2014 IEEE 28th International Parallel & Distributed Processing Symposium

DataMPI: Extending MPI to Hadoop-like Big Data Computing*

Xiaoyi Lu, Fan Liang, Bing Wang, Li Zha, and Zhiwei Xu
Institute of Computing Technology, Chinese Academy of Sciences
{luxiaoyi, liangfan, wangbing, char, zxu}@ict.ac.cn

***Abstract*—MPI has been widely used in High Performance Computing. In contrast, such efficient communication support is lacking in the field of Big Data Computing, where communication is realized by time consuming techniques such as HTTP/RPC. This paper takes a step in bridging these two fields by extending MPI to support Hadoop-like Big Data Computing jobs, where processing and communication of a large number of key-value pair instances are needed through distributed computation models such as MapReduce, Iteration, and Streaming. We abstract the characteristics of key-value communication patterns into a bipartite communication model, which reveals four distinctions from MPI: Dichotomic, Dynamic, Data-centric, and Diversified features. Utilizing this model, we propose the specification of a minimalistic extension to MPI. An open source communication library, DataMPI, is developed to implement this specification. Performance experiments show that DataMPI has significant advantages in performance and flexibility, while maintaining high productivity, scalability, and fault tolerance of Hadoop.**

***Keywords*-MPI; Hadoop; MapReduce; Big Data; DataMPI**

I. Introduction

Message Passing Interface (MPI) has been widely used in the field of High Performance Computing. This includes the MPI specifications [1] and various MPI implementations, such as MPICH [2], MVAPICH [3], and Open MPI [4]. MPI not only enables fast communication, but also provides a standard interface facilitating the development of parallel and distributed application programs in a portable way. Recently, Big Data has attracted much attention in the IT industry and academic communities. Much work has been done on Big Data Computing models and systems, such as MapReduce [5], MapReduce Online [6], Twister [7], Dryad [8], Spark [9], and S4 [10]. In the open-source community, Hadoop [11] is a widely used implementation of the MapReduce model for Big Data. These systems have one commonality: they feature the processing and communication of a large number of key-value pair instances. We call such systems Hadoop-like Big Data Computing systems, or Big Data Computing systems for short.

Differing from the HPC field, such Big Data Computing systems lack efficient communication support such as MPI. Instead, communication is realized by time consuming techniques such as HTTP, RPC. This paper takes a step in bridging these two fields by extending MPI to support Hadoop-like Big Data Computing jobs, so that knowledge and techniques from the HPC field can be transferred to the Big Data Computing field. For example, many optimized MPI communication techniques on different networks (e.g. InfiniBand and 1/10GigE) developed for HPC can be utilized by Big Data Computing applications.

*This research is supported in part by the Strategic Priority Program of Chinese Academy of Sciences (Grant No. XDA06010401), the Guangdong Talents Program, the Hi-Tech Research and Development (863) Program of China (Grant No. 2013AA01A209, 2013AA01A213).

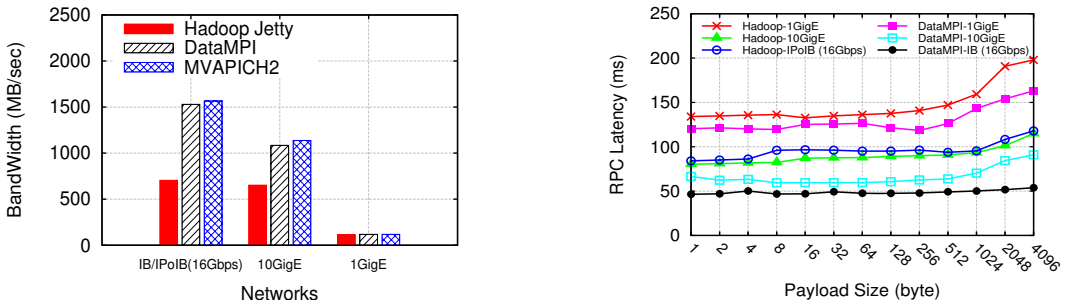
A. Motivation

An obvious starting point for this line of research is to extend MPI for Big Data, reusing MPI abstractions, mechanisms, and operations. However, we should first assess how much performance improvement we can attain by extending MPI to Big Data. As mentioned above, the communication mechanisms, involving HTTP and RPC, are commonly used in different types of network interactions in Hadoop, like Hadoop RPC calls among different components and MapReduce Shuffle. The achievable peak bandwidth and RPC latency are the two most important indicators to show the performance improvement potential from the basic communication primitive level. Compared with MPI communication primitives, the performance of Hadoop communication primitives is low. Figure 1(a) shows the comparison of achieved peak bandwidths for three systems, Hadoop HTTP Jetty, our proposed DataMPI, and MVAPICH2, on three networks (IB/IPoIB, 10GigE, and 1GigE). The peak bandwidth is measured by varying both total data size and packet size. As we can see, DataMPI and MVAPICH2 drive bandwidth more than twice as Hadoop Jetty on the networks of IB/IPoIB and 10GigE. DataMPI and MVAPICH2 use network more efficiently than Hadoop Jetty even on 1GigE. The bandwidth of DataMPI is slightly lower than that of MVAPICH2. This is because the Java binding (See Section IV) of DataMPI introduces some minor overheads in the JVM layer. We further implement an RPC system based on DataMPI by using the same data serialization mechanism as default Hadoop RPC. Then, we can use similar benchmarks to measure the performance of our DataMPI RPC and default Hadoop RPC. Figure 1(b) shows the latency comparison between them. When the payload size ranges from 1 B to 4 KB, the latency of DataMPI RPC is better than that of Hadoop RPC up to 18% on 1GigE, 32% on 10GigE, and 55% on IB. From these comparisons, we can see that our proposed DataMPI can provide much better performance than Hadoop in the primitive-level communication, which means the performance improvement potential of extending MPI to Big Data is attractive.

