SDAFT: A Novel Scalable Data Access Framework for Parallel BLAST

Jiangling Yin, Junyao Zhang, Jun Wang
Department of Electrical Engineering and Computer Science
University of Central Florida
Orlando, Florida 32826
{jyin,junyao,jwang}@eecs.ucf.edu

Wu-chun Feng
Department of Computer Science
Virginia Tech
Virginia Tech, Blacksburg, VA 2406
wfeng@vt.edu

ABSTRACT

To run search tasks in a parallel and load-balanced fashion, existing parallel BLAST schemes such as mpiBLAST introduce a data initialization preparation stage to move database fragments from the shared storage to local cluster nodes. Unfortunately, a quickly growing sequence database becomes too heavy to move in the network in today's big data era.

In this paper, we develop a Scalable Data Access Framework (SDAFT) to solve the problem. It 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 to 10 and double the overall execution performance as compared with existing schemes.

Keywords

MPI/POSIX I/O, HDFS, Parallel Sequence Search, mpiBLAST

1. INTRODUCTION

Research in computational biology is extremely important to help people understand the composition and functionality of biological entities and processes. In the past decade, the ever-increasing computational power and parallel techniques [14] have significantly advanced the computing capabilities of the gene database search. However, in today's big data era, a rapidly growing sequence database [1] becomes too heavy to move among existing computational systems. These systems rely on a single or handful network links to transfer data to and from the cluster [8]. Unfortunately, these links do not scale with the big data problem size.

DISCS-2013 November 18, 2013, Denver, CO, USA

http://dx.doi.org/10.1145/2534645.2534647 .

Modern MPI-based parallel applications adopt a compute-centric model, *i.e.*, moving data to compute processes. For instance, the mpiBLAST work [6, 9], a parallel implementation of NCBI BLAST, implements a dynamic load balancing process scheduler in its workflow by taking every node load into consideration. It schedules each search task to execute at an available idle node, which gets needed data from a shared global file system. To mitigate data movement overhead, a local disk cache is implemented at every worker node that enables its running process be able to reuse locally stored history data. Although this scheme works well when database is small, it does not scale up with the exponentially growing database 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) is designed for MapReduce applications to realize local data access. The idea behind HDFS is that it is faster and more efficient to send the compute executables to the stored data and process 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 complexity expression in the implementation of scientific applications.

In this paper, we develop a Scalable Data Access Framework (SDAFT) to solve the scalability and data movement problem for MPI-based parallel applications. 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. There are two important components in SDAFT. 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 a incompatibility issue, a virtual translation layer(SDAFT-IO) is developed to enable computational processes to execute parallel I/O on underlying distributed file systems. By experimenting the SDAFT prototype on two clusters, we found that SDAFT doubles the overall performance of existing parallel solutions when the search becomes I/O intensive.

The rest of this paper is organized as follows: Section 2

presents our proposed framework. Section 3 shows performance results and analysis. Section 4 discusses related work and Section 5 concludes the paper.

2. SDAFT DESIGN AND IMPLEMENTATION

2.1 Design Motivations and System Architectures

In computational biology, genomic sequencing search is well recognized important for identifying new sequences and studying their effects. However, when sequence search becomes both computational 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 we discussed, parallel sequence search 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 for achieving scalable data access for parallel search applications that execute at compute-centric platforms. The first one is to implement the co-located compute and storage property in a compute-centric MPI runtime platform. Since data are often statically distributed in HDFS, we need to dynamically move the computation task to appropriate data. The second one is to achieve load balance at runtime.

In order to solve the aforementioned issues, we developed 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 the database fragment that has not been searched against by the given query. According to the updated per node load reports, the master scheduler asks fast nodes to handle more fragments than slow nodes.

The SDAFT framework consists of several important components, a translation I/O layer, a data centric load-balanced scheduler called DC-scheduler and a fragments location monitor, as illustrated in Figure 1. SDAFT-IO is developed to make existing well-recognized parallel programs transparently interface with Hadoop Distributed File System (HDFS), which is actually designed for MapReduce programs. The SDAFT-I/O is implemented as an non-intrusive software component added to existing application codes such that many successful performance tuned parallel algorithms and high performance noncontiguous I/O optimization methods are inherited in SDAFT [9]. 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, DC-scheduler schedules worker processes at the appropriate compute nodes, namely, moves 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 unit.

