Achieving Load Balance for Parallel Data Access on Distributed File Systems

Dan Huang, Dezhi Han, Jun Wang, Jiangling Yin, Xunchao Chen, Xuhong Zhang, Jian Zhou, and Mao Ye

Abstract—The distributed file system, HDFS, is widely deployed as the bedrock for many parallel big data analysis. However, when running multiple parallel applications over the shared file system, the data requests from different processes/executors will unfortunately be served in a surprisingly 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 system such as HDFS store each data unit, referred to as chunk 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 probability 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*, resulting in a degraded I/O performance. In this paper, we first conduct a complete analysis on how remote and imbalanced read/write patterns occur and how they are affected by the size of the cluster. We then propose novel methods, referred to as Opass, to optimize parallel data reads, as well as to reduce the imbalance of parallel writes on distributed file systems. Our proposed methods can benefit parallel data-intensive analysis with various parallel data access strategies. Opass adopts new matching-based algorithms to match processes to data so as to compute the maximum degree of data locality and balanced data access. Furthermore, to reduce the imbalance of parallel writes, Opass employs a heatmap for monitoring the I/O statuses of storage nodes and performs HM-LRU policy to select a local optimal storage node for serving write requests. Experiments are conducted on PRObE's Marmot 128-node cluster testbed and the results from both benchmark and well-known parallel applications show the performance benefits and scalability of Opass.

Index Terms—Parallel data access, distributed file systems, HDFS, bipartite matching

1 Introduction

With the explosive growth of scientific information and simulation data [29], [38], applications in analysis/visualization require more time than ever before. Parallel computing techniques [27] speed up the performance of these applications by exploiting the inherent parallelism in data mining/rendering algorithms [9], [50]. Commonly used parallel strategies [9] 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. The de facto standard of parallel computing for decades has been the MPI programming model [36]. In this paper, we refer to each operator on data partitions as *a data processing task*. We specifically refer to *parallel data analysis* as the set of parallel processes running simultaneously to perform the tasks and terminating at the end of analysis.

Conventionally, in parallel data analysis, multiple processes running on different cluster nodes share a separate dedicated storage system. Once a data processing task is scheduled to a process, the data will be transferred from the shared storage to the process. However, in today's big data era, large amounts of data movement over the shared network could incur an extra overhead during parallel execution, especially during iterative data analysis, which involves moving data from storage to processes repeatedly.

Distributed file systems [7], [18], such as GFS, HDFS, QFS or Ceph, could be directly deployed on the disks of cluster nodes [19] to reduce data movement. When storing a data set, distributed file systems will usually divide the data into small *chunk 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. If the required data is not on the local disk, the process will then read from another node that contains the required data. The data requests from the parallel processes are referred as *parallel data requests*. These data requests can be issued not only from Hadoop Hadoop MapReduce applications, but also MPI applications [10], [14], [24], [31].

Unfortunately, the data requests from parallel processes/executors in big data processing will be served in an imbalanced fashion on the distributed storage servers. And these parallel requests over the storage will compete for shared resources such as hard disk head and network bandwidth.

• D. Huang, J. Wang, J. Yin, X. Chen, X. Zhang, J. Zhou, and M. Ye are with the Department of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816.
E-mail: {dhuang, jwang, jyin, xchen, xzhang, jzhou, mye}@eecs.ucf.edu.
• D. Han is with the Merchant Marine College, Shanghai Maritime University, Shanghai 201306, China. E-mail: dzhan@shmtu.edu.cn.

Manuscript received 15 Apr. 2017; revised 14 Aug. 2017; accepted 28 Aug. 2017. Date of publication 6 Sept. 2017; date of current version 16 Feb. 2018.
(Corresponding authors: Dezhi Han and Jun Wang.)
Recommended for acceptance by F. Douglis.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2017.2749229

Because of this, the makespan of an entire program could be significantly prolonged and the overall I/O performance will degrade. We specifically summarize the following challenges residing in distributed file systems in regard of parallel read/write performance.

1) *Simultaneous data requests in big data processing*: From the perspective of applications, parallel processes/executors in big data processing could simultaneously issue a large number of data read requests to file systems [9], [36], [50], and the data loading phase takes several minutes even hours, such as mpiBLAST [15] and ParaView [37]. These parallel data requests could create a serious contention on certain storage nodes. In general, the main goal associated with parallel data analysis over HDFS is to identify an assignment of processes to tasks such that the maximum amount of data can be accessed locally and in a balanced fashion while also adhering to the constraints of data distribution in HDFS as well as the load balance requirements of each parallel process/executor. Some of the challenges of achieving this goal arise because, *a*) the number of data requests per process is difficult to predict in dynamic data processing architectures [15], [48] and parallel processes/executors in static architectures 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 requires multiple data inputs, the inputs needed by a task may be stored on multiple nodes [20].
2) *Imbalanced concurrent data requests among storage servers from multi-tenant analytic programs*: On today's clusters, there are often multi-tenant analytic programs being run simultaneously. These analytic programs access hundreds or thousands of data files from a shared distributed storage. In a scale-out distributed environment, individual data sets requested by each program are usually distributed unevenly among storage nodes. This can cause some storage nodes to serve more data requests than others and thus create a bottleneck in the system, resulting in a prolonged I/O completion delay for all analysis programs. To address this problem, we need to define an efficient scheduling method which could dispatch the I/O workload to storage nodes such that a high degree of global balance can be achieved for concurrent data accesses.
3) *Prolonged parallel big data writing time in replication pipelining*: By default, HDFS uses a replication pipeline method and random policy to write chunk files into distributed storage nodes with several copies. It inevitably introduces imbalanced parallel writes, I/O congestion and uneven data placement. Our experiment results show that the fastest write process is twice as fast as the slowest one (Fig. 3). This is because the job scheduler in HDFS does not take into account the I/O statuses of the nodes. Tracking the I/O status of each node over time could introduce non-negligible overhead. This is especially true when we scale out the cluster.

In this paper, we first present a systematic analysis for parallel data accesses on distributed file systems. We then propose novel algorithms for improving parallel access. Our goal is to reduce remote data accesses on HDFS for parallel data analysis and thus achieve a high balance of data requests between cluster nodes. For parallel data reads access, we retrieve the data layout information from the underlying distributed file system and model the assignment of processes to data as a one-to-many matching in a *Bipartite Matching Graph*. We then use the matching-based algorithms to compute a solution that enables parallel data reads to be served on HDFS in a local and balanced fashion. Moreover, we demonstrate how to adopt our method to optimize the dynamic data requests in heterogeneous and multi-tenant environments. For parallel data write, we design a *HM-LRU Policy* to reduce the imbalance of parallel writes in distributed file systems. This method randomly chooses three candidate DataNodes for storing a chunk replica. Then, the DataNode with least transmission connections and least recently used is chosen for saving chunks. We evaluate our proposed methods on both benchmark and well-known parallel applications. The evaluation results confirm the performance and scalability benefits of Opass.

The initial idea of this work has appeared in 2015 IEEE Parallel and Distributed Processing Symposium, which only demonstrates the problem and solution for parallel data read requests from a single MPI-based application [46]. In this version, we extend the initial ideas and provide a more robust design for parallel data read requests from multiple applications as well as parallel writes over shared storage. We also conduct experiments and analysis for scalable data processing.

2 Background

In this section, we first discuss how parallel applications can access data from distributed file systems, e.g, HDFS. Then we briefly describe challenges associated with parallel data accesses on the file systems.

2.1 The Hadoop File System and Parallel Data accesses

Parallel Data Access on Distributed File Systems: The Hadoop file system can allow parallel programs to access data by using its libHDFS library [2], [32]. Currently, there are two methods that have been implemented to access data from HDFS. The first is to include *hdfs.h* and use Hadoop C/C++ API (libhdfs.so). The I/O interface, like *hdfsread* and *hdfswrite*, will be used to read/write data from/to HDFS. Another method is to use an I/O virtual translation layer to translate the parallel I/O operations, e.g POSIX I/O or MPI-I/O, into HDFS I/O operations [12], [44], [45] during data access. In the following experiments, we use the first method to access data in HDFS unless otherwise specified.

