DL-MPI: Enabling Data Locality Computation for MPI-based Data-Intensive Applications

Jiangling Yin, Andrew Foran and Jun Wang
Department of Electrical Engineering & Computer Science
University of Central Florida, Orlando, Florida 32826
jyin, jwang@eecs.ucf.edu

***Abstract*—Currently, most scientific applications based on MPI adopt a compute-centric architecture. Needed data is accessed by MPI processes running on different nodes through a shared file system. Unfortunately, the explosive growth of scientific data undermines the high performance of MPI-based applications, especially in the execution environment of commodity clusters. In this paper, we present a novel approach to enable data locality computation for MPI-based data-intensive applications and refer to it as DL-MPI. DL-MPI allows MPI-based programs to obtain data distribution information for compute nodes through a novel data locality API. In addition, the problem of allocating data processing tasks to parallel processes is formulated as an integer optimization problem with the objectives of achieving data locality computation and optimal parallel execution time. For heterogeneous runtime environments, we propose a scheduling algorithm based on probability to dynamically schedule tasks to processes by evaluating the unprocessed local data and the computing ability of each compute node. We demonstrate the functionality of our methods through the implementation of scientific data processing programs as well as the incorporation of DL-MPI with existing HPC applications.**

***Keywords*-MPI; Hadoop file system; HPC application;**

I. Introduction

Scientific analytic programs are usually developed with MPI and run parallel processes to perform analysis on different portions of a large data set. For example, bioinformatics [1] and scientific visualization applications constitute an emerging category of scientific applications that perform sophisticated calculation routines on a given dataset. Usually, a network file system is used to store the dataset, which will transfer data to the compute nodes for processing. Unfortunately, the rapidly growing datasets [2] pose a great challenge for these applications and can limit performance due to long network wait time. Accordingly, this leads to a reduction in scalability and an overall inability to handle large data sets well.

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 and allows programs to realize local data access and avoid data movement in the network. The idea behind HDFS is that it is faster and more efficient to send the compute executables to the stored data and process data locally rather than to pull the data from storage and send it via a network to the compute node. Compared to the MPI programming model, however, MapReduce does not allow for the flexibility and efficiency of complexity expression in the implementation of scientific applications.

Therefore, to address the issues associated with the growth of data, we present a novel approach in support of Data Locality computation for MPI-based, data-intensive applications (DL-MPI). We propose a data locality API to allow MPI-based programs to retrieve data distribution information from the underlying distributed file system. The proposed API is designed to be easily integrated into existing MPI-based applications or new MPI programs. In addition, to balance data locality computation and the parallel execution time in heterogeneous environments, we propose a novel probability scheduler algorithm, which schedules data processing tasks to MPI processes through evaluating the unprocessed local data and the computing ability of each compute node. Finally, we demonstrate the functionality of our method through the implementation of data analytic applications as well as the incorporation of DL-MPI with mpiBLAST, an open-source parallel BLAST tool.

II. DL-MPI Design and Methodologies

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

A. Design Goals and System Architecture

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

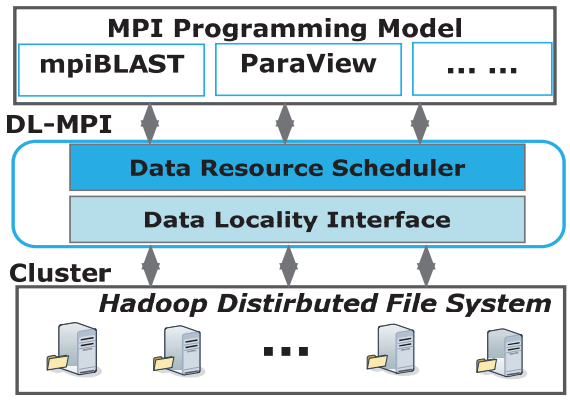

Figure 1. The DL-MPI software architecture.

Table I
DESCRIPTIONS OF DATA LOCALITY API IN DL-MPI

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

data processing tasks to parallel processes in a heterogeneous running environment is needed.

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

The DL-MPI system consists of two important components, a Data Locality Interface and a Data Resource Scheduler as illustrated in Figure 1. The data to be processed is stored, along with several copies, in a distributed file system, which supports scalable data access. Through our DL-MPI system, the MPI-based application on top could do complexity computation and achieve data locality computation with HDFS as the underlying storage system.

B. Data Locality Interface for MPI-based Programs

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

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

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

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

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

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

To show how the API is executing for a call from a client application, we take the *DLI_map_process_chunks* function

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

C. Data Resource Allocation

To achieve data locality computation for data-intensive applications, not only the information of data distribution among nodes is required, but also an effective strategy for mapping data to processes is needed.

