2011 International Conference on Parallel Processing Workshops

Can MPI Benefit Hadoop and MapReduce Applications?

Xiaoyi Lu[1,2], Bing Wang[1,2], Li Zha[1], and Zhiwei Xu[1]
[1]*Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*
[2]*Graduate University of Chinese Academy of Sciences, Beijing 100049, China*
Email: {luxiaoyi, wangbing}@software.ict.ac.cn, {char, zxu}@ict.ac.cn

***Abstract*—The Message Passing Interface (MPI) standard and its implementations (such as MPICH and OpenMPI) have been widely used in the high-performance computing area to provide an efficient communication infrastructure. This paper investigates whether MPI can be adapted to the data intensive computing area to substantially speed up Hadoop and MapReduce applications, by reducing communication overheads. Three specific issues are studied. First, is the potential for reducing communication overheads significant, if MPI is used? Second, what are the main technical challenges to adapt MPI to Hadoop? Third, what are the minimal extensions to the MPI standard that can help alleviate the challenges while promise to significantly improve performance? To answer the first question, we identify important and basic communication primitives in both MPI and Hadoop, and make fair comparisons of their performance through experiments. The results show that the potential for improvement could be high. To answer the second and the third questions, we analyze the Hadoop codebase to identify communication related programmers' needs. Furthermore, we propose a minimal interface extension to the MPI standard (only one pair of library calls are added), which capture the key-value pair nature commonly found in data intensive computing. This extension is implemented in a prototype library called MPI-D. Benchmark tests based on simulation show that Hadoop augmented with MPI-D could significantly speed up MapReduce application performance.**

***Keywords*-MPI; MPI-D; Hadoop; MapReduce**

I. Introduction

Data intensive computing [1] refers to the fast growing field of parallel and distributed computing where massive data need to be stored, processed, analyzed, visualized and used. A frontier application area of data intensive computing is the Internet services sector, where PB-scale data are processed daily [2]. Hadoop [3, 4] is an open source software suite widely used in Internet services datacenters to process massive data based on the MapReduce model [5].

When comparing the two areas of data intensive computing and high-performance computing, we see a clear difference in the underlying communication technology used. Although both areas need to reduce communication overhead, the HPC area seems to have a more efficient communication infrastructure, including high-performance interconnects such as the Infiniband and high-performance communication libraries such as implementations of the Message Passing Interface (MPI) standard [6–10]. In contrast, the Hadoop software suite realizes communication based on traditional network protocols such as TCP/IP, RPC, and HTTP.

To put it in historical perspective, the communication infrastructure of the high-performance computing area went through three overlapping development stages. First, the HPC system vendors provide low-level proprietary communication libraries. Although these libraries had reasonable communication performance, their non-standard nature was a big hurdle for HPC applications developers. Many application developers, when possible, instead used traditional network protocols such as TCP/IP and RPC as the communication substrate to build their parallel programs. In contrast to these two approaches, MPI was then developed to offer a standard communication interface that allows high-performance implementations such as MPICH [8, 9] and OpenMPI [10]. Current MPI libraries can achieve communication performance over 90% of the hardware peak. It is a fair assessment that the communication infrastructure of the current data intensive computing platforms (e.g., Hadoop) is comparable to the pre-MPI stage of high-performance computing. Historical lessons tell us that data intensive computing needs a standard and efficient communication infrastructure.

An obvious starting point for this line of research is to adapt MPI to serve as the communication infrastructure for Hadoop. But we must answer three questions:

- **Is it worth it**? Is the communication overhead of a MapReduce application in Hadoop worthy to be optimized? Is the performance improvement potential high if we can successfully adapt MPI to Hadoop, a few percentage points or orders of magnitude?
- **How difficult it is to adapt MPI to Hadoop**? MPI is originally designed for HPC applications. Can it be adapted to suit the data intensive computing needs? What are the main technical challenges?
- **Can we impact the ecosystems in a minimal way**? Both Hadoop and MPI have accumulated significant code bases and developer/user communities. It is not a viable approach if we need to rewrite a large portion of the Hadoop or MPI libraries, or significantly change the interfaces.

This paper makes three contributions: First, we investigate the performance improvement potential for adapting MPI

1530-2016/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPPW.2011.56

to Hadoop. We barrow a successful research method from the HPC field: when assessing the potential for improving a communication subsystem, first focus on the most basic primitives. In the case of MPI, these are the point-to-point send and receive primitives. Much work on optimizing MPI implementations focuses on send and receive. A good example is the invention of user-level communication techniques, which improves the point-to-point communication performance by an order of magnitude [11]. We identify two comparable primitives in the Hadoop suite, Hadoop RPC [4] and HTTP over Jetty [12]. Experimental results show that the potential of improving communication performance by adapting MPI is high.

Second, we identify a number of challenges adapting MPI to the Hadoop environment. The most challenging issue is to identify a minimal extension to MPI that inherits the well established MPI style but captures the essential communication needs of Hadoop. We design a pair of new MPI calls supporting Hadoop data communication specified via key-value pairs.

