Accelerating Apache Hive with MPI for Data Warehouse Systems

Lu Chao [1,2], Chundian Li [1,2], Fan Liang [1,2], Xiaoyi Lu [1], Zhiwei Xu [1]

[1]*Institute of Computing Technology, Chinese Academy of Sciences*
[2]*University of Chinese Academy of Sciences*
{chaolu,lichundian,liangfan,luxiaoyi,zxu}@ict.ac.cn

***Abstract*—Data warehouse systems, like Apache Hive, have been widely used in the distributed computing field. However, current generation data warehouse systems have not fully embraced High Performance Computing (HPC) technologies even though the trend of converging Big Data and HPC is emerging. For example, in traditional HPC field, Message Passing Interface (MPI) libraries have been optimized for HPC applications during last decades to deliver ultra-high data movement performance. Recent studies, like DataMPI, are extending MPI for Big Data applications to bridge these two fields. This trend motivates us to explore whether MPI can benefit data warehouse systems, such as Apache Hive. In this paper, we propose a novel design to accelerate Apache Hive by utilizing DataMPI. We further optimize the DataMPI engine by introducing enhanced non-blocking communication and parallelism mechanisms for typical Hive workloads based on their communication characteristics. Our design can fully and transparently support Hive workloads like Intel HiBench and TPC-H with high productivity. Performance evaluation with Intel HiBench shows that with the help of light-weight DataMPI library design, efficient job startup and data movement mechanisms, Hive on DataMPI performs 30% faster than Hive on Hadoop averagely. And the experiments on TPC-H with ORCFile show that the performance of Hive on DataMPI can improve 32% averagely and 53% at most more than that of Hive on Hadoop. To the best of our knowledge, Hive on DataMPI is the first attempt to propose a general design for fully supporting and accelerating data warehouse systems with MPI.**

***Keywords*—Data Warehouse Systems, Apache Hive, Hadoop, MPI, DataMPI, TPC-H**

I. Introduction

Distributed management and analytics of large volumes of data is a significant challenge being faced by the Big Data community. Modern data warehouse systems have been regarded as effective and important tools to obtain valuable knowledge from Big Data. As one of the most important examples, Apache Hive [1] is an open-source data warehouse system upon the Hadoop framework [2] . It supports a convenient query language named HiveQL, which is a SQL-like declarative language and easy to use [3]. Hive has a compiler and an execution engine. The compiler translates the query into MapReduce jobs and the execution engine submits the jobs to Hadoop framework. Containing the basic elements of SQL, HiveQL provides support to the tables composed of primitive types, arrays, maps and the combination of them. Since Hive is constructed upon the MapReduce programming model, it can scale out easily and tolerate faults. It also supports user-defined query functions. With these advantages, Hive gets recognized and used widely in the distributed computing field.

However, current generation Apache Hive has not fully embraced High Performance Computing (HPC) technologies to deliver optimal performance on modern clusters for the queries, even though the trend of converging Big Data and HPC is emerging. For example, in traditional HPC field, Message Passing Interface (MPI) [4] and its implementations have been optimized and used popularly for HPC applications during last decades to deliver ultra-high communication performance on various high-performance networks (e.g. InfiniBand [5] and 10/40 GigE). Recent studies [6], [7], [8], [9], [10], [11] have shown that HPC communication technologies (e.g. MPI, RDMA on high-performance networks) can improve the performance of Big Data applications significantly. As a specific example of the trend, DataMPI [7], [8] is an efficient communication library aiming at extending MPI for Big Data applications, which can productively support MapReduce paradigm with better performance.

The trend of converging Big Data and HPC technologies motivates us to explore whether MPI can benefit data warehouse systems, such as Apache Hive. However, using MPI to accelerate Apache Hive still has many questions to be answered.

- **What kinds of opportunities and challenges are there to use MPI to accelerate Apache Hive?** MPI is mainly optimized for data movement. What kinds of communication characteristics do typical Hive workloads have? Do we have opportunities to optimize the data movement process? And how difficult can it be? Is it feasible that the proposed design has minimal modifications in Hive?
- **Is it possible to use DataMPI to accelerate Apache Hive?** Although DataMPI has taken a further step to bridge HPC and Big Data fields, can it be used to accelerate Apache Hive? What kind of efficient design can be proposed by utilizing DataMPI?
- **How many performance benefits can be achieved by using DataMPI?** The major goal is to accelerate Hive query performance by using MPI. Can DataMPI significantly improve the performance for different Hive workloads, like all the queries in TPC-H?

To address the above problems, this paper first investigates the opportunities and challenges of accelerating Hive with DataMPI to further discuss the motivation of our design. Through the investigation, we find that typical Hive workloads have irregular communication characteristics, causing

the difficulties for the underlying data movement subsystem design. Then, we present a plug-in-based design called Hive on DataMPI to replace the low-level execution engine of Hive from Hadoop to DataMPI. Aiming at Hive's irregular communication characteristics, we propose optimized non-blocking communication with tuning and enhanced parallelism mechanisms for the DataMPI engine. Through our systematical performance evaluation, we observe that the performance of Hive on DataMPI improves 30% on average than that of Hive on Hadoop MapReduce for Intel HiBench workloads. The further performance breakdown experiments with Intel HiBench show that our design can improve the performance of the shuffle phase (major data movement phase) by 20%-70% in different jobs. Moreover, the factors of light-weight framework, efficient job startup and data movement mechanisms accelerate the Hive job execution. By further evaluating our design with TPC-H [12], the results show that Hive on DataMPI can fully and transparently support all TPC-H queries, offering 20% and 32% better performance averagely than Hive on Hadoop with Text and ORCFile formats [13], respectively. The best case can achieve 53% improvement in TPC-H Q12 query with a 20 GB ORCFile data set. Results also show that Hive on DataMPI has good scalability, efficient resource utilization, and high productivity.

To the best of our knowledge, Hive on DataMPI is the first attempt to propose a general design to fully support and accelerate data warehouse systems with MPI.

The rest of the paper is organized as follows. Section II reviews the background of DataMPI. Section III presents the opportunities and challenges of the whole idea. We present the design of accelerating Apache Hive with DataMPI in Section IV. Section V describes our detailed evaluation with Hive on DataMPI. Section VI discusses the related work. Section VII concludes this paper and presents our future work.

II. Overview of DataMPI

DataMPI [6], [7], [8] is aiming at extending MPI for Big Data applications by proposing a bipartite communication model and key-value pair based communication operations. The bipartite communication model in DataMPI defines that intermediate data moves from the tasks in communicator O (Operators, like Mappers) to those in communicator A (Aggregators, like Reducers) with user configurable data movement behaviors. The key-value pair based communication operations are more suitable for the data movement requirements of Big Data applications than the buffer-to-buffer communication operations in traditional MPI. Consequently, DataMPI can efficiently and productively support different kinds of Big Data applications [7], [9].