2.2 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 reality, there exist several heterogeneity issues that could potentially result in load imbalance. First, in parallel sequence search, 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. 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 a lot and is hard to track according to the input data size and different computing capacities per node [9].

We implement a fragment location monitor as a background daemon to periodically report updated unassigned fragment status to a master scheduler. At any point of time, DC-scheduler always tries to launch a search task at the node that holds its requested fragments when the node is available. There exist 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, which are to be launched search processes. 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 host disk that could make the number of unassigned fragments on all other nodes as balanced as possible will be assigned to the idle process. Figure 2 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 $< f2, f4, f6 >$. The $argmin$ value for $f2$ is 2, which 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.

The problem of scheduling tasks to multiple nodes, in order to minimize the longest execution time, is known to be NP-complete [7] when the number of nodes is greater than 2, even for the case that all tasks are executed locally. The scheduling problem becomes much harder when we take the

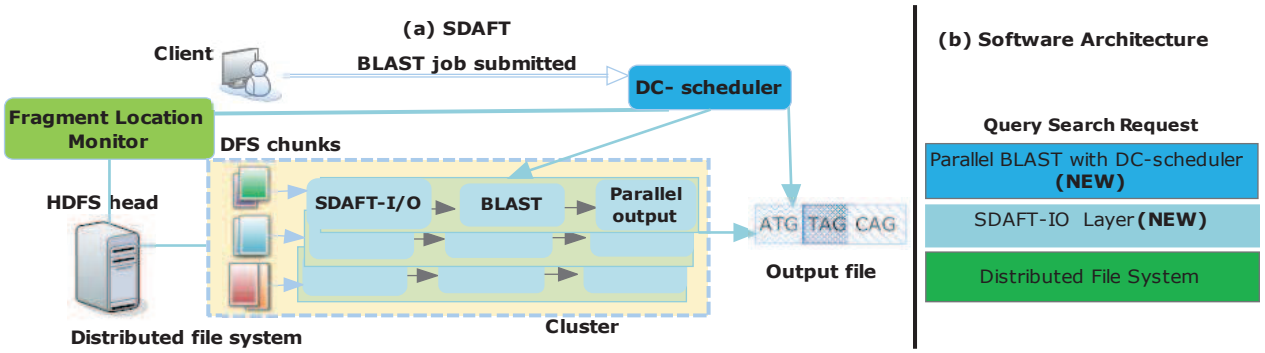

Figure 1: Proposed BLAST workflow. (a) The DC-scheduler employs a Fragment Location Monitor to snoop the fragments location and dispatches unassigned fragments to computation processes such that each process could read the fragments locally, *i.e.*, reading HDFS chunks via SDAFT-IO. (b) The SDAFT software 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.

Algorithm 1 Data centric load-balanced Scheduler Algorithm

Let $C = \{c_1, c_2, ..., c_n\}$ be the set of participating nodes
Let $F = \{f_1, f_2, ..., f_m\}$ 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 *Locationmonitor* 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
      $x = \underset{x}{\text{argmax}}(\underset{1 \leq k \leq n}{\text{argmin}}(|F_k|))$
      Assign f_x to the idle process on node i
    else
      Find $f_x \in F$ such that
      $x = \underset{x}{\text{argmax}}(\underset{1 \leq k \leq n}{\text{argmin}}(|F_k|))$
      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

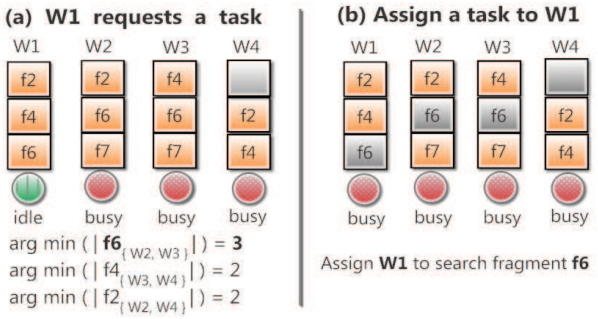

Figure 2: A simple example where the scheduler receives the idle message from $W1$. The scheduler finds the available unassigned fragments on $W1$. The $f6$ will be assigned to $W1$ since the minimum unassigned fragments value is 3 on $W2$ and $W3$, which also contains $f6$. After assigning $f6$ to $W1$,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

location variable into consideration. However, we will conduct real experiments to examine its locality and parallel execution in Section 3.

