Parallel Computing 40 (2014) 697–709

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

SDAFT: A novel scalable data access framework for parallel BLAST

Jiangling Yin [a,*], Junyao Zhang [a], Jun Wang [a], Wu-chun Feng [b]

[a] EECS, University of Central Florida, Orlando, United States
[b] Department of Computer Science, Virginia Tech, Blacksburg, VA 2406, United States

ARTICLE INFO

Article history:
Available online 26 August 2014

Keywords:
MPI/POSIX I/O
HDFS
Parallel sequence search
mpiBLAST

ABSTRACT

In order to run tasks in a parallel and load-balanced fashion, existing scientific parallel applications such as mpiBLAST introduce a data-initializing stage to move database fragments from shared storage to local cluster nodes. Unfortunately, with the exponentially increasing size of sequence databases in today's big data era, such an approach is inefficient.

In this paper, we develop a scalable data access framework to solve the data movement problem for scientific applications that are dominated by "read" operation for data analysis. SDAFT employs a distributed file system (DFS) to provide scalable data access for parallel sequence searches. SDAFT consists of two interlocked components: (1) a data centric load-balanced scheduler (DC-scheduler) to enforce data-process locality and (2) a translation layer to translate conventional parallel I/O operations into HDFS I/O. By experimenting our SDAFT prototype system with real-world database and queries at a wide variety of computing platforms, we found that SDAFT can reduce I/O cost by a factor of 4–10 and double the overall execution performance as compared with existing schemes.

1. Introduction

Research in computational biology is extremely important to help people understand the composition and functionality of biological entities and processes [4,2,23,25,6,7]. Over the past decade, ever-increasing computational power and innovative parallel techniques [33] have significantly advanced the computing capabilities in this field. However, in today's big data era, rapidly growing data sets [2] are too large in size making it too costly to move data in existing computational systems. Specifically, these systems rely on a single link or a handful of network links to transfer data to and from the cluster [15]. Unfortunately, these links do not scale with the problem size of big data due to their limited throughput in relation to the data.

Modern MPI-based parallel applications adopt a compute-centric model, *i.e.*, moving data to compute processes. For instance, mpiBLAST [12,17], a parallel implementation of BLAST [6,7], implements a dynamic load balancing process scheduler in its workflow by taking every node load into consideration. The scheduler schedules each search task to execute at an available idle node, which retrieves needed data from a shared network file system. To mitigate data movement overhead, a local disk cache is implemented at every worker node that enables its running process to reuse locally stored history data.

* Corresponding author.

http://dx.doi.org/10.1016/j.parco.2014.08.001
0167-8191/

Although this scheme works well when a database is small, it does not scale up with exponentially growing databases [10,31,24] and can become a major stumbling block to high performance and scalability.

Distributed file systems, constructed from machines with locally attached disks, can scale with the problem size and number of nodes as needed. For instance, the Hadoop Distributed File System (HDFS) could support MapReduce applications to realize local data access. 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 needed data from storage and send it via a network to the compute node. Compared to the MPI-based parallel programming model, however, MapReduce does not allow for the flexibility and efficiency of complex expressions and communications with regards to scientific parallel applications [30,17].

In this paper, we develop a scalable data access framework to solve the scalability and data movement problems of MPI-based parallel applications which are dominated with read operation. SDAFT is an adaptive, data locality-aware middleware system that employs a scalable distributed file system to supply parallel I/O and dynamically schedules compute processes to access local data by monitoring physical data locations. Our main focus here is on well-recognized scientific parallel applications that are developed with the MPI programming model and that originally execute on compute-centric platforms rather than data-centric platforms. SDAFT allows these applications to benefit from data locality exploitation in HDFS, while also leveraging the flexibility and efficiency of the MPI programming model.

SDAFT consists of two important components: A process-to-data mapping scheduler (DC-scheduler) changes the compute-centric mapping relationship to a data-centric scheme: a computational process always accesses the data from a local (or nearby) computation node unless the computation is not executable in a distributed fashion. To solve all incompatibility issues between parallel I/O, such as *MPI_File_read()*, and DFS I/O, such as *hdfsRead()*, a virtual translation layer(SDAFT-IO) is developed to enable computational processes to execute parallel I/O on underlying distributed file systems.

We realize a SDAFT prototype system and use mpiBLAST as a case-study application to demonstrate the efficacy of SDAFT. In our work, SDAFT runs on the popular Hadoop Distributed File System (HDFS). Because every BLAST task is always scheduled to execute on an idle compute node and accesses the necessary data directly from local attached disks, the I/O cost is significantly reduced. By experimenting with the SDAFT prototype on PRObE's Marmot 128-node cluster testbed and a university on-site 46-node cluster, we find that SDAFT doubles the overall performance of existing mpiBLAST solutions when the problem becomes I/O-intensive by reducing the average parallel I/O cost prior to parallel BLAST execution by a factor of five as compared to current approaches.

The remainder of this paper is organized as follows: Section 2 discusses background information on parallel sequence search tools and the underlying file systems. Section 3 presents our proposed framework. Section 4 shows performance results and analysis. Section 5 discusses related work and future work, and Section 6 concludes the paper.

2. Background

2.1. Genomic sequence search and mpiBLAST

In computational biology, genomic sequencing tools are used to compare given query sequences against datatbases sequences. This is well recognized as important for identifying new sequences and studying their effects. Different alignment algorithms, such as Smith and Waterman [29], Needleman and Wunsch [22], FASTA [25], and BLAST [6,7] are representatives of this field. Among them, the BLAST family of algorithms is the most widely used in the study of biological and biomedical research. BLAST 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 [12] is a parallel implementation of NCBI BLAST that organizes all parallel processes into one master process and many workers processes. Before performing the actual search, the raw sequence database is formatted into many fragments and stored in a shared network file system with the support of MPI I/O or POSIX I/O operations. mpiBLAST follows a compute centric model and moves the requested database fragments to corresponding compute processes. A detailed workflow is illustrated in Fig. 1. To search in a large database, the I/O cost, which takes place before the real BLAST execution, takes a significant amount of time, especially on commodity clusters.