Third, to validate the above ideas and results, we implement a prototype MPI library called MPI-D (short for MPI Data Extension). Benchmark tests based on simulation show that Hadoop augmented with MPI-D could significantly speed up application performance.

The rest of this paper is organized as follows. Section 2 assesses the communication improvement potential. Section 3 identifies the main challenges for adapting MPI to Hadoop. Section 4 presents the MPI-D prototype and evaluates its performance via an application benchmark. Section 5 discusses related work. Section 6 offers concluding remarks.

II. Assessment of Performance Improvement Potential

To assess the performance improvement potential, we first need to expose the percentage of the communication time in the total execution time of a MapReduce application in Hadoop to judge whether the communication time is worthy to be optimized. Second, we should compare the communication mechanisms in MPI and Hadoop for estimating improvement potential. In this paper, we will show the comparisons of latency and bandwidth in point-to-point communication primitives in MPI and Hadoop.

All experiments in this paper are run in the following platform. The hardware is a cluster consisting 8 nodes interconnected by a Gigabit Ethernet switch. Each node is equipped with two CPUs, and each CPU has Quad-cores. The CPU is Intel(R) Xeon(R) E5620 processor (2.40GHz). It has shared 1 MB L2 cache and 12 MB L3 cache. Each node has 16 GB memory and 170 GB disk. Each node runs the CentOS5.3 x86_64. The version of Hadoop is 0.20.2, which is run on JDK 1.6.0_22. The MPI implementation is MPICH2 1.3.

A. Estimate the communication percentage

This section focuses on estimating communication overhead of shuffle in Hadoop under different input data sizes and configurations, because the shuffle stage is a communication intensive process in the whole lifecycle of a MapReduce application. This process will contain large amounts of RPC invocations and big intermediate data transferring.

We deploy the Hadoop platform on 8 cluster nodes (1 master, 7 slaves), and use the JavaSort benchmark in GridMix [13] to measure Hadoop's communication overhead. The processed data size is 150 GB, and the block size adopts the default value of 64 MB. We set the max map/reduce number to 8/8 in each tasktracker. Through Hadoop's logs, we gather all reducers' running time and the consuming time of shuffle. Figure 1 shows the time of copy, sort, and reduce stages in every reducer. The X-axis means the reducer id (from 0 to 2344), while the Y-axis means the consuming time of copy, sort, and reduce stages. (Note: to make Figure 1 clear, we delete 56 $(7 * 8)$ values of reducers as their time reaches 4000 s.)

In Figure 1, the time of the copy stage in shuffle is from 48 s to 178 s, and the average shuffle time is about 128.5 s. The points of the sort stage are always near X-axis, and the average sort time is 0.0102 s. The max time of the reduce stage is 58 s, while the min is 2 s. The average time of the reduce stage is 6.7995 s. The total time of the copy stage in shuffle within all reducers occupies about 95% of the all reducers' whole life cycles.

Furthermore, we conduct another group of tests, in which the input data sizes vary from 1 GB to 150 GB, and the max map/reduce number in each node changes from 4/2 to 16/16, as shown in Table I. We sum all the time of copy stage in shuffle within all reducers in each experiment, and use this result to divide the sum of all mappers and reducers' execution time as the values in this table. Table I presents the overhead distributions of copy stage in shuffle under different input data sizes and configurations. From these results, we find the smallest percentage of the copy stage is 33.9%, while the biggest one is 82.7%. From all tests in this section, we conclude that the copy stage in shuffle is a time consuming phase.

However, we know that not all of the time in copy stage in shuffle is caused by RPC or Jetty. We argue that the statistical data in Figure 1 and Table I can expose the percentage of the communication overhead in a MapReduce application execution to a certain extent. The test result shows the communication time occupies a big percentage. Therefore, the communication process of a MapReduce application in Hadoop is indeed worthy to be optimized.

B. Compare the basic point-to-point communication primitives between MPI and Hadoop

This section compares the communication mechanisms in MPI and Hadoop for estimating improvement potential. In

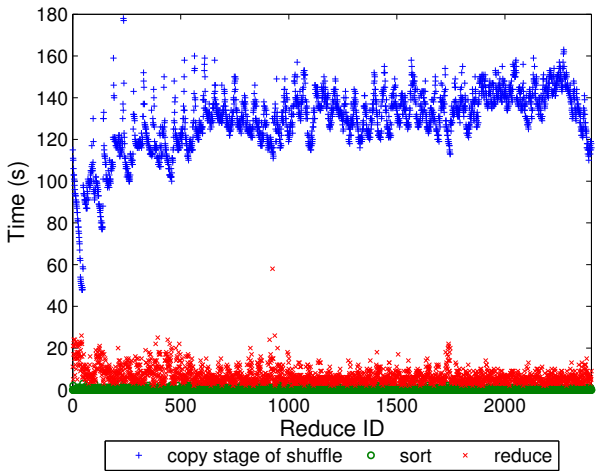


Figure 1: Overhead of Copy Stage in Shuffle of JavaSort Benchmark. (150 GB data and processed across 7 work nodes)

Table I: Percentage Distributions of All Copy Stage Time in Total Mappers and Reduceers Execution Time under Different Input Data Sizes and Configurations.