2.3 SDAFT-IO: A Translation Layer

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

To solve the problem, we implement a translation layer—SDAFT-IO to handle the incompatible I/O semantics. The basic idea is to transparently transform high-level I/O operations of parallel applications to standard HDFS I/O calls. We elaborate how SDAFT-IO works as follows. SDAFT-IO first connects to the HDFS server using hdfsConnect() and mounts HDFS as a local directory at correspondent 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.

How to handle concurrent write is a real challenge in SDAFT. Parallel applications may produce distributed results and leave every engaged process to write to a shared file. For instance, mpiBLAST takes the advantage of Independent/Collective I/O to optimize the searched output.

Figure 3: The flow of I/O call in our prototype from mpiBLAST to HDFS through SDAFT-I/O

The *WorkerCollective* output strategy introduced by Lin *et. al.* [9] realizes a concurrent write semantics, which can be interpreted as "multiple processes write to a single file at the same time ". These concurrent write schemes often work well with parallel file systems or network file systems. However, recent data-intensive distributed file systems such as HDFS only support appended write, and most importantly, only one process is allowed to open the file for writing.

To resolve the incompatible I/O semantics issue, we revise "concurrent write to one file "to "concurrent write to multiple files ". We demonstrate how SDAFT handles the concurrent write to output results as follows. The master scheduler process creates an output directory in HDFS under which it maintains an index file to record the 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 query sequence 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.

In the experiments, we prototyped SDAFT-IO using FUSE [2], a framework for running stackable file systems in a non-privileged mode. The flow of an I/O call from application to Hadoop file system is illustrated in Figure 3. The Hadoop file system is mounted on all participating cluster nodes through the SDAFT-IO layer. The I/O operations of mpiBLAST are passed through a virtual file system (VFS), taken over by SDAFT-IO, and then forwarded to HDFS. HDFS is responsible for the actual data storage management.

3. 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 and housed at CMU in Pittsburgh. The system has 128 nodes / 256 cores and each node in the cluster has dual 1.6GHz AMD Opteron processors, 16GB of memory, Gigabit Ethernet, and a 2TB Western Digital SATA disk drive. CASS consists of 46 nodes on two racks, one rack including 15 compute nodes and one head node and the other rack containing 30 compute nodes. Each node is equipped with dual 2.33GHz Xeon Dual Core processors, 4GB of memory, Gigabit Ethernet and a 500GB SATA hard drive.

In both clusters, MPICH [1.4.1] is installed as parallel programming framework on CENTOS55-64 with kernel 2.6. We installed PVFS2 version [2.8.2] on the cluster nodes. We also run experiments on NFS as the developers of mpiBLAST use NFS as shared storage [9]. Both PVFS and NFS are used with default configuration. We chose Hadoop 0.20.203 as the distributed file system, which is configured as follows: one node for the NameNode/JobTracker, one node for the secondary NameNode, and other nodes as the DataNode/TaskTracker. When we studied the performance of parallel BLAST, we scaled up the number of data nodes

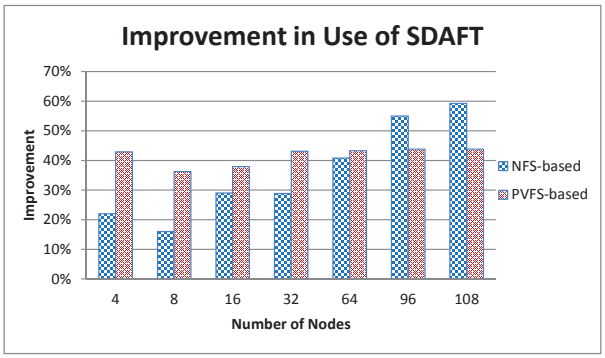

Figure 4: The performance gain of execution time when searching the *nt* database in use of SDAFT as compared to NFS-based and PVFS-based.