Data locality: Data locality access [16], [25] is an important technique used for shortening data access time, as it can reduce network contention and data transfer time. This technique could significantly expedite data analysis with large amounts of data as input. For instance, the time for tasks to execute remotely could be more than 2X slower than to execute locally [45], [47]. However, most parallel data requests

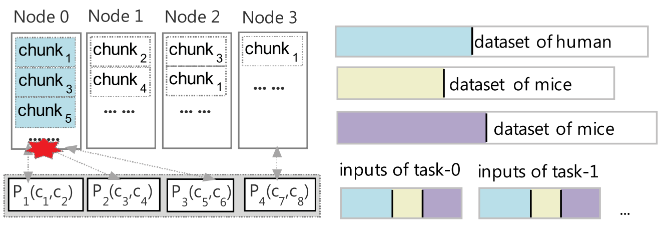


Fig. 1. The left figure shows the contention of parallel data requests on replica-based data storage. The right figure shows an example of parallel tasks with multiple parallel data inputs.

will be served remotely. We discuss the parallel data accesses associated with specific applications in next section.

Replication Pipeline Write: By default, HDFS adopts a random selection based replication pipeline to save multiple copies of chunk files into DataNodes. First, the client receives a list of randomly chosen DataNodes (e.g., three DataNodes) from the NameNode, which are the designated candidate DataNodes for storing replicas of the chunk files. The client then flushes its chunk data into the first DataNode. The first DataNode starts receiving the data in small portions (e.g., 4 KB), writes each portion to its local repository and transfers it to the second DataNode in the list. The second DataNode, starts receiving and writing that portion to its repository and then flushes it to the third DataNode.

2.2 Applications with Parallel Data Accesses

Parallel applications and BigData applications such as Paraview [37] and MapReduce [1] 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 requirement [9], [36], [50]. We specifically discuss parallel data accesses challenges in parallel data analysis.

Parallel Single-Data Read 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. 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 Fig. 1 (left), three processes can read three data chunks from Node 0 and no process will read data from Node 1, resulting in a lower parallel utilization of cluster nodes/disks.

Parallel Multi-Data Read 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 ford1955simplespecies [20]. For instance, to compare the genome sequences of humans, mice and chimpanzees, a single task needs to read three inputs, as shown in Fig. 1 (right). These inputs may be stored on different cluster nodes and, without consideration of data distribution, their data requests could cause some storage nodes to suffer a contention, thus degrading the execution performance.

Parallel Multi-tenant Read Access. On today's clusters, there are often multi-tenant analytic programs running simultaneously. These analytic programs access hundreds or thousands of data files from a shared distributed storage. In a scale-out distributed environment, individual data sets requested by each program are usually distributed unevenly among storage nodes. This can cause some storage nodes to serve more data requests than others and thus create a bottleneck in the system, resulting in a prolonged I/O completion delay for all analysis programs. To address this problem, we need to define an efficient method which could dispatch the I/O workload to storage nodes such that a higher degree of global balance can be achieved for concurrent data accesses.

Parallel Write Access. Parallel write is an important access pattern in HDFS that can provide a highly aggregated bandwidth for simulation or big data analytic workflows. For instance, in HPC settings, scientific simulations take advantage of MPI-IO [4] to output data into storage in parallel. Currently, a set of methods and middlewares [6], [44], have been proposed to allow parallel writes to achieve high performance in HDFS. Both Facebook and Linkedin write transaction logs and user activities in parallel from web servers into Hadoop clusters at certain intervals (e.g., 15 minutes to 1 hour) [39], [40], so as to feed fresh data to big data analytics, such as news/friends recommendations.

3 Motivations and Theoretical Analysis

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

3.1 Remote Read 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 have 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 32 GB dataset consisting of 512 chunks, in Fig. 2, we plot the cumulative distribution function of X for $k = 0, 1, 2, \ldots, 20$ with cluster sizes of 64, 128, 256 and 512. The probability of reading more than 5 chunks locally is

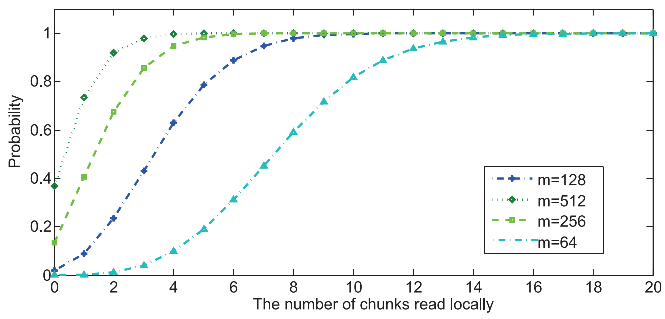


Fig. 2. CDF of the number of chunks read locally. The cluster size, m, changes from 64 to 512.

$$\begin{aligned} P(X > 5)|_{m=64} &= 1 - P(X \le 5)|_{m=64} = 81.09\%,\\ P(X > 5)|_{m=128} &= 1 - P(X \le 5)|_{m=128} = 21.43\%,\\ P(X > 5)|_{m=256} &= 1 - P(X \le 5)|_{m=256} = 1.64\%,\\ P(X > 5)|_{m=512} &= 1 - P(X \le 5)|_{m=512} = 0.46\%. \end{aligned}$$

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

3.2 Imbalanced Read 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 each 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 $node_j$, let Z be the random variable denoting the number of chunks served by $node_j$ and Y be the number of chunks on $node_j$. By default, data are randomly distributed within HDFS, so the probability of node $node_j$ containing a certain chunk is r/m. Thus, the probability that $node_j$ 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 provided in Section 3.1, we can assume that almost all data requests served by $node_j$ are remote requests. For any chunk on $node_j$, the probability of the process requesting that chunk being served by $node_j$ is $1/r$. Given that $node_j$ contains exactly a chunks, the conditional probability $P(Z \le k|Y = a)$ is a Binomial cumulative distribution function, and according to the Law of Total Probability, the probability that $node_j$ will serve at most k chunks is

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

Given $r = 3, n = 512,$ and $m = 128$, the expected number of nodes serving at most 1 chunk is $512 \times P(Z \le 1) = 11$ while the expected number of nodes serving more than 8 chunks is $512 \times (1 - P(z \le 8)) = 6$, which implies that some storage nodes will serve more than $8\times$ 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. These *imbalanced* data access patterns will result in inefficiency in parallel use of storage nodes/disks and hence a low I/O performance.

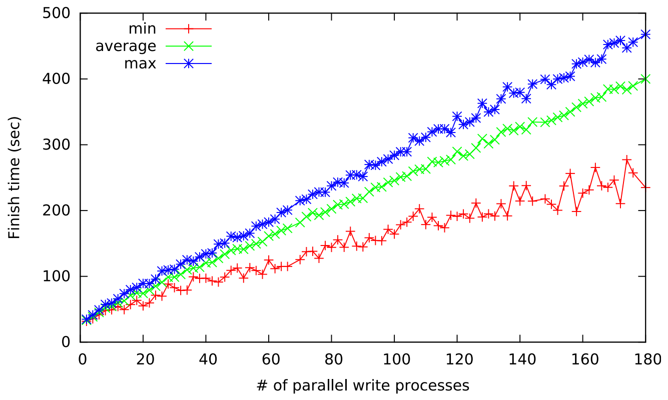


Fig. 3. Imbalanced parallel write.