DataMPI provides kinds of modes for Big Data applications (e.g. *common*, *iteration* [14] and *streaming*). As the most basic one, the *common* mode implements the bipartite communication model and supports SPMD-style programming like traditional MPI programming. And MapReduce applications can be rewritten with key-value pair based communication operations productively. A typical DataMPI application is launched by the *mpidrun* command, then a set of DataMPI working processes will be spawned to receive and execute data-centric scheduled tasks. To maintain the context of corresponding communicators in each scheduled task, all *MPI_D* routines should be surrounded by a pair of *MPI_D.init()* and *MPI_D.finalize()* routines. The O tasks and A tasks are created with *MPI_D.COMM_BIPARTITE_O* and *MPI_D.COMM_BIPARTITE_A*, respectively. According to the scheduling policy, the A tasks will be executed only when all O tasks finish. To transfer each key-value pair from O-side to A-side with a relaxed all-to-all communication pattern (like Shuffle), *MPI_D.send()* routine in O tasks sends the pairs and *MPI_D.recv()* routine receives the pairs in the A tasks. Detailed runtime information can be obtained from *MPI_D.Comm_rank()* and *MPI_D.Comm_size()*. Moreover, user-defined functions (Partitioner, Combiner and Comparator, etc.) can be set up in the configuration to satisfy various requirements.

III. Opportunities and Challenges

We take micro tests on our testbed, which will be introduced in Section V, and breakdown the execution time of the benchmarks with Intel HiBench [15] over a 20 GB data set. Two queries are used including the AGGREGATE query which groups the rows with one column attribute and the JOIN query which joins one small table with another big table. We breakdown each job execution time into three sections according to the operations, including startup, Map-Shuffle and others. The JOIN query has three stages in Hive and each of which is a MapReduce job. The Map-Shuffle operation overlaps the Map phase and Shuffle phase in Hadoop. As shown in Figure 1, the average Map-Shuffle operation takes up over 50% time of a MapReduce job, implying that optimizing the data movement process of I/O bound workloads is an opportunity to speed up Hive. Besides, startup time occupies 5% of total time, providing another opportunity to optimize.

Using MPI to accelerate data warehouse systems is full of challenges, which are mainly listed as below: 1. MPI just provides simple point-to-point/collective communication operations which is impossible to use MPI operations to replace all underlying communication operations in Hive. It is a challenge to propose a feasible design to make MPI fully support typical Hive workloads, like Intel HiBench and TPC-H. We may have to implement all complex partition/sort/spill operations, etc, by using MPI for Hive. 2. It is hard to ensure that the design has performance portability to adapt for different cluster environments and fit various key-value types. However, using DataMPI is a good choice to solve these challenges.

We also analyze the different communication patterns in Hadoop under three situations (all are 20GB data sets): 1. HiBench AGGREGATE benchmark over HiBench data; 2. TeraSort benchmark over TeraGen data; 3. TPC-H Q3 query with three tables' JOIN operations over TPC-H data. As shown in Figure 2, we observe two characteristics in Hive's communication pattern:

1) **The skew time sequences of collecting operations:** Collect operations in Hadoop are called when key-value pairs are ready to transfer. To analyze the overhead between two consecutive collect operations in Hive and Hadoop, we modify the collect operators in Hadoop without actually sending key-value pairs and record the time sequences when the Map collect

operations are called. Figure 2(a) and Figure 2(b) show that when running HiBench AGGREGATE benchmark in Hive, the ending time of Map tasks is irregularly distributed from 19 seconds to 25 seconds, while that of TeraSort benchmark in Hadoop is almost centralized at 25 seconds. This is because when executing Hive benchmark, the processing complexity is affected by varied operator execution paths and sizes of input splits, while that of typical Hadoop benchmark is well-distributed.

2) **The variance between key-value pairs of collecting operations:** Figure 2(c) and Figure 2(d) show that when executing HiBench AGGREGATE benchmark, the sizes of the sending key-value pairs are centralized at 32 Bytes, while those in TPC-H Q3 benchmark are mainly distributed around 14 Bytes and 32 Bytes. This is because the length of a key-value pair sent by Hive applications differs from table types and column types.

Our previous studies [16] show that DataMPI can benefit MapReduce-like jobs for Big Data applications, but it doesn't optimize for irregular communication pattern in Hive's workloads. So it's a challenge to solve the mismatch between Hive's irregular computation-communication pattern and MPI's blocking/non-blocking communication.

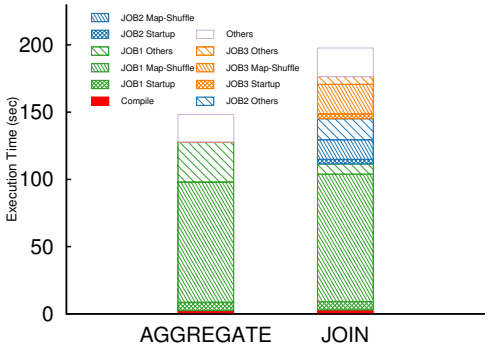


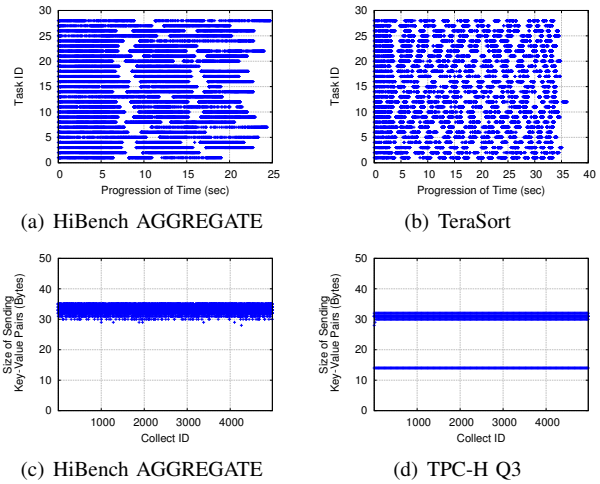
Fig. 1. Performance Breakdown of Hive in HiBench

IV. Design of Hive on DataMPI

A. Design Overview

In this section, we present the design of Hive on DataMPI. In order to make DataMPI fully compatible with HiveQL, we adopt the plug-in-based design principle, which means that we mainly insert new functions and modify with minimal impacts on original codes. In the meantime, plug-in-based Hive on DataMPI could be easily integrated with newer Hive releases.