Even though we can get good performance improve-

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.90

IEEE computer society



(a) Peak Bandwidth (higher is better)

(b) RPC Latency (lower is better)

Figure 1. Performance Improvement Potentials of Hadoop Communication Primitives by Using MPI

ments in basic communication primitives, it is still full of challenges to extend MPI to real Big Data applications. One main reason is that the buffer-to-buffer communication model of MPI does not fit the key-value communication patterns of Big Data [12]. This paper overcomes these challenges by carefully analyzing the open-source codes of three representative Big Data Computing workload models: Hadoop (the MapReduce model), PageRank [13] (the Iteration model), and S4 (the Streaming model). We focus on presenting the detailed analysis of Hadoop but will also present evaluation results for the Iteration and the Streaming benchmarks. This paper addresses the following issues:

1) What main characteristics can we abstract to help design Big Data communication facilities?
2) How can we extend MPI to Hadoop-like Big Data Computing in a minimalistic way? And how can we design a high-performance communication library to implement our proposed extensions?
3) How much performance improvement can be obtained by our library over different Big Data processing workloads? Can the primitive level performance benefits be really achieved in the application level?

B. Contributions

The main contributions of this paper are:

1) Abstracting the requirements of a 4D (Dichotomic, Dynamic, Data-centric, and Diversified) bipartite communication model and key-value pair based communication, which capture the essential communication characteristics of Hadoop-like Big Data Computing.
2) Proposing a simple programming specification to extend MPI to Big Data. Several representative Big Data benchmarks (WordCount, TeraSort, PageRank, K-means [14] , Top-K) are used to show that our extension is easy to use and flexible.
3) Presenting a novel design of high-performance communication library called DataMPI for Big Data Computing. Performance evaluations on testbeds show that our library can achieve significant better performance on a variety of workloads.

The rest of this paper is organized as follows. Section II discusses Big Data communication characteristics. Section III introduces our proposed extensions. Section IV presents our library design. Section V describes our experiments and evaluation. Section VI discusses related work. Section VII offers concluding remarks and future work.

II. Communication Characteristics of Big Data

By the analysis on Hadoop MapReduce, we abstract the essential communication characteristics of Big Data Computing systems in this section.

A. A 4D Bipartite Communication Model

From the perspective of task management and data movement, we can find that the MapReduce framework sustains independence among map tasks, and also makes no data exchange among reduce tasks. The data is moved from one set of tasks (maps) to the other set (reduces). This paradigm can be represented as a bipartite graph, which is constructed by two groups of tasks and the data movements between them. As shown in Figure 2, the intermediate data moves from the tasks in Communicator O (Operation) to those in Communicator A (Aggregation). In fact, if we analyze the execution paradigms of other Hadoop-like Big Data Computing systems (e.g. Dryad, S4), we will find that the bipartite communication model is also concealed in the intermediate processing steps or stages of those systems. Through analyzing Hadoop MapReduce and other Big Data systems, we can further identify the 4D communication characteristics from the bipartite model.

Dichotomic. The MapReduce and other Big Data systems show that communications happen between two communicators. The underlying communication is a bipartite graph, i.e., the processes are dichotomic and belong to either the O communicator or the A communicator.

Dynamic. Big Data communication has a dynamic characteristic, which means the number of concurrent running tasks in each communicator of Figure 2 often changes dynamically because of task finish and launch. This indicates we usually do not explicitly assign a destination to the emitted data and it will be automatically distributed by the system. For example, when we write a map function in a MapReduce program, we only need to emit the processed data, but do not assign a destination (reduce) for it. Even though we want

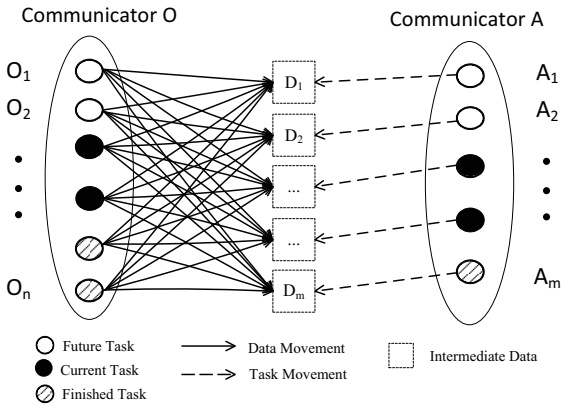


Figure 2. The Bipartite Communication Model

to assign the destination, it may cause confusion, because the assigned destination task may not be launched yet. In this case, it is a wise choice to schedule the data movement by the system or communication library implicitly, just like Hadoop hiding the communication details in the framework.

Data-centric. Jim Gray's Laws [15] tell us we should move computations to the data, rather than data to the computations in Big Data Computing. Such principle can be found in the most popular Big Data Computing models and systems. We also observed the data-centric characteristic from our bipartite model. As shown in Figure 2, tasks in Communicator O emit their data, while tasks in Communicator A can be scheduled to the corresponding data locations to process them. However, in current Hadoop MapReduce, most of the maps load data locally from HDFS, but the data locality feature is not designed for reduces. In fact, the miss of data locality in reduces is also discussed by the recent studies [16, 17]. We will further discuss this in Section IV-B.

Diversified. Although we can find many similar communication behaviors among different Big Data Computing systems, there still exists diversities. For MapReduce applications, the arrows of data movements in Figure 2 are dense and the model is a complete bipartite graph. For applications in Dryad or S4, the arrows of data movements may not be dense and Figure 2 becomes a more sparse bipartite graph. In addition, the intermediate data processing steps in different Big Data Computing scenarios are also diversified. Such characteristic indicates Big Data Computing has diversified communication modes. In this paper, we summarize four kinds of modes of Big Data Computing, which are *Common*, *MapReduce*, *Iteration*, and *Streaming*.

B. Key-Value Pair based Communication

From the perspective of programming, we can observe that many Big Data Computing systems (e.g. Hadoop MapReduce, S4, HBase) choose key-value pair as the core data representation structure, which is simple but has a strong ability to carry rich information. Therefore, it is a good idea to provide key-value pair based communication interfaces for Big Data Computing systems and applications, but not the traditional buffer-to-buffer interface signature. Such high level interface abstraction can reduce much programming complexity in parallel Big Data applications.

III. Proposed Extensions

Based on the above abstractions, this section presents our proposed MPI extensions.

