKSearch are 20%, 39.1%, 40.6% and 42% respectively.

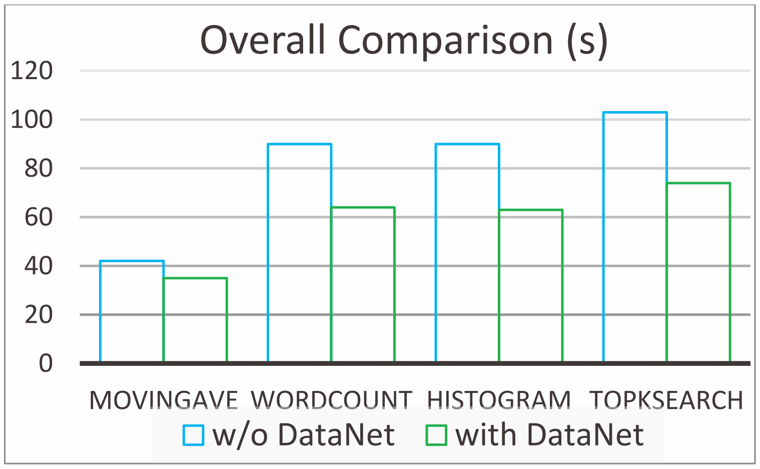


Figure 6.4: Overall execution comparison.

Figure 6.5 shows the sub-dataset distribution over the HDFS blocks, where a small number of blocks contain most of our target data due to content clustering; that is, most reviews about a movie are clustered around the time of the release. Figure 6.6 shows the workload corresponding to the size of the filtered sub-datasets over the cluster nodes. As we can see, without DataNet, the workload of node 25 was significantly higher than the workload of node 17. Such a distribution is far from being balanced for the subsequent analysis and this explains the performance gain in Figure 6.4.

6.4.1.2 Map Execution Time on the Filtered Sub-dataset

To gain further insight into the performance, we monitor the map execution time comparison on the filtered data for the sub-dataset analysis jobs. Figure 6.7 shows the local execution time of Top K Search on all 32 cluster nodes. From the figure, we can find that the slowest execution time is 64 seconds while the fastest execution time is 5 seconds. This could lead to a longer synchronization time to execute the next analysis phase and result in a longer overall execution as shown in Figure 6.4.

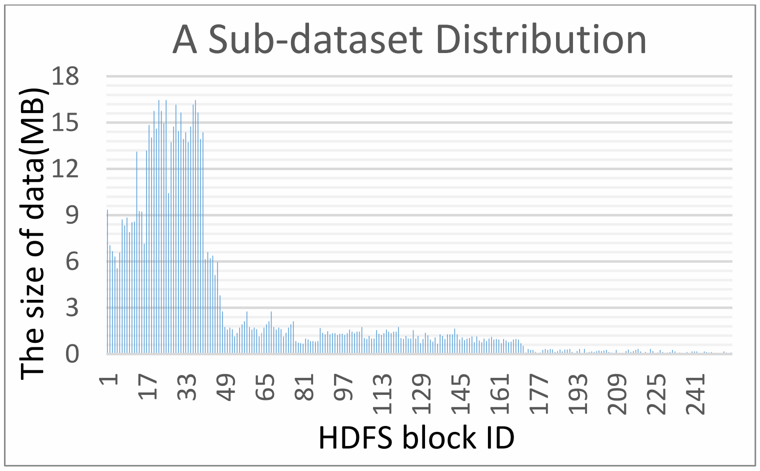


Figure 6.5: Size of data over HDFS blocks.

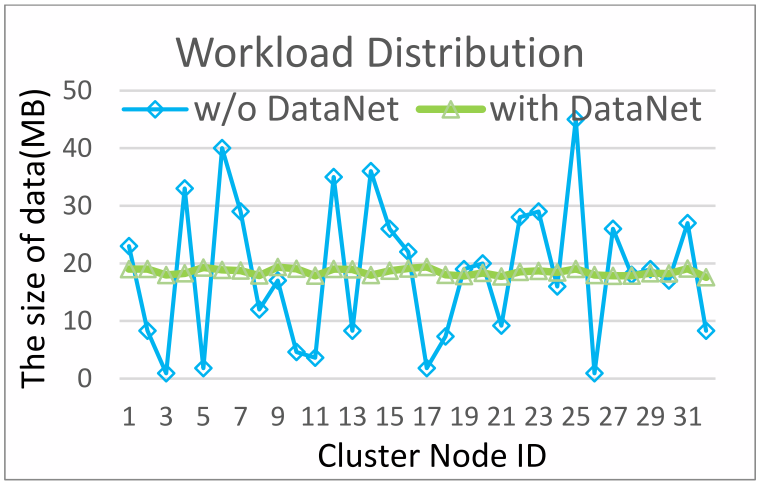


Figure 6.6: Workload distribution after selection.

With different computational requirements, the imbalanced workload could have different effects on the performance of analysis jobs. To demonstrate this, In Figure 6.8 and 6.9, we show the min, average and max execution times on the the filtered sub-dataset for Moving Average and Word Count on 32 nodes. From the figure, we can find that the gap between the min and max times for Moving Average is much smaller than that of Word Count. This is because Word Count needs to combine words while Moving Average only needs to iterate the data. Therefore, with greater computational requirements, the issue of imbalance becomes more serious.