Input Data Size	Max Mapper / Reducer number in each node			
	4/2	**4/4**	**8/8**	**16/16**
1 *GB*	43.1%	43.0%	38.5%	35.7%
3 *GB*	35.0%	**33.9%**	35.9%	46.3%
9 *GB*	43.1%	42.9%	42.8%	39.7%
27 *GB*	44.3%	47.9%	43.18%	36.4%
81 *GB*	60.0%	71.0%	74.6%	73.9%
150 *GB*	69.6%	82.0%	**82.7%**	80.6%

MPI, we choose `MPI_Send` and `MPI_Recv` as the basic point-to-point communication primitives in our experiments. There are two important and basic communication primitives used in Hadoop and MapReduce applications:

- **Hadoop RPC**: It is the fundamental communication mechanism in Hadoop, and it can be found everywhere in the Hadoop code. It mainly used to exchange data among processes. Its major applications include block management operations between HDFS clients and the name node or data nodes, job management and learning system status operations between job clients and the jobtracker, status monitoring and information exchanging between tasktrackers and taskrunners, system status and information exchanging among nodes, and so on.
- **HTTP over Jetty**: Hadoop creates several Jetty embedded servers to answer http requests. The primary goal is to serve up status information for the server. One of its main application is used to transfer intermediate data for mappers and reducers in the copy stage of shuffle in MapReduce applications.

We conduct two groups of experiments to compare the basic point-to-point communication primitives between MPICH2 and Hadoop. The first group is to compare the data exchanging latency among varies sizes of messages (small / medium / large) between MPICH2 and Haddop RPC. The second group compares the bandwidth among Hadoop RPC, HTTP over Jetty, and MPICH2. The aim of these tests is to show how much benefit can be achieved if all system information exchanging and intermediate data transferring are implemented by an MPI-like efficient communication library.

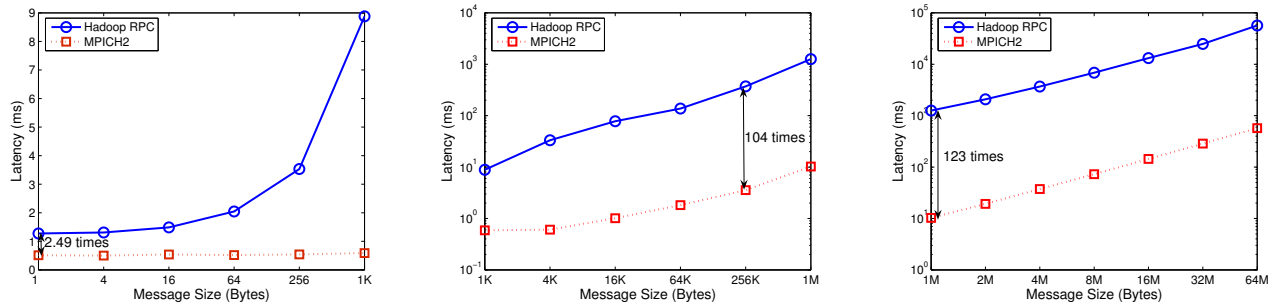
In Hadoop RPC, we implement a basic class extending from `VersionedProtocol` Interface (required in Hadoop RPC) with a simple `recv` method, which only checks the received data size. In order to test the ping-pong time, this method will return the received data back to the invoker. We choose the value of the ping-pong time divided by 2 as the latency. This class is registered in the Hadoop RPC proxy and server. In MPICH2, we also implement the same program logic in sending and receiving sides with `MPI_Send` and `MPI_Recv` operations. Each test result in the following experiments is an average value of 100 tests. In order to avoid the overhead caused by class loading and object instantiation, we drop the first 5 test values of Hadoop, which is implemented by Java.

Figure 2 shows the comparison results of the latency in the basic point-to-point send/recv operations between MPICH2 and Hadoop RPC. As Figure 2(a) shows, when the message size is small (from 1 byte to 1 KB), the latency of MPICH2 does not exceed 1 ms. When the message size varies from 1 byte to 16 bytes, the latency of Hadoop RPC is about 1.3 ms. In the case of 1 byte message, the latency of Hadoop RPC is 2.49 times of that in MPICH2, which is the smallest gap between them in the whole test. However, as the message size increasing over 16 bytes, the gap dramatically risies up. When the message size reaches 1 KB, the latency of Hadoop RPC is 15.1 times of that in MPICH2.

When the message size varies from 1 KB to 1 MB, as shown in Figure 2(b) , the MPICH2 latency rises from 0.6 ms to 10.3 ms. However, the Hadoop RPC latency grows even faster, from about 8.9 ms to 1259 ms. When the message size exceeds 256 KB, the Hadoop RPC latency is 100 times higher than that in MPICH2.