A. Programming Specification

As shown in Table I and Table II, the extended functions are categorized into two sets. Table I summarizes three pairs of basic library functions for Big Data applications. Due to the dichotomic feature of the bipartite model, two build-in communicators (`COMM_BIPARTITE_O`, `COMM_BIPARTITE_A`) are introduced to organize the parallel execution of tasks. For different tasks, the corresponding communicator will be initialized by the `MPI_D_INIT` function call and they are finalized by invoking `MPI_D_FINALIZE`. We highlight two parameters (`mode` and `conf`) of `MPI_D_INIT`. They are used to support the diversified modes of Big Data communication. Currently, we have defined four kinds of modes. The `Common` mode is used to support SPMD-style programming and execution, just like traditional MPI programs. The `MapReduce` mode is devised based on the `Common` mode to support MPMD-style MapReduce applications. The `Streaming` mode is used to process real-time data streams. The `Iteration` mode is for supporting iterative computations. Each mode defines a group of configurations. The advanced users can define their own configurations and transfer them to `MPI_D_INIT` by the `conf` parameter. A set of reserved keys for configurations are also defined. Due to the space limitation, we just show two commonly used keys: `KEY_CLASS` and `VALUE_CLASS`, which are used to define data types for *key* and *value*. Note that the implementations can choose their preferred approaches to handle serialization issues.

In addition, a pair of naming functions is needed for applications. `MPI_D_COMM_RANK` is used to get the rank of task in different communicators of the bipartite model. `MPI_D_COMM_SIZE` returns the total number of tasks in Communicator O or A. According to the observation of key-value pair based communication requirement, we propose two functions (`MPI_D_SEND` and `MPI_D_RECV`) to exchange key-value pairs among tasks. As discussed in Section II-A, due to the dynamic feature, these two proposed communication functions do not require applications to assign destinations in interfaces. It means the library should schedule the data movement implicitly. The library should also satisfy the data-centric characteristic when moving data. These functions can help applications to achieve both productivity and high performance.

Table II lists three optional user-defined functions for flexibility. `MPI_D_COMPARE`: due to the requirement of sorting key-value pairs in some modes, the key should be

Table I
EXTENDED LIBRARY FUNCTIONS

Function	Parameter	Action
MPI_D_INIT	In: args, mode, conf	init env
MPI_D_FINALIZE	void	finalize env
MPI_D_COMM_SIZE	In: comm; Out: size	get size of tasks
MPI_D_COMM_RANK	In: comm; Out: rank	get rank of task
MPI_D_SEND	In: key, value	send KV
MPI_D_RECV	Out: key, value	receive KV

Table II
OPTIONAL USER-DEFINED FUNCTIONS

Function	Parameter	Action
MPI_D_COMPARE	In: k1, k2; Out: res	compare keys
MPI_D_PARTITION	In: k, v, comm; Out: dest	partition KV
MPI_D_COMBINE	In: k, ivs; Out: ovs	combine KVs

comparable. Applications can provide this function to tell the library how to compare the keys. `MPI_D_PARTITION`: this function is used to define the distribution policy of key-value pairs from one communicator to the other. The implementation should at least provide a default policy (e.g. hash-modulo). Applications can also define this function to override the default behavior. `MPI_D_COMBINE`: applications can provide this function to combine the intermediate data for reducing the size of exchanged data.

Listing 1. Example of Sort in the Common Mode

```
1  public class Sort {
2   public static void main(String[] args) {
3    try {
4     int rank, size;
5     HashMap<String, String> conf = new HashMap<String, ↩
          String>();
6     conf.put(MPI_D_Constants.KEY_CLASS, java.lang.String.↩
          class.getName());
7     conf.put(MPI_D_Constants.VALUE_CLASS, java.lang.↩
          String.class.getName());
8     // Init
9     MPI_D.Init(args, MPI_D.Mode.Common, conf);
10    if (MPI_D.COMM_BIPARTITE_O != null) {
11     // Get rank and size
12     rank = MPI_D.Comm_rank(MPI_D.COMM_BIPARTITE_O);
13     size = MPI_D.Comm_size(MPI_D.COMM_BIPARTITE_O);
14     // Users can load KVs from their preferred sources
15     String[] keys = loadKeys(rank, size);
16     // Users can do any computation logic here
17     if(keys != null)
18      for (int i = 0; i < keys.length; i++)
19       // Send
20       MPI_D.Send(keys[i], "");
21    } else if (MPI_D.COMM_BIPARTITE_A != null) {
22     rank = MPI_D.Comm_rank(MPI_D.COMM_BIPARTITE_A);
23     size = MPI_D.Comm_size(MPI_D.COMM_BIPARTITE_A);
24     // Receive
25     Object[] keyValue = MPI_D.Recv();
26     while (keyValue != null) {
27      // Users can do any computation logic here and store KVs to their ↩
            preferred destinations
28      outputKeyValue(rank, keyValue[0], keyValue[1]);
29      keyValue = MPI_D.Recv();
30     }
31    }
32    // Finalize
33    MPI_D.Finalize();
34   } catch (MPI_D_Exception e) {
35    e.printStackTrace();
36   }
37  }
38 }
```

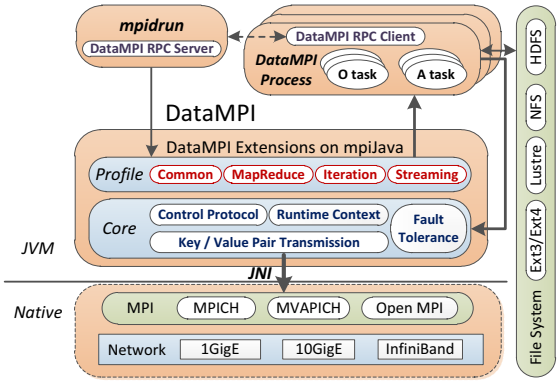


Figure 3. Architecture Overview of DataMPI

B. A Simple Example of Sorting

Listing 1 shows a simple example of sorting in the Common mode. This example is based on the Java-binding. In order to integrate and fairly compare with Hadoop MapReduce, which is Java-based, we choose to implement the Java-binding of our proposed extensions firstly. Our implementation of the Common mode contains default values for all required configurations, so that this example does not need to define optional functions. Our Java-binding can support the serialization mechanisms of both Java (`Serializable` and primitives) and Hadoop (`Writable`) currently. From this parallel sort example, we can see that the major part only contains 38 lines, which demonstrates our proposed extensions are easy-to-program. As our evaluations (Section V) show, such a simple example is scalable on both varied data sizes and cluster sizes.

IV. THE DATAMPI LIBRARY IMPLEMENTATION

This section presents our implementation, called **DataMPI**, of the bipartite communication model and the proposed extensions to MPI.