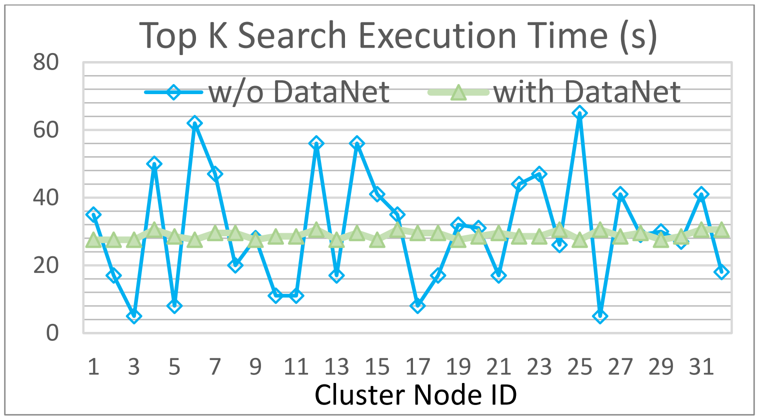


Figure 6.7: The map execution time distribution

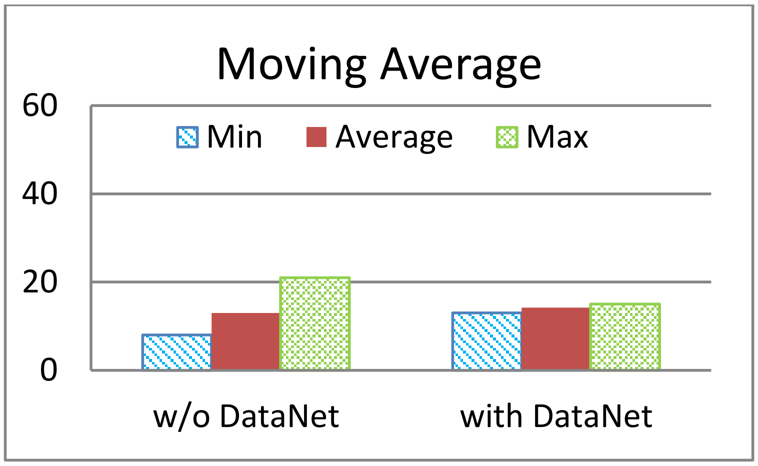


Figure 6.8: The Moving Average map time(s).

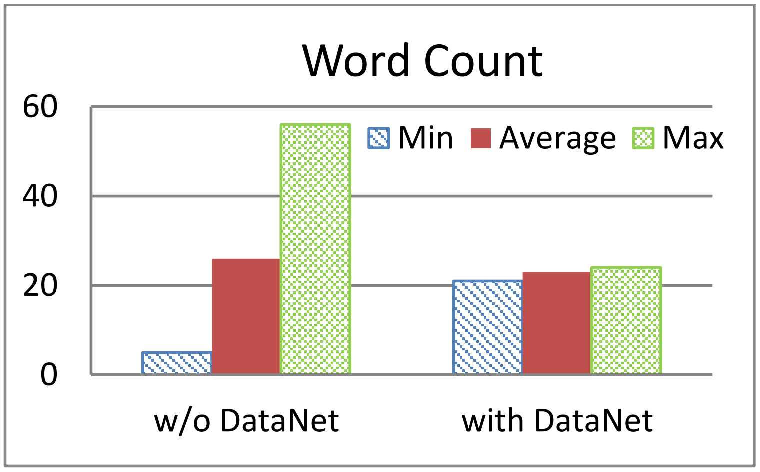


Figure 6.9: The Word Count map time(s).

6.4.1.3 Shuffle Execution Time Comparison

The shuffle phase [85] starts whenever a map task is finished and ends when all map tasks have been executed. We expect that the the shuffle time would be much longer with an imbalanced workload among the cluster nodes. To demonstrate how the imbalance affects the shuffle phase, we collect the min, average and max execution times for shuffle tasks in the Top K Search and Word Count analyses, and show the comparison in Figure 6.10 and Figure 6.11. From the figure, we can find that the shuffle phase without the use of DataNet takes 4-5X longer than with DataNet. We also find that the speedup of Top K Search is greater than that of Word Count. This is because the Top K Search takes more time for map execution as shown in Figure 6.7.

6.4.1.4 More Results and Discussion

We also run experiments on GitHub event log data [5]. The datasets provide more than 20 event types ranging from new commits and fork events to opening new tickets, commenting, and adding members to a project. The size of the raw data is around 34 GB. The experimental setting is the same as with the movie data. We run analysis jobs on “IssueEvent”. Figure 6.12 shows the sub-dataset distribution on the first 128 HDFS blocks. As we can see, the sub-dataset distribution doesn’t satisfy the property of content clustering. However, since the distribution over HDFS blocks is imbalanced, with the use of ElasticMap, we still can optimize task assignment to meet the balanced computation requirement using Algorithm 6.3.1. Specifically, to run the Top K Search job, the longest map execution time is 125 seconds without the use of DataNet and 107 seconds with DataNet. However, we find that the overall improvement is much less than that of the movie dataset. This is because the movie dataset has a more imbalanced sub-dataset distribution due to content clustering, which could cause a more imbalanced workload distribution when scheduling tasks without the distribution knowledge provided by ElasticMap. This can be seen by comparing Figure 6.9 and Figure 6.13.