Figure 3 shows the loose coupling Hive on DataMPI architecture and details about the compiler component. Hive mainly provides client and JDBC/ODBC accesses, and Metastore which stores metadata for Hive tables and partitions [17]. Besides the above components, the compiler and the execution engine are the key components of Hive. The compiler contains three stages. Firstly, semantic parser translates the query into Abstract Syntax Tree (AST), does type checking and semantic analysis to constitute the intermediate representation named



(a) HiBench AGGREGATE (b) TeraSort

(c) HiBench AGGREGATE (d) TPC-H Q3

Fig. 2. Observation of Hive's Irregular Operator Pattern Without Data Transfer (20 GB Data Set)

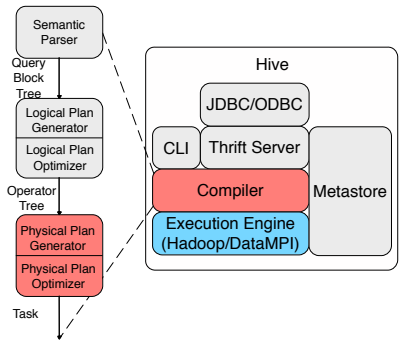


Fig. 3. Architecture Overview of Hive on DataMPI

the query block tree. Then, logical plan generator will organize the operator DAG, or named operator tree. The optimizer is important in Hive to get low latency upon Hadoop/DataMPI. Hive defines framework independent operators in logical plan with elaborate implementations, such as SelectOperator and FilterOperator. The physical plan is generated and optimized for Hadoop/DataMPI execution engine which is like a DAG of tasks. Finally, the execution engine converts the plan into different jobs and submits the jobs to the cluster.

To design the Hive on DataMPI, we conform to three principles:

1) Keep the Hive architecture intact.
2) Support Hive with minimal modifications.
3) Optimize the execution engine of DataMPI according to Hive's communication pattern.

B. A Light-weight Design of Hive on DataMPI

To start, we observe the similarities and differences between Hadoop and DataMPI to determine what would be kept or modified in Hive. They share the same or similar parts as below:

1) **HDFS support:** DataMPI also supports HDFS data access, so DataMPI can share the same input and out-

put files using existed Hive implementation without extra modifications.

2) **Similar semantics with MapReduce:** Since DataMPI's *common* mode uses a bipartite communication model to perform similar operations like MapReduce, DataMPI can inherit the same MapReduce operations in *map()* and *reduce()* by adding a few extra codes to handle the key-value pairs with *MPI_D_send()* and *MPI_D_recv()*.

Benefited from the above two points, Hive on DataMPI will not modify the operators already defined in Hive, and we continue to share the query plan optimized for Hadoop. This observation will simplify our implementation significantly.

But DataMPI still has two differences with MapReduce:

1) **The runtime environment:** DataMPI is a communication library instead of a framework. Hadoop jobs can be submitted to JobTracker via JobClient, but DataMPI jobs are launched from a *mpidrun* command which is similar to MPI.
2) **The execution flow in job:** DataMPI's *common* mode supports SPMD-style programming like MPI programs, rather than MPMD-style MapReduce programming in Hadoop.

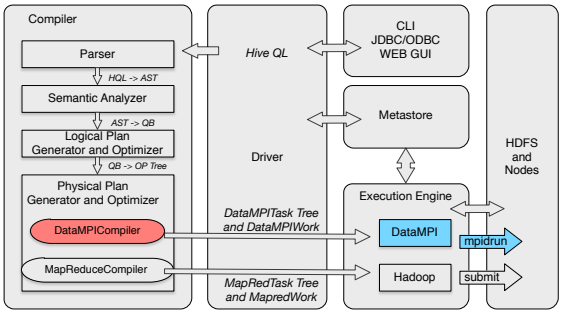


Fig. 4. Execution Flow of Hive on DataMPI

So Hive on DataMPI reserves the most parts of compiling components in Hive. In Figure 4, we still keep the compiler's execution flow from a HiveQL query to an operator tree. In order to start a DataMPI execution engine rather than a Hadoop execution engine, the physical plan generator and optimizer which converts an operator tree into a task tree has to be replaced. Thus, we add a new physical plan generator called DataMPICompiler. To make the new compiler selectable, users need to set *hive.execution.engine* = *"datampi"* in the Hive configuration.

DataMPICompiler generates DataMPIWork and DataMPITask, which are physical query plans converted from the existed operator tree. DataMPIWork describes the detailed operators in a stage, and DataMPITask describes how to configure and start a physical job. Due to the *common* mode of DataMPI covering the semantics of MapReduce, we currently adopt the same optimizing rules as Hadoop to divide the logical operator tree into different stages. Therefore, DataMPIWork inherits the same contents of MapredWork, including the MapWork and ReduceWork. After a task tree is generated, the execution engine will execute different jobs according to the information wrapped in tasks. When compared with Hive on Hadoop, DataMPIWork and DataMPITask are similar to MapredWork and MapRedTask, respectively.

Hence, when a new query comes to Hive Driver, it is first compiled into a series of DataMPITask objects. Then, Hive Driver starts to execute this query plan's task tree, and DataMPITask's *execute()* method constructs a DataMPI starting command line and serializes the necessary objects onto HDFS including DataMPIWork, configuration and split information. The starting command line with *mpidrun* command is shown as below:

```
mpidrun -f [hostfile]
-mode COM -O [ONum] -A [ANum] -jar hive-exec-0.13.1.jar
org.apache.hadoop.hive.ql.exec.dm.DataMPIHiveApplication
Dplan [tmp_dir]/plan.xml
Djobconf [tmp_dir]/jobconf.xml
Dsplit [tmp_dir]/split.xml
```

DataMPI runner spawns CommonProcess instances with the entry class of DataMPIHiveApplication running on the nodes configured in *hostfile*, and the processes are prepared for executing *ONum* O tasks and *ANum* A tasks which are specified by Hive. The parameters of *plan*, *jobconf* and *split* indicate the locations of serialized DataMPIWork, configuration and splits.

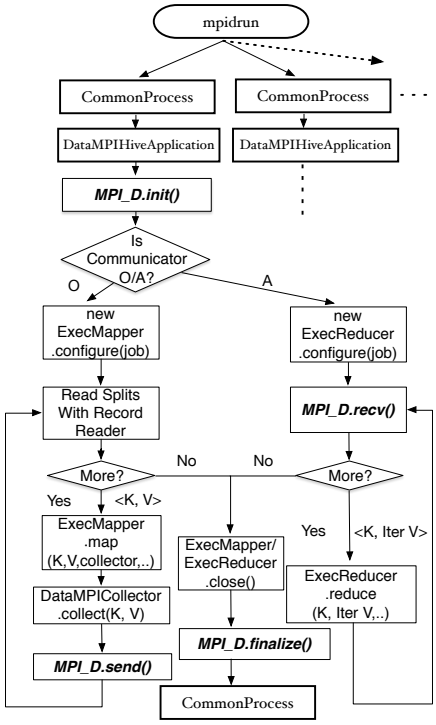


Fig. 5. Execution Flow of DataMPIHiveApplication

When O tasks with *MPI_D.COMM_BIPARTITE_O* are scheduled, objects are first deserialized from HDFS and installed to support the remaining execution in DataMPIHiveApplication. Then, as shown in Figure 5, *MPI_D* context's life cycle is from *MPI_D.init()* routine to *MPI_D.finalize()* routine. Since one key-value pair goes through the same procedure as it does in the MapReduce Mapper, the process will invoke a new ExecMapper to handle each key-value pair with configuring the

physical plan in JobConf. We replace the common MapReduce Collector to a special DataMPICollector. Thus, whenever a new record is read, the record is first transfered to the ExecMapper's *map()* method. And when *map()* is returned, the DataMPICollector's *collect()* method will use *MPI_D.send()* to send the generated key-value pairs to the A-side (Reducer).