A. Architecture Overview

Figure 3 shows the two-layer architecture of DataMPI. In the JVM layer, DataMPI extends the mpiJava [18] design. The major extensions include dynamic process management (DPM) in the Java layer, optimized buffer management by native direct IO, the implementation of our proposed specification, bug fixes, etc. The lower layer is the native layer, in which we utilize JNI to connect upper Java-based routines to native MPI libraries. Compared with Hadoop, DataMPI provides a more light-weight and flexible library to users. For example, many HPC clusters do not have many local disks to build a big capacity HDFS storage, while what they often have are NFS, Parallel File System (e.g. Lustre), etc. Regarding these hosting environments, DataMPI can support applications to choose different storage systems flexibly. By utilizing the portable MPI standard, DataMPI can efficiently run Big Data applications over different high performance networks. We further introduce the important DataMPI components as follows.

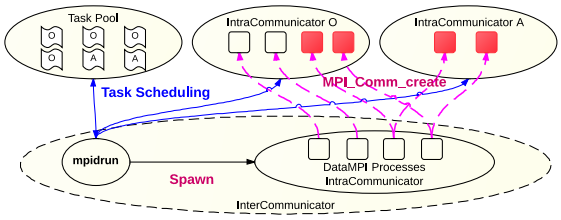


Figure 4. Communicator and Process Management & Task Scheduling

mpidrun: A `mpidrun` program is designed as a job launcher and scheduler. It supports the 4D features of the bipartite model. Support Dichotomic: `mpidrun` launches an application through a command like:

```
$ mpidrun -f hostfile -O n -A m -M mode -jar jarname classname params
```

Then a group of DataMPI processes will be launched. `mpidrun` separates all tasks into two queues for communicator O & A for further scheduling. Support Dynamic: `mpidrun` schedules tasks to run on processes, controls their executions, manages namespaces without conflicts, etc. Support Data-centric: `mpidrun` schedules tasks in communicator O & A to the relevant processes to load data from local when the data is ready. Support Diversified: the parameter of "`-M mode`" can be used to change the runtime behavior for different communication modes.

Profile: Each communication mode has a kind of profile, which contains a set of typical configurations and related extensions to the DataMPI core. For example, the MapReduce mode requires the intermediate data to be sorted by keys, while the Streaming mode may not need this feature. The Iteration mode needs the communication to be bi-directional, which means the data can be moved from O tasks to A tasks, and vice versa. However, the MapReduce mode only needs one-way communication. The typical features of each mode should be defined in the profile. And the related extensions can be loaded by the core component.

Core: By abstracting common functions of different modes, we design a core component in DataMPI, which supports functionalities of control protocol, runtime context, key-value pair transmission, and fault tolerance. Each mode can work on a certain profile, hold a runtime context for current profile, use the corresponding protocol to control the task execution and transfer intermediate data as key-value pairs. The details will be further discussed in following sections.

B. Runtime Execution and Data-centric Task Scheduling

Runtime Execution: Figure 4 shows the interactions among `mpidrun`, DataMPI processes and tasks. When the application is launched, `mpidrun` will spawn DataMPI working processes. These processes are in the same intracommunicator, while they are also connected with their parent, `mpidrun`, by an intercommunicator. DataMPI uses this intercommunicator to build up an RPC communication channel for exchanging control messages between working processes and `mpidrun`. After all processes are launched, they will receive and execute tasks scheduled by `mpidrun`. As shown in Listing 1, when tasks execute `MPI_D_INIT`, this function will initialize the execution environment. In this step, if the tasks are O tasks, `COMM_BIPARTITE_O` will be created. Similar things will be done for A tasks. In user-defined logic, we can use `MPI_D_RANK` and `MPI_D_SIZE` to do the problem decomposition as what we commonly do in traditional MPI programs.

Data-centric Task Scheduling: Hadoop MapReduce provides the data-local feature for maps, but not for reduces. Current Hadoop architecture adopts a two-phase and proxy-based data movement approach. Firstly, each map task writes the intermediate data to local disk, then each reduce task downloads the data from different maps by the proxies, which are the built-in HTTP servers in TaskTrackers. Such design is hard to achieve data locality for reduces, because the outputs of maps are distributed over the whole cluster. A basic idea for improvement is: before reduces are launched, can we collect the intermediate data for them in advance? If yes, then the reduce-side data locality can be achieved by task scheduling. Regarding this, DataMPI provides a data-centric task scheduling design. With HDFS, a utility function is designed to locally load data from HDFS for O tasks by their ranks and the communicator size. During the executions of O tasks, their intermediate data will be sent out by `MPI_D_SEND`. The emitted data is partitioned and stored among the processes' local accessible spaces (in memory or spilling to disk). After the O phase is finished, the processes will run A tasks also. `mpidrun` knows the information of the intermediate data distribution, so it can directly schedule A tasks to the corresponding processes to read their data locally by `MPI_D_RECV`. In this way, our proposed design can guarantee the data-local feature for both O and A tasks. As shown in Figure 4, an application has four O tasks and two A tasks. After finishing O tasks, two A tasks are scheduled to the two corresponding processes which already have the intermediate data.

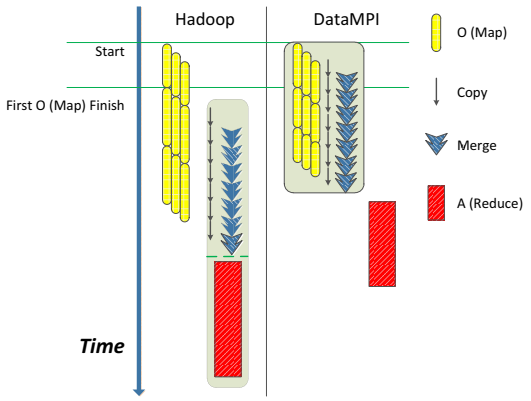
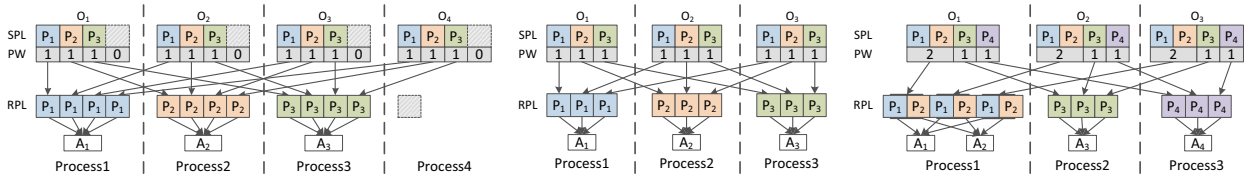


Figure 5. Overlapping Comparison of Hadoop and DataMPI



(a) $NUM_O > NUM_A$ (b) $NUM_O = NUM_A$ (c) $NUM_O < NUM_A$

Figure 6. Buffer Management and Communication in DataMPI (SPL: Send_Partition_List, RPL: Receive_Partition_List, PW: Partition Window)

C. O-side (Map-side) Data Shuffle Pipeline

As shown in Figure 5, Hadoop MapReduce overlaps map and shuffle (copy and merge) stages, but it does not launch reducers to copy data remotely until the first map is finished at least. The shuffle stage in reducers often lags behind all finished maps, because the reducers need to pull map completion events, copy data remotely, and merge them totally. As a result, such kind of overlapping design in Hadoop will make the execution of reduce function significantly delayed and the performance degraded.