3.3 Parallel Write Access Pattern Analysis

By default, HDFS adopts a random selection based replication pipeline to save multiple copies of chunk files into DataNodes. However, there are several performance issues that need to be thoroughly investigated. The first is reducing the disparity of parallel writes bandwidth between the fastest and the slowest processes. In our preliminary experimental results, the fastest write process outperforms the slowest by up to 100 percent. Reducing this difference is essential in many parallel applications, since the uneven bandwidth can cause severe overall performance degradation to write-read workflows. For example, many scientific simulations, such as GTC-P [30], create a barrier that pauses the progress of the simulation until all processes finish writing scientific simulation data at the end of each iteration. After analyzing the parallel write procedure, we realize that there is an awareness gap between parallel write behaviors and HDFS. In HDFS, data is split into small chunks (64 MB) to be stored on different DataNodes. Processes are unaware of the current I/O status, such as I/O congestion, network latency and imbalanced workloads, of the DataNodes. Those parallel processes might create a contention on some specific DataNodes for shared I/O resources, such as disk head, network channel etc. In the worst case, if a process continues sending synchronized write requests to a severely congested DataNode, this process could halt for an unacceptable amount of time. Second, HDFS adopts a random policy to place chunks on DataNodes, which will cause an issue called imbalanced data placement, since in most circumstances random placement can not guarantee balanced data placement. Further, the imbalanced placement will result in imbalanced parallel data reads in analytic programs, because some DataNodes storing more hot chunks have to serve more parallel read requests.

To show the problem of imbalanced parallel writes in HDFS, assume a set of parallel processes are launched on an m-node cluster with an r-way replication storage architecture

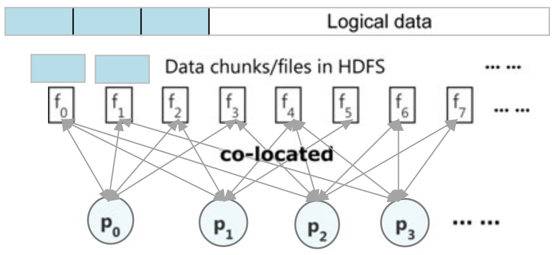


Fig. 4. A bipartite matching example of processes and chunk files.

to write a dataset consisting of n chunks and the nodes are randomly chosen to store data, the probability of a node to be chosen is r/m (r out of m cluster nodes has to be chosen). Let X be the random variable denoting a node to be chosen, X has a Binomial Distribution and its *cumulative distribution function* is

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

Given $r = 3, n = 512,$ and $m = 128$, the expected number of nodes serving at most 1 chunk is $128 \times P(x \leq 1) = 11$ while the expected number of nodes serving more than 7 chunks is $128 \times (1 - P(z < 7)) = 14$, which implies that some storage nodes will serve more than $7X$ the number of chunk requests as others. On the nodes serving 7 chunks, the write requests from different processes will compete for the *hard disk head* and *network bandwidth*, while the nodes serving 1 chunk will be idle for a long time.

To further illustrate the imbalance, we run preliminary experiments on *Marmot* and *HDFS*. In Marmot, 61 nodes are configured as 1 NameNode and 60 DataNodes. Parallel write processes are evenly assigned to DataNodes. Each process writes 1 GB of data (split into 16 chunks, 64 MB per chunk) to HDFS and the number of parallel processes varies from 2 to 180. We record the finish times of write processes and compare the average, minimum and maximum time of all processes in the Fig. 3. As the number of parallel writes increases, the difference between the maximum and minimum time increases. In the worst case, the maximum time is 2× that of the minimum. These results show that the random-selection policy of HDFS is not an efficient and effective method for providing balanced parallel writes.

4 Methodology and Design of Opass

In this section, for parallel read access, we first retrieve the data distribution information from the underlying distributed file system and build the processes to data matching in Section 4.1. We then find a matching from processes to data through matching based algorithms with the constraints of locality and load balance. Further, we apply Opass to dynamic parallel data accesses in heterogeneous and multi-tenant environments in Section 4.5. For parallel write access, we create a heatmap to record the current active connection on each DataNode. Based on this heatmap, we design and develop HM-LRU policy to reduce the imbalance of parallel writes in HDFS.

4.1 Encoding Process to Data Matching

Based on the data read policy in HDFS and our analysis in Section 3, we can allow parallel data read requests to be served in a balanced way through maximizing the degree of data locality read. 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 Sections 4.2 and 4.3. The locality relationship is represented as a *Bipartite Matching Graph* $G = (P, F, E)$, where $P = \{p_0, p_1, \ldots, p_n\}$ and $F = \{f_0, f_1, \ldots, f_m\}$ 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 will have several copies stored on different cluster nodes. We show a bipartite matching example in Fig. 4. 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 data locality, 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*.

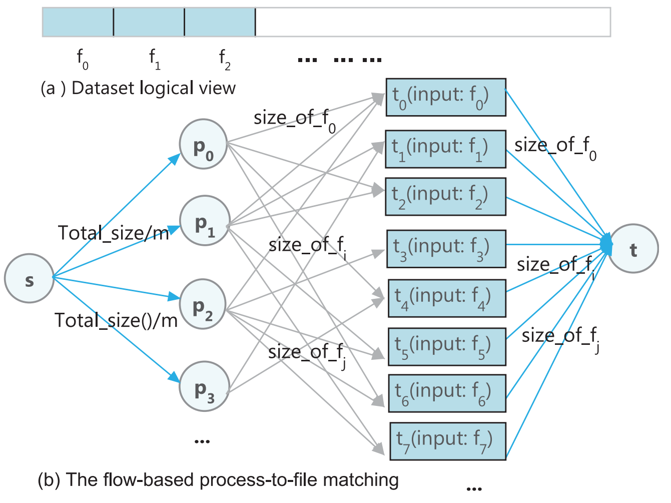
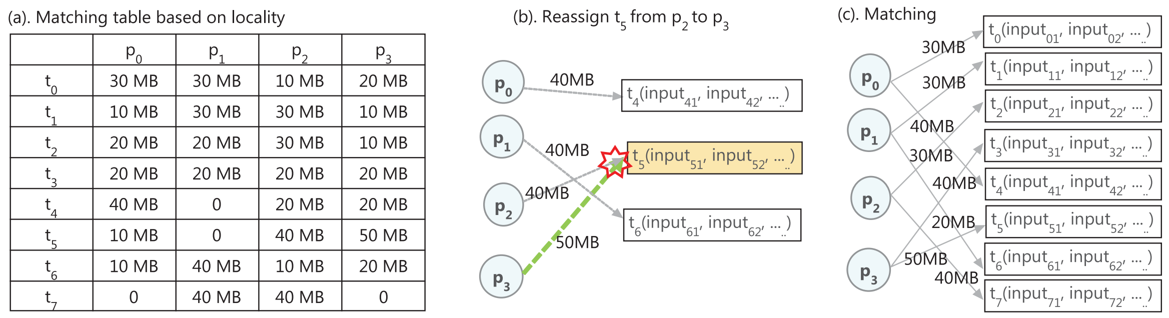


Fig. 5. The matching-based process-to-file configuration for single-data access in equal-data assignment. Each data processing task has only one input.

4.2 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, 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 and we call this *Single-data Access*. We can encode this type of matching problem as a flow network as shown in Fig. 5.

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 graph. Then, m edges, each connecting s and a vertex in P are added with equal capacity


(a). Matching table based on locality
(b). Reassign t5 from p2 to p3
p0
40MB
t4(input41, input42, ...)
p1
40MB
t5(input51, input52, ...)
p2
40MB
t6(input61, input62, ...)
p3
50MB
(c). Matching
p0
p1
p2
p3
30MB
30MB
40MB
30MB
40MB
20MB
50MB
40MB
t0(input01, input02, ...)
t1(input11, input12, ...)
t2(input21, input22, ...)
t3(input31, input32, ...)
t4(input41, input42, ...)
t5(input51, input52, ...)
t6(input61, input62, ...)
t7(input71, input72, ...)

	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

Fig. 6. The process-to-data matching example for multi-data assignment.

$TotalSize/m$, where $TotalSize$ is the net size of all data to be processed. Next, edges are added between processes and tasks, each with a capacity of the file size. As a result, n edges, connecting vertices in F with t, are added, each with 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 64 M).

In order to achieve a maximum amount of local data reads, we employ the standard max-flow algorithm, Ford-Fulkerson [17], 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*, 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 [17]. 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 Fig. 4. This method could allow us the achieve the optimal task assignment, which enables tasks to access data locally and avoids network data transfer.