1) Problem Formalization: To allow a process to achieve data locality computation, a simple way is always to assign task to the process which has access to a local chunk. However, with such a greedy strategy, there exist several heterogeneity issues that could potentially result in low execution performance. First, the HDFS random chunk placement algorithm may pre-distribute the target data unevenly within the cluster, leaving some nodes with more local data than others. Second, the execution time of a specific computation task could vary a lot among different nodes, due to the heterogeneous run-time environment, e.g. multiple users and multiple applications. These issues could make the processes in some compute nodes take remote computation and thus have a long I/O wait time. We formally discuss the assignment issue next.

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

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

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

Suppose the optimal parallel execution time for D being OPT_D and the execution time of process p_i on assigned data F_i being $T(F_i)$. The goal of the assignment algorithm is to assign chunks for each process i that maximizes the degree of data locality and the degree of execution time optimization, subject to a constraint that the union of chunks processed by all processes covers the entire D. More formally, we need to solve for F_i in the following optimization problem:

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

subject to

$$\sum_{1 \le i \le m} F_i \ge (1, 1, \cdots, 1) \tag{2}$$

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

Suppose the process speed of each cluster node could be estimated according to historical execution performance and the time to finish processing the chunks is determined when a F_i is given. The OPT_D is a constant reference value and given a fixed α and β, e.g. 0.5, the problem can be converted into an integer programming problem and be solved using the CPLEX solver.

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

2) Probability based Data Scheduler: In this section we introduce the probability based data scheduler algorithm that balances data locality computation and the parallel execution time among MPI processes.

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

The pseudo code of the probability scheduler algorithm is listed in Algorithm 1. Whenever a process i requests a task, we initially try to launch a task with its requested chunks stored at the node. We calculate for each candidate chunk d_x the probability to be assigned, based on an estimated execution time of other processes. Due to the replication mechanism of HDFS, for each candidate chunk d_x, we may have other processes j that could take local computation on d_x. We estimate the remaining local execution time of these processes j, as the number of all unprocessed chunks on process j divided by the process speed s_k. We choose the minimum execution time over all these processes j to decide the probability for assigning d_x to process i. This is because, the process with less remaining local execution time will have a larger probability to take remote computation in future. That is,

$$T_{d_x} = \min_{1 \le k \le n, d_x \in D_k, k != i} (\frac{|D_k|}{s_k}) \tag{3}$$

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

$$P(d_x) = \frac{T_{d_x}}{\sum_{X \in D_i} T_X} \quad (4)$$

Algorithm 1 Probability based Data Scheduler Algorithm

1: Let $C = \{c_1, c_2, ..., c_n\}$ be the set of participating nodes
2: Let $S = \{s_1, s_2, ..., s_n\}$ be the data-process speed set of n nodes;
3: Let $D = \{d_1, d_2, ..., d_m\}$ be the set of unprocessed data chunks;
4: Let D_i be the set of unprocessed data chunks located on node i;

Steps:

5: **while** $|D| \neq 0$ **do**
6: **if** an mpi process on node i requests a new task **then**
7: Update s_i according to the historical execution performance
8: **if** $|D_i| \neq 0$ **then**
9: **for** $d_x \in D_i$ **do**
10: $T_{d_x} = \min_{1 \leq k \leq n, d_x \in D_k, k!=i} (\frac{|D_k|}{s_k})$
11: **end for**
12: **for** $d_x \in D_i$ **do**
13: $P(d_x) = \frac{T_{d_x}}{\sum_{X \in D_i} T_X}$
14: **end for**
15: Assign d_x to the process on node i with probability $P(d_x)$
16: **else**
17: **for** $d_x \in D$ **do**
18: $T_{d_x} = \min_{1 \leq k \leq n, d_x \in D_k, k!=i} (\frac{|D_k|}{s_k})$
19: **end for**
20: **for** $d_x \in D$ **do**
21: $P(d_x) = \frac{T_{d_x}}{\sum_{X \in D_i} T_X}$
22: **end for**
23: Assign d_x to the process on node i with probability $P(d_x)$
24: **end if**
25: Remove d_x from D
26: **for all** D_k s.t. $d_x \in D_k$ **do**
27: Remove d_x from D_k
28: **end for**
29: **end if**
30: **end while**

The probability assignment algorithm could be implemented in applications with dynamic scheduling algorithms, such as mpiBLAST, in which scheduling is determined by what nodes are idle at any given time. This kind of scheduling adopts a master-slave architecture and the assignment algorithm could be incorporated into master process. The probability assignment algorithm could also be implemented in applications with static scheduling, such as ParaView, which uses static data partitioning so the work allocation can be determined beforehand. For this kind of scheduling, we can assume a round-robin request order for assignment in Step 6 of Algorithm 1.