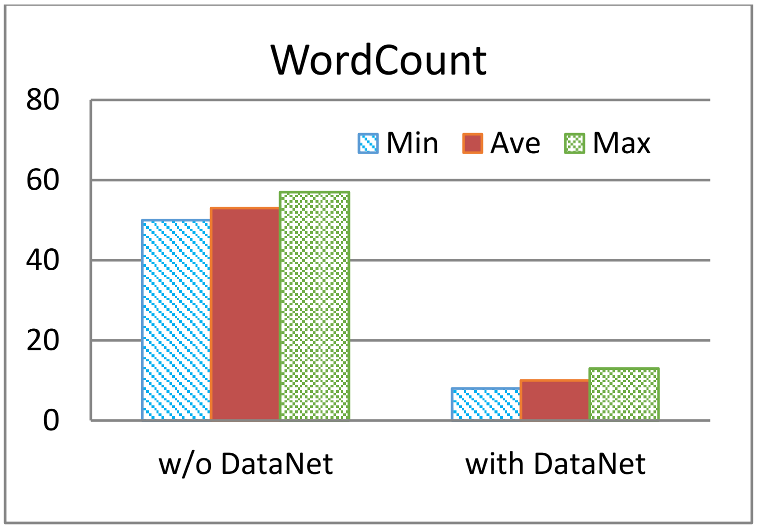


Figure 6.10: The Word Count shuffle time(s).

In order to achieve a workload-balance computation for parallel execution, an alternative method is to dynamically monitor the runtime status [51] and migrate workloads when necessary. Specifically, for sub-dataset processing, we can rebalance the sub-dataset distribution among cluster nodes after the map task's execution on the sub-dataset selection. With the example without DataNet in Figure 6.6, we find that almost every cluster node will transfer or receive sub-datasets and the overall percentage of data migration is more than 30%. Besides the overhead of collecting statistics and adjusting workload during runtime, the data migration could occupy the network resource and prolong the overall execution in comparison with DataNet, which can foresee the imbalanced issue in advance. In comparison, DataNet will scan the raw data once to build all sub-dataset distributions, while the method of dynamic adjugement will migrate the workload for each sub-dataset analyses during runtime. We will specifically discuss the efficiency of DataNet in the next section.

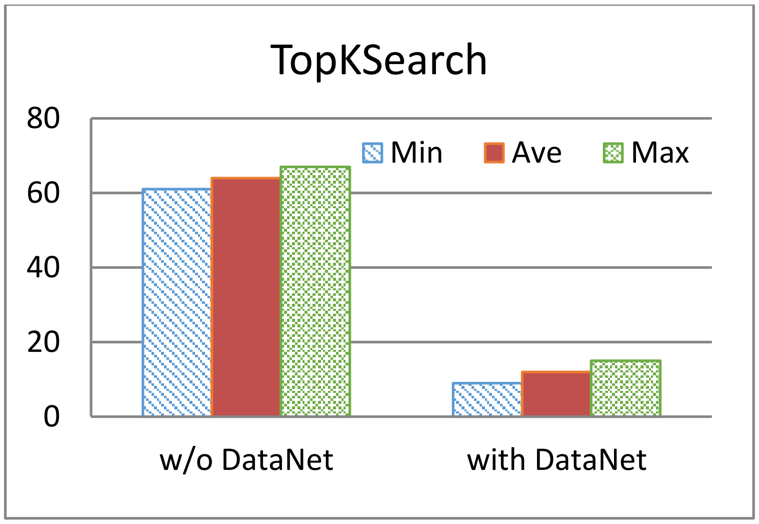


Figure 6.11: The Top K Search shuffle time(s).

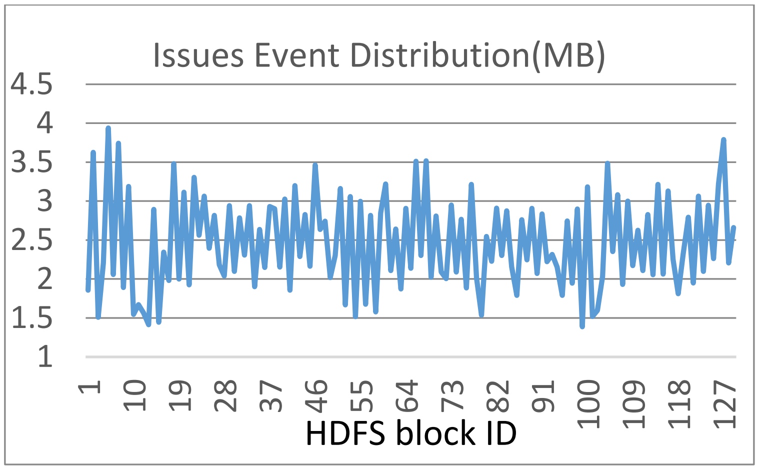


Figure 6.12: Size of data in HDFS blocks.

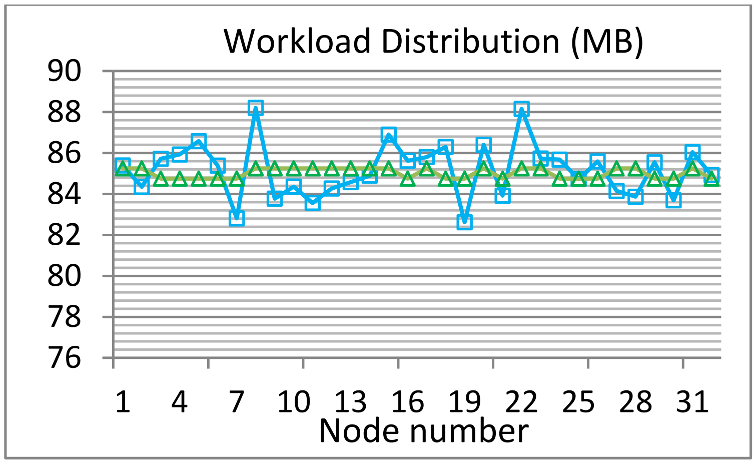


Figure 6.13: Workload distribution.