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 not be a full matching, which implies that some processes are assigned less than $TotalSize/m$ of data. To rectify this, we assign unmatched tasks to each such process until all processes are matched to $TotalSize/m$ of data.

4.3 Optimization of Parallel Multi-Data Read Access

For a single task with multiple data inputs as shown in Fig. 1(right), the needed 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 only deals with one-to-one matchings [11]. In this section, we propose a novel matching-based algorithm for this type of parallel data accesses.

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 4.1, as shown in Fig. 4 to find co-located data between tasks and parallel processes. Fig. 6a 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. Fig. 6b 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 form 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.

The method is detailed in Algorithm 1. Assume that we have a set of n tasks $T = \{t_0, t_1, \ldots, t_{n-1}\}$ and a set of m parallel processes $P = \{p_0, p_1, \ldots, p_{m-1}\}$. Each process will be assigned n/m tasks for parallel execution. First, through the matching information in Fig. 4, 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 1 consists of the tasks assigned to each process i, denoted as $T(p_i)$. 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 1. Matching-Based Algorithm for Tasks with Multi-Data Inputs

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

4.4 Opass for Dynamic Parallel Data Read 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 [29]. To address this problem, applications such as mpiBLAST [29] usually combine task parallelism and SPMD parallelism by adopting a *master process* that controls and assigns tasks to *worker 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 section, we will demonstrate how to adopt our proposed methods to enable parallel applications 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 worker process, and will assign tasks to worker 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 worker 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.

Algorithm 2. Distribution Algorithm of Read Requests for File Chunks

```
Let the file $F$ consist of $m$ chunks stored on $n$ DataNodes
Let the set $NC = \{nc_0, nc_1, \ldots, nc_{n-1}\}$ represent the current
number of chunk data requests served by the DataNodes.
Let the set $C = \{c_0, c_1, \ldots, c_{n-1}\}$ represent the number of
chunk read requests for file $F$ that the DataNodes will serve.
Let the set $L = \{l_0, l_1, \ldots, l_{n-1}\}$ represent the number
of chunks of f that each DataNode holds.
Input: $m, n, C, NC$; Output: $C$
Steps:
Find the maximum number of read requests served by
current DataNodes: $nc_j = Max(NC)$
\* assign m chunks to n DataNodes *\
for $nc_i$ in $NC$ and $nc_i < nc_j$ do
   if $nc_j - nc_i > l_i$ then
      $c_i = l_i; m = m - c_i;$
   else
      $c_i = nc_j - nc_i; m = m - c_i;$
   end if
end for
while $m > 0$ do
   for $nc_i$ in $NC$ and $m > 0$ do
      if $c_i < l_i$ then
         $c_i = c_i + 1; m = m - 1$
      end if
   end for
end while
```

4.5 Opass for Multi-Tenant Parallel Data Read Access

HDFS employs an *r*-way replication to provide high availability. Files in HDFS are split into chunks and each chunk will be copied to *r* DataNodes. When a client initiates a read request for a chunk, by default, the NameNode will return a sorted list of DataNodes that hold the requested chunk. The sorting is based on the proximity of DataNodes to the client process. The nearest DataNode will be picked to serve the read request. When there are many concurrent read requests, this simple strategy may cause some DataNodes to serve more read requests than others.

We define the degree of imbalance as the difference between the amount of data being served by the node serving the most requests and that of the node serving the fewest requests. When the degree of imbalance exceeds a predefined threshold, we will replace the default locality driven read strategy with our matching based balanced read method. We use network flow algorithms to solve this matching problems. First, we obtain the number of files each application is going to read. For each file, we retrieve the location relationship between chunks and DataNodes, where each chunk read request can be served by *r* DataNodes. The location relationship is represented as a *Bipartite Matching Graph* $G = (F, DN, E)$, where $DN = \{dn_0, dn_1, \ldots, dn_{n-1}\}$ and $F = \{f_0, f_1, \ldots, f_{m-1}\}$ are the vertices representing the

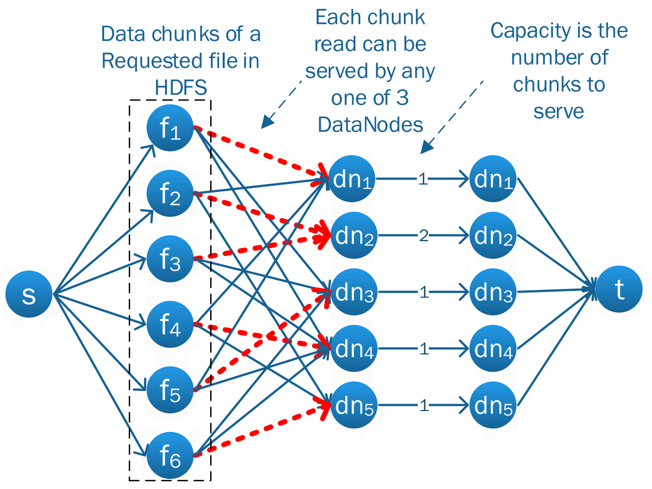

Data chunks of a Requested file in HDFS
Each chunk read can be served by any one of 3 DataNodes
Capacity is the number of chunks to serve
s
f1
f2
f3
f4
f5
f6
dn1
dn2
dn3
dn4
dn5
1
2
1
1
1
t

Fig. 7. Balanced file read requests matching using network flow. Capacity is determined by each DataNode's current number of chunks served and the number of chunks of newly requested file. Expected matching is denoted in red doted line.

DataNodes and chunks respectively and $E \subset F \times DN$ is the set of *edges* between F and DN. Before reading this file, suppose that DataNode i is already serving NC_i chunks where $i = 1, 2, 3, \ldots, n$. After assigning the read requests for this file's m chunks to n DataNodes, we want the number of chunk read requests each DataNode serves to be as balanced as possible. Algorithm 2 is used to evenly distribute the read requests for this file's chunks. After determining the number of chunk read requests that each DataNode should serve for this file, represented as $C = \{c_0, c_1, \ldots, c_{n-1}\}$, we want to find a one to one matching in G that could satisfy this new serving distribution.

Here, we use a maximum flow algorithm to solve this matching problem. Graph G is extended to $G' = (F, DN, E')$ to apply the maximum flow algorithm, where $E' \subset F \times DN \times DN$, for $DN \times DN$, an edge only exists between dn_i *and* dn_i and the edge capacity is c_i. Fig. 7 gives an example of using network flow to find a balanced file read request matching (denoted in red doted line).

4.6 Opass for Parallel Data Write Access

To reduce the imbalance of parallel writes, we introduce an heuristic method, *HM-LRU policy* as shown in Algorithm 3. This method considers both workload size and workload starting time on DataNodes. The heatmap table in NameNode records the number of active client connections to DataNodes, and a timestamp table records the least recently timestamps of DataNode being selected to store blocks. These clients access to a Datanode for reading or writing data chunks. In each DataNode, there is a thread pool to maintain active connection threads. When a client connects to the DataNode for sending/receiving data, the thread pool will generate a thread to handle the client's I/O request. Traditionally, NameNode can access DataNodes' thread pool status via remote RPC at runtime. But NameNode might become a performance bottleneck while it makes bursty RPC requests to access DataNodes for thread pool statuses. Alternatively, in our method the heatmap is updated while a DataNode reports its status to NameNode via heartbeat routines. In HDFS, DataNode sends a heartbeat message to the NameNode periodically to report live status. Heartbeats carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. We add the thread pool status into the heartbeat message. While NameNode receiving the DataNode's heartbeat information, it will update the active clients connections in the heatmap. The timestample table is also maintained in NameNode and records the timestamp when a client starts an IPC connection to a DataNode for data transmission. The timestampe implies that the earlier a DataNode serves a workload, the higher probability that the workload has ended or will end sooner. With the heatmap and timestamp table, we can choose the coolest DataNode in candidates to store the replica. HM-LRU policy is an improvement upon the pure random strategy which randomly chooses one among all possible DataNodes.