DataMPI adopts an alternative design of O-side (Map-side) data shuffle pipeline. Each process initialized for an O task has three kinds of threads. The main thread will do the computation and send the key-value pairs to the communication thread. The communication thread will partition, sort (if needed), and save the key-value pairs to appropriate buffers. When the size of buffered data exceeds a threshold, the communication thread will send the data to corresponding processes, while it also receives data from other processes. For received data, another thread is responsible for merging them. In this way, the computation, copy, and merge are fully overlapped by multi-threading. Since DataMPI caches the intermediate data in memory by default and exchanges the data by high efficient MPI communication, it improves the O phase performance (See Section V-C). The O-side shuffle benefit is due to the help from the MPI-based bipartite communication model. Figure 5 shows DataMPI achieves better performance and more overlapping in O-side. By shuffling in O-side, A tasks can be executed in data-local and this further improves the overall performance consequently.

D. Buffer Management and Communication Optimization

From the perspective of communication pattern, the communication operations in the bipartite model can be seen as an relaxed all-to-all pattern globally. For supporting MPI point-to-point and collective based communication, we design a Partition List (PL) based buffer management scheme. One partition list is composed by several data partitions, which are used to store key-value pairs for corresponding A tasks. The data partition also contains multiple kinds of meta information, such as key-value offsets, indexes, etc. When a key-value pair is emitted, it is first cached in a selected partition from Send Partition List (SPL) according to the `MPI_D_Partition` function. When an SPL is full, it will be inserted to a send queue to wait for transferring by the communication thread. On the receiver side, the received data is cached in a Receive Partition List (RPL) which will be added to a merge queue. When the total size of merge queue is over a threshold, some of the cached RPLs are merged and spilled over to in-memory buffer or disk. The partition lists make buffer layout clear, easy-to-program and support both all-to-all or point-to-point communication.

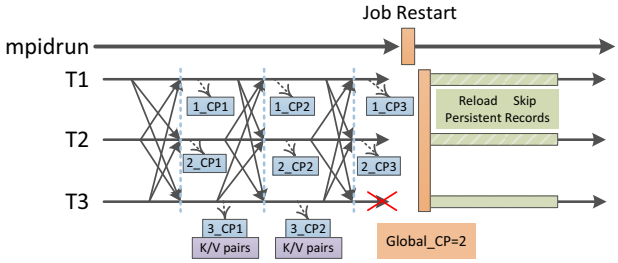


Figure 7. Overview of DataMPI Fault Tolerance

Note that the MPI communication happens in process-level, which may have mismatches with task-level data movements. Figure 6 shows three different cases of O/A task numbers. In different cases, the data partitions may be moved to different processes. It is fussy and error-prone especially when tasks are dynamically launched and exited. To control the buffers' offsets and lengths easily, we introduce a structure of Partition Window (PW) to redirect partitions to different processes. Through PW, we can reduce the complexity of buffer management. All of above techniques are transparent for applications, which just need to use our proposed send/receive interfaces to achieve high performance and productivity.

E. Key-Value based Library Level Checkpoint

For traditional MPI applications, we usually implement fault tolerance by application/library/system-level checkpoint/restart. Different with buffer-to-buffer communication interfaces, our proposed key-value pair based interfaces are also helpful for the fault tolerance design. As a core data structure in Hadoop-like Big Data systems, a key-value pair is usually considered as an intact business record. When applications send key-value pairs by our proposed interface, we can know what data should be checkpointed and how many records have been checkpointed when we do recovery. In this way, DataMPI provides a key-value pair based library level checkpoint mechanism, which is transparent to deterministic algorithm based Big Data applications. As Figure 7

shows, each task makes the checkpoint (CP) separably after a round of data exchanging. When a job recovers, all processes can coordinate with each other to get the global maximum checkpoint number among all successfully generated checkpoints and skip those processed records. DataMPI provides an option to enable fault tolerance (FT) support.

V. Evaluation

This section presents the evaluations of DataMPI.

A. Experiment Setup

We use two different testbeds for our evaluations.

(1) Testbed A: Testbed A has 17 nodes connected with 1 GigE. Each node is equipped with dual octa-core 2.1 GHz AMD Opteron(TM) processors, 64 GB RAM and single 500 GB HDD. Each node runs Linux 2.6.18-128.el5.

(2) Testbed B: 65 nodes in Testbed B are equipped with Intel Xeon dual quad-core processors operating at 2.67 GHz. Each node has 12 GB RAM, a 1 GigE NIC and single HDD (less than 80 GB free space). Each node runs 2.6.32-358.el6.

Each testbed has one master node and the rest nodes are slaves. Most of our tests are taken on Testbed A, while the scalability tests are conducted on Testbed B. We compare DataMPI performance with Hadoop 1.2.1. We use MVAPICH2 1.9 as the native MPI library. We store all the input/output data on HDFS. We use DataMPI to implement all the benchmarks (e.g. TeraSort, PageRank, K-means) for comparisons. Hadoop PageRank is self-developed and Hadoop K-means is chosen from Mahout 0.8 [19].