When the message size is enlarged from 1 MB to 64 MB, as shown in Figure 2(c), the Hadoop RPC latency rises from about 1259 ms to 56827 ms, nearly increasing 50 times, while MPICH2 latency moves from 10.2 ms to 572 ms. In this picture, we find when the message size is 1 MB, the latency of Hadoop RPC is 123 times of that in MPICH2, which is the biggest multiple between them in the whole test.

From this group of tests, we find that the potential for improvement on message latency of using MPI to benefit



(a) Small Messages (Size from 1 B to 1KB) (b) Medium Messages (Size from 1 KB to 1 MB) (c) Large Messages (Size from 1 MB to 64 MB)

Figure 2: Comparisons of Message Latency between Hadoop RPC and MPICH2.

Hadoop is indeed high and could reach two orders of magnitude.

The second group of tests focuses on the bandwidth. We use the Hadoop RPC, Jetty, and MPICH2 to transfer fix-sized of data (128 MB) when the message packet size varies from 1 byte to 64 MB. In Hadoop RPC, we transfer the data through the parameter in the RPC method. HTTP over Jetty is another basic mechanism of intermediate data exchanging in Hadoop. One of the most important usages is adopted in the copy stage of shuffle in MapReduce applications. In the copy stage, the reducer process will copy output from mappers remotely by a HTTP servlet, which is served in an embedded Jetty server in a tasktracker. Typically, it is a multi-threaded copying in this stage, but in order to keep simple, we analyze the situation of point-to-point single thread copying. We carefully extracted the minimal codes of data transferring logic from it. Then, we develop a program with those codes and deploy it in an independent HTTP server over Jetty to complete this test. We still use `MPI_Send` and `MPI_Recv` to transfer data in MPICH2.

Figure 3 shows that the Hadoop RPC cannot use bandwidth efficiently. The largest bandwidth achieved by the Hadoop RPC is only 1.4 MB per second. However, Jetty and MPICH2 can use the bandwidth effectively since the message size exceeding 256 bytes. When the message size varies from 256 bytes to 64 MB, the bandwidth of Jetty is about 80 MB per second to more than 100 MB per second, and the bandwidth of MPICH2 is about 60 MB per second to more than 110 MB per second. This indicates Jetty and MPICH2's bandwidths are about 100 times of that in the Hadoop RPC. However, the average value of peak bandwidth achieved by MPICH2 is about 111 MB per second, while Jetty is about 108 MB per second. Furthermore, during our tests, the peak bandwidth of MPICH2 is much smoother than Jetty.

From this group of tests, we find that the potential for improvement on transferring bandwidth of using MPI to benefit Hadoop could reach two orders of magnitude

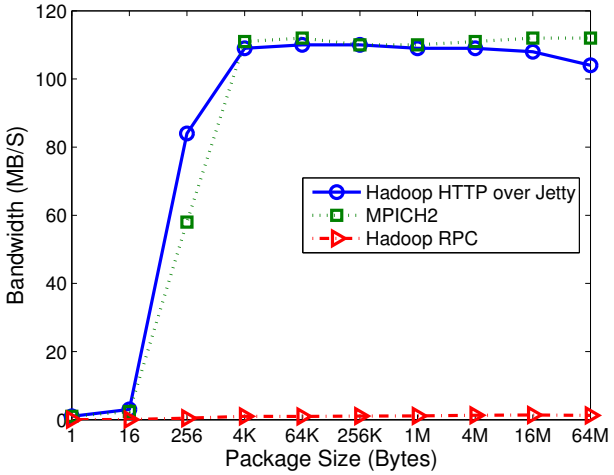


Figure 3: Comparison of Bandwidth among Hadoop RPC, Hadoop HTTP over Jetty, and MPICH2.

when compared with Hadoop RPC, and could reach a few percentage points (2%-3%) when compared with Hadoop HTTP over Jetty.

C. Summary

It is hard to give an exact formula to identify the communication speedup that can be achieved by MPI, due to the parallel and overlapping patterns of computing and communication. And we also hardly to identify how many percentages in the communication time are caused by RPC, Jetty, or others.

However, from all of above tests, we can see that the message latency of MPI is about 100 times less than Hadoop primitives. The average peak bandwidth of MPI is about 100 times higher than Hadoop RPC, while the average peak bandwidth of MPI is about 2%-3% higher than HTTP over Jetty. Therefore, when using Hadoop to process large data, if a data intensive communication library is implemented, with

performance close to MPI, application performance gains could be high.

III. Challenges of Adapting MPI to Hadoop and MapReduce Applications

The above tests illustrate that if we can successfully adapt MPI to Hadoop, high performance improvement could be achieved on Hadoop and MapReduce Applications. In order to check how much performance improvement potential could be achieved in MapReduce applications, this paper decides to make a completed test of application examples.