It is worth mentioning that DataMPI has overlapped computation and communication operations by calling *MPI_D.send()* directly after each key-value pair is processed. Hadoop's reduce task starts to copy data until the first map is finished at least. But receiving processes in DataMPI have threads responsible for collecting and merging data when the send data size of some O task exceeds a threshold but without any O tasks finished. In this way, DataMPI can cache most of the intermediate data in memory by default and send them using high performance MPI communication directly. This characteristic will make the shuffle phase between MapReduce tasks efficient.

After all O tasks finalize, the scheduled A-side processes will invoke DataMPIHiveApplication with *MPI_D.COMM_BIPARTITE_A* as A tasks, and receive the key-value pairs by using *MPI_D.recv()*. Similar to the ExecMapper used in the O task, the A task invokes a new configured ExecReducer by passing the key and the iterator of same key's value-list to ExecReducer's *reduce()* function.

After all the A tasks finalize the *MPI_D* context, a physical plan ends. DataMPITask will do some cleaning work and finally return the results to the Driver.

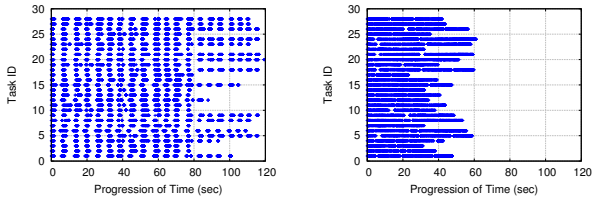
C. Optimizing DataMPI Engine for Hive Workloads

The shuffle operation in MapReduce will send distinct data in Mappers to each of the Reducers, which is similar to the MPI-AlltoAll communication. The Reducers have to wait for the completion of the data transmission from Mappers to continue execution because of the data dependency between Map and Reduce. To overlap the data transmission and Map operations, Hadoop launches Map and Reduce tasks concurrently, where Reduce tasks copy the intermediate data from completed Map tasks. Different from the coarse-grained communication mechanism of Hadoop, DataMPI provides partition-based communication mechanism which can overlap the data movement and computation efficiently.

DataMPI supports the relaxed all-to-all communication pattern with blocking style and non-blocking style. Users can set the communication mechanism according to the different applications. In the buffer manager, DataMPI designs Send Partition Lists (SPL), and each partition is used to store key-value pairs for corresponding A tasks. When the send partitions are full, they will be pushed into the send queue in the shuffle engine, and wait for transmission. In the blocking style, the shuffle engine starts a thread for transferring data, who will create send/receive requests with *MPI_Isend*, *MPI_Irecv* and use *MPI_Waitall* to wait for all the send/receive operations to return. The communication thread will be blocked until all the data has been received successfully. Because each send partition is small, the communication among the tasks will be invoked multiple times in a relaxed all-to-all pattern. The non-blocking style communication does not block all the communication threads when the data is needed to move among the tasks. To achieve the communication efficiently, each communication manager starts two threads for transferring data to manage the send queue and receive queue, respectively. Once the data is in the send queue, it will be delivered without waiting for the other tasks participating in the relaxed all-to-all communication.

We compare both communication styles with the typical HiBench AGGREGATE benchmark over a 20GB data set. We record the time of each send operation and plot the time sequences for each O task in one graph. Figure 6 shows that the execution time of O tasks in blocking style is 120 seconds, while that in non-blocking style is 61 seconds. The lines of the blocking communication are cut into several fragments, and the span between two successive fragments reflects the communication waiting overhead. Because of the data skew in the AGGREGATE benchmark, each task operates different sizes of data. When communicating in the blocking style, much of time is cost to wait for the tasks to participate in communication invocations, resulting in the synchronization overhead and worse performance among the working threads in a DataMPI task.

Based on the above observation, we further optimize the non-blocking communication design in DataMPI for Hive workloads. Figure 7 shows the communication design in DataMPI. It has two components, the buffer manager and the shuffle engine. The buffer manager is composed of partitions, each of which has a data structure, including the raw buffer data and the meta-information, such as the size of buffer used, the number of cached key-value pairs, the offsets and indices of each key-value pair in the buffer. The shuffle engine will take one send partition and create a send request by invoking *MPI_Isend*. The request handlers will be cached in the shuffle engine and the engine will test for the completion. The shuffle engine also binds each receive request with one receive partition which has sufficient space to receive data. When one of the receive operations is completed, related data partition will be cached in the buffer manager again.



(a) Blocking Communication (b) Non-blocking Communication

Fig. 6. HiBench AGGREGATE Benchmark with a 20GB Data Set

D. Parallelism Tuning and Optimization

Usually, the data distribution in a table is not uniform, because the number of rows with a key may be much larger or smaller than the number of rows with another key. With the normal partition strategy, such as hash-partition, most of the key-value pairs may be partitioned into several Reducers in MapReduce paradigm, while the other Reducers may take much less data to process. For example, we analyze the workload of TPC-H Q9 with a 40 GB data set to show the data skew in Hive applications. By default, Hive launches 16 A tasks in one of the stages when executing TPC-H Q9. The maximum

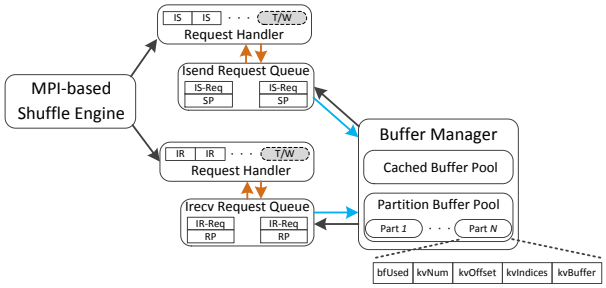
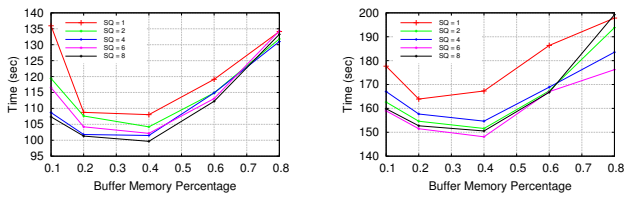


Fig. 7. DataMPI Non-blocking Communication Design for Hive Application

number of records processed by one of the A tasks is 13x more than the minimum number of records in another A task, and the stage costs long time to finish. When the number of A tasks increases to 28, which is the maximum number of executing slots in Hive, the maximum number of records is 4x more than the minimum one, and the execution time is cut down to the 27% of the previous. This means increasing the execution parallelism may alleviate the data skew problem. We provide a parameter *hive.datampi.parallelism* = *"enhanced/default"* in Hive on DataMPI for tuning the parallelism for applications. When using default mode, the number of O tasks is based on the number of input splits and less than the maximum number of executing slots, and the number of A tasks conforms to the Hive task scheduling policy. When using enhanced mode, the number of A tasks is equal to the number of O tasks, and when the job is the last stage in a query, the number of A tasks is 1.