B. Parameter Tuning

Both Hadoop and DataMPI have many tunable parameters. Since we only have single HDD per node in our testbeds, the disk will easily become the bottleneck when we test with larger data size. HDFS block size and concurrent task number are the two most important performance parameters. We use TeraSort to tune both of them on Testbed A. As shown in Figure 8(a), we vary HDFS block size from 64 MB to 1024 MB and measure TeraSort throughput by processing 96 GB data. Both Hadoop and DataMPI can achieve the best throughput when the block size is 256 MB. In Figure 8(b), we increase concurrent A (reduce) tasks per node from 2 to 8 and measure TeraSort throughput by processing 2 GB data per task. Hadoop and DataMPI can get the best throughput when the number of concurrent A (reduce) tasks on each node is 4. Thus, we choose 256 MB as HDFS block size and run 4 A (reduce) tasks per node in our following tests when using Testbed A. We choose 4 concurrent O (map) tasks per node by similar tunings.

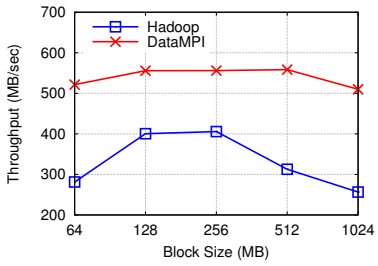
C. Overall Performance

DataMPI supports the diversified feature of the bipartite model and provides different modes to applications. This section will show the overall performance of different representative benchmarks for different modes. First, we compare DataMPI and Hadoop MapReduce by using TeraSort. Figure 9 shows the progress comparison between DataMPI and Hadoop MapReduce when they process 168 GB data on Testbed A. As we can see, Hadoop requires 475 sec to complete the job, whereas DataMPI only needs 312 sec. And we can also find DataMPI improves the performance for both O (map) and A (reduce) phases. Figure 10(a) further shows the situations when the input data size varies from 48 GB to 192 GB, DataMPI gains 32%-41% improvement compared to Hadoop. We also evaluate the performance with WordCount, which has smaller data movement compared with TeraSort. The results still show that DataMPI speeds up the performance of WordCount by 31% compared to Hadoop. Due to the similar results, we do not include the graph. These results demonstrate that the designs like data-centric task scheduling, O-side data shuffle pipeline, and optimized buffer management and communication in DataMPI are beneficial for both CPU and communication intensive applications.

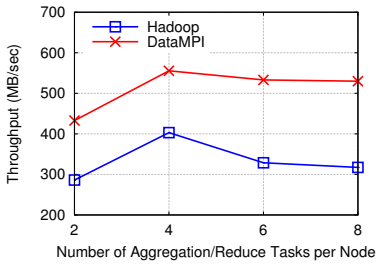
We include some initial results for Iteration and Streaming modes, which will be introduced in future work. We compare DataMPI Iteration with Hadoop by using PageRank and K-means with 40 GB data. Each of the benchmarks runs seven rounds. As shown in Figure 10(b), DataMPI averagely improves the performance of PageRank by 41% and K-means by 40% compared to Hadoop. S4 is a streaming data processing system. We run the Top-K benchmark to compare DataMPI Streaming with S4 (v0.5.0). Figure 10(c) shows the distribution of streaming data processing latencies with a message rate as 1 K msg/sec and 100 B for each. The latencies of DataMPI range from 0.5 to 4 sec, while S4 latencies range from 1.5 to 12 sec. This means DataMPI can gain better performance for Streaming applications.

D. Profile of Resource Utilization

We further measure the resource utilization of DataMPI and Hadoop with 168 GB TeraSort. As Figure 11(a) shows, DataMPI has lower average CPU utilization compared to Hadoop with less execution time. The beginning part of DataMPI CPU utilization is slightly higher than that of Hadoop. This is because the O-side data shuffle pipeline design in DataMPI is able to leverage system resource to overlap the computation/communication/merge operations efficiently. Figure 11(b) shows the disk utilization. The read throughput of DataMPI is averagely 65.8 MB/sec during the O phase, which is 69% higher than that of Hadoop (38.9 MB/sec during the map phase). This shows our O-side shuffle pipeline design achieves better disk read throughput. DataMPI writes near half of data compared to Hadoop because DataMPI will cache intermediate data in memory by default. Note that the total amount of disk read is nearly equal between DataMPI and Hadoop, which is because the system-level disk cache helps Hadoop to hold intermediate data and cuts down the read overhead. DataMPI caches the data by its own design. Figure 11(c) shows DataMPI



(a) HDFS Block Size Tuning



(b) Task Number Tuning

Figure 8. Parameter Tuning

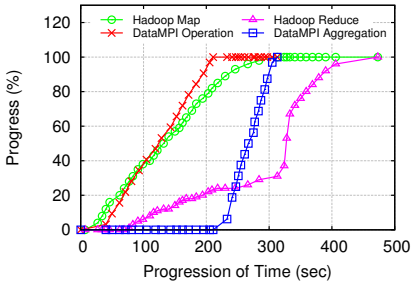
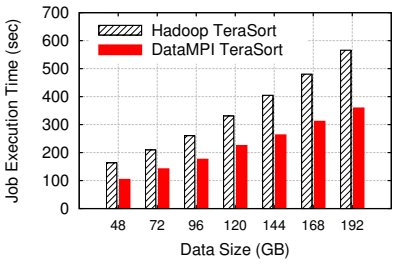
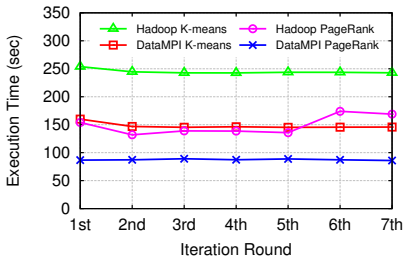


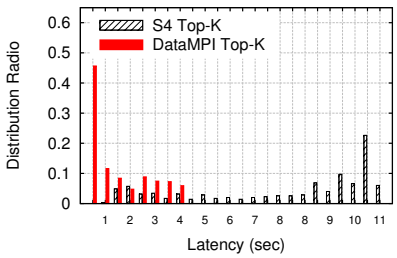
Figure 9. Progress of TeraSort



(a) TeraSort



(b) PageRank and K-means



(c) Top-K (More left is better)

Figure 10. Performance Comparison of Different Benchmarks

communication mainly occurs in the O phase because of the data-centric scheduling design for A tasks. On the average, the network throughput of DataMPI is 74.3 MB/sec, which is 47% higher (better) than that of Hadoop (50.6 MB/sec). Figure 11(d) shows during 0 - 475 sec, the average memory used by DataMPI is 26.6 GB which is less than 29.3 GB in Hadoop. This indicates data caching and in-memory shuffle in DataMPI do not make extra memory overhead compared with Hadoop which uses system-level caching.