However, adapting MPI to Hadoop is not an easy task. In a programmer's perspective, there are mainly three technical issues:

- **Core data structures are quite different between MPI and MapReduce applications**. Traditional MPI programs usually operate on contiguous and fix-sized data (e.g., array, matrix) in memory, while MapReduce programs generally operate on non-contiguous and variable sized key-value pair data. If a programmer directly uses the traditional MPI to develop MapReduce applications, he or she must handle data non-contiguity and size variability by extra effort, even though MPI can supply some functional supports, like `MPI_Pack`/`MPI_Unpack`.
- **Collective communications in MPI cannot be easily adopted by Hadoop MapReduce applications**. The MPI interfaces for collective communications have highly efficient implementations and seem to be very appropriate for mapper and reducer's all-to-all or all-to-one communication patterns. But from Hoefler's work [14], we can see that there are fussy details that need to be maintained by a programmer. For example, the number of different intermediate keys needs to be known beforehand by all processes. If not all processes contain a value for each key, the programmer must still need to supply an identity element with respect to the target key without value. In addition, the collective communication operations in MPI (e.g., `MPI_Reduce`) generally require participation by all processes. However, this communication pattern is not appropriate for arbitrary MapReduce applications. In this paper, we try to make the interfaces of the communication library simple and lean while supporting the communication behaviors in the MapReduce model.
- **Communication styles are different in MPI and MapReduce applications**. Communication destinations are always not assigned between mappers and reducers in MapReduce applications. When a programmer writes a map function, he or she does not and needs not know the destination of the output key-value pair. But with the traditional MPI, we must assign the rank of target process in a communication operation.

Table II: The Message Passing Interfaces of MPI_D.

void MPI_D_Send(S_KEY_TYPE key, S_VALUE_TYPE value);
void MPI_D_Recv(R_KEY_TYPE key, R_VALUE_TYPE value);

Separate approaches can be identified to solve the above problems individually. The biggest challenge is to find a minimal extension to the MPI standard that can help alleviate all these challenges, while promise to enhance performance.

We propose such a minimal extension to MPI, shown in Table II. The extension maintains the MPI style and includes only two operations, one for the sending side and another for the receiving side. This extension is implemented in a prototype library called MPI-D, which is built on the basic point-to-point primitives in MPI to support arbitrary MapReduce operations. The `MPI_D_Send` operation is used to send the intermediate data (key-value pair) outputted by mappers. The `MPI_D_Recv` operation is used to collect the intermediate data for reducers. In addition, MPI-D also supplies two environment initialization interfaces called `MPI_D_Init` and `MPI_D_Finalize`.

This extension is the exposed interfaces of the MPI-D library. The MapReduce application can be implemented utilizing MPI-D. The main features of this extension include:

- It is simple and captures the key-value pair nature commonly found in data intensive computing.
- It is a minimal extension to the traditional MPI and does not impact legacy MPI applications.
- Applications only need to send a pair of $\langle key, value \rangle$ data, and the communication process can be automatically completed in MPI-D library space. Communication details are transparent to the applications, which is appropriate for the MapReduce computing model.
- Advanced optimization on communication can be achieved in the MPI-D library. For example, local combination of key-value pairs with the same key to reduce message size, automatic realigning of data in memory to construct contiguous and fix-sized data for MPI communication, and transparent recovering them to key-value pairs for the MapReduce applications.

IV. Simulation with the MPI-D Prototype

In order to make a completed test of application examples to show performance improvement potential, which could be achieved by Hadoop augmented with MPI-D, we design and implement a simulation system with the MPI-D prototype for MapReduce applications. We also develop a WordCount example as a micro application benchmark. Then a performance experiment is conducted between WordCount in Hadoop and in the simulated system with MPI-D.

A. Overview

This section will introduce the current implementation of MPI-D prototype, and explain the MapReduce application running process on the simulated system with MPI-D library. Our currently implementation of MPI-D prototype is based on MPICH2, and it is built on top of MPI as a convenience high-level library. In order to make our experiment environment more comparable with ordinary Hadoop, which contains MapReduce computing framework and Hadoop distributed file system, we use rank 0 process in the simulation system to simulate the master process, like the jobtracker process in Hadoop. Other processes are used to simulate workers. In addition, we distribute all input data across all nodes to guarantee the data accessing locally as in Hadoop.

Figure 4 shows the overview of MapReduce application running process in our simulation system with MPI-D library. When the MapReduce application starts up, the mapper processes will scan input data records continuously, and send the records to map function user defined.

Map function parses the records and produces outputs in the form of key-value pairs. These output key-value pairs can be directly sent out by `MPI_D_Send` interface. In the common case, `MPI_D_Send` routine will buffer the key-value pairs in a hash table, and return the invocation procedure immediately, which aims to achieve much more overlapping between computing and communication. In the `MPI_D_Send` routine, the key-value pair will be local combined by a combiner, who is also can be implemented by applications. The combiner commonly gathers pairs of the same key together, and constructs a key and value list pair. For instance, the key-value pairs $\langle K_1, V_1 \rangle$, $\langle K_1, V_1' \rangle$ will be combined as a key and value list pair $\langle K_1, \{V_1, V_1'\} \rangle$. The aim of combining is to reduce the memory consuming and the transmission quantity. Similar to Hadoop, in a common case, the combine function can be user defined and is always assigned as the reduce function.