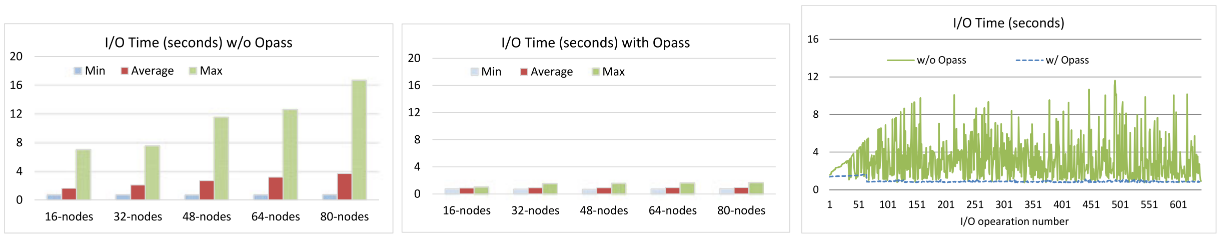
Algorithm 3. HM-LRU Policy for Parallel Write

```
Let $r$ be r-way replica write
let set $DN = \{N_0, N_1, \ldots, N_{n-1}\}$ be the n DataNodes.
Let the hash map $HM$ represents the heatmap that records
pairs of DataNodes and the number of client connections.
let the hash map $TM$ represents the latest timestamp of a
DataNode being selected to store data replica.
Let set $Results = \{N_0, N_1, \ldots, N_{r-1}\}$ be the selected $r$
DataNodes to store the new created block.
Input: $block$, $DN$; Output: $Results$
Steps:
for $i$ from 0 to $r - 1$ do
  1) randomly select three candidates, $R = \{a, b, c\}$,
  from set $DN$
  2) calculate $weight(x) = rank(HM(x)) + rank(TM(x))$,
  where $x \in R = \{a, b, c\}$ and rank() returns the rank of x in
  the ascending sorted set R.
  2) let $result = argmin_x(weight(x))$ . (find the DataNode
  with least workload and least recently selected).
  3) put $result$ into set $results$
end for
```

5 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 [19] that is housed at CMU in Pittsburgh. The system has 128 nodes / 256 cores and each node in the cluster has dual 1.6 GHz AMD Opteron processors, 16 GB 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 the parallel programming framework on all compute nodes running CentOS55-64 with kernel 2.6, in which CFQ-Scheduler is adopted as the block I/O scheduler. 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 DataNodes/TaskTrackers. 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



(a) Read data from HDFS without Opass. (b) Read data from HDFS with Opass. (c) I/O time On a 64-node cluster.

Fig. 8. I/O times comparison on parallel single-data read access.

will read data from another node that is chosen at random. We adopts MPI-IO benchmark and ParaView to evaluate our proposed method and algorithms. The amount of access data is averagely ranging from hundreds to thousands of MBs per storage nodes.

5.1 Opass Evaluation

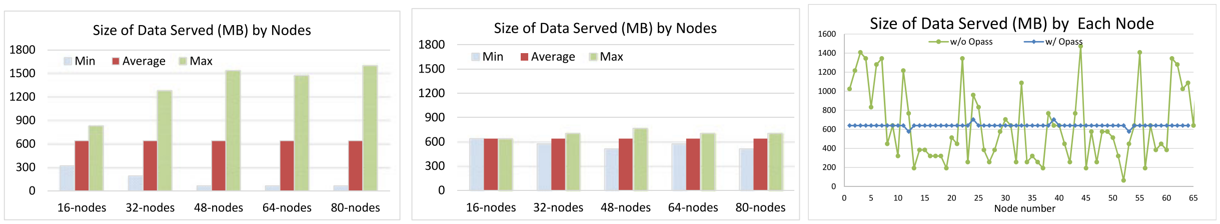
5.1.1 Evaluating Opass on Parallel Single-Data Access

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 as shown in Section 2. 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 Fig. 8. As we can see in Fig. 8a, 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 9× that of the minimum. However, on an 80-node cluster, this value becomes 21×. 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 Fig. 8b, 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 percent of the data are accessed remotely. The detailed analysis will be presented along with Fig. 9. 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 Fig. 8c. 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.

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 Fig. 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 Fig. 9a, the imbalance of data accesses becomes more serious as the size of the cluster increases. For instance, on an 80-node cluster, the maximum amount of data served by a node is 1,500 MB while the minimum is 64 MB. To gain further insight into this imbalance, we plot the amount of data served by each node on a 64 node cluster in Fig. 9c. 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 1,400 MB of data while some nodes serve 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 Fig. 8. This also confirms our analysis in Section 3. In contrast, with the use of Opass, every storage node serves approximately 640 MB.

5.1.2 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



(a) Access patterns on HDFS without Opass. (b) Access patterns on HDFS with Opass. (c) Access patterns On a 64-node cluster.

Fig. 9. Access patterns comparison on parallel single-data read access.

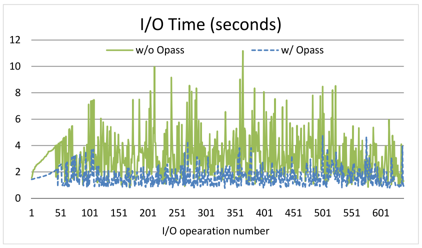


Fig. 10. I/O times of tasks with multiple inputs on a 64-node cluster.

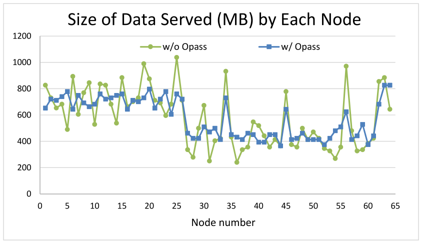


Fig. 11. Access patterns of multi-inputs on a 64-node cluster.

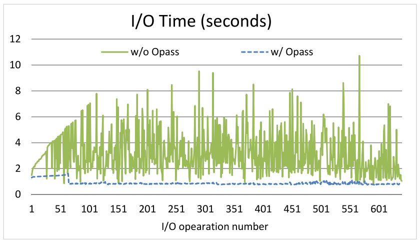


Fig. 12. I/O times of dynamic data assignment on a 64-node cluster.

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 Fig. 10. The improvement of Opass in this test is not as great as that in Fig. 8c. This is because each task involves several data inputs that are distributed throughout the cluster; therefore, to execute a task, part of the 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.

We also record the amount of data served by every node on the cluster and plot the results in Fig. 11. While the balance of data accesses 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.

5.1.3 *Evaluating Opass on Dynamic Parallel Data Accesses*

For the dynamic data accesses tests, we allow a master process to control the task assignments with an architecture similar to that of mpiBLAST [29]. The master process assigns tasks to worker processes, 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

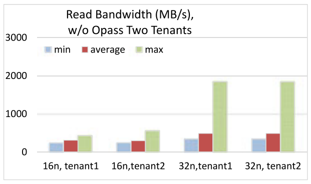


Fig. 13. Read bandwidths on 16-node and 32-node cluster without opass, two tenants.

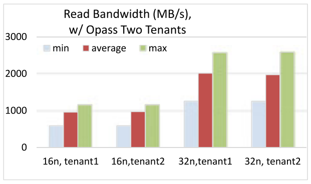


Fig. 14. Read bandwidths on 16-node and 32-node cluster with opass, two tenants.

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 Fig. 12. The results obtained from these tests are similar to those of the equal data assignment tests shown in Fig. 8c. 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.

5.1.4 *Evaluating Opass on Multi-Tenants Data Access*

For the multi-tenant data access tests, we configure two tenants performing read operations on 16-node or 32-node clusters in Marmot. The 16-node cluster comprises 1 NameNode and 16 DataNodes and the 32-node cluster comprises 1 NameNode and 32 DataNodes. Two tenants share the cluster and each tenant creates 16 and 32 parallel read processes on the 16-node cluster and 32-node cluster respectively. These read processes are evenly assigned to DataNodes. Each process reads 1 GB data (split into 16 chunks, 64 MB per chunk) from HDFS. We record the read bandwidth of each process and compare the average, minimum and maximum bandwidth of all processes. Figs. 13 and 14 show the comparison result in the two clusters. With Opass, the average bandwidth of each tenant can be increased by 3× in the 16-node cluster and by 4× in the 32-node cluster. With Opass, the minimum read bandwidth, which is the read bandwidth of the tenant's slowest process, is promoted by 2.4× in the 16-node cluster and 4.5× in the 32-node cluster.