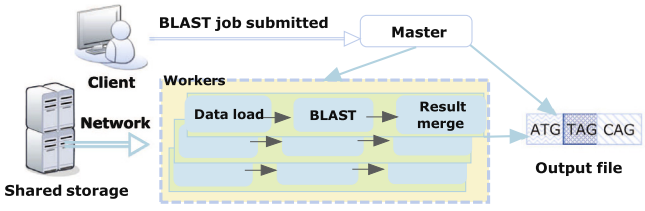

Fig. 1. In the default mpiBLAST framework, mpiBLAST workers load database fragments from a shared network file system and perform BLAST task according to the master scheduling.

2.2. The Hadoop Distributed File System

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 originally designed for the MapReduce framework, HDFS can be deployed to interface traditional parallel programs using its libHDFS library for C/C++ based programs. Client programs are able to make connections to the HDFS NameNode using libHDFS, and request chunk location information by querying 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 such information, the application can make informed decisions as to which node should execute a given piece of code so as to take advantage of data locality. This realizes local data access and avoids data movement in the network.

While HDFS can support analysis programs to benefit from data locality computation, there are several limitations 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 local UNIX file system or a shared network file system. However, HDFS has its own I/O interfaces, which allow programs to read and write data and are different from traditional MPI-I/O and POSIX-I/O. Moreover, MPI-based parallel applications usually produce distributed results and employ "concurrent write" method to output result, while HDFS only supports "append" write.

3. SDAFT design and implementation

3.1. Design motivations and system architectures

In computational biology, genomic sequence searching is well recognized as important for identifying new sequences and studying their effects. However, when sequence searches become both computationally and data intensive, running searches on a large-scale cluster in parallel could suffer from potentially long I/O latency resulting from non-negligible data movement, especially in commodity clusters. As discussed, parallel sequence searches could significantly benefit from local data access in a distributed fashion, similarly adopted by successful MapReduce systems.

There are two difficult problems to be resolved in order to achieve scalable data access for parallel search applications that execute on compute-centric platforms. The first is the ability to efficiently and effectively execute a traditional compute-centric MPI program on a cluster with co-located compute and storage resources. Since data is often statically distributed in HDFS, we need to dynamically move the computation tasks to the appropriate data. The second major challenge involves achieving load balance at runtime.

In order to solve the aforementioned issues, we develop a master–slave workflow in SDAFT. A master process is introduced to work as a front-end server in charge of process and data scheduling. It first collects unassigned fragment lists at all participating nodes, and then directs the slave processes to handle their local unassigned fragments. An unassigned fragment is defined as a database fragment that has not been searched against by a given query. According to the updated per node load reports, the master scheduler asks fast nodes to handle more fragments than slow nodes.

As illustrated in Fig. 2 the SDAFT framework consists of three important components, a translation I/O layer, a data centric load-balanced scheduler called DC-scheduler and a fragments location monitor. SDAFT-IO is developed to allow existing well-recognized parallel programs to transparently interface with Hadoop Distributed File System (HDFS), which is designed for MapReduce programs. The SDAFT-I/O is implemented as a non-intrusive software component added to existing application codes such that many successful performance tuned parallel algorithms and high performance noncontig-

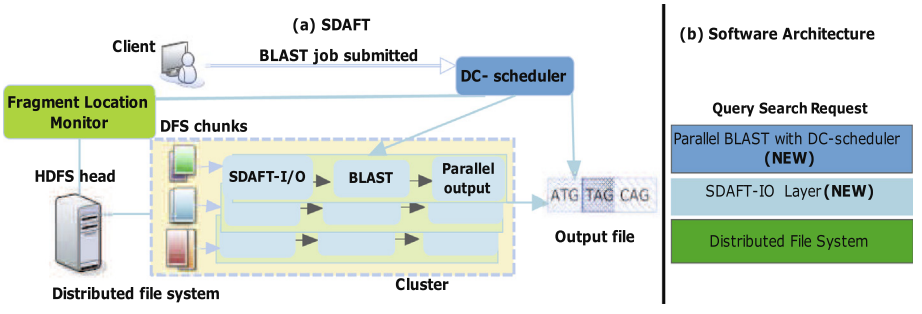

Fig. 2. Proposed BLAST workflow. (a) The DC-scheduler employs a Fragment Location Monitor to snoop the fragments location and dispatches unassigned fragments to computation processes such that each process could read the fragments locally, *i.e.*, reading HDFS chunks via SDAFT-IO. (b) The SDAFT software 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.

uous I/O optimization methods are inherited in SDAFT [17]. The DC-scheduler determines which specific fragments each node is assigned to search for. It aims to minimize the number of fragments pulled over the network. To get the physical location of all unassigned fragments, a Fragment Location Monitor is implemented between the DC-scheduler and HDFS. The monitor is invoked by the master scheduler process to report the database fragment locations. By tracking the location information, the DC-scheduler schedules worker processes to the appropriate compute nodes, moving computation to data. Through SDAFT-I/O, search processes can directly access fragments treated as chunks in HDFS from local hard drives, which is part of the entire HDFS storage. In general, the chunk size (set as 128 MB in our experiment) is bigger than a database fragment.

3.2. A data centric load-balanced scheduler

The key to realizing scalability and high-performance in big data parallel applications is to achieve both data locality and load balance at the same time. In practice, there exists several heterogeneity issues that could potentially result in load imbalance. First, in parallel sequence searches, the global database is formatted into many fragments. The search for query sequences is separated into a list of tasks corresponding to the number of fragments. The HDFS random chunk placement algorithm may distribute database fragments unevenly within the cluster, leaving some nodes with more data than others. Second, the execution time of a specific search task could vary greatly and is difficult to track according to the input data size and different computing capacities per node [17].

We implement a Fragment Location Monitor as a background daemon to periodically report updated unassigned fragment statuses to a master scheduler. At any point in time, the DC-scheduler always tries to launch a search task at the node that holds its requested fragments when the node is available. There exists a good chance to realize such data locality, as each data fragment has three physical copies in HDFS; namely, there are three different node candidates available for scheduling.