When the hash table buffer exceeds a particular size, a thread will be created to spill out the data from the hash table to partitions, which are a set of continuous arrays with fixed size. Data in each partition will be sent to the corresponding reducer. This process contains two important functions. One is the hash-mod based partition selection. The key and value list pairs in the hash table buffer will be moved to partitions through a hash-mod selector. The selector selects the pairs according to their keys' hash values. So, we redirect outputs from mappers to reducers by partitions. Our implementation is similar to the `HashPartitioner` in the Hadoop MapReduce framework. The other important function is data realignment, which is reformatting key and value list pairs from a discrete hash table to an address-sequential and fix-sized partition. In addition, when MPI-D puts the key value list pairs to a partition, it can also sort the value list for each key on demand. Data realignment is an important step of MPI-D library, so it can be improved in several aspects, like high performance sorting and compressing data. At last, the data in partitions will be sent out by the `MPI_Send` primitive and the destination will be assigned automatically according to the partition number.

In the receiving side, each reducer adopts the `MPI_Recv` primitive in the wildcard reception style to receive messages from any source. Multiple data flows in mappers' partitions are sent to the corresponding reducer concurrently, while reducers receive and combine them in memory. The reducer will adopt a streaming mode to process the data for saving memory space. When the data partition is full, it will trigger reverse realignment process. In this process, the sequential data stream will be re-constructed as key-value pairs and reduce runners will dispatch them to the reduce function user defined.

MPI-D makes the processes between `MPI_D_Send(K, V)` and `MPI_D_Recv(K, V)` transparently to MapReduce application developers and supplies them with easy-to-programming interfaces. In current status, this library is just a prototype, which can be optimized in the future, like dynamic process management of mapper and reducer processes, `MPI_Isend` and `MPI_Irecv` adoption to achieve much more overlapping of computing and communication, trying to find opportunities to use collective operations inner MPI-D library, etc.

From above discussion shown, the computing process in our simulation system with MPI-D library is similar to a MapReduce application running on ordinary Hadoop. Therefore, we can develop some micro-benchmark examples to do evaluations between our system and Hadoop.

B. Programming examples on MPI-D

In order to evaluate MPI-D, we write a common example, WordCount, which is implemented based on above simulation system with the MPI-D library, and use this example as a micro benchmark to compare with the ordinary WordCount example in Hadoop. Figure 5 is the simple example of WordCount implemented by MPI-D interfaces. From this example, the features listed in above section are demonstrated a lot.

The WordCount example in Figure 5 is developed explicitly and directly by MPI-D interfaces, which is good for clearly demonstration. However, the typical MapReduce applications in Hadoop always do not directly invoke communication operations, but through context collectors to hide the communication processes. Actually, our MPI-D interfaces can be also adopted inner the map and reduce runners, and we can keep them transparently for the developers. Then, all existing developed MapReduce applications can be effected as little as possible.

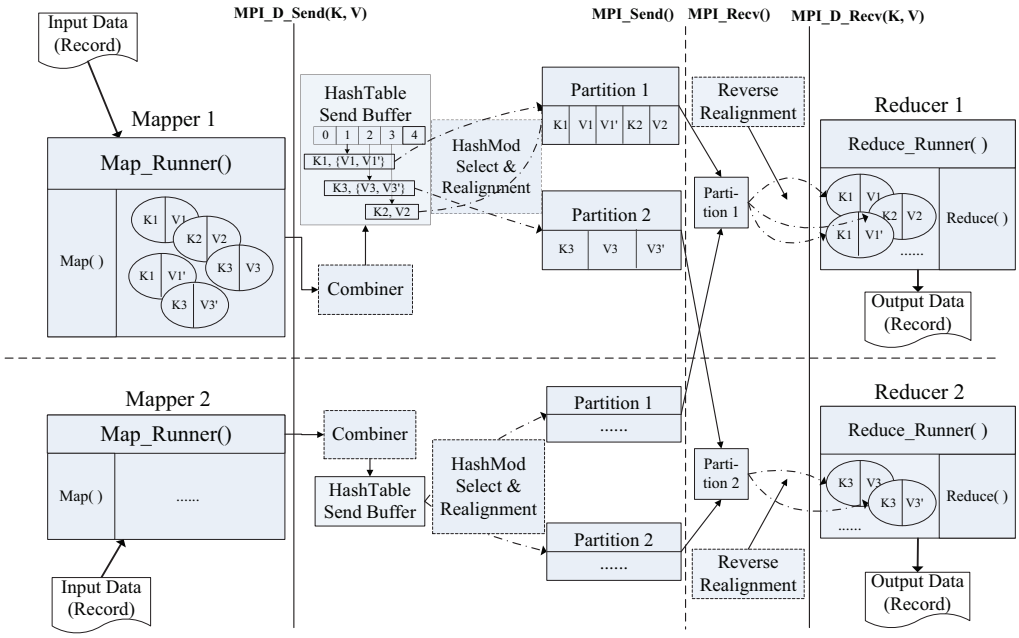


Figure 4: Overview of MapReduce Application Running Process in Our Simulation System with the MPI-D Library.

```
// WordCount example implemented on MPI-D interfaces

void map (MAP_KEY mk, MAP_VALUE mv)
{
    // do some map logic ...
    REDUCE_KEY[] kt = parse(mv);
    for(i = 0; i < kt.length; i++)
    {
        MPI_D_Send( kt[i], 1);
    }
}

void reduce (REDUCE_KEY rk, REDUCE_VALUE rv)
{
    MPI_D_Recv( rk, rv);
    increment (rk, rv);
}
```

Figure 5: WordCount Example Implemented on MPI-D Interfaces.