E. Spill Over Efficiency

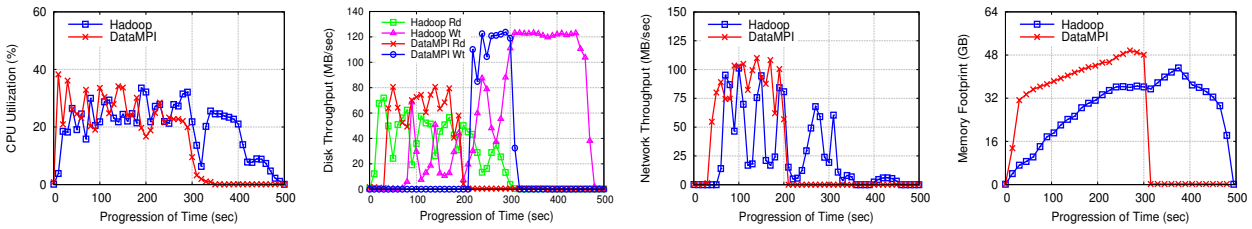
Most of above TeraSort tests ensure the slave nodes have enough memories to store intermediate data. However, when the slave nodes do not have enough memories, DataMPI needs to spill intermediate data to disk. An interesting question is: can our design still guarantee better performance when data is spilled over? In order to check this, we configure DataMPI to cache different amount of intermediate data in memory and measure the job execution time. Figure 12 shows with more intermediate data cached in memory, the job costs less time. The DataMPI performance degrades slightly (up to 9%) from full caching to zero caching and DataMPI with zero caching still has better performance than Hadoop. This is because DataMPI provides the data-local feature for A tasks and the intermediate data on disk can be prefetched at the initial stage of A phase.

F. Fault Tolerance

DataMPI supports fault tolerance by the key-value based light-weight library level checkpoint technique. When the DataMPI application enables checkpoint, the intermediate data will be saved to disk periodically. This will introduce some overheads. We evaluate the DataMPI performance with different percentages of checkpointed data sizes. We test 100 GB TeraSort on 10 slave nodes in Testbed A. Figure 13(a) shows the performance comparison between default DataMPI and checkpoint-enabled DataMPI. We see the checkpoint-enabled DataMPI has only about 12% performance loss compared with the default DataMPI. If we compare it with Hadoop, we see the checkpoint-enabled DataMPI can still show 21% improvement. If we kill the job intentionally when DataMPI has persisted different sizes of checkpoints, we can see that each job restart only costs less than 3 seconds. With increasing of the checkpointed data sizes, the checkpoint reload time increases proportionally and the total execution time has a slight augment. Figure 13(b) shows the CPU utilization in detail of the fault tolerant job, which has 60% checkpointed data. These results demonstrate the key-value pair based communication can really help the fault tolerance design.

G. Scalability

We evaluate the DataMPI scalability on Testbed B. After parameter tuning on this testbed, we choose 128 MB as HDFS block size and run 2 concurrent O (map) and A (reduce) tasks on each slave node. We measure the job execution time of TeraSort in two scaling patterns: strong scale and weak scale. In strong scale, we fix the total data size (256 GB) and increase the parallelism degree by increasing the number of slave nodes. In weak scale, we fix the data size (2 GB) for each A (reduce) task and increase the number of slave nodes. Figure 14(a) shows DataMPI has the similar trend with Hadoop in strong scale and reduce the



(a) CPU Utilization (b) Disk Throughput (c) Network Throughput (d) Memory Footprint

Figure 11. Profile of Resource Utilization

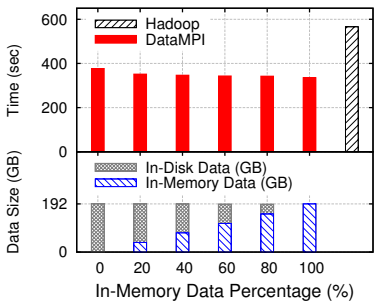
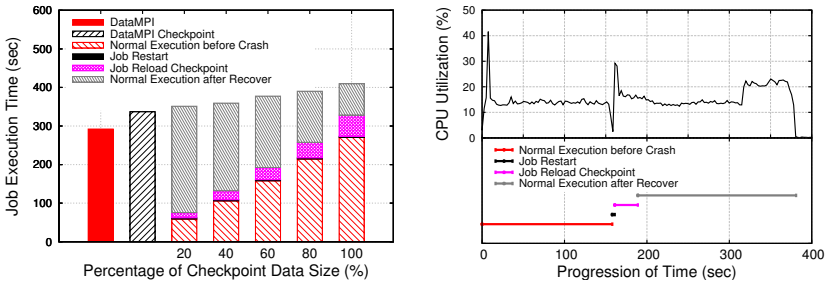


Figure 12. Spill Over Efficiency



(a) Efficiency with Different Checkpoints (b) CPU Utilization (60% checkpointed data)

Figure 13. Fault Tolerance Efficiency

job execution time by 35%-40% compared to Hadoop. In weak scale, Figure 14(b) shows both DataMPI and Hadoop can achieve good scalability and DataMPI can improve the performance by 40% compared to Hadoop. Therefore, both DataMPI and Hadoop can achieve linear scalability and DataMPI can achieve better performance.

VI. Related Work

Big Data Computing models and systems: The MapReduce model was introduced in 2004 by Dean *et.al.* in [5]. MapReduce Online [6] allows data to be streamed between mappers and reducers, supporting pipelined execution, continuous queries, and online aggregation. Twister [7] extends MapReduce to support iterative jobs through keeping static data in memory among jobs. Dryad [8] supports directed acyclic graph based tasks. This paper complements these work by abstracting and implementing a key-value pair 4D bipartite communication model for Big Data Computing.