The scheduler is an independent running process as detailed in Algorithm 1. At first, the scheduler initializes a list of nodes C, where search processes will be launched. The input query sequences are divided into a list of individual tasks F. Each of these tasks searches a specific query sequence against a distinct database fragment. Also, the scheduler process will connect to the location monitor to get the distribution information for all fragments and generate an unassigned fragment list for each participating node. While F is not empty, each search process periodically reports to the scheduler for assignments. Upon receiving a task request from an idle computation process on node i, the scheduler process determines a task for the process as follows:

1. If the node i has some database fragments on its local disk, then the fragment x on its local disk that could make the number of unassigned fragments on all other nodes as balanced as possible will be assigned to the idle process. Fig. 3 illustrates an example to demonstrate how to assign an unassigned fragment to an idle process. In the example, there are 4 search processes running on 4 nodes, assuming $W1$ is idle and the unassigned fragment on $W1$ being $\langle f2, f4, f6\rangle$. The *argmin* value for $f2$ is 2, which is the minimum number of unassigned fragments on the nodes containing $f2$, namely node 2 and node 4. After assigning the fragment with largest *argmin* to $W1$, the number of unassigned fragments on node $2, 3$ and 4 are $2, 2, 2$.
2. If node i, on its local disk, does not contain any unassigned fragments, the scheduler will calculate *argmin* for all unassigned fragments and assign the fragment with the largest *argmin* to the idle process, which needs to pull data over network.

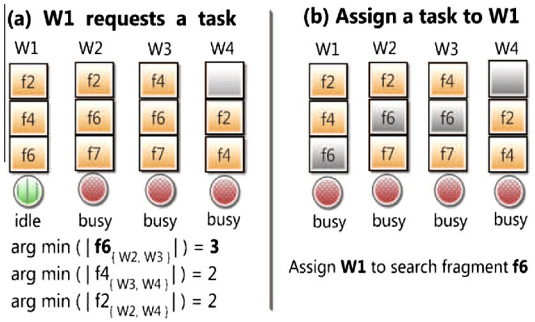

Fig. 3. A simple example where the scheduler receives the idle message from $W1$. The scheduler finds the available unassigned fragments on $W1$. The $f6$ will be assigned to $W1$ since the minimum unassigned fragments value is 3 on $W2$ and $W3$, which also contains $f6$. After assigning $f6$ to $W1$, the number of unassigned fragments on $W1$–4 is 2.

Algorithm 1. Data centric load-balanced scheduler algorithm

Let $C = \{c_1, c_2, \ldots, c_n\}$ be the set of participating nodes
Let $F = \{f_1, f_2, \ldots, f_m\}$ be the set of unassigned database fragments;
Let F_i be the set of unassigned fragments located on node i;
Steps:
Initialize C from MPI initialization
Initialize F for input query sequences
Invoke *Location monitor* and initialize F_i for each node i
while $|F| \neq 0$ **do**
  if a searching process on node i is idle **then**
    if $|F_i| \neq 0$ **then**
      Find $f_x \in F_i$ such that
      $x = \underset{x}{\operatorname{argmax}}(\underset{1 \leqslant k \leqslant n}{\operatorname{argmin}}(|F_k|))$
      Assign f_x to the idle process on node i
    else
      Find $f_x \in F$ such that
      $x = \underset{x}{\operatorname{argmax}}(\underset{1 \leqslant k \leqslant n}{\operatorname{argmin}}(|F_k|))$
      Assign f_x to the idle process on node i
    end if
    Remove f_x from F
    for all F_k s.t. $f_x \in F_k$ **do**
      Remove f_x from F_k
    end for
  end if
end while

The problem of scheduling tasks to multiple nodes, in order to minimize the longest execution time, is known to be NP-complete [13] 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.

Suppose there are $m = |F|$ search tasks with actual execution times of $t_1, t_2, \ldots, t_m$ on $n = |C|$ nodes. We use T^* to denote the maximum execution time of a node in the optimal solution, and T_k to denote the overall execution time on node k. First, notice that T^* cannot be smaller than the maximum execution time of a single task. Second, the best assignment we can wish for is $T_1 = T_2 = \cdots = T_n = \frac{\sum_{i=1}^{m} t_i}{n}$. These two observations gives us a lowerbound for T^*:

$$T^* \geqslant \max\left(\max_{1 \leqslant k \leqslant m}(t_k), \frac{\sum_{i=1}^{m} t_i}{n}\right). \tag{1}$$

Let T be the maximum execution time of a node in a solution given by our algorithm. We assume that the last completed task f_n is scheduled on node n; otherwise, we renumber the nodes. 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 idle nodes earlier. Therefore we have $T^* \geqslant s_{f_n}$. Because $T^* \geqslant \max_{1 \leqslant k \leqslant m}(t_k)$, we have $T^* \geqslant t_n$. This gives us the desired approximation bound: $T = s_{f_n} + t_n \leqslant 2T^*$.

The scheduling problem becomes 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. SDAFT-IO: a translation layer

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 shared network file system. SDAFT uses HDFS to replace these file systems, and therefore the I/O compatibility issues between MPI-based programs and HDFS must be resolved. More specifically, MPI-based parallel applications access files through MPI-I/O or POSIX-I/O interfaces, which are supported by local UNIX file systems or shared network file systems. These I/O interfaces are different from the I/O operations in HDFS. For example, *hdfsRead()*, instead of *MPI_File_read()* is used to read data from HDFS. These compatibility issues make scientific parallel applications unable to run on HDFS without some translational stage.

To address these compatibility issues, we implement a translation layer—SDAFT-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. SDAFT-I/O works as follows: SDAFT-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. At last, the correspondent hdfs I/O calls are triggered to execute specific I/O functions *e.g.* open/read/write/close.

Handling concurrent writes is a challenge in SDAFT. Parallel applications usually produce distributed results and may leave every engaged process write to disjointed 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. [17] 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 conventional shared network file system. However, HDFS only supports appended write, and most importantly, only one process is allowed to open and write to a file at a time, otherwise an open error will occur.

To resolve above incompatible I/O semantics, we revise "concurrent write to one file" to "concurrent write to multiple files". Fig. 4(a) and (b) demonstrate how SDAFT handles the concurrent write to output results. The master scheduler process creates an output directory in HDFS under which it maintains an index file to record the disjointed write ranges of each worker. Every worker will output their results independently into a physical file in HDFS. Logically, all output files produced for a data processing job are allocated in a same directory. The overall results are retrieved by reading the index file and joining all physical files under the same directory. Such a solution adopts the similar strategy with PLFS [11], which utilized the log structured file systems to improve the performance of checkpointing [27] for HPC simulation data. In this paper, we more focus on improving the read performance of data analysis program.