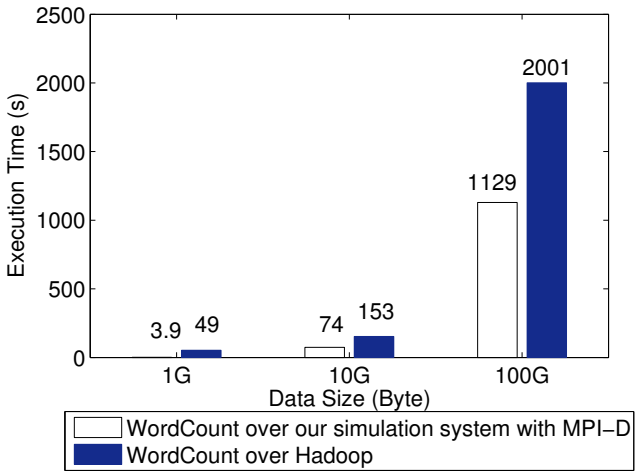


Figure 6: Performance Comparison of WordCount Example between Our Simulation System with the MPI-D Prototype and Hadoop.

C. Performance evaluation

We conduct a performance evaluation experiment of WordCount examples based on ordinary Hadoop and our simulation system with the MPI-D prototype. 8 nodes are used to build the experimental platform, and 7 nodes of them are used as worker nodes. Our simulation system with MPI-D also run on these 8 machines. We set the maximum concurrent number of mappers and reducers are 7/7, and left one slot to the OS and other processes. The input data sizes vary from 1 GB to 100 GB. And we configure the number of MPI processes in our simulation system same as the number of parallel mappers and reducers in Hadoop, that is 49 processes as concurrent mappers, and 1 process as the reducer. Another one process is the rank 0 process as the master.

Figure 6 shows the results of performance comparison of WordCount examples in the two cases. As the data size changing from 1 GB to 100 GB, WordCount on Hadoop

consumes about 49 s to 2001 s, while the time of WordCount on our simulation system is from 3.9 s to 1129 s. Therefore, we find that our simulation system reduce execution time to 8%, 48%, 56% of the ordinary Hadoop, when the processed data size is 1 GB, 10 GB, and 100 GB across 7 nodes.

From this simulation based test, we can see that performance improvement can be really achieved by adopting MPI to benefit MapRedcue applications.

V. Related Work

The MapReduce programming model was introduced in 2004 by Dean et. al in [5]. Hadoop [3, 4] is the most successful open source implementation version of MapReduce. MPI is a library specification for message passing, proposed as a standard by a broadly based committee of vendors, implementers, and users [6–10]. These results form the basis and environment for our research.

Hoefler et.al. [14] is the perfect attempt to write MapReduce applications using MPI blocking and non-blocking collective operations. The authors also pay attention to exploit MPI-2.2 and MPI-3 features to achieve performance improvement of up to 25% on multiple cluster nodes. There are three major differences between our work and theirs. First, the motivations are quite different. Hofeler et.al. focuses on utilizing operations and features provided by MPI to implement highly efficient MapReduce applications. They share very valuable practice experiences and also reveal limitations of MPI in current versions. Our work tries to adapting MPI to efficiently support Hadoop and MapReduce applications. Second, the deliverables of our work and theirs are different. We plan to deliver a minimally extended MPI library for the data intensive computing environment in general and Hadoop in particular. The deliverable of [14] is experiences, insights and suggestions. Third, the approaches are different. We try to find a minimal and efficient extension to MPI to benefit Hadoop and MapReduce applications. They use MPI operations to write efficient MapReduce applications. We are indebted to [14] for valuable information, especially on the challenges identified.

Plimpton et. al. [15, 16] is another good example of using MPI to implement MapReduce applications. They describe their MR-MPI library and several MapReduce graph algorithms. The MR-MPI library is implemented on MPI and provides a MapReduce interface to application developers. Our work differs from theirs in that we try to provide a minimal extension to MPI, and improve Hadoop application performance. Furthermore, we hope our work effects the Hadoop ecosystem and existing Hadoop applications as little as possible.

Sur et. al. [17] have shown that adopting high performance interconnects (e.g., InfiniBand or 10 Gigabit Ethernet) with or without Solid State Drive (SSD) can achieve performance improvement varying from 11% to 219% for the sort, random write, and sequential write benchmarks in the Hadoop distributed file system. From this work, we are happy to see the big performance improvement possibility by using high performance communication technologies, and the authors' work inspires us to investigate the communication detail in Hadoop system, and make comparison with MPI.