In Hive workloads, much of memory in the application level needs to be used to process the rows. To balance the memory between application and DataMPI library, Hive on DataMPI provides the buffer tuning parameter *hive.datampi.memusedpercent* and *hive.datampi.sendqueue*. We tune the size of send block queue (SQ) and the percentage of cache memory on JOIN and AGGREGATE workloads of HiBench with a 20 GB data set. As shown in Figure 8, when the percentage of cache memory is 0.4, both workloads achieve the best performance. When it is close to 0, the intermediate data is spilled on disk and the performance decreases. When it is close to 1, less memory for the application increases the overhead of Java memory garbage collection and hurts the performance. Besides, the waiting time in DataMPI between calculation and data movement threads is based on the send block queue. When increasing the size of send queue, the waiting time decreases. Results show that when the size of send queue is larger than 6, the performance becomes stable.



(a) AGGREGATE (b) JOIN

Fig. 8. The Tuning of Cache Memory and Send Queue Size with a 20 GB Data Set in Intel HiBench

V. Evaluation

The section presents the evaluation of Hive on DataMPI. We first present the performance benefits of Hive on DataMPI with Intel HiBench and TPC-H workloads. Then, we analyze the evaluation of Hive between Hadoop and DataMPI with the execution breakdown and resource utilization. We also compare the performance of Hive on DataMPI with Text and ORCFile formats. Finally, we show the productivity statistics of Hive on DataMPI with the plug-in-based approach.

A. Experiment Setup

Our testbed is a cluster with 8 nodes which are connected by a Gigabit Ethernet switch. Each node is equipped with 2 Intel Xeon E5620 (2.4 Ghz) CPU processors with disabling hyper-threads. Each E5620 CPU processor has 4 physical cores with private L1, L2 caches and a shared LLC cache. And each node has 4X4 GB DDR3 RAM and one 2TB SATA disk with 7200 RPM.

Each node is configured with the same software environment, CentOS 6.5(Final) with kernel 2.6.32-431.el6.x86-64 is installed. In the following benchmarks, we adopt Intel HiBench 3.0 [15], TPC-H 2.17.0 [12], MVAPICH2-2.0b [18], DataMPI 0.6 [8], Hadoop 1.2.1 and JDK 1.7.0_25. Our proposed design is based on Hive 0.13.1. For the sake of fair, Hadoop and DataMPI are assigned with the same concurrency level, heap size, 1 master node and 7 slave nodes. The number of MapReduce slots in Hadoop and O/A tasks in DataMPI is configured to 4 in each node. The HDFS block size is configured as the default value (64MB). For Hive, parameters *hive.datampi.memusedpercent*, *hive.datampi.sendqueue* and *hive.datampi.parallelism* are set to 0.4, 6 and default, respectively in consideration of trade-off.

TABLE I. HiBench and TPC-H Data Set Size Used in Experiment

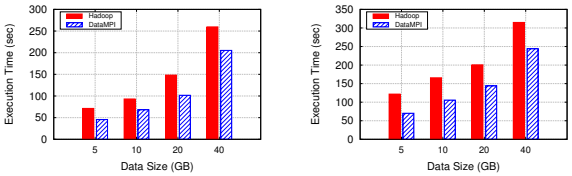
Benchmark	Table/Size	5 GB	10 GB	20 GB	40 GB
HiBench	rankings	234 MB	467 MB	935 MB	1.83 GB
	uservisits	4.4 GB	8.7 GB	17 GB	34 GB
TPC-H	customer		234 MB	469 MB	938 MB
	lineitem		7.3 GB	15 GB	30 GB
	nation		4.0 KB	4.0 KB	4.0 KB
	orders		1.7 GB	3.3 GB	6.6 GB
	partsupp		1.2 GB	2.3 GB	4.6 GB
	part		233 MB	466 MB	932 MB
	region		4.0 KB	4.0 KB	4.0 KB
	supplier		14 MB	28 MB	55 MB

We adopt the data sets generated by the HiBench generator with various sizes of 5 GB, 10 GB, 20 GB, 40 GB and the data sets generated by TPC-H data generator with sizes of 10 GB, 20 GB and 40 GB. The detailed data sizes are listed in Table I. In order to perform a TPC-H benchmark using Hive, the queries are modified to adapt for the HiveQL [19].

B. Performance Benefits on Intel HiBench

We select Intel HiBench's Hive workloads as the micro benchmarks. The data set of HiBench conforms to the Zipfian distribution. By default, the input data uses sequence format. Two workloads for Hive are chosen, including AGGREGATE and JOIN. Figure 9 shows that Hive on DataMPI can averagely

achieve 29% and 31% improvements compared with Hadoop.



(a) AGGREGATE (b) JOIN

Fig. 9. Performance of Intel HiBench

To analyze the benefits of Hive on DataMPI, we break down the execution of a query into three sections, including query compiling, MapReduce jobs execution, and others. Each MapReduce job is decomposed into startup phase, Map-Shuffle (abbr. MS) phase and others. The time of startup is counted from the job being submitted, to MapTask/OTask being invoked. The MS time covers the copy phase in Hadoop and O phase in DataMPI. The time of others is counted by subtracting the time of startup and MS from the total job execution time, which contains the operations of merge, reduce, computation and synchronization, etc. The results over a 20 GB data set are shown in Figure 10.

The common difference is that all of the queries in Hive on DataMPI have nearly 30% shorter startup time in DataMPI than Hadoop, because DataMPI is more light-weight than Hadoop.

AGGREGATE workload has one MapReduce job. It uses the GroupBy operator to make the rows with the same key aggregated in the same group. Figure 10 shows DataMPI has 40% performance improvement on MS time. Because DataMPI can overlap the computation and communication efficiently with multi-threading design. The A phase locality consideration makes data processed expediently.

JOIN workload contains three jobs which need to filter the records from the two different tables within a particular data range and join them with the matching values for corresponding field, calculate and finally output the results. Figure 10 shows the MS time of DataMPI in JOB1 and JOB2 is 20% and 55% less than that of Hadoop, respectively. JOB1 benefits from the light-weight design of DataMPI and less overhead of reading data than Hadoop. While JOB2 is similar to AGGREGATE workload. JOB3 which only has 1 Mapper and 1 Reducer simply sinks the results. The MS time of DataMPI in JOB3 is 70% less than that of Hadoop, because of the light-weight process management.