In the experiments, we prototyped SDAFT-I/O using FUSE [1], a framework for running stackable file systems in a non-privileged mode. The flow of an I/O call from an application to the Hadoop file system is illustrated in Fig. 5. The Hadoop file system is mounted on all participating cluster nodes through the SDAFT-I/O layer. The I/O operations of mpiBLAST are passed through a Virtual File System (VFS), taken over by SDAFT-I/O through FUSE, and then forwarded to Hadoop namenode, which will handle the I/O operations. Finally, HDFS is responsible for the actual data storage management. In regards to concurrent write, SDAFT-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/searchNTResult*30.

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. Specifically, each individual formatted file size should not exceed the configured chunk size, otherwise the file will be broken up into multiple chunks with each chunk being 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 would be lost. As a result, we should keep the file size of each database fragment smaller than the chunk size when formatting the database.

4. Experiments and analysis

We conducted comprehensive testing on our proposed scalable framework SDAFT on both Marmot and CASS clusters. Marmot is a cluster of PRObE on-site project [14] and housed at Carnegie Mellon University (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 2 TB 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 2.33 GHz Xeon Dual Core processors, 4 GB of memory, Gigabit Ethernet and a 500 GB SATA hard drive.

In both clusters, MPICH [1.4.1] is installed as parallel programming framework on CENTOS55–64 with kernel 2.6. We installed PVFS2 version [2.8.2] on the cluster nodes. PVFS uses with default configuration. We choose 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. We also run experiments on NFS as the developers of mpiBLAST use NFS as shared storage [17] and NFS is the default shared storage in our experimental cluster. The NFS configuration is a Dell MD1000 raid array (8 disks) connected with dual SAS channels and the bandwidth to the head node is also 1 Gb Ethernet. When we studied the performance of parallel BLAST, we scaled up the number of data nodes in the cluster and compared the performance with PVFS2, NFS and HDFS, respectively. For clarity, we labeled them as NFS-based, PVFS-based and SDAFT-based BLAST. We selected the nucleotide sequence database *nt* as our experimental database. At the time when we performed experiments, the *nt* database contained 17,611,492 sequences with a total raw size of about 45 GB. The data is downloaded from ncbi website [3] and saved to nfs, pvfs and hdfs with similar time cost.

The queries to input to search *nt* were generated as follows: we randomly chose some sequences from *nt* and revised them, which would guarantee that we will find some close matches in the database. In addition, we made up some

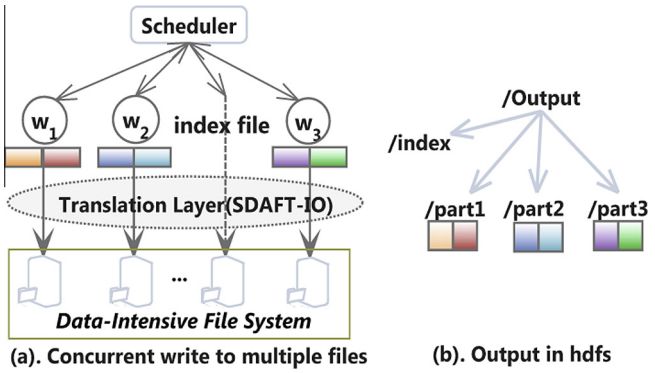

Fig. 4. How to handle "concurrent write" in SDAFT/HDFS. Figure (a) shows an instance consisting of three workers and one scheduler. Once the tasks are finished, the workers notify the scheduler and the scheduler issues an invalid partial result to each worker. The scheduler creates a directory in HDFS and writes the index file under the directory. Afterwards, all three workers write their results independently into the same directory in HDFS; figure (b) shows a sample logical organization of the output in HDFS.

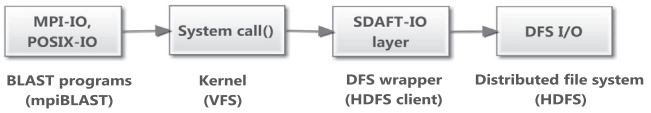

Fig. 5. The I/O call in our prototype. The kernel (VFS) redirects file system calls from parallel I/O to SDAFT-I/O. SDAFT-I/O wraps HDFS clients and translates the I/O call to DFS I/O.

sequences which may or may not find similar sequence matches from the database. We mixed up these sequences and fixed the query size to be 50 KB in all running cases, which generated the same output result in the amount of around 5.8 MB. The *nt* database was partitioned into 200 fragments.

4.1. Evaluating SDAFT with mpiBLAST

We compare SDAFT with a state-of-the-art mpiBLAST [1.6.0] solution in reference [17]. When running mpiBLAST on the cluster in parallel, users usually run *fastasplitn* and *formatdb* methods to format the database into fragments and reuse the formatted database fragments. Since each formatted fragment includes seven related files, the Hadoop Archive method should be applied to ensure that the seven files of a fragment are stored together in HDFS during a formatted execution.

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 next sequences searches. mpiBLAST is configured with two file systems—NFS and PVFS2 and both work as baselines. SDAFT employs HDFS as a distributed storage. Therefore, there is no need for gathering a fragment from multiple data nodes such as with PVFS, and we do not cache fragments in local disks either.

We study how SDAFT could improve the performance of 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 SDAFT-based BLAST. During the experiments, we mount NFS, HDFS and PVFS2 as 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 be 50 KB, which generated an output result in the amount of around 5.8 MB. The *nt* database was partitioned into 200 fragments.

4.1.1. Results from an Marmot cluster

The experimental results collected from Marmot are illustrated in Figs. 6–9.