6.4.2 *Efficiency of DataNet*

6.4.2.1 Memory Efficiency and Accuracy of ElasticMap

The goal of ElasticMap is to store the information of sub-datasets' distribution in a compact fashion. With different data distributions, the memory cost on the meta-data storage could vary. For datasets with a high degree of content clustering or a limited number of events, such as GitHub event logs, the size ratio of raw data to meta-data could be very large. We studied the memory efficiency of ElasticMap over the movie dataset and show the results in Table 6.2. The first column of the table represents the percentage of elements stored in the hash map. The last column represents the size ratio of raw data to meta-data. The second column represents the accuracy of ElasticMap calculated as,

$$\chi = 1 - \frac{\sum_{b_j \in \tau_1}(|S \cap b_j| + \delta * |\tau_2|) - Size\_of\_raw\_data}{Size\_of\_raw\_data}$$

where S is the union of all sub-datasets, and b_j, τ_1, τ_2, and δ have the same meaning as in Equation 6.6

α in Equation 6.5	Accuracy(χ)	Representation ratio
51%	97%	1857
40%	93%	2270
31%	88%	2751
25%	83%	3196
21%	80%	3497

Table 6.2: The efficiency of ElasticMap

As we can see, for cases in which a small percentage of elements are stored in the hash map, there is a higher representation ratio but a lower overall accuracy. For example, in the case where 21% of the elements are stored in the hash map, 1 MB of meta-data in the ElasticMap can represent around 3497 MB of raw data. On the other hand, in the case where 51% of the elements are stored in the hash map, the overall accuracy of DataNet rises to 97% but the representation ratio dropped to 1857. This is due to the fact that the bloom filter can only indicate the existence of a sub-dataset within a block rather than the real size of the sub-dataset.

We also perform accuracy evaluations on individual movies with different sizes. The results are shown in Figure 6.14. As we can see, for sub-datasets with larger sizes, the difference between the actual size of the sub-dataset and the size calculated through Equation 6.6 is smaller. This is because the sub-datasets are dominant on most blocks and so they are precisely recorded in the hash map.

A greater difference occurs for sub-datasets with a size less than 32 MB. This could be explained by the fact that these sub-datasets are not dominant on most HDFS blocks and they are inaccurately recorded in the bloom filter. Nevertheless, as these sub-datasets have little data, there will be a lower probability for them to cause imbalanced computing. On the other hand, with the knowledge of ElasticMap, we can reduce the I/O cost, since we don't need to process blocks that don't contain our target data (no records in the hash map and bloom filter).

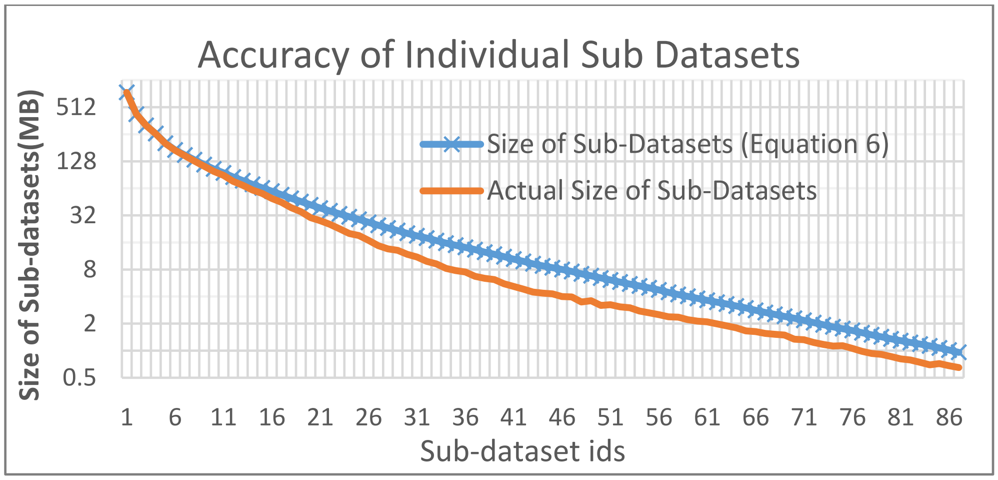


Figure 6.14: The accuracy of ElasticMap with respect to different sub-datasets.

In a real-time or interactive execution environment, recording the meta-data in memory could be efficient. However, as the problem size becomes extremely large, the meta-data may not be able to reside in memory. In such cases, the meta-data can be stored into a database or distributed among multiple machines. We leave this problem as future work and focus on the study and analysis of the imbalanced sub-dataset distribution and computing over Hadoop clusters.