Twister [18] uses a pub-sub broker network to connect mappers and reducers using topics and adopts direct TCP links for data communication. The direct TCP links are used to prevent the broker network from large data transfers. This work is interesting, but the motivation and the adopted techniques are quite different between us.

VI. Conclusion and Future Work

This paper attempts to answer the question: can MPI be adapted to substantially speed up Hadoop and MapReduce applications, by reducing communication overheads? We conduct two groups of experiments using the MPI send/recv communication primitives and the Hadoop RPC and HTTP over Jetty communication primitives. Compared to the Hadoop RPC, the MPI point-to-point communication primitives show highly better latency and bandwidth. In particular, MPICH2 shows two orders of magnitude improvement in latency when the message size exceeds 1 KB. Meanwhile, when transferring 128 MB data within different packet sizes, the average peak bandwidth of MPICH2 is about 100 times higher than Hadoop RPC and about 2%-3% higher than Jetty over Ethernet.

Extending the conventional MPICH library, we implement a prototype communication library for data intensive applications, called MPI-D (short for MPI Data Extension). The MPI-D library presents to programmers a key-value pair based communication interface, more appropriate for MapReduce programming scenarios than the original MPI interfaces. We test the WordCount example based on the original Hadoop and the same benchmark on our simulation system over the MPI-D library. Experimental results show that our simulation system can reduce application execution time by 44%, when the processed data size is 100 GB across 7 nodes.

Our research is in an early stage currently, and it shows that there is indeed much room for speedup, when MPI is adapted in a data intensive computing environment such as Hadoop. Future research directions include (1) to compare the primitives between MPI and Socket over Java NIO, which is mainly used to transfer data blocks between datanodes in Hadoop; (2) to identify and refine the programming interface to comply with the MPI standard, while at the same time meet the data-intensive application requirements; (3) to optimize the MPI-D library to exploit its potential, especially improving scalability; and (4) to utilize high performance interconnects such as the Infiniband and datacenter networks.

VII. Acknowledgment

We are indebted to Hao Wang of Ohio State University and Shicai Wang of Institute of Computing Technology for helpful discussions. This research is supported in part by China Hi-Tech Research and Development Program (the 863 Program) (Grant No. 2009AA01A131 and 2009AA01A130) and China Basic Research Program (the 973 Program) (Grant No. 2011CB302502 and 2011CB302803). We also thank the reviewers for valuable comments.

References

[1] R. E. Bryant, "Data intensive scalable computing: Harnessing the power of cloud computing," http://www.cs.cmu.edu/~bryant/pubdir/disc-overview09.pdf, Feb. 2009, revised manuscript (2009) based on his talk at the 2007 Federated Computing Research Conference.

[2] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu, "RCFile: a fast and space-efficient data placement structure in MapReduce-based warehouse systems," in *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE 2011)*. IEEE, Apr. 2011, pp. 1199–1208.

[3] "Apache hadoop," http://hadoop.apache.org/.

[4] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., Oct. 2010.

[5] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[6] "Message passing interface (MPI) forum home page," http://www.mpi-forum.org/.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard." *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sep. 1996.

[8] "MPICH2 : High-performance and widely portable MPI," http://www.mcs.anl.gov/research/projects/mpich2/.

[9] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[10] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall, "Open MPI: goals, concept, and design of a next generation MPI implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface (Euro PVM/MPI 2004)*, ser. Lecture Notes in Computer Science, D. Kranzlmller, P. Kacsuk, and J. Dongarra, Eds. Berlin / Heidelberg: Springer, 2004, vol. 3241, pp. 97–104.

[11] K. Hwang and Z. Xu, *Scalable Parallel Computing: Technology, Architecture, Programming*. New York, NY, USA: McGraw-Hill, Inc., 1998.

[12] "Jetty WebServer," http://jetty.codehaus.org/jetty/.

[13] "Gridmix," http://hadoop.apache.org/mapreduce/docs/current/gridmix.html.

[14] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards efficient MapReduce using MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface (Euro PVM/MPI 2009)*, ser. Lecture Notes in Computer Science, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Berlin, Heidelberg: Springer, Sep. 2009, vol. 5759, pp. 240–249.

[15] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for large-scale graph algorithms," *A special issue of Parallel Computing*, pp. 1–39, 2011.

[16] "MapReduce-MPI library," http://www.sandia.gov/~sjplimp/mapreduce.html.

[17] S. Sur, H. Wang, J. Huang, X. Ouyang, and D. K. Panda, "Can High-Performance interconnects benefit hadoop distributed file system?" in *Proceedings of the Workshop on Micro Architectural support for Virtualization, Data Center Computing, and Clouds (MASVDC'10), in Conjunction with the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2010)*, Atlanta, GA, USA, Dec. 2010.

[18] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010)*. Chicago, Illinois: ACM, 2010, pp. 810–818.