When running parallel BLAST on a 108-node configuration system, we found the total program execution times on NFS-based, PVFS-based and SDAFT-based BLAST are 589.4, 379.7 and 240.1 s, respectively. We calculate the performance gain according to Eq. 2, where $T_{SDAFT-based}$ denotes the overall execution time of parallel BLAST based on SDAFT and $T_{NFS/PVFS-based}$ is the overall execution time of mpiBLAST based on NFS or PVFS.

$$improvement - percentage = 1 - \frac{T_{SDAFT-based}}{T_{NFS/PVFS-based}}. \quad (2)$$

As seen from Fig. 6, we conclude that SDAFT-based BLAST could reduce overall execution latency by 15% to 30% for small-sized clusters less than 32 nodes as compared to NFS-based BLAST. Given an increasing cluster size, SDAFT reduces overall execution time by a greater percentage, reaching to 60% for a 108-node cluster setting. This indicates that NFS-based

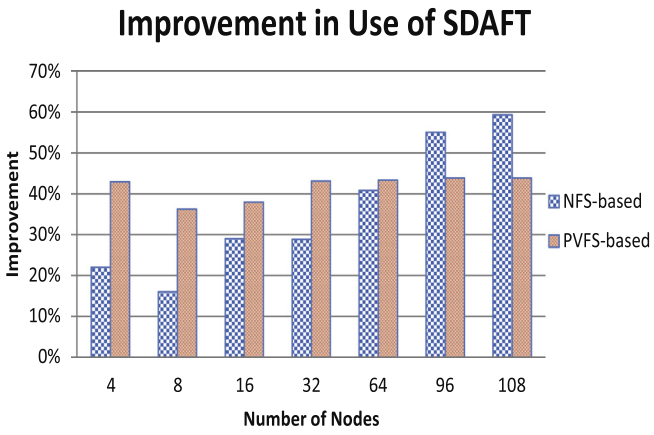

Fig. 6. The performance gain of mpiBLAST execution time when searching the *nt* database in use of SDAFT as compared to NFS-based and PVFS-based.

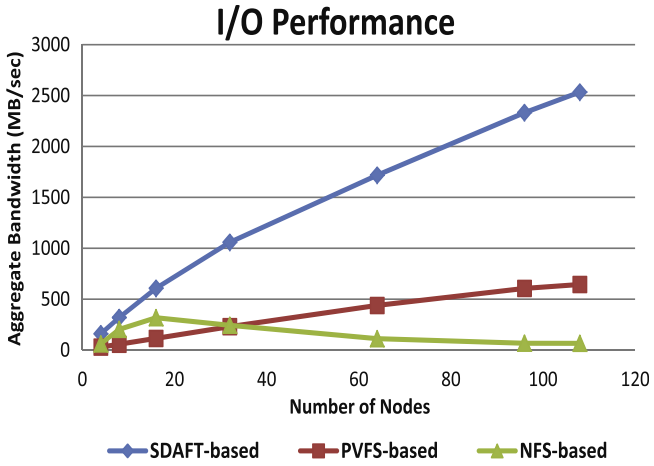

Fig. 7. The input bandwidth comparison of NFS-based, PVFS-based and SDAFT-based BLAST schemes. The key observation is that SDAFT scales linearly well for search workloads. NFS quickly peaks its bandwidth at a small cluster size and then decreases as the number of nodes increases. Similarly, PVFS also peaks in its ability to deliver data and then slightly grows with an increasing cluster size.

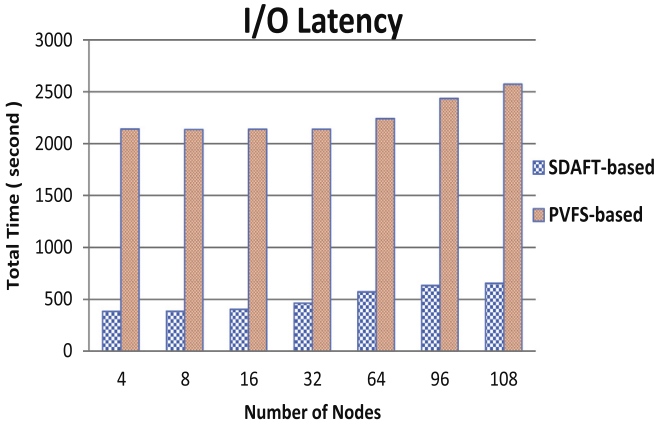

Fig. 8. The I/O latency comparison of PVFS-based and SDAFT-based BLAST schemes on the *nt* database with an increasing number of nodes.

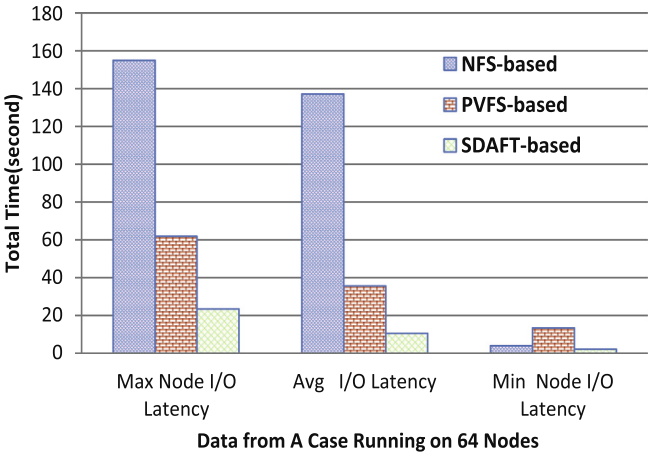

Fig. 9. The max and min node I/O time comparison of NFS-based, PVFS-based and SDAFT-based BLAST on the *nt* with varying number of nodes.

settings do not scale well. In comparison to PVFS-based BLAST, SDAFT runs consistently faster by about 40% for all cluster settings.

For a scalability study, we collected results of aggregated I/O bandwidth comparison illustrated in Fig. 7. As seen from the figure, we find that the I/O bandwidth scales up well with an increasing number of nodes for SDAFT. However, the NFS and PVFS based BLAST schemes achieve a much lower bandwidth. The bandwidth gap between SDAFT and the other two cases is further widened as the number of nodes increases. This indicates that a huge data movement overhead exists in both baselines that becomes a performance barrier to the system scalability.