6.4.2.2 *The Degree of Balanced Computing*

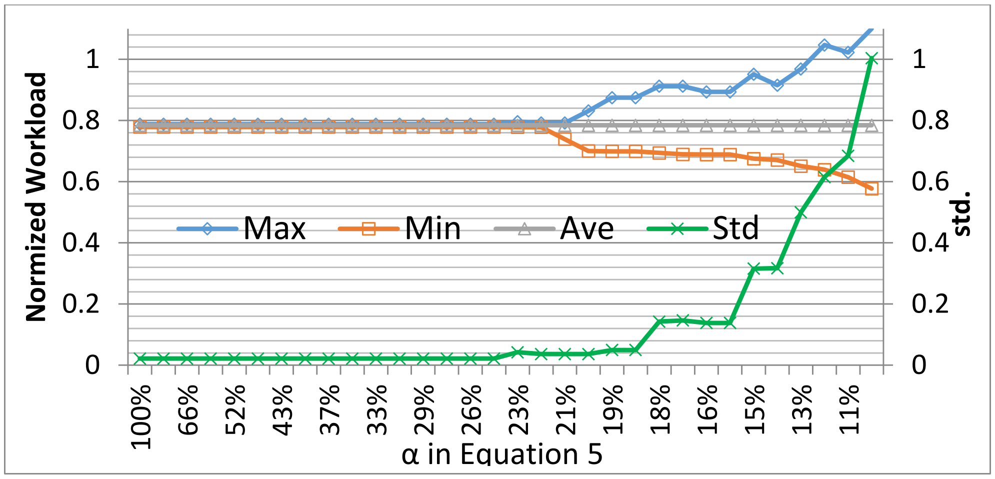


Figure 6.15: Balancing evaluation; the comparison of maximum, minimum, average workload and the std deviation on 32 compute nodes with different α in Equation 6.6.

When more sub-datasets are stored in the hash map with a higher memory cost, a higher accuracy could be achieved. This can produce a better workload balance through distribution-aware scheduling. We also run experiments with different α values and study the degree of load balance. The test sub-dataset shares the same distribution in Figure 6.5. The results are shown as Figure 6.15. From the Figure, we can find that with only about 15% of the sub-datasets recorded in the hash map, DataNet is able to achieve a satisfactory workload balance, i.e. the max workload is around 0.9 while the min is around 0.7. Changing the percentage from $15\ to\ 100$ will have little effect on workload balance. The main reason behind these results is that content clustering is the main cause of workload imbalance, and with about 15% of the sub-datasets stored in the hash map, these clustered data could be detected and thus handled using Algorithm 6.3.1.

6.5 Discussion and Related Works

To provide faster execution on the log files, Yin [92] et al. proposed a framework with a group-order-merge mechanism and Logothetis [59] et al. proposed a in-situ MapReduce architecture to mine the data "on location". To efficiently process ordered datasets in HDFS, Chen [26] et al. proposed a bloom filter-based approach to avoid unnecessary sorting and improve the performance of MapReduce. VSFS [88] is a searchable distributed file system for addressing the needs of data filtering at the file system-level. HBase [36] is an open source implementation of Google's BigTable [24] and the bloom filter used by HBase can greatly reduce the I/O cost during data selection. However, none of these methods address the sub-datasets' imbalanced distribution in parallel computing.

There have also been researches proposed to address data skew problem for MapReduce applications. LIBRA [25] addresses the data skew problem among the reducers of MapReduce applications through sampling the intermediate data. SkewTune [51] can mitigate skew in MapReduce applications through observing the job execution and re-balancing workload among the computing resources. Coppa [28] designs a novel profile-guided progress indicator which can predict data skewness and stragglers so as to avoid excessive costs. DataNet is orthogonal to these techniques and can proactively address the imbalanced computing through its sub-dataset distribution aware algorithm.

In oder to achieve better parallelism performance in cluster computing, many computational frameworks such as Dryad and Spark translate a job into a Directed Acyclic Graph (DAG) consisting of many small tasks, and execute them in parallel. Dmac [94] optimizes the DAG scheduler by calculating the matrix dependency in the matrix program. CooMR [54] is a cross-task coordination framework that can enable the sorting/merging of Hadoop intermediate data without actually moving the data over the network. KMN [83] schedules the sampling-based approximate query based on the run time status of the cluster. Delay scheduling [96] allows tasks to wait for a small amount of time for achieving locality computation. Quincy [46] is proposed to schedule concurrent distributed jobs with fine-grain resource sharing. Different from the aforementioned methods, DataNet studies the problem of content clustering in sub-dataset analysis and solves the imbalanced computing through distribution-aware techniques.

CHAPTER 7: CONCLUSION

In this chapter, we present concluding remarks of the proposed designs.

Firstly, we present our “Scalable Locality-Aware Middleware” (SLAM) for HPC scientific applications in order to address the data movement problem. SLAM leverages a distributed file system (DFS) to provide scalable data access for scientific applications. Since the conventional and parallel I/O operations from the high-performance computing (HPC) community are not supported by DFS, we propose a translation layer to translate these I/O operations into DFS I/O. Moreover, a novel data-centric scheduler (DC-scheduler) is proposed to enforce data-process locality for enhanced performance. We prototype our proposed SLAM system along with the Hadoop distributed file system (HDFS) across a wide variety of computing platforms. By testing two state-of-the-art real scientific applications, i.e., mpiBLAST and ParaView, we find that SLAM can greatly reduce the I/O cost and double the overall performance, as compared to existing approaches.

Secondly, we conduct a complete analysis on how remote and imbalanced read patterns occur and how they are affected by the size of the cluster when parallel data requests are issued to distributed file systems. We then propose a novel method to optimize parallel data access on distributed file systems. We aim to achieve a higher balance of data read requests between cluster nodes. To achieve the goal, we represent the data read requests that are issued by parallel applications to cluster nodes as a graph data structure where edges weights encode the demands of data locality and load capacity. Then we propose new matching-based algorithms to match processes to data based on the configurations of the graph data structure so as to compute the maximum degree of data locality and balanced access. Our proposed method can benefit parallel data-intensive analysis with various parallel data access strategies. Experiments are conducted on PRObEs Marmot 128-node cluster testbed and the results from both benchmark and well-known parallel applications show the performance benefits and scalability of Opass.