Further, we plot the finish time taken to read every chunk on a 32-node cluster by the two tenants. Each tenant reads 512 chunks and the size of each chunk is 64 MB. As shown in Fig. 15, the average time on each I/O operation is 4.5× less than with use of the default dynamic assignment method.

5.1.5 *Evaluating Opass on Parallel Writes*

To show Opass reducing the imbalance of parallel writes in HDFS, the same testbeds, using *Marmot* and *HDFS*, is adopted to run comparison experiments. In Marmot, 16 nodes or 32 nodes are configured as 1 NameNode and 16 or 32 are configured as DataNodes. Parallel write processes are evenly assigned to DataNodes. Each process writes 1

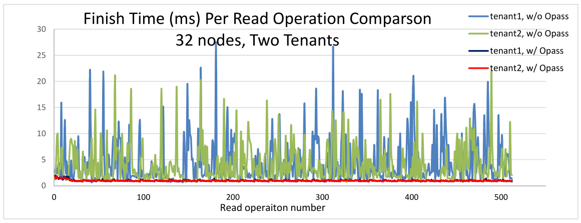


Fig. 15. Finish times of multi-tenant read operations on a 32-node cluster.

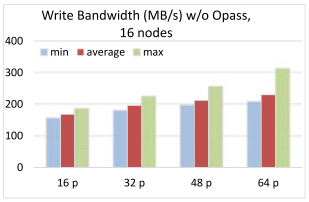


Fig. 16. Write bandwidths of processes on a 16-node cluster without Opass.

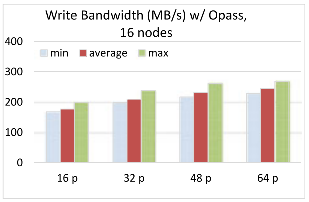


Fig. 17. Write bandwidths of processes on a 16-node cluster with Opass.

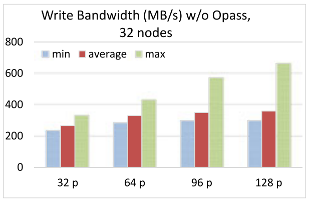


Fig. 18. Write bandwidths of processes on a 32-node cluster without Opass.

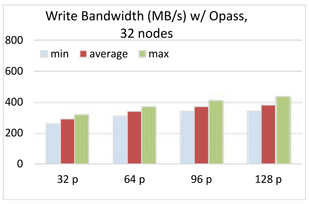


Fig. 19. Write bandwidths of processes on a 32-node cluster with Opass.

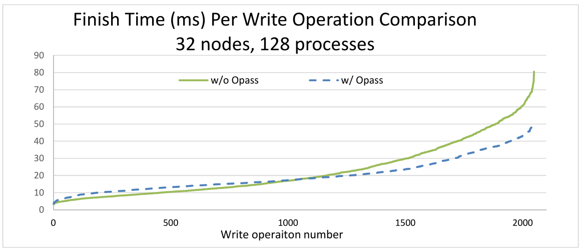


Fig. 20. Sorted finish times of parallel write operations on a 32-node cluster.

GB data (split into 16 chunks, 64 MB per chunk) to HDFS and the number of parallel processes varies from 16 to 128. We record the write bandwidth of each process and compare the average, minimum and maximum bandwidth of all processes. Figs. 16 and 17 show the comparison results in the 16-nodes testbed. With Opass, the imbalance of parallel writes (maximum bandwidth - minimum bandwidth) is reduced by 53.4 percent while the number of parallel write processes is increased to 64. The minimum bandwidth increases by more than 20 MB/s while the number of processes increased to 48 and 64. Figs. 18 and 19 show the comparison results in the 32-nodes testbed. With Opass, the imbalance of parallel writes is reduced by 72.6 percent while the number of parallel write processes is increased to 128. The minimum bandwidth increases by more than 40 MB/s while the number of processes increased to 96 and 128. In addition, as the number of parallel writes increases, the difference between the maximum and minimum bandwidth is controlled in a stable range (30 percent of minimum bandwidth).

To gain further insight into the finish time distribution of parallel writes, we plot the finish time taken to write every chunk on a 32-node cluster with 128 processes writing into HDFS. This testcase contains 2,048 chunks and the size of each chunk is 64 MB. We sort the finish times in ascending order and show the results in Fig. 20. Without the use of Opass, the finish times are initiated at lower level, but increases dramatically after the median. This is due to the fact that as the application runs, an increasing number of write requests are issued from parallel processes to HDFS, which causes contention on the disks and the network transfer on some hot DataNodes. In contrast, with the use of Opass, the finish times during the entire execution is much smoother. In all, this figure shows the detail of the effectiveness of Opass.

5.2 Evaluating ParaView with Opass

We also conduct experiments to test Opass on a real parallel application, ParaView. In our tests, 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]. Opass is incorporated on VTK MultiBlock datasets reader for the data task assignment. A multiblock dataset is a series of sub datasets, together they represent an assembly of parts or a collection of meshes.

To deal with MultiBlock datasets, a meta-file 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. Specifically, Opass is added into the vtkXMLCompositeDataReader class and called in the function ReadXMLData(), which assigns the data pieces to each data server after processing the meta-file. Through Opass, each data server process can receive the proper task assignment with it's associated data locally accessible. For our test data, we use the Macromolecular datasets obtained from a Protein Data Bank [5]. The processed output of these protein datasets are polygonal images. In our test, we take each dataset as a time step and convert it to a subset of ParaView's MultiBlock file. Due to the cost of downloading multiple datasets to the test system, we duplicate some datasets with a small revision and save them as new datasets in

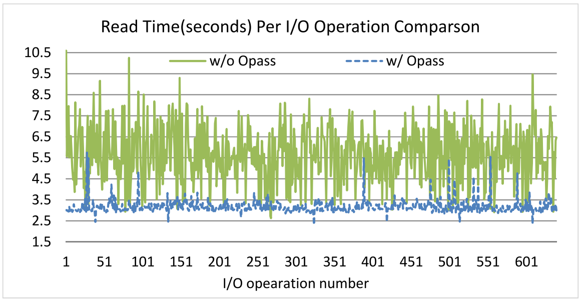


Fig. 21. Trace of time taken for each call to vtkFileSeriesReader with and without ppass supported ParaView.

“binary” mode. For each rendering, 64 datasets were selected from 640 datasets. As a result, our test set was approximately 26 GB in total size and 3.8 GB per rendering step. We use a python script to set up the visualization environment and needed filters to create a reproducible test. The script was submitted to the ParaView server via the provided pvbatch utility to produce a test.

Fig. 21 shows the traces of the request time for each call into vtkFileSeriesReader on a 64-node cluster on marmot. Each I/O operation performs on data about 56 MB in size. Without Opass, ParaView, in certain instances, is able to achieve very low read times with the fastest time of 2.63 seconds. However, these instances of quick access are negated by the instances where data is not locally available and must be fetched remotely. Overall, ParaView achieves an average read time of 5.48 seconds with a standard deviation of 1.339. With Opass, ParaView consistently achieves low read times, with a few outliers in which an I/O operation read time was above usual. Opass achieves an average read time of 3.07 seconds with a standard deviation of 0.316. We run the tests 5 times and the average execution time of Paraview with Opass is around 98 second while that of Paraview without Opass is around 167 seconds. This shows that the varied I/O time prolongs the overall execution.

5.3 Efficiency and Overhead Discussion

5.3.1 Efficiency of Opass

With the comparison of Figs. 8c, 10, and 12, we find that the improvements of the I/O time and data access balance vary between the three tests. This is due to the different I/O requirements 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 [34] may be needed. Data reconstruction or redistribution is beyond the scope of this paper.