To gain insight on how expensive data preparation is, we collected results of specific I/O latency as illustrated in Fig. 8. The I/O latency is the time cost on reading data from file system to the MPI processes' private space (local memory). We find that the total I/O latency of PVFS-based BLAST remains at about 2,000 s before 32-node settings and slightly stretches up for larger-sized clusters. The I/O latency of PVFS-based BLAST is as high as five times that of SDAFT-based BLAST. Moreover, we studied a case of running three schemes on a 64-node cluster and collected results of two extreme-case nodes, the slowest I/O and the fastest as illustrated in Fig. 9. It tells an intense contention for limited network bandwidth results in significant distinct node I/O latency, particularly severe in NFS and PVFS.

4.1.2. Results from a CASS cluster

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

Fig. 10 illustrates the average actual BLAST computation times (excluding I/O latency) for an increasing cluster size. We find that the average actual BLAST time in Fig. 10 decreases sharply as the number of nodes grows. Interestingly, the three systems obtain comparable BLAST performance. This matches our conjecture, as SDAFT only targets the I/O rather than the

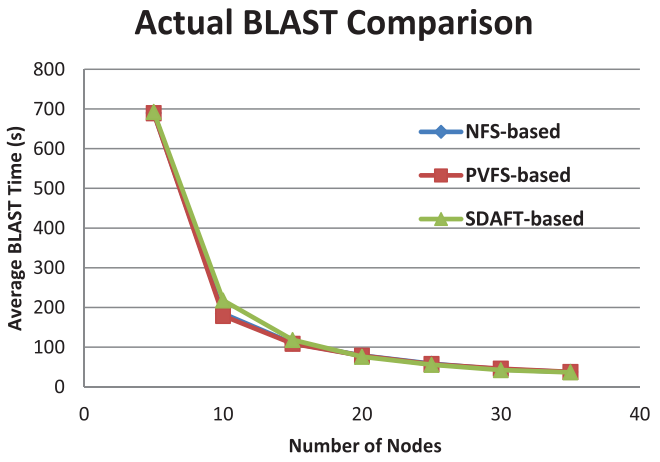

Fig. 10. The actual BLAST time comparison of NFS-based, PVFS-based and SDAFT-based BLAST programs on the *nt* database with different number of nodes.

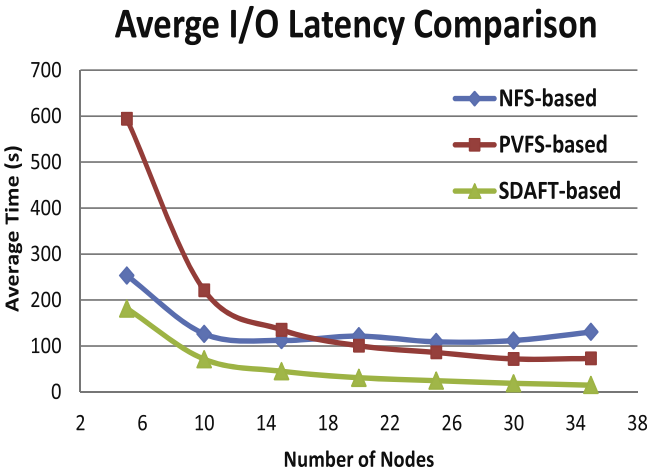

Fig. 11. The average I/O time of NFS-based, PVFS-based and SDAFT-based BLAST on the *nt* database with different number of nodes.

real BLAST computation. Different file system configurations—NFS, PVFS, and HDFS account for the differences among three BLAST programs. Fig. 11 illustrates the I/O phase of the BLAST workflow. In NFS and PVFS baselines, the average I/O cost stops declining after the cluster size exceeds 15. In contrast, SDAFT proves to be a scalable solution and achieves a decreasing I/O latency as cluster size grows. From Figs. 10 and 11, we can infer that as the query size becomes larger, the BLAST search time will increase under the three different configurations. However, the data input time will remain unchanged if the size of database is constant. Thus, the improvement due to I/O latency reduction will be decreased with an increasing size of query. Also, with an increasing size of database and an unchanged size of query, the improvement will be further enhanced.

From Figs. 7 and 11, we can find that shared network file system will peak in its ability to deliver data with an increasing cluster size in the environment of commodity cluster. In fact, even with high interconnect network on traditional HPC platforms, the systems' ability to deliver data will still be limited. For instance, our previous work VisIO [21] shows that Lustre file system will peak its bandwidth around 32-nodes in the traditional HPC platform.

The priority of our DC-scheduler is to achieve data-task locality subjected to a load balance constraint. To explore how effectively the DC-scheduler works (*i.e.*, to what extent search processes are scheduled to access fragments from local disks), Fig. 12 illustrates one snapshot of the fragments searched on each node and the fragments access by the network. We specifically ran the experiments five times to verify how much data moves through the network in a 30-node cluster, and track down a total number of fragments 150, 180, 200, 210, 240 respectively. As seen from the Fig. 12, most nodes search a comparable number of fragments locally. More than 95% of data fragments are read from local storage.

4.1.3. Comparing with Hadoop-based BLAST

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

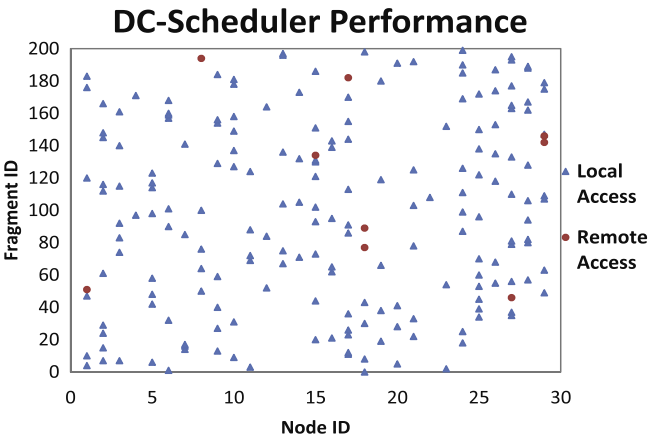

Fig. 12. This displays which data fragments are accessed locally on which node and involved in the search process. The blue crosses represent the data fragments accessed locally during the search, while the red dots represent the fragments accessed remotely.