Finally, we investigate the issues of imbalanced sub-dataset analyses over Hadoop clusters. Due to the missing information of sub-datasets' locality, the content clustering inherent in most sub-datasets prevents applications from efficiently processing them. Through a theoretical analysis, we conclude that an uneven sub-dataset distribution almost always leads to a lower-performance in parallel data analysis. To address this problem, we propose DataNet to support sub-dataset distribution-aware computing. DataNet uses an elastic structure, called ElasticMap, to store the sub-dataset distributions. Also, a dominant sub-dataset separation algorithm is proposed to support the construction of ElasticMap. We conduct comprehensive experiments for different sub-dataset applications with the use of DataNet and the experimental results show the promising performance of DataNet.

Acknowledgment

This material is based upon work supported by the National Science Foundation under the following NSF program: Parallel Reconfigurable Observational Environment for Data Intensive Super-Computing and High Performance Computing (PRObE).

This work is supported in part by the US National Science Foundation Grant CCF-1527249, CCF-1337244 and National Science Foundation Early Career Award 0953946.

LIST OF REFERENCES

[1] Cplex linear programming. http://www.aimms.com/aimms/solvers/cplex.

[2] Flume: Open source log collection system. https://flume.apache.org/.

[3] Fuse:. http://fuse.sourceforge.net/.

[4] genomes:. http://aws.amazon.com/1000genomes/.

[5] Github events. https://www.githubarchive.org/.

[6] http://www.mpi-forum.org/docs/mpi21-report.pdf.

[7] Running hadoop-blast in distributed hadoop. http://salsahpc.indiana.edu/tutorial/hadoopblast.html.

[8] Vtk readers. http://www.vtk.org/.

[9] World cup 1998 dataset. http://goo.gl/2UqlS.

[10] Enrique E Abola, Frances C Bernstein, and Thomas F Koetzle. Protein data bank. Technical report, Brookhaven National Lab., Upton, NY (USA), 1984.

[11] Sameer Agarwal, Srikanth Kandula, Nicolas Bruno, Ming-Chuan Wu, Ion Stoica, and Jingren Zhou. Reoptimizing data parallel computing. In *NSDI*, pages 281–294, 2012.

[12] James Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. *The Visualization Handbook*, 717:731, 2005.

[13] Stephen F Altschul, Warren Gish, Webb Miller, Eugene W Myers, David J Lipman, et al. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.

[14] Jo-Ellen Asbury. Overview of focus group research. *Qualitative health research*, 5(4):414–420, 1995.

[15] A. Atanasov, M. Srinivasan, and T. Weinzierl. Query-driven parallel exploration of large datasets. In *Large Data Analysis and Visualization (LDAV), 2012 IEEE Symposium on*, pages 23–30, Oct 2012.

[16] Haim Avron and Anshul Gupta. Managing data-movement for effective shared-memory parallelization of out-of-core sparse solvers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 102:1–102:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[17] Janine C. Bennett, Hasan Abbasi, Peer-Timo Bremer, Ray Grout, Attila Gyulassy, Tong Jin, Scott Klasky, Hemanth Kolla, Manish Parashar, Valerio Pascucci, Philippe Pebay, David Thompson, Hongfeng Yu, Fan Zhang, and Jacqueline Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 49:1–49:9, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[18] Dennis A Benson, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, and Eric W Sayers. Genbank. *Nucleic acids research*, 38(suppl 1):D46–D51, 2010.

[19] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[20] Dhruba Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.

[21] Dhruba Borthakur. Hdfs architecture guide. *HADOOP APACHE PROJECT http://hadoop. apache. org/common/docs/current/hdfs design. pdf*, 2008.

[22] Joe B Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. Scihadoop: Array-based query processing in hadoop. In *Pro-*

ceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, page 66. ACM, 2011.

[23] Mario Cannataro, Domenico Talia, and Pradip K. Srimani. Parallel data intensive computing in scientific and commercial applications. *Parallel Comput.*, 28(5):673–704, May 2002.

[24] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[25] Qi Chen, Jinyu Yao, and Zhen Xiao. Libra: Lightweight data skew mitigation in mapreduce. *Parallel and Distributed Systems, IEEE Transactions on*, 26(9):2520–2533, Sept 2015.

[26] Zhijian Chen, Dan Wu, Wenyan Xie, Jiazhi Zeng, Jian He, and Di Wu. A bloom filter-based approach for efficient mapreduce query processing on ordered datasets. In *Advanced Cloud and Big Data (CBD), 2013 International Conference on*, pages 93–98, Dec 2013.

[27] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. *ACM SIGOPS Operating Systems Review*, 43(4):5–10, 2010.

[28] Emilio Coppa and Irene Finocchi. On data skewness, stragglers, and mapreduce progress indicators. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 139–152, New York, NY, USA, 2015. ACM.

[29] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[30] Chuck Cranor, Milo Polte, and Garth Gibson. Structuring plfs for extensibility. In *Proceedings of the 8th Parallel Data Storage Workshop*, PDSW '13, pages 20–26, New York, NY, USA, 2013. ACM.

[31] Aaron Darling, Lucas Carey, and Wu-chun Feng. The design, implementation, and evaluation of mpiblast. *Proceedings of ClusterWorld*, 2003, 2003.

[32] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. Efficient big data processing in hadoop mapreduce. *Proceedings of the VLDB Endowment*, 5(12):2014–2015, 2012.

[33] Simon Dooms, Toon De Pessemier, and Luc Martens. Movietweetings: a movie rating dataset collected from twitter. In *Workshop on Crowdsourcing and Human Computation for Recommender Systems, CrowdRec at RecSys 2013*, 2013.