In summary, using DataMPI in Hive has performance improvements in the following three aspects: 1. The light-weight library design reduces the overhead for process management; 2. DataMPI has an efficient data movement mechanism; 3. DataMPI can benefit from the advantages of efficient MPI communication.

C. Performance Benefits on TPC-H

This section shows the evaluation of Hive on Hadoop and DataMPI with TPC-H workloads. Firstly, we compare

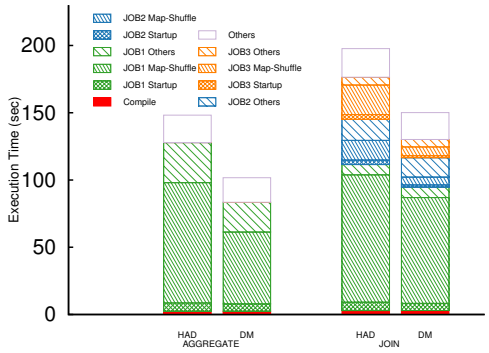


Fig. 10. Performance Breakdown for Intel HiBench with a 20 GB Data Set

the performance of both systems, with the Text format and Optimized Row Columnar File (ORCFile) format. Table II shows the results with a 40 GB data set. We observe the performance with ORCFile format for Hadoop and DataMPI has nearly 22% improvement compared with Text format. This is because ORCFile format uses highly efficient way to store Hive data.

TABLE II. PERFORMANCE OF TWO FILE FORMATS IN TPC-H BENCHMARK WITH 40 GB DATA SETS. (UNIT: SEC)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
HAD-TEXT	208.65	202.07	340.40	350.08	422.68	193.08	684.37	541.92	1,045.68	388.11	160.60
HAD-ORC	91.48	183.72	242.02	274.08	291.55	55.91	439.43	422.5	1343.19	258.92	141.54
DM-TEXT	229.93	142.25	282.47	280.29	400.08	178.91	616.68	403.69	956.46	344.05	104.29
DM-ORC	90	130.46	196.74	174.54	222.78	44.16	423.09	359.19	1365.70	204.23	85.41
	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
HAD-TEXT	252.58	196.33	201.67	276.05	224.50	470.32	633.45	352.53	353.49	1,042.18	228.11
HAD-ORC	149.62	276.11	81.04	106.62	268.46	323.7	505.63	302.68	204.49	728.59	274.41
DM-TEXT	229.09	137.57	178.52	203.31	196.16	432.55	549.09	252.62	307.89	897.08	159.35
DM-ORC	94.6	195.34	59.7	73.63	241.18	319.92	465.69	304.54	152.72	660.78	192.04

Furthermore, we break down the execution time of Hadoop and DataMPI with a 40 GB data set in ORCFile format with two different parallelism strategies, namely *enhanced* and *default* as introduced in Section IV-D. For fair comparison, we add the enhanced strategy of tasks schedule in Hive on Hadoop as the same as that used in Hive on DataMPI. Figure 11 shows the results. The tags of *h* and *H* represent tests of Hadoop with default parallelism and enhanced parallelism, respectively, and the tags of *d* and *D* represent tests of DataMPI with default parallelism and enhanced parallelism, respectively.

The number of jobs in each query of DataMPI and Hadoop is equal because of the same mechanism to generate physical plan. Each bar in the figures is broken down into several jobs. Each job is decomposed into Map/O and Reduce/A phases. The length of the bar represents the total execution time of the corresponding query.

The results show that Hadoop and DataMPI perform averagely 14% and 23% improvements when using the enhanced parallelism strategy compared with the default parallelism strategy. When running Q9, the improvement of Hadoop with enhanced strategy is 42% compared with default strategy, while that of DataMPI is 56%. This is because when increasing the parallelism degree, the data is partitioned more uniformly, and the computation is paralleled more efficiently among tasks. Besides, some queries do not benefit from the parallelism strategies, such as Q1, Q6, Q11 and Q14. Because both strategies get the similar execution parallelism.

Considering the enhanced strategy, DataMPI achieves 29% performance improvement compared with Hadoop on average. 12 queries have more than 30% performance improvement, and the minimum improvement (9%) comes from Q1 which is a Map-only job to find the particular records. The benefits of the queries come from the efficient communication design of DataMPI, which have been pointed out before.

Adopting the enhanced strategy, we conduct experiments with data sets varied from 10 GB to 40 GB to demonstrate the scalability of DataMPI.

Figure 12 shows the execution time of 22 queries on Hadoop and DataMPI with Text format and ORCFile format over different data sizes. From the results, we observe that the execution time has the similar growth trend in Hadoop and DataMPI. The best case occurs in Q12 with a 20 GB ORCFile data set with 53% improvement.

The performance improvements of DataMPI over Hadoop for 22 queries are averagely 20% and 32% with Text format and ORCFile format, respectively. This implies that DataMPI has the similar scalability to Hadoop when running TPC-H workloads. And when ORCFile format optimizes the table storage, DataMPI can gain a 12% average improvement in data movements.

D. Resource Utilization Analysis

This section shows the resource utilization of the TPC-H Q9 between Hadoop and DataMPI with enhanced parallelism strategy over a 40 GB data set. Because of the limited space, we do not show the others. The **dstat** [20] tool is used to collect the sampling data. The execution time of TPC-H Q9 with Hadoop and that of DataMPI is 802 seconds and 598 seconds, respectively.

Figure 13(a) shows the CPU utilization and the I/O-wait CPU usage which indicates the overhead of CPU spent on waiting for I/O operations. Results show that DataMPI has slightly higher CPU utilization than Hadoop. However, the execution time is 25% less than that of Hadoop.

As shown in Figure 13(b), the average bandwidths of disk writing for Hadoop and DataMPI are 24 MB/sec and 25 MB/sec, while the peak bandwidths of that are 123 MB/sec and 124 MB/sec. Both of Hadoop and DataMPI's average and peak bandwidths of disk reading reach nearly 3 MB/sec and 28 MB/sec, respectively. Since DataMPI caches a certain amount of intermediate data in memory, it costs less time to perform I/O operations than Hadoop.

Figure 13(c) shows that both of Hadoop and DataMPI will achieve the maximum memory footprint. However, DataMPI can achieve the upper memory footprint faster than Hadoop, which means it can utilize the memory resource more efficiently.

Figure 13(d) shows the bandwidth of network utilization. The average achieved bandwidths of Hadoop and DataMPI are 20 MB/sec and 30 MB/sec, respectively. The peak bandwidths of them are 79 MB/sec and 80 MB/sec. This indicates that DataMPI can perform more efficiently to transfer data than Hadoop. This benefits from the non-blocking design with MPI in the shuffle engine and lightly optimized pipeline.