Since Opass does not modify the design of HDFS, HDFS still controls 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 a local and balanced fashion.

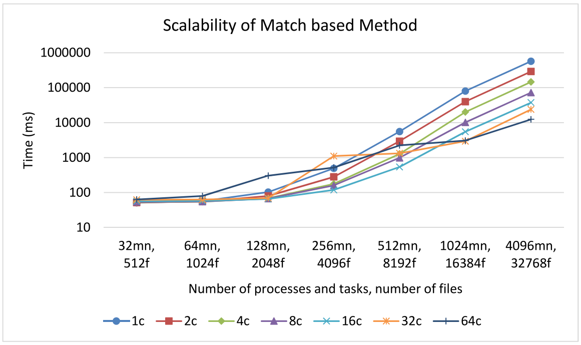


Fig. 22. Scalability of match based methods. Match based methods run on 1 to 64 cpu cores in a distributed fashion. The x-axis represents the scale of matching size. For example “32mn, 512f” means optimally matching 32 tasks and processes to 512 files. Each file has 16 data chunks.

5.3.2 Scalability

With our current experimental settings, we found that the overhead created by the matching method was less than 1 percent of the overhead involved with accessing the whole dataset. However, as the problem size becomes extremely large, the matching method may not be scalable. In order to scale our match method, we extend it via MPI framework, and run this parallel match method with various problem sizes. The idea of this parallel version is that we partition the computation intensive parts of the match method and executing them on multiple DataNodes as the problem size increases. The computation intensive parts include finding the optimal matching from tasks to processes and optimally distributing requests to target DataNodes. As shown in Fig. 22, with the computing cores increasing from 1 to 64, the performance of match method can speedup by 45× in the extreme problem size (4096 mn, 32768 f). In the extreme problem sizes, the performance can be proportionally increased as the number of cpu cores increase. The performance speedup 1.94× as the number of cores increases from 1 to 2, or 32 to 64.

Previous studies [35] have demonstrated that HDFS’s NameNode is a single container of the file system metadata, hence it naturally becomes a limiting factor for file system growth. In order to make metadata operations fast, the NameNode loads the whole namespace into its memory, and therefore the size of the namespace is limited by the amount of RAM available to the NameNode. The solutions to this scalability issue is based on partitioning and distributing the namespace to multiple metadata servers for workload balancing and reducing the single server’s workloads. For instance, Ceph [43] has a cluster of namespace servers and uses a dynamic subtree partitioning algorithm to map the namespace tree to servers evenly.

6 Related Works

Distributed file systems are ever-increasing in popularity for data-intensive analysis. 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. Tachyon [28] is a distributed file system enabling reliable data sharing at memory speed across cluster computing frameworks. Cranor et al. [12] propose methods to write parallel data into HDFS and achieve high I/O performance. MRAP [34] is proposed to reconstruct scientific data according to data access patterns to assist data processing using the Hadoop system. Sci-Hadoop [8] allows scientists to specify logical queries over array-based data models. Wang et al. [32], [42], [45] 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 reads 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 [47] allows tasks to wait for a small amount of time for achieving locality computation. Yarn [41], the new generation of Hadoop system, supports both MPI programs and MapReduce workloads. Mesos [22] is a platform for sharing commodity clusters between MPI and Hadoop jobs and guarantees performance isolation for these two parallel execution frameworks. Huang et al. [23] takes advantage of OS-Level virtualization to allocate shared distributed storage resources to users, in order to guarantee quality of services. 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.

Cranor et al. [6], [13], Jin et al. [26], [44] and Wang et al. [42] proposed methods to allow MPI-based programs to write data, in parallel, into HDFS and implememted an I/O middleware to fill the semantic gap between HPC concurrent random access and HDFS append access such as N-to-1 concurrent write on both PVFS [3] and HDFS. Cranor et al. [33] proposed a new middleware, IndexFS, based on table architecture to promote metadata and small file operations in distributed file systems such as PVFS and HDFS. Starfish et al. [21] is a self-tuning system for tuning big data analytics within the Hadoop Stack. It provides job-level, workflow-level and workload-level tunings for map-reduce jobs. To reduce the imbalance data placement of write-read workflow, it adopts a round-robin policy for chunk placement instead of the default random policy. Zaharia et al. [49] introduces a new job scheduling policy for heterogenous cluster to promote hadoop job throughputs, such as parallel write-intensive jobs. In summary, those research works achieved job-level or platform-level load balance and promoted the performance of parallel write in HDFS. However, they did not cover the imbalance issues of parallel writes at the storage level such as imbalanced parallel writes, parallel I/O congestion and imbalanced data placement. Our approaches give a set of middleware systems, algorithms and comprehensive analysis to tackle those issues.

7 Conclusion

In this paper, we investigate the problems of parallel data reads/writes on distributed file systems. Due to the lack of consideration of data distribution, parallel data requests are often served in an *imbalanced* and *remote* fashion, resulting in I/O contention on some storage nodes. To address the read problem, we model the data requests as a *bipartite matching* problem and propose novel matching based algorithms for optimizing parallel data reads. In terms of imbalanced write, we design HM-LRU policy to select the local optimal storage node. We also optimize the dynamic data requests in heterogeneous and multi-tenant environments. We conduct comprehensive experiments for different parallel data access strategies on PRObE's Marmot and the experimental results show the scalability and performance of Opass. With the use of our method, parallel data read requests can be served such that they benefit from locality and balanced read and thus achieve high I/O performance, e.g., averagely 2.7× speedup on dynamical parallel read requests. The imbalance of serving parallel write requests on DataNodes is reduced by 53.4 and 72.6 percent, as the number of parallel write processes is set to 64 and 128 respectively.

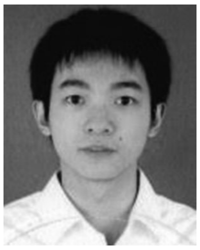
Acknowledgments

This work is supported in part by the US National Science Foundation Grant 1717388, CCF-1527249 and CCF-1337244. The experiments of this work are conducted on the PRObE Marmot cluster, which is supported in part by National Science Foundation awards CNS-1042537 and CNS-1042543 (PRObE). This work is supported in part by the National Science Foundation of China (Grant nos. 61373028 and 61672338).

References

[1] Hadoop MapReduce framework. (2016). [Online]. Available: http://en.wikipedia.org/wiki/mapreduce
[2] Libhdfs - hadoop wiki. apache software foundation, [Online]. Available: http://wiki.apache.org/hadoop/LibHDFS
[3] Parallel virtual file system version 2. (2003). [Online]. Available: http://www.pvfs.org/
[4] MPI-2: Extensions to the message-passing interface, Jul. 1997. [Online]. Available: http://parallel.ru/docs/parallel/mpi2
[5] E. E. Abola, F. C. Bernstein, and T. F. Koetzle, "Protein data bank," Neutrons in Biology. Springer US, p. 441, 1984.
[6] J. Bent, et al., "PLFS: A checkpoint filesystem for parallel applications," in *Proc. Conf. High Perform. Comput. Netw. Storage Anal.*, 2009, Art. No. 21.
[7] D. Borthakur, HDFS architecture guide, 2008. [Online]. Available: HADOOP APACHE PROJECT http://hadoop.apache.org/common/docs/current/hdfsdesign.pdf
[8] J. B. Buck, et al., "Scihadoop: Array-based query processing in Hadoop," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, Art. no. 66.
[9] M. Cannataro, D. Talia, and P. K. Srimani, "Parallel data intensive computing in scientific and commercial applications," *Parallel Comput.*, vol. 28 no. 5, pp. 673–704, May 2002.
[10] L. Chao, C. Li, F. Liang, X. Lu, and Z. Xu, "Accelerating Apache hive with MPI for data warehouse systems," in *Proc. IEEE 35th Int. Conf. Distrib. Comput. Syst.*, 2015, pp. 664–673.
[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, vol 2. Cambridge, MA, USA: MIT Press, 2001.
[12] C. Cranor, M. Polte, and G. Gibson, "Structuring PLFS for extensibility," in *Proc. 8th Parallel Data Storage Workshop*, 2013, pp. 20–26.
[13] C. Cranor, M. Polte, and G. Gibson, "Structuring PLFS for extensibility," in *Proc. 8th Parallel Data Storage Workshop*, pp. 20–26, 2013.
[14] T.-C. Dao and S. Chiba, "HPC-reuse: Efficient process creation for running MPI and Hadoop MapReduce on supercomputers," in *Proc. 16th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2016, pp. 342–345.
[15] A. Darling, L. Carey, and W.-C. Feng, "The design, implementation, and evaluation of mpiblast," presented at the *ClusterWorld*, San Jose, CA, USA, 2003.
[16] J. Dean and S. Ghemawat,"MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[17] L. R. Ford Jr and D. R. Fulkerson, "A suggested computation for maximal multi-commodity network flows," *Manag. Sci.*, vol. 5, no. 1, pp. 97–101, 1958.
[18] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, pp. 29–43, 2003.