We chose Hadoop-Blast [2.2.23] [5] as the Hadoop-based approach. The database for both programs are '*nt*' and the input query is same. With 25-nodes setting on marmot, SDAFT-based BLAST takes 568.664 s while Hadoop-Blast takes 1427.389 s. We run the tests several times, and the SDAFT-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 a heartbeat mechanism, (2) the advantages of I/O optimization based on MPI are not adapted by Hadoop-Blast, and (3) the difference of efficiency in Java and C/C++ implementations [32].

4.2. Evaluating SDAFT with newly developed MPI programs

In order to show that our proposed I/O layer and DC-scheduler could benefit other MPI-based applications with data locality computation, we develop a test program upon SDAFT with the use of the MPI programming model. The test program uses a master–slave architecture and employs the DC-scheduler to assign data processing tasks to parallel MPI processes. Specifically, the MPI process with rank 0 is developed to dynamically assign data processing tasks to slave processes that execute the assigned tasks. In our execution, we analyze genomic datasets of varying sizes. Since we are more concerned with the I/O performance, our test programs read all gene data into memory for sequence length analysis and exchange messages between MPI processes.

We compare the process time of our test program using the I/O layer with and without DC-scheduler, referred to as "SDAFT-based (with DC-scheduler), SDAFT-based (without DC-scheduler)", and no locality configuration with PVFS, referred to as "PVFS-based", on CASS for variable file sizes. We use 32 nodes for these experiments and show the performance comparison in Table 1. We find that the benchmark program using SDAFT consistently obtains lower process times than without SDAFT. With increased size of input data, the running time increases quickly for the test program without using SDAFT. This confirms that the data movement in a commodity cluster is the performance bottleneck. With the use of data locality computation, the performance for MPI-based application could be greatly improved. Moreover, through running our test program layer without the DC-scheduler, we find that the performance is only slightly better than that of PVFS-based test case. This is because the parallel processes need to read data remotely without the coordination of a data locality scheduler.

4.3. Efficiency of SDAFT-IO layer and HDFS

In the SDAFT framework, a translation layer—SDAFT-IO is developed to make parallel search codes able to execute on a distributed file system. In our prototype, we chose FUSE mount to transparently relink MPI program I/O operations to DFS I/O interface. This choice was made because FUSE is a popular program recognized in the community and we require a working system for proof of concept.

However, the overhead of a FUSE-based implementation must be evaluated. Since SDAFT-IO is built through 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 through the VFS's generic system call first (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 introduce.

We run the search programs and measured the time it takes to open the 200 formatted fragments. We performed two kind of tests. The first test directly uses the HDFS library while the other utilizes a default POSIX I/O, running HDFS file open through our translation 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 SDAFT-IO is around 15 s. The time through direct HDFS I/O was 25 s. This may derive from the overhead incurred by connecting and disconnecting with hdfsConnect() independently for each file. A second experiment involved running the BLAST process on multiple nodes through SDAFT-IO. The average time to open a file in HDFS is around 0.075 s, which is negligible compared to the overall I/O latency and BLAST time. In all, a FUSE based implementation has negligible overhead. Sometimes, SDAFT-IO 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 SDAFT, 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 communicating with Hadoop's Namenode. We evaluated the HDFS overhead by retrieving the physical location of 200 formatted fragments. The average time is around 1.20 s, which accounts for a very small portion out of the overall I/O latency.

Table 1
Data processing time comparison of our test program(s).

File size (GB)	7.5	30	120	335	670
PVFS-based	24	98	338	1084	2161
SDAFT-based (with DC-scheduler)	12	38	143	369	787
SDAFT-based (without DC-scheduler)	19	77	263	845	1631

5. Related work

There are many other methods that seek to increase the efficiency of parallel BLAST. ScalaBLAST [23] is a highly efficient parallel BLAST, which organizes processors into equally sized groups and assigns a subset of input queries to each group. ScalaBLAST can use both distributed memory and shared memory architectures to load the target database. Lin et al. [16] developed another efficient data access method to deal with initial data preparation and result merging in memory. MR-MPI-BLAST [32] is a parallel BLAST application employing a MapReduce-MPI library developed by Sandia Labs. These parallel applications benefit from the flexibility and efficiency of the HPC programming model but still follow a compute-centric model. AzureBlast [18] is a case study of developing science applications such as BLAST on the cloud. CloudBLAST [19] 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 [5] and bCloudBLAST [20] present a MapReduce-parallel implementation for BLAST but do not adopt existing advanced techniques like collective I/O or computation and I/O coordination. Our SDAFT is orthogonal to these techniques and allow parallel BLAST applications to benefit from data locality exploitation in HDFS while maintaining the flexibility and efficiency of the MPI programming model.

There are existing methods that are used to improve the parallel I/O performance. Sigovan et al. [28] 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. Prabhakar et al. [26] propose a disk-cache and parallelism aware I/O scheduling to improve storage system performance. FlexIO [35] is a middleware that offers simple abstractions and diverse data movement methods to couple simulation with analytics. Sun et al. [34] propose a data replication scheme(PDLA) to improve the performance of parallel I/O systems. Janine et al. [9] develop a platform which realizes efficient data movement between in situ and in-transit computations that perform on large-scale scientific simulations. Haim Avron et al. [8] develop an algorithm that uses a memory management scheme and adaptive task parallelism to reduce the data-movement costs. VisIO [21] obtains a linear scalability of I/O bandwidth for ultra-scale visualization. The VisIO implementation calls the HDFS I/O library directly from the application programs, which is an intrusive scheme and requires significant hard coding effort to rewrite the ParaView read methods. Different from above methods, our SDAFT uses an I/O middleware to allow parallel applications to achieve scalable data access with an underlying distributed file system.

6. Conclusions

In this paper, we developed a new scalable distributed framework to dramatically improve the I/O performance for MPI-based parallel search applications. We proposed a data-centric load-balancing scheduler to exploit data-task locality and enforce load balance within the cluster. The scheduler is independent of specific search tools and it can be adopted for other MPI-based applications that benefit from some form of data locality computation. In addition, we developed a SDAFT-IO layer to allow traditional MPI or POSIX based applications to run over a data-intensive distributed file system. Although we prototyped SDAFT-IO for mpiBLAST applications, it is applicable to any MPI-based parallel applications to run over HDFS without any recompiling effort. By conducting comprehensive experiments over two different clusters, we found that SDAFT can reduce I/O cost by a factor of 4–10 and double the overall execution performance as compared with existing schemes.