Overall, the benefits come from the efficient buffer management and data movement design in DataMPI to overlap the computation, communication and I/O efficiently.

E. Productivity Analysis

Another advantage shown in transplanting DataMPI to Hive's execution engine is that DataMPI uses fairly small amounts of codes. As an MPI extended data-computing library, DataMPI can fully support the workloads for Hive with minimal changes. After the current design work, we make a statistical analysis on the related code lines to support the execution engine with Hadoop and DataMPI. The result is shown in Table III. In the compiler part, we mainly concern the components of MapReduceCompiler and DataMPICompiler. The number of main different code lines in namespaces is only 0.01K. In the execution engine part, we count the major components (e.g Task, Work, Mapper, Reducer) listed in the execution engine package. Except for inheriting nearly 1.1K lines and refactoring 2.6K lines from Hive's MapReduce engine directly, we mainly change about 0.2K code lines. In the component of others, we add a few DataMPI attributions and configurations in Hive's namespaces with nearly 0.1K lines. Therefore, the shared code lines of entire Hive framework are not listed in the table.

In summary, the main changed code lines are about 0.3K totally. Hive on DataMPI's high productivity is contributed by a light-weight design of DataMPI, minimal impacts on Hive and efficient support for HiveQL queries.

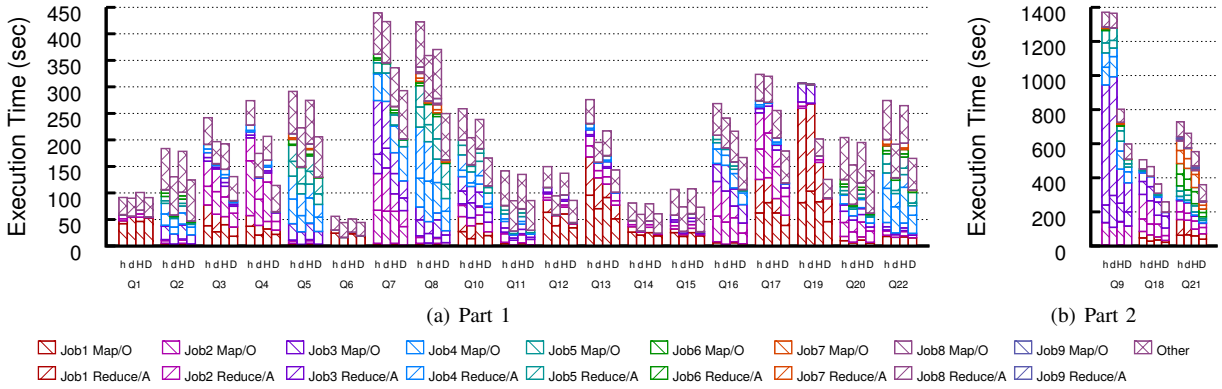
TABLE III. THE MAIN CHANGES FOR HIVE

	Code Lines for Hadoop	Code Lines for DataMPI	Main Changes
Compiler	0.3K	0.3K	0.01K
Execution Engine	3.9K	2.8K	0.2K
Others	—	—	0.1K

VI. RELATED WORK

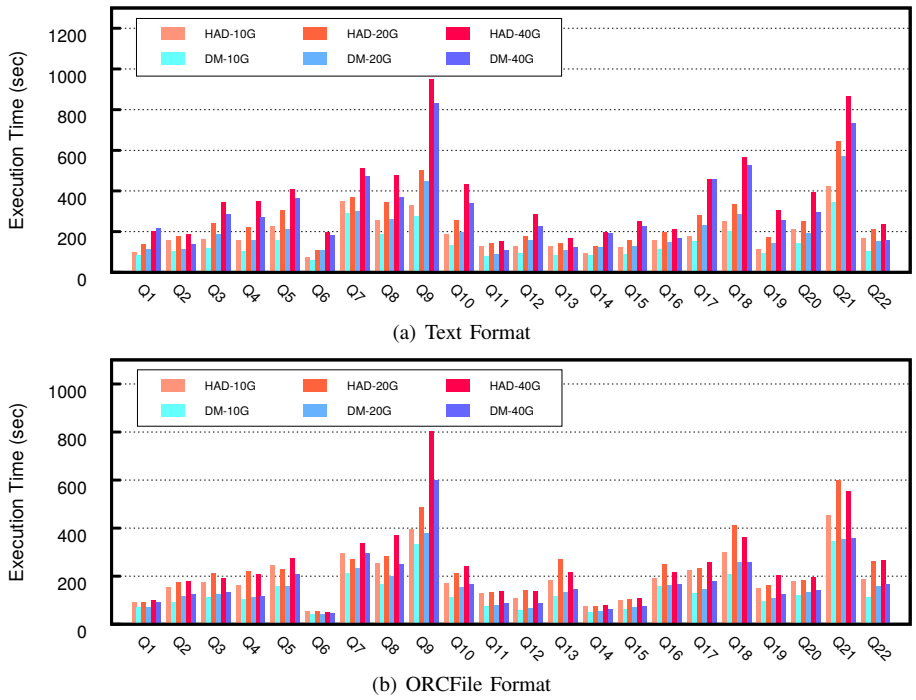
Big Data Computing with HPC technologies: Heofler *et al.* [21] attempted to anticipate MPI to write MapReduce-like applications. Plimpton *et al.* [22] proposed MR-MPI to support graph algorithms with MapReduce operations based on an MPI context. Recent work [10] showed the potential to adapt RDMA [11] (Remote Direct Memory Access) for Spark applications. Matsuda *et al.* [23] attempted to support TPC-H with MPI and rewrote the workloads with MapReduce-like programs. They stored the data in memory without external storage support. Different from their work, this paper proposes a design to fully support data warehouse system based on DataMPI with small modifications on Hive, and provides fine performance and scalability.

Other techniques to support Big Data in data warehouse systems: Authors in [24] pointed out the shared nothing architecture with partitioning the data in the horizontal or vertical way, and compression-aware databases to support traditional data warehouse systems in high-performance computers. Data Cube [25] used precomputed aggregation calculations to provide efficient query processing in high-performance computers to accelerate the On-Line Analytical Processing (OLAP). YSmart [26] used a set of rules to



(a) Part 1 (b) Part 2

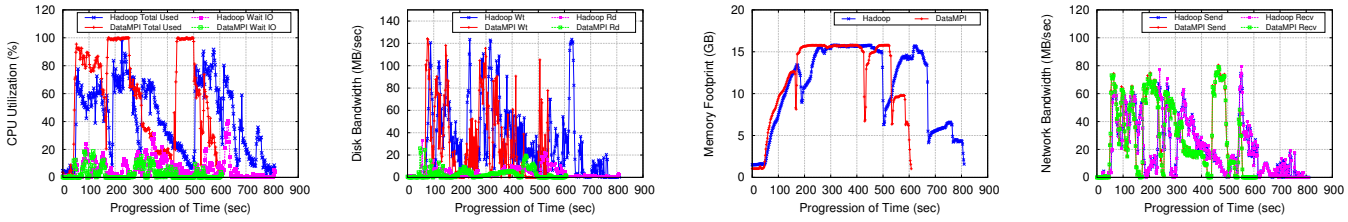
Fig. 11. Performance Breakdown for TPC-H with a 40 GB Data Set in ORCFile Format



(a) Text Format

(b) ORCFile Format

Fig. 12. Performance Benefits of Hive on DataMPI for TPC-H in Two Different Table Formats (Text vs. ORCFile)



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

Fig. 13. Resource Utilization of TPC-H Q9 with a 40 GB Data Set

minimize the number of MapReduce jobs in Hive's compiling and speed up execution time. Our work focuses on improving the communication efficiency in the data warehouse systems with the high-performance techniques.