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. As the time when we did experiments, the *nt* database contained 17,611,492 sequences with a total raw size of about 45 GB. The queries to input to search *nt* are generated as follows: we randomly choose some sequences from *nt* and revise them, which would guarantee that we can find some close matches in the database. In addition, we make up some sequences which may or may not find similar sequence matches from the database. We mix up these sequences and fix 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.

When running parallel BLAST on Marmot with 108-nodes, we found the total program execution time with NFS-based, PVFS-based and SDAFT-based BLAST are 589.4, 379.7 and 240.1 seconds, respectively. We calculate the performance gain as, $improvement = 1 - \frac{T_{SDAFT-based}}{T_{NFS/PVFS-based}}$, where $T_{SDAFT-based}$ denotes the execution time of parallel BLAST based on SDAFT and $T_{NFS/PVFS-based}$ is the execution time of mpiBLAST based on NFS or PVFS. As seen from Figure 4, we concluded 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 setting is not scaling well. In comparison to PVFS-based BLAST, SDAFT runs consistently faster by about 40% for all cluster settings.

For a scalability study on Marmot, we collected results of aggregated I/O bandwidth comparison illustrated in Figure 5. 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 baselines is further widening as the number of nodes increase. This indicates huge data movement overhead in both baselines does become a performance barrier to the system scalability.

For comprehensive testing, we carried out similar exper-

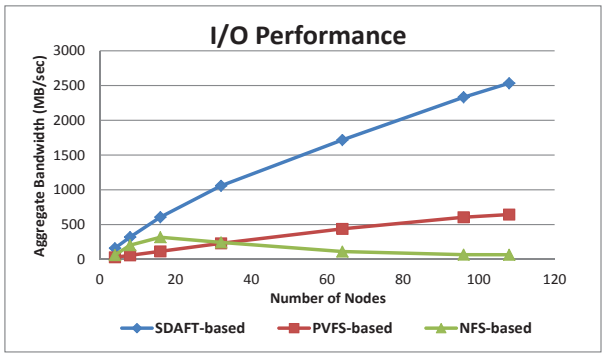

Figure 5: The input bandwidth comparison of NFS-based, PVFS-based and SDAFT-based BLAST schemes.

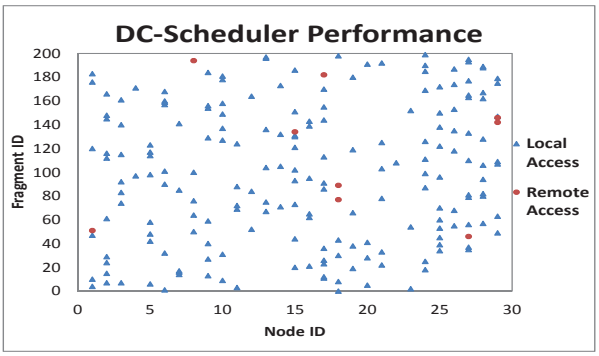

Figure 6: This displays which data fragments are accessed locally on which node and involved in the search process. The blue crosses represent the data fragments accessed locally, while the red dots represent the fragments accessed remotely.

iments and achieved similar results on the CASS cluster. Due to limited space, we don't present them here. To explore how effectively DC-scheduler works (*i.e.*, to what extent search processes are scheduled to access fragments from local disks), Figure 6 illustrates one snapshot from CASS on the fragments searched on each node and the fragments access by the network. We specifically ran experiments five times to check how much data move through the network at a 30-node setting, and track down a total number of fragments 150, 180, 200, 210, 240 respectively. As seen from the Figure 6, most nodes search a comparable number of fragments locally. More than 95% of data fragments are read from local storage.

We also conducted experiments to quantify how much overhead the translation I/O layer for parallel BLAST would introduce. We did two kind of tests. The first one is directly using the HDFS library while the other is with 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 and repeated the experiment several times. We found that the average total time through SDAFT-IO is around 15 seconds. The time through direct HDFS I/O was actually 25 seconds. This may derive from the overhead connecting and disconnecting with hdfsConnect() independently for each file. A second experiment we did was running BLAST process on multiple nodes through SDAFT-IO. The average time to open a file in HDFS is around 0.075 seconds, which is negligible compared with the overall I/O latency and BLAST time.