Acknowledgments

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 Grants CNS-1115665, CCF-1337244 and National Science Foundation Early Career Award 0953946.

References

[1] Fuse: <http://fuse.sourceforge.net/>.
[2] Genomes: <http://aws.amazon.com/1000genomes/>.
[3] Genomes database: <ftp://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/>.
[4] Genomes to life project proposal. <www.genomes2life.org/SNL-ORNL-GTL-Proposal.doc>.
[5] Running hadoop-blast in distributed hadoop. <http://salsahpc.indiana.edu/tutorial/hadoopblast.html>.
[6] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, D.J. Lipman, et al, Basic local alignment search tool, J. Mol. Biol. 215 (3) (1990) 403–410.
[7] S.F. Altschul, T.L. Madden, A.A. Schäffer, J. Zhang, Z. Zhang, W. Miller, D.J. Lipman, Gapped blast and psi-blast: a new generation of protein database search programs, Nucleic Acids Res. 25 (17) (1997) 3389–3402.
[8] H. Avron, A. 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, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, pp. 102:1–102:11.
[9] J.C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, J. 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, IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, pp. 49:1–49:9.
[10] D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, E.W. Sayers, Genbank, Nucleic Acids Res. 38 (Suppl. 1) (2010) D46–D51.

[11] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, M. Wingate. PLFS: a checkpoint filesystem for parallel applications, in: 2009 ACM/IEEE Conference on Supercomputing, November 2009.
[12] A. Darling, L. Carey, W.-C. Feng. The design, implementation, and evaluation of mpiBLAST. Proceedings of ClusterWorld, vol. 2003, 2003.
[13] M.R. Garey, D.S. Johnson, R. Sethi, The complexity of flowshop and jobshop scheduling, Math. Oper. Res. 1 (2) (1976) 117–129.
[14] G. Gibson, G. Grider, A. Jacobson, W. Lloyd. Probe: A Thousand-node Experimental Cluster for Computer Systems Research. vol. 38, June 2013.
[15] G. Grider, H. Chen, J. Nunez, S. Poole, R. Wacha, P. Fields, R. Martinez, P. Martinez, S. Khalsa, A. Matthews, G. Gibson. Pascal – A new parallel and scalable server IO networking infrastructure for supporting global storage/file systems in large-size Linux clusters, in: 25th IEEE International Performance, Computing, and Communications Conference, IPCCC 2006, 2006, pp. 10,340.
[16] H. Lin, X. Ma, P. Chandramohan, A. Geist, N. Samatova. Efficient data access for parallel blast, in: Parallel and Distributed Processing Symposium, Proceedings of 19th IEEE International, April 2005, pp. 72b–72b.
[17] H. Lin, X. Ma, W. Feng, N.F. Samatova, Coordinating computation and i/o in massively parallel sequence search, IEEE Trans. Parallel Distrib. Syst. 22 (4) (Apr. 2011) 529–543.
[18] W. Lu, J. Jackson, R. 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, ACM, New York, NY, USA, 2010, pp. 413–420.
[19] A. Matsunaga, M. Tsugawa, J. Fortes. Cloudblast: combining mapreduce and virtualization on distributed resources for bioinformatics applications, in: eScience, eScience '08, December 2008, pp. 222–229.
[20] Z. Meng, J. Li, Y. Zhou, Q. Liu, Y. Liu, W. Cao. bcloudblast: an efficient mapreduce program for bioinformatics applications, in: 2011 4th International Conference on Biomedical Engineering and Informatics (BMEI), vol. 4, 2011, pp. 2072–2076.
[21] C. Mitchell, J. Ahrens, J. Wang. Visio: enabling interactive visualization of ultra-scale, time series data via high-bandwidth distributed i/o systems, in: Parallel Distributed Processing Symposium (IPDPS), IEEE International, May 2011, pp. 68–79.
[22] S.B. Needleman, C.D. Wunsch, et al, A general method applicable to the search for similarities in the amino acid sequence of two proteins, J. Mol. Biol. 48 (3) (1970) 443–453.
[23] C. Oehmen, J. Nieplocha, Scalablast: a scalable implementation of blast for high-performance data-intensive bioinformatics analysis, IEEE Trans. Parallel Distrib. Syst. 17 (8) (2006) 740–749.
[24] J. Ostell, Databases of discovery, Queue 3 (3) (2005) 40–48.
[25] W.R. Pearson, D.J. Lipman, Improved tools for biological sequence comparison, Proc. Nat. Acad. Sci. 85 (8) (1988) 2444–2448.
[26] R. Prabhakar, M. Kandemir, M. Jung, Disk-cache and parallelism aware i/o scheduling to improve storage system performance, Int. Symp. Parallel Distrib. Process. (2013) 357–368.
[27] M. Rosenblum, J.K. Ousterhout, The design and implementation of a log-structured file system, ACM Trans. Comput. Syst. 10 (1) (1992) 26–52.
[28] C. Sigovan, C. Muelder, K. Ma, J. Cope, K. Iskra, R.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.
[29] T. Smith, M. Waterman, Identification of common molecular subsequences, J. Mol. Biol. 174 (2) (1881) 195–197.
[30] A.H. Squillacote, The ParaView Guide: A Parallel Visualization Application, Kitware, 2007.
[31] B.J. Strasser, Genbank–natural history in the 21st century?, Science 322 (5901) (2008) 537–538
[32] S.-J. Sul, A. Tovchigrechko. Parallelizing blast and som algorithms with mapreduce-mpi library, in: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011, pp. 481–489.
[33] C. Wu, A. Kalyanaraman, An efficient parallel approach for identifying protein families in large-scale metagenomic data sets, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC '08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 35:1–35:10.
[34] Y. Yin, J. Li, J. He, X.-H. Sun, R. Thakur. Pattern-direct and layout-aware replication scheme for parallel i/o systems, in: 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS), 2013, pp. 345–356.
[35] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, H. Yu. Flexio: I/o middleware for location-flexible scientific data analytics, in: 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS), 2013, pp. 320–331.