III. Experiments and Analysis

A. Experimental Setup

We conducted comprehensive testing on our proposed DL-MPI at 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 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 2.33GHz 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 $[2.8.2]$ with default configuration on the cluster nodes. 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.

B. Evaluating DL-MPI

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

We firstly compare the process time of our benchmark program using DL-MPI probability based scheduling, referred to as 'With DL-MPI', and non locality scheduling, referred to as 'Without DL-MPI', on CASS for variable file sizes. We use 32 nodes for these experiments and show the performance comparison in Figure 2. From the figure, we find that the benchmark program using DL-MPI consistently obtains much lower process times than without using DL-MPI. With increased size of input data, the running time increases quickly for the benchmark program without using DL-MPI.

We also compare the bandwidth performance through varying the number of nodes in the Marmot cluster. We

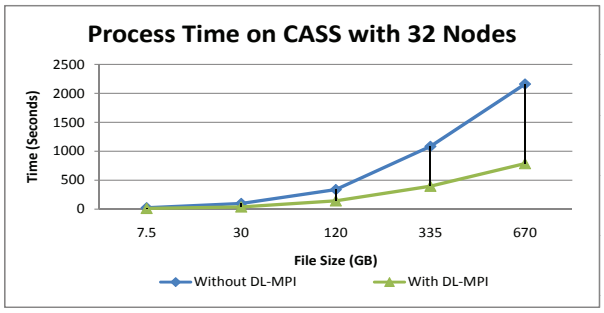

Figure 2. Process time Comparison of our benchmark program with and without DL-MPI on CASS using variable file sizes.

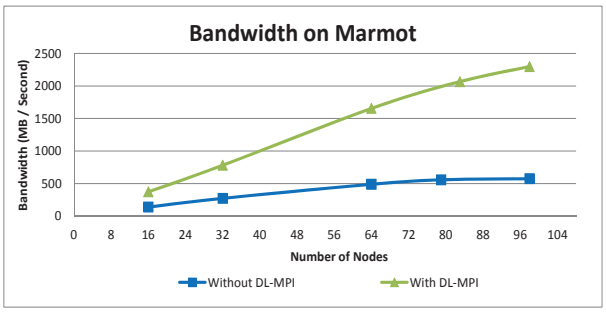

Figure 3. Bandwidth comparison of our benchmark program on PVFS and HDFS using DL-MPI on Marmot using variable number of nodes.

measure the bandwidth for processing a 1 TB file and show the bandwidth comparison in Figure 3. We find a massive improvement for the benchmark program using DL-MPI as the number of nodes increases, resulting in approximately a four fold increase when the number of nodes reaches around 100. We also see the bandwidth of the benchmark program using PVFS begin to level off at around 60 nodes while the test with DL-MPI scales much better.

2) mpiBLAST with DL-MPI: BLAST algorithms are widely used in the study of biological and biomedical research. They compare a query sequence with database sequences by a two-phased heuristic-based alignment algorithm. mpiBLAST [1] is a parallel implementation of NCBI BLAST. It organizes all parallel processes into one master process and many worker processes. Before performing the actual search, the raw sequence database is formatted into many fragments and stored in a shared network file system. It implements a dynamic load balancing scheduler, which schedules search tasks on available idle nodes. The worker process gets the requested data from the network parallel file system.

In our experiments, we use mpiBLAST as an example of existing HPC applications to make use of DL-MPI and achieve data locality computation (DL-mpiBLAST in brief). We incorporate the probability scheduler algorithm into the master scheduler of mpiBLAST. Whenever the master process is going to assign a new task to an idle process, the

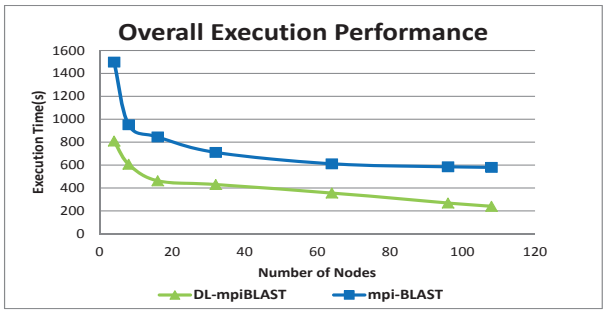

Figure 4. The execution performance comparison of DL-mpiBLAST and default mpiBLAST.

master process will call the Probability scheduler. For the data processing speed, we initialize it as 1 for all processes and update it by tracking the number of tasks finished by the process.