VII. Conclusions and Future Work

In this paper, we have analyzed the performance improvement potential of a data warehouse system Apache Hive brought by an MPI extended data communication library – DataMPI. We have proposed a design of Hive on DataMPI with full compatibility with HiveQL. Based on our study, we make the following conclusions from the view points of

functionality, productivity, and performance:

- **Hive on DataMPI can fully support and accelerate data warehouse workloads.** Our design has supported all Hive workloads including 2 micro queries of Intel HiBench and 22 business oriented queries of TPC-H. The efficient implementation can produce correct results and improve the performance.
- **DataMPI's bipartite communication model and key-value communication interfaces can enable Hive on DataMPI productively.** We only need to change about 0.3K lines of core codes in Hive to implement the required functions with DataMPI.
- **DataMPI can benefit Hive's execution performance.** Through Hive on DataMPI with enhanced non-blocking communication and parallel mechanisms, our design has achieved 30% better average performance in Intel HiBench and can gain 32% performance improvement than Hive on Hadoop averaged over all the queries in TPC-H. The best case can achieve 53% performance improvement in TPC-H Q12 query with a 20 GB ORCFile data set.

We have shown early experiences of accelerating data warehouse system with MPI. Hive on DataMPI is a pioneering work to connect the HPC technologies with data warehouse systems. We plan to extend our future research in three aspects: 1. explore more bottlenecks in Hive on DataMPI, and optimize the performance; 2. evaluate Hive on DataMPI on different high-performance clusters with larger data set sizes; 3. reduce the overhead of intermediate files storing by supporting DAG (Directed Acyclic Graph) distributed computing models.

ACKNOWLEDGMENT

This work is supported in part by the Hi-Tech Research and Development (863) Program of China (Grant No. 2013AA01A209, 2013AA01A213), the Strategic Priority Program of Chinese Academy of Sciences (Grant No. XDA06010401) and the Guangdong Talents Program (Grant No. 201001D0104726115).

REFERENCES

[1] "Apache Hive," https://hive.apache.org.

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

[3] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive - A Petabyte Scale Data Warehouse Using Hadoop," in *Proceedings of the 2010 International Conference on Data Engineering*, 2010, pp. 996–1005.

[4] "MPI: A Message-Passing Interface Standard Version 3.0," http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[5] I. T. Association, *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[6] X. Lu, B. Wang, L. Zha, and Z. Xu, "Can MPI Benefit Hadoop and MapReduce Applications?" in *Proceedings of the 2011 International Conference on Parallel Processing Workshops*, 2011, pp. 371–379.

[7] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "DataMPI : Extending MPI to Hadoop-like Big Data Computing," in *Proceedings of the 2014 International Parallel and Distributed Processing Symposium*, 2014, pp. 829–838.

[8] "DataMPI: Extending MPI for Big Data with Key-Value based Communication," http://datampi.org/.

[9] F. Liang, C. Feng, X. Lu, and Z. Xu, "Performance Characterization of Hadoop and Data MPI Based on Amdahl's Second Law," in *Proceedings of the 2014 International Conference on Networking, Architecture, and Storage*, 2014, pp. 207–215.

[10] X. Lu, M. Wasi-ur Rahman, N. Islam, D. Shankar, and D. K. Panda, "Accelerating Spark with RDMA for Big Data Processing: Early Experiences," in *Proceedings of the 2014 Annual Symposium on High-Performance Interconnects*, 2014, pp. 9–16.

[11] "High-Performance Big Data (HiBD)," http://hibd.cse.ohio-state.edu/.

[12] "The TPC Benchmark H," http://www.tpc.org/tpch/.

[13] Y. Huai, S. Ma, R. Lee, O. O'Malley, and X. Zhang, "Understanding Insights into the Basic Structure and Essential Issues of Table Placement Methods in Clusters," *Proceedings of Very Large Data Bases Endowment*, vol. 6, no. 14, pp. 1750–1761, 2013.

[14] F. Liang and X. Lu, "Accelerating Iterative Big Data Computing Through MPI," *Journal of Computer Science and Technology*, vol. 30, no. 2, pp. 283–294, 2015.

[15] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench Benchmark Suite: Characterization of the MapReduce-based Data Analysis," in *Proceedings of the 2010 International Conference on Data Engineering Workshops*, 2010, pp. 41–51.

[16] F. Liang, C. Feng, X. Lu, and Z. Xu, "Performance Benefits of DataMPI: A Case Study with BigDataBench," in *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, ser. Lecture Notes in Computer Science, J. Zhan, R. Han, and C. Weng, Eds. Springer International Publishing, 2014, pp. 111–123.

[17] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A Warehousing Solution over a Map-Reduce Framework," *Proceedings of Very Large Data Bases Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009.

[18] "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE," http://mvapich.cse.ohio-state.edu.

[19] "JIRA's HIVE-600:Running TPC-H queries on Hive," https://issues.apache.org/jira/browse/HIVE-600.

[20] B. T. Johnson, *"DSTAT: Software for the Meta-analytic Review of Research Literatures"*. Erlbaum, 1989.

[21] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards Efficient MapReduce Using MPI," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2009, pp. 240–249.

[22] 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.

[23] M. Matsuda, S. Takizawa, and N. Maruyama, "Evaluation of Asynchronous MPI Communication in Map-Reduce System on the K Computer," in *Proceedings of the 2014 European MPI Users' Group Meeting*, 2014, pp. 163:163–163:168.

[24] D. J. DeWitt, S. Madden, and M. Stonebraker, "How to Build a High-Performance Data Warehouse," http://db.lcs.mit.edu/madden/high-perf.pdf.

[25] S. Goil and A. Choudhary, "High Performance OLAP and Data Mining on Parallel Computers," *Data Mining and Knowledge Discovery*, vol. 1, no. 4, pp. 391–417, 1997.

[26] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang, "YSmart: Yet Another SQL-to-MapReduce Translator," in *Proceedings of the 2011 International Conference on Distributed Computing Systems*, 2011, pp. 25–36.