Moreover, 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. 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 talking to Hadoop's Namenode. We evaluated the HDFS overhead by retrieving the physical location of 200 formatted fragments. The average time is around 1.20 seconds, which accounts for a very small portion out of overall I/O latency.

4. RELATED WORK

The data locality provided by a data-intensive distributed file system is a desirable feature to improve the I/O performance. This is especially important when dealing with the ever-increasing amount of data in parallel computing. AzureBlast [10] is a case study of developing science applications such as BLAST on the cloud. CloudBLAST [11] adopts a MapReduce paradigm to parallelize genome index and search tools and manage their executions in the cloud. However, AzureBlast and CloudBLAST have been recently redeveloped and have not incorporated existing advanced techniques such as parallel result output. Moreover, both solutions only implement query segmentation but exclude database segmentation. Hadoop-BLAST [3] and bCloudBLAST [12] present a MapReduce-parallel implementation for BLAST but don't include existing advanced techniques like computation and I/O coordination presented in [9]. Our SDAFT is orthogonal to these techniques and allow MPI-based parallel programs to benefit from data locality exploitation in HDFS. SDAFT could also be suited for other parallel data applications like ParaView.

There exist other methods that are used to efficiently handle data movement or data management. Janine *et. al.* [5] developed a platform which realizes efficient data movement between in-situ and in-transit computations for large-scale scientific simulations. Haim Avron *et. al.* [4] developed an algorithm that uses a memory management scheme and adaptive task parallelism to reduce the data-movement costs. Our prior VisIO work [13] obtains a linear scalability of I/O bandwidth for ultra-scale visualization by exploiting data locality of HDFS. VisIO implementation calls the hdfs I/O library directly from the application programs, which is an intrusive scheme that requires significant hard coding effort. To support runs of large-scale parallel applications in which a shared file system abstraction is used, Zhang *et. al.* [15] developed a scalable MTC data management system to efficiently handle data movement. The aforementioned data movement solutions work at different contexts from SDAFT.

5. 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 could 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 to 10 and double the overall execution performance as compared with existing schemes.

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

7. REFERENCES

[1] 1000genomes project. http://aws.amazon.com/1000genomes/.

[2] Fuse: Filesystem in userspace. http://fuse.sourceforge.net/.

[3] Running hadoop-blast in distributed hadoop. http://salsahpc.indiana.edu/tutorial/hadoopblastex3.html.

[4] H. Avron and 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, pages 102:1–102:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[5] 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, and 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, pages 49:1–49:9, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[6] A. Darling, L. Carey, and W.-c. Feng. The design, implementation, and evaluation of mpiblast. *Proceedings of ClusterWorld*, 2003, 2003.

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

[8] G. Grider, H. Chen, J. Nunez, S. Poole, R. Wacha, P. Fields, R. Martinez, P. Martinez, S. Khalsa, A. Matthews, and G. Gibson. Pascal - a new parallel and scalable server io networking infrastructure for supporting global storage/file systems in large-size linux clusters. In *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International*, pages 10 pp.–340, 2006.

[9] 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.*, 22(4):529–543, Apr. 2011.

[10] W. Lu, J. Jackson, and 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, pages 413–420, New York, NY, USA, 2010. ACM.

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

[12] Z. Meng, J. Li, Y. Zhou, Q. Liu, Y. Liu, and W. Cao. bcloudblast: An efficient mapreduce program for bioinformatics applications. In *Biomedical Engineering and Informatics (BMEI), 2011 4th International Conference on*, volume 4, pages 2072–2076, 2011.

[13] 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 *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 68–79, May.

[14] C. Wu and 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, pages 35:1–35:10, Piscataway, NJ, USA, 2008. IEEE Press.

[15] Z. Zhang, D. S. Katz, J. M. Wozniak, A. Espinosa, and I. Foster. Design and analysis of data management in scalable parallel scripting. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, 2012.