[19] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd, "Probe: A thousand-node experimental cluster for computer systems research," The magazine of USENIX & SAGE, vol. 38, pp. 37–39, Jun. 2013.
[20] Y. Hahn and B. Lee, "Identification of nine human-specific frameshift mutations by comparative analysis of the human and the chimpanzee genome sequences," *Bioinf.*, vol. 21, no. suppl 1, pp. i186–i194, 2005.
[21] H. Herodotou, et al., "Starfish: A self-tuning system for big data analytics," in *Proc. Conf. Innovative Data Syst. Res.*, 2011, pp. 261–272.
[22] B. Hindman, et al., "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 22–22.
[23] D. Huang, J. Wang, Q. Liu, J. Yin, X. Zhang, and X. Chen, "Experiences in using OS-level virtualization for block I/O," in *Proc. 10th Parallel Data Storage Workshop*, 2015, pp. 13–18.
[24] S. Huang and A. W.-C. Fu, "k-balanced sorting and skew join in MPI and MapReduce," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 225–230.
[25] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, 2009, pp. 261–276.
[26] H. Jin, J. Ji, X.-H. Sun, Y. Chen, and R. Thakur, "Chaio: Enabling HPC applications on data-intensive file systems," in *Proc. 41st Int. Conf. Parallel Process.*, 2012, pp. 369–378.
[27] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Redwood City, CA, USA: Benjamin/Cummings, 1994.
[28] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proc. ACM Symp. Cloud Comput.*, 2014, pp. 1–15.
[29] H. Lin, X. Ma, W. Feng, and N. F. Samatova, "Coordinating computation and I/O in massively parallel sequence search," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 4, pp. 529–543, Apr. 2011.
[30] Z. Lin, S. Ethier, T. Hahm, and W. Tang, "Size scaling of turbulent transport in magnetically confined plasmas," *Phys. Rev. Lett.*, vol. 88, no. 19, 2002, Art. no. 195004.
[31] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "Datampi: Extending MPI to Hadoop-like big data computing," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 829–838.
[32] C. Mitchell, J. Ahrens, and J. Wang, "Visio: Enabling interactive visualization of ultra-scale, time series data via high-bandwidth distributed I/O systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, May 2011, pp. 68–79.
[33] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 237–248.
[34] S. Sehrish, G. Mackey, J. Wang, and J. Bent, "MRAP: A novel MapReduce-based framework to support HPC analytics applications with access patterns," in *Proc. 19th ACM Int. Symp. High Performance Distrib. Comput.*, 2010, pp. 107–118.
[35] K. V. Shvachko, "HDFS scalability: The limits to growth," *Mag. USENIX*, vol. 35, no. 2, pp. 6–16, 2010.
[36] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference*. Cambridge, MA, USA: MIT press, 1995.
[37] A. H. Squillacote, *The ParaView Guide: A Parallel Visualization Application*. New York, NY, USA: Kitware, 2007.
[38] B. J. Strasser, "Genbank–natural history in the 21st century?" *Sci.*, vol. 322, no. 5901, pp. 537–538, 2008.
[39] R. Sumbaly, J. Kreps, and S. Shah, "The big data ecosystem at linkedin," in *Proc. Int. Conf. Manag. Data*, 2013, pp. 1125–1134.
[40] A. Thusoo, et al., "Data warehousing and analytics infrastructure at Facebook," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2010, pp. 1013–1020.
[41] V. K. Vavilapalli, et al., "Apache Hadoop yarn: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, Art. no. 5.
[42] J. Wang, et al., "Sideio: A side I/O system framework for hybrid scientific workflow," *J. Parallel Distrib. Comput.*, vol. 18, pp. 45–58, 2017.
[43] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. 7th Symp. Operating Syst. Des. Implementation*, 2006, pp. 307–320.
[44] X. Yang, Y. Yin, H. Jin, and X.-H. Sun, "Scaler: Scalable parallel file write in HDFS," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2014, pp. 203–211.
[45] J. Yin, J. Wang, W.-C. Feng, X. Zhang, and J. Zhang, "SLAM: Scalable locality-aware middleware for I/O in scientific analysis and visualization," in *Proc. 23rd Int. Symp. High-Performance Parallel Distrib. Comput.*, 2014 pp. 257–260.
[46] J. Yin, J. Wang, J. Zhou, T. Lukasiewicz, D. Huang, and J. Zhang, "Opass: Analysis and optimization of parallel data access on distributed file systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 623–632.
[47] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.
[48] M. Zaharia, et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 2–2.
[49] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. Symp. Operating Syst. Des. Implementation*, vol. 8, 2008, Art. no. 7.
[50] M. J. Zaki, "Parallel and distributed association mining: A survey," *IEEE Concurrency*, vol. 7, no. 4, pp. 14–25, Oct. 1999.



Dan Huang received the bachelor and master's degrees from Jilin University and Southeast University, respectively. He is currently working toward the PhD degree in computer engineering program at University of Central Florida. Before this, he worked as a research assistant in Georgia State University. His research interests include distributed storage systems, virtualization technology, and the I/O of distributed system.

Dezhi Han received the BS degree from Hefei University of Technology, Hefei, China, the MS and PhD degrees from Huazhong University of Science and Technology, Wuhan, China. He is currently a professor of computer science and engineering with Shanghai Maritime University. His specific interests include storage architecture, cloud computing, cloud computing security, and cloud storage security technology.

Jun Wang is a full professor of computer science and engineering, and director of the Computer Architecture and Storage Systems (CASS) Laboratory, University of Central Florida, Orlando, Florida. He received the National Science Foundation Early Career Award 2009 and Department of Energy Early Career Principal Investigator Award 2005. He has authored more than 120 publications in premier journals such as the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, and leading HPC and systems conferences such as HPDC, EuroSys, ICS, Middleware, FAST, IPDPS.

Jiangling Yin received the BS and MS degrees in software engineering from the University of Macau, in 2011. He is working towards the PhD degree in computer engineering in the Electrical Engineering and Computer Science Department, University of Central Florida. His research focuses on energy-efficiency computing and file/storage systems.

Xunchao Chen received the MS degree in electrical engineering from the University of Texas at Dallas. Currently he is working towards the PhD degree in computer engineering at University of Central Florida. His research interests include data management in memory subsystem, NAND flash SSD, and emerging nonvolatile memory technologies (e.g., STT-RAM, DWM). He was also involved in various FPGA prototyping and embedded system design projects during his internships prior to PhD study.

Xuhong Zhang received the bachelor degree in software engineering from Harbin Institute of Technology, in China, 2011, the master's degree in computer science from Georgia State University, in 2013. His research areas include approximate computing, graph processing, and big data storage.

Jian Zhou is currently working toward the PhD degree in computer engineering at University of Central Florida. Before this, he was a research assistant with Huazhong University of Science and Technology. His current focus is developing new applications and data processing frameworks for persistent memory systems.

Mao Ye received the bachelor's degree from Zhejiang University in China and the master's degree in computer science from University of Central Florida, in 2010. Her research interest include machine learning, big data, and data-intensive high performance computing.