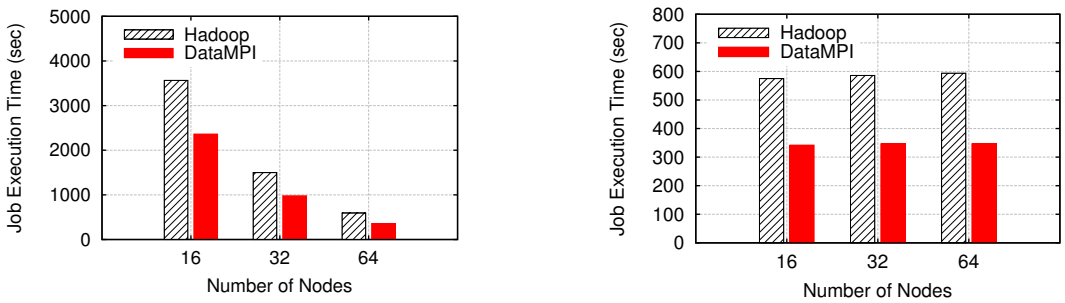
Adopting HPC Technologies to Improve Data Processing Performance: Hoefler *et.al.* [20] attempted to write efficient MapReduce applications by using MPI. Plimpton *et.al.* [21] described their MR-MPI library and several MapReduce graph algorithms examples. Hadoop-A [22] introduces an RDMA-levitated merge algorithm in shuffle phase of Hadoop to improve the performance. Hadoop-RDMA [23–25] leverages RDMA-capable interconnects to improve Hadoop performance with enhanced designs of various components, such as MapReduce, HDFS, RPC. The major difference between our work and theirs is that we propose a 4D bipartite communication model for Big Data Computing, and implement a general communication library based on the model. Evaluations have shown the merits of our model and design.

In addition, our previous work [12] identifies the opportunities and challenges of adapting MPI to MapReduce. This paper makes three significant progresses. First, we abstract a general bipartite communication model from these practices; Second, we refine the programming interfaces to support the diversified application requirements; Third, we implement and open source a real bipartite model based communication library by extending MPI.

VII. Conclusion and Future Work

We present **DataMPI**, an efficient communication library for Big Data Computing that features the processing and communication of large numbers of key-value pairs. The design of DataMPI is guided by carefully analyzing the essential features of key-value pair communication requirements in Big Data Computing. We observe that such communication requirements are abstracted as a bipartite communication model with four distinctions from traditional MPI: Dichotomic, Dynamic, Data-centric, and Diversified features. DataMPI provides a programming specification that extends MPI in a minimalistic way, requiring only three pairs of new MPI library calls in default cases.

We implement DataMPI with a Java binding and conduct experiments on Ethernet clusters, running application benchmarks with up to 65 nodes. Using five application benchmarks (WordCount, TeraSort, PageRank, K-means, Top-K) that represent three popular Big Data Computing models (MapReduce, Iteration, Streaming), we show that DataMPI has significant advantages in following aspects.



(a) Strong Scale (b) Weak Scale

Figure 14. Scalability of DataMPI

- **Efficiency**: DataMPI speeds up varied Big Data workloads and improves job execution time by 31%-41%.
- **Fault Tolerance**: DataMPI supports fault tolerance. Evaluations show that DataMPI-FT can attain 21% improvement over Hadoop.
- **Scalability**: DataMPI achieves high scalability as Hadoop and 40% performance improvement.
- **Flexibility**: DataMPI can support different models of Big Data Computing and can be used with different file systems and networks.
- **Productivity**: The coding complexity of using DataMPI is on par with that of using traditional Big Data application frameworks such as Hadoop.

We have open sourced DataMPI at [26]. We plan to continuously update this package with newer designs and carry out evaluations with more applications.

REFERENCES

[1] "Message Passing Interface Forum," http://www.mpi-forum.org.

[2] "MPICH2," http://www.mcs.anl.gov/research/projects/mpich2/.

[3] "MVAPICH," http://mvapich.cse.ohio-state.edu/.

[4] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proc. of the 11th European PVM/MPI Users' Group Meeting*, ser. EuroMPI '04, Budapest, Hungary, 2004.

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," in *Proc. of the 7th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI '10, Berkeley, CA, USA, 2010.

[7] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A Runtime for Iterative MapReduce," in *Proc. of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10, New York, NY, USA, 2010.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks," in *Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, ser. EuroSys '07, New York, NY, USA, 2007.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proc. of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud '10, Berkeley, CA, USA, 2010.

[10] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *Proc. of the 2010 IEEE International Conference on Data Mining Workshops*, ser. ICDMW '10, Sydney, Australia, 2010.

[11] "The Apache Hadoop Project," http://hadoop.apache.org.

[12] X. Lu, B. Wang, L. Zha, and Z. Xu, "Can MPI Benefit Hadoop and MapReduce Applications?" in *Proc. of the 40th International Conference on Parallel Processing Workshops*, ser. ICPPW '11, Taipei, China, 2011.

[13] S. Brin and L. Page, "The Anatomy of A Large-scale Hypertextual Web Search Engine," in *Proc. of the 7th International Conference on World Wide Web*, ser. WWW7, Amsterdam, The Netherlands, 1998.

[14] J. B. MacQueen, "Some Methods for Classification and Analysis of MultiVariate Observations," in *Proc. of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, Berkeley, CA, USA, 1967.

[15] Tony Hey, Stewart Tansley, and Kristin Tolle, *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Redmond, Washington: Microsoft Research, 2009.

[16] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud," in *Proc. of the 2nd International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM '10, Washington, DC, USA, 2010.

[17] M. Hammoud and M. F. Sakr, "Locality-Aware Reduce Task Scheduling for MapReduce," in *Proc. of the 3rd International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM '11, Washington, DC, USA, 2011.

[18] "mpiJava," http://www.hpjava.org/mpiJava.html.

[19] "The Apache Mahout Project," http://mahout.apache.org.

[20] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards Efficient MapReduce Using MPI," in *Proc. of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. EuroMPI '09, Berlin, Heidelberg, 2009.

[21] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for Large-scale Graph Algorithms," *Parallel Computing*, vol. 37, no. 9, pp. 610–632, 2011.

[22] Y. Wang, X. Que, W. Yu, D. Goldenberg, and D. Sehgal, "Hadoop Acceleration Through Network Levitated Merge," in *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11, New York, NY, USA, 2011.

[23] M. Wasi-ur Rahman, N. S. Islam, X. Lu, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-Performance RDMA-based Design of Hadoop MapReduce over InfiniBand," in *Proc. of the 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '13, Washington, DC, USA, 2013.

[24] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda, "High Performance RDMA-based Design of HDFS over InfiniBand," in *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, Los Alamitos, CA, USA, 2012.

[25] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-Performance Design of Hadoop RPC with RDMA over InfiniBand," in *Proc. of the 42nd International Conference on Parallel Processing*, ser. ICPP '13, Lyon, France, 2013.

[26] "The DataMPI Project," http://datampi.org.