[34] Mark K. Gardner, Wu-chun Feng, Jeremy Archuleta, Heshan Lin, and Xiaosong Mal. Parallel genomic sequence-searching on an ad-hoc grid: experiences, lessons learned, and implications. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[35] Michael R Garey, David S Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[36] Lars George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[37] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[38] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. Probe: A thousand-node experimental cluster for computer systems research. volume 38, June 2013.

[39] Jeff Gill. *Bayesian methods: A social and behavioral sciences approach*, volume 20. CRC press, 2014.

[40] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced features of the message-passing interface*. MIT press, 1999.

[41] Salman Habib, Adrian Pope, Zarija Lukic, David Daniel, Patricia Fasel, Nehal Desai, Katrin Heitmann, Chung-Hsing Hsu, Lee Ankeny, Graham Mark, Suman Bhattacharya, and James Ahrens. Hybrid petacomputing meets cosmology: The roadrunner universe project. *Journal of Physics: Conference Series*, 180(1):012019, 2009.

[42] Yoonsoo Hahn and Byungkook Lee. Identification of nine human-specific frameshift mutations by comparative analysis of the human and the chimpanzee genome sequences. *Bioinformatics*, 21(suppl 1):i186–i194, 2005.

[43] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, and Xian-He Sun. I/o acceleration with pattern detection. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 25–36, New York, NY, USA, 2013. ACM.

[44] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 22–22. USENIX Association, 2011.

[45] Jiangling Yin Jun Wang Jian Zhou Tyler Lukasiewicz Dan Huang and Junyao Zhang. Opass: Analysis and optimization of parallel data access on distributed file systems. In *Accepted to appear in Proc. of 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS'15), Hyderabad, India*, May 2015.

[46] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of*

the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09, pages 261–276, New York, NY, USA, 2009. ACM.

[47] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why we twitter: Understanding microblogging usage and communities. In *Proceedings of the 9th WebKDD and 1st SNA-KDD 2007 Workshop on Web Mining and Social Network Analysis*, WebKDD/SNA-KDD '07, pages 56–65, New York, NY, USA, 2007. ACM.

[48] Hui Jin, Jiayu Ji, Xian-He Sun, Yong Chen, and Rajeev Thakur. Chaio: enabling hpc applications on data-intensive file systems. In *Parallel Processing (ICPP), 41st International Conference on*, pages 369–378. IEEE, 2012.

[49] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Publishing Company Redwood City, CA, 1994.

[50] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. A study of skew in mapreduce applications. *Open Cirrus Summit*, 2011.

[51] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 25–36, New York, NY, USA, 2012. ACM.

[52] Jerald F Lawless. *Statistical models and methods for lifetime data*, volume 362. John Wiley & Sons, 2011.

[53] Yanfang Le, Jiangchuan Liu, Funda Ergun, and Dan Wang. Online load balancing for mapreduce with skewed data input. In *INFOCOM, 2014 Proceedings IEEE*, pages 2004–2012. IEEE, 2014.

[54] Xiaobing Li, Yandong Wang, Yizheng Jiao, Cong Xu, and Yu Weikuan. Coomr: Cross-task coordination for efficient data management in mapreduce programs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–11, Nov 2013.

[55] Heshan Lin, Xiaosong Ma, P. Chandramohan, A. Geist, and N. Samatova. Efficient data access for parallel blast. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 72b–72b, April.

[56] Heshan Lin, Xiaosong Ma, Wuchun Feng, and Nagiza F. Samatova. Coordinating computation and i/o in massively parallel sequence search. *IEEE Trans. Parallel Distrib. Syst.*, 22(4):529–543, April 2011.

[57] Gordon S Linoff and Michael JA Berry. *Data mining techniques: for marketing, sales, and customer relationship management*. John Wiley & Sons, 2011.

[58] Dionysios Logothetis, Chris Trezzo, Kevin C. Webb, and Kenneth Yocum. In-situ mapreduce for log processing. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[59] Dionysios Logothetis, Chris Trezzo, Kevin C. Webb, and Kenneth Yocum. In-situ mapreduce for log processing. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[60] Wei Lu, Jared Jackson, and Roger Barga. Azureblast: a case study of developing science applications on the cloud. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 413–420, New York, NY, USA, 2010. ACM.

[61] A. Matsunaga, M. Tsugawa, and J. Fortes. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 222–229, Dec.

[62] K. Mehta, J. Bent, A. Torres, G. Grider, and E. Gabriel. A plugin for hdf5 using plfs for improved i/o performance and semantic analysis. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 746–752, Nov 2012.

[63] Zhen Meng, Jianhui Li, Yunchun Zhou, Qi Liu, Yong Liu, and Wei Cao. bcloudblast: An efficient mapreduce program for bioinformatics applications. In *Biomedical Engineering and Informatics (BMEI), 2011 4th International Conference on*, volume 4, pages 2072–2076, 2011.

[64] C. Mitchell, J. Ahrens, and Jun Wang. Visio: Enabling interactive visualization of ultra-scale, time series data via high-bandwidth distributed i/o systems. In *IPDPS, 2011 IEEE International*, pages 68–79, May.

[65] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. f4: Facebooks warm blob storage system. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 383–398. USENIX Association, 2014.

[66] Saul B Needleman, Christian D Wunsch, et al. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[67] C. Oehmen and Jarek Nieplocha. Scalablast: A scalable implementation of blast for high-performance data-intensive bioinformatics analysis. *Parallel and Distributed Systems, IEEE Transactions on*, 17(8):740–749, 2006.

[68] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[69] James Ostell. Databases of discovery. *Queue*, 3(3):40–48, 2005.