The *nt* nucleotide sequence database is selected as our experimental database. At the time when we did the experiments, the total raw size of the *nt* database is about 53 GB. The input queries to search against the *nt* database are randomly chosen from *nt* and revised, which guarantees that we find some close matches in the database. The total query size is around 60KB and the output is about 9.7 MB.

We track the parallel execution time for DL-mpiBLAST on HDFS and mpiBLAST on PVFS and show the performance comparison in Figure 4. From the figure, we can find that the parallel execution time of DL-mpiBLAST is smaller than that of mpiBLAST. Given an increasing cluster size, DL-mpiBLAST reduces overall execution time as well. However, mpiBLAST using PVFS does not scale well as the increasing number of nodes. This indicates the data movement overhead in the baseline does become a performance barrier to the system scalability.

IV. Related Work

With the explosive growth of data, research works [3], [4], [5], [6], [7] are presented to solve the data movement and data management. Specially, SciMATE [8] is a framework that is developed to improve the I/O performance by allowing scientific data in different formats to be processed with a MapReduce like API. Kshitij *et. al.* [9] developed a new plugin for HDF5 using PLFS to convert the single-file layout into a data layout that is optimized for the underlying file system. Jun *et. al.* [10] demonstrated how patterns of I/O within scientific applications can significantly impact the effectiveness of the underlying storage systems and utilized the identifying patterns to mitigate the I/O bottleneck. These methods improve the I/O performance though using data access regularity. While our DL-MPI improves the I/O performance by capitalizing on data locality computation.

The data locality provided by a data-intensive distributed file system is a desirable feature to improve I/O performance.

This is especially important when dealing with the ever-increasing amount of data in parallel computing. Mesos [11] is a platform for sharing commodity clusters between multiple diverse cluster computing frameworks. Mesos shares resources in a fine-grained manner, allowing frameworks to achieve data locality by taking turns reading data stored on each machine. VisIO [12] obtains a linear scalability of I/O bandwidth for ultra-scale visualization by exploiting data locality of HDFS. The aforementioned data movement solutions work in different contexts from DL-MPI.

V. Conclusion

In this paper, we propose a data locality interface and a dynamic scheduler to support flexible mapping of compute processes for MPI-based data-intensive programs, allowing them to take advantage of the locality information provided by HDFS. The interface is designed to be easily adopted by existing MPI programs, so that traditional applications can take advantage of it without massive code rewriting. By testing our DL-MPI prototype system with real-world data, we saw significant performance improvement over traditional MPI architectures. These improvement were enhanced even further by increasing the data and cluster size. We also saw performance and scalability improvement in mpiBLAST by using our DL-MPI probability scheduler algorithm. By allowing for the easy integration of data locality awareness in the MPI programming model, in the form of our DL-MPI, we believe that MPI-based data-intensive applications can efficiently run on commodity cluster.

VI. 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 CCF-0811413, CNS-1115665, and National Science Foundation Early Career Award 0953946.

References

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

[2] "Super computing 2008 exascale workshop," http://www.lbl.gov/CS/html/SC08ExascalePowerWorkshop/index.html.

[3] 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*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 49:1–49:9.

[4] X. Wu, K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, "Probabilistic communication and i/o tracing with deterministic replay at scale," in *Proceedings of the 2011 International Conference on Parallel Processing*, ser. ICPP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 196–205.

[5] S. Lakshminarasimhan, J. Jenkins, I. Arkatkar, Z. Gong, H. Kolla, S.-H. Ku, S. Ethier, J. Chen, C. S. Chang, S. Klasky, R. Latham, R. Ross, and N. Samatova, "Isabela-qa: Query-driven analytics with isabela-compressed extreme-scale scientific data," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, 2011, pp. 1–11.

[6] 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*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 102:1–102:11.

[7] 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*, 2012, pp. 1–11.

[8] Y. Wang, W. Jiang, and G. Agrawal, "Scimate: A novel mapreduce-like framework for multiple scientific data formats," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 443–450.

[9] K. Mehta, J. Bent, A. Torres, G. Grider, and E. Gabriel, "A plugin for hdf5 using plfs for improved i/o performance and semantic analysis," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, ser. SCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 746–752.

[10] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun, "I/o acceleration with pattern detection," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 25–36.

[11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 22–22.

[12] 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*, May, pp. 68–79.