[70] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The quantcast file system. *Proc. VLDB Endow.*, 6(11):1092–1101, August 2013.

[71] William R Pearson and David J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

[72] Ramya Prabhakar, Mahmut Kandemir, and Myoungsoo Jung. Disk-cache and parallelism aware i/o scheduling to improve storage system performance. *Parallel and Distributed Processing Symposium, International*, 0:357–368, 2013.

[73] Robert B Ross, Rajeev Thakur, et al. Pvfs: A parallel file system for linux clusters. In *in Proceedings of the 4th Annual Linux Showcase and Conference*, pages 391–430, 2000.

[74] Shun-Tak Leung Sanjay Ghemawat, Howard Gobioff. The google file system. *OPERATING SYSTEMS REVIEW*, 37:29–43, 2003.

[75] Philip Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, volume 2003, 2003.

[76] Saba Sehrish, Grant Mackey, Jun Wang, and John Bent. Mrap: A novel mapreduce-based framework to support hpc analytics applications with access patterns. HPDC '10, pages 107–118, New York, NY, USA, 2010. ACM.

[77] C. Sigovan, C. Muelder, K. Ma, J. Cope, K. Iskra, and Robert B. Ross. A visual network analysis method for large scale parallel i/o systems. In *International Parallel and Distributed Processing Symposium (IPDPS 2013)*. IEEE, IEEE, 2012.

[78] Marc Snir, Steve W Otto, David W Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: the complete reference*. MIT press, 1995.

[79] A.H. Squillacote. *The ParaView Guide: A Parallel Visualization Application*. Kitware, 2007.

[80] Bruno J Strasser. Genbank–natural history in the 21st century? *Science*, 322(5901):537–538, 2008.

[81] Seung-Jin Sul and A. Tovchigrechko. Parallelizing blast and som algorithms with mapreduce-mpi library. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 481–489, 2011.

[82] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[83] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J Franklin, and Ion Stoica. The power of choice in data-aware cluster scheduling. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 301–316. USENIX Association, 2014.

[84] Yi Wang, Wei Jiang, and Gagan Agrawal. Scimate: A novel mapreduce-like framework for multiple scientific data formats. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 443–450, Washington, DC, USA, 2012. IEEE Computer Society.

[85] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[86] D.N. Williams, T. Bremer, C. Doutriaux, J. Patchett, S. Williams, G. Shipman, R. Miller, D.R. Pugmire, B. Smith, C. Steed, E.W. Bethel, H. Childs, H. Krishnan, P. Prabhat, M. Wehner, C.T. Silva, E. Santos, D. Koop, T. Ellqvist, J. Poco, B. Geveci, A. Chaudhary, A. Bauer, A. Pletzer, D. Kindig, G.L. Potter, and T.P. Maxwell. Ultrascale visualization of climate data. *Computer*, 46(9):68–76, September 2013.

[87] Xing Wu, K. Vijayakumar, F. Mueller, Xiaosong Ma, and P.C. Roth. Probabilistic communication and i/o tracing with deterministic replay at scale. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 196–205, Sept 2011.

[88] Lei Xu, Ziling Huang, Hong Jiang, Lei Tian, and David Swanson. Vsfs: A searchable distributed file system. In *Proceedings of the 9th Parallel Data Storage Workshop*, PDSW '14, pages 25–30, Piscataway, NJ, USA, 2014. IEEE Press.

[89] Xi Yang, Yanlong Yin, Hui Jin, and Xian-He Sun. Scaler: Scalable parallel file write in hdfs.

[90] Jiangling Yin, Jun Wang, Wu-chun Feng, Xuhong Zhang, and Junyao Zhang. Slam: Scalable locality-aware middleware for i/o in scientific analysis and visualization. HPDC '14, pages 257–260, New York, NY, USA, 2014. ACM.

[91] Jiangling Yin, Junyao Zhang, Jun Wang, and Wu chun Feng. Sdaft: A novel scalable data access framework for parallel {BLAST}. *Parallel Computing*, 40(10):697 – 709, 2014.

[92] Jiangtao Yin, Yong Liao, Mario Baldi, Lixin Gao, and Antonio Nucci. A scalable distributed framework for efficient analytics on ordered datasets. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, UCC '13, pages 131–138, Washington, DC, USA, 2013. IEEE Computer Society.

[93] Yanlong Yin, Jibing Li, Jun He, Xian-He Sun, and R. Thakur. Pattern-direct and layout-aware replication scheme for parallel i/o systems. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 345–356, 2013.

[94] Lele Yu, Yingxia Shao, and Bin Cui. Exploiting matrix dependency for efficient distributed matrix computation. 2015.

[95] Yongen Yu, Jingjin Wu, Zhiling Lan, D.H. Rudd, N.Y. Gnedin, and A. Kravtsov. A transparent collective i/o implementation. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 297–307, 2013.

[96] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

[97] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[98] Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, October 1999.

[99] Xuechen Zhang, Ke Liu, Kei Davis, and Song Jiang. ibridge: Improving unaligned parallel file access with solid-state drives. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 381–392, Washington, DC, USA, 2013. IEEE Computer Society.

[100] Zhao Zhang, Daniel S. Katz, Justin M. Wozniak, Allan Espinosa, and Ian Foster. Design and analysis of data management in scalable parallel scripting. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, 2012.

[101] Fang Zheng, Hongbo Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, Tuan-Anh Nguyen, Jianting Cao, H. Abbasi, S. Klasky, N. Podhorszki, and Hongfeng Yu. Flexio: I/o middleware for location-flexible scientific data analytics. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 320–331, 2013.