Vol.2015-HPC-150 No.10
2015/8/4

# Improving Hadoop MapReduce Performance on the FX10 supercomputer with JVM Reuse

Thanh-Chung Dao[1,a) Shigeru Chiba[1,b)

**Abstract:**
Hadoop is known as the most popular open-source MapReduce implementation that is used widely in practice to process large datasets. In the aspects of productivity and maturity, it is a good choice to run MapReduce on supercomputers that are urgently dealing with data-intensive problems. Our examination reveals that in iterative MapReduce computation, such as PageRank and K-means clustering, the start-up and initializing time is relatively long in comparison with overall execution time. Furthermore, on the FX10 supercomputer, MPI is the de facto communication and required for high-speed communication, but MPI is not available over Hadoop processes due to the FX10's specification. In this work, we propose JVM Reuse and its implementation, a process pool, for the sake of Hadoop avoiding the overhead of JVM start-up time and enabling MPI communication. We also present a MPI-based shuffling engine as a proof to show MPI advantage. Our performance evaluation demonstrates that JVM Reuse helps reduce job completion time up to 35%.

## 1. Introduction

A supercomputer is a very expensive cluster consisting of identical computers equipped with multi-core processors and large capacity of main memory and connected to each other through high-speed networks. It is typically employed for scientific computation. So far, supercomputing has focused mainly on compute-intensive applications, but many data-intensive workloads, for example, graph processing and pre-processing of simulation data, are emerging as supercomputing problems that supercomputers need to tackle for the sake of leveraging their high-performance hardware. MapReduce [1] is a programming paradigm used widely to process large-scale datasets and it comes as a relevant solution to deal with data-intensive problems on supercomputers. It provides an easy-to-use and scalable computation model into which naturally many scientific algorithms can fit. MapReduce implementation often makes parallelization and communication (file I/O and data exchange) hidden from users that helps run applications on hundreds and thousands of nodes more easily.

Hadoop [2] is known as the most popular open-source MapReduce implementation that is used widely in practice and by leading companies including Yahoo! and Facebook. Hadoop is a better choice to run MapReduce on supercomputers in the aspects of productivity and maturity rather than developing a new MapReduce framework from scratch that is time-consuming.

Although Hadoop is a good adoption, but it does not support directly iterative computation [3][4] that is used by a broad class of scientific algorithms including graph ranking (PageRank [5]), clustering (K-Means), and machine learning training. In order to deploy iterative MapReduce computation on Hadoop, each iteration is considered as a small MapReduce job and run consecutively in a loop. A job often consists of lots of MapTask and ReduceTask processes that are newly started every iteration. It leads to a problem that many processes are created during the iterative program running. Hadoop MapReduce is written in Java and its process (the JVM) is considered to have high start-up cost. Supercomputers are naturally expensive, and thus it is very costly to run an iterative job whose duration can be several hours or a day. Long start-up time of processes is waste of resources.

Contemporary supercomputers have been equipped with high-speed network infrastructure, such as 3D-Torus [6] or Tofu [7] interconnection and InfiniBand, on which MPI is optimized and considered as the de facto communication. Unfortunately, Hadoop is designed to rely mainly on TCP/IP-based communication that prevents it from taking advantage of high bandwidth and low latency of supercomputer networks. On our supercomputer at the University of Tokyo (FX10) [8], the throughput difference between MPI and TCP is tenfold. However, MPI is not available among Hadoop processes on the FX10 since Hadoop requires dynamic process creation to start new MapTask and ReduceTask processes, but dynamically created processes on FX10 cannot use MPI. That is the specification existing on the FX10. Note that MPI-Spawn command [9] can be used, but it is only possible to spawn a new process on a totally new node on which there is no process running.

In order to address the problem of long start-up time and the need of using MPI over Hadoop processes, we propose an idea of reusing, called *JVM Reuse*, for the sake of Hadoop avoiding the overhead of JVM start-up and enabling MPI communication.

[1] Dept. of Creative Informatics, The University of Tokyo
a) chung@csg.ci.i.u-tokyo.ac.jp
b) chiba@acm.org

© 2015 Information Processing Society of Japan

To avoid starting and terminating processes, JVM Reuse keeps a process running and when it is free, a new program's instruction is loaded on top of its. Also, since the process is kept running without terminating, Hadoop does not need dynamic process creation; instead, Map or Reduce tasks can be created and executed on top of that process. MPI is of course always available among processes started at the beginning when their job is submitted. JVM Reuse can be integrated with any Hadoop version 2.x and resource managers, such as Mesos [10] and YARN [11], since its implementation, our proposed process pool, provides a mechanism to allocate and de-allocate process slots that can work independently. Compared to other JVM Reuse implementation, our design considers optimization including *Reflection* and *Clean-up*.

Overall, we have made four contributions as follows:

- We have revealed iterative MapReduce computation on Hadoop has long start-up time.
- We have designed and implemented JVM Reuse to avoid long start-up time and enable MPI over Hadoop processes on the FX10 supercomputer.
- We have carried out evaluation of JVM Reuse on PageRank computation.
- We have designed a MPI shuffling engine to show MPI advantage on Hadoop MapReduce workloads.

In the following, firstly we present our motivation and describe problems in Section 2. We then discuss the idea of reusing and its implementation in Section 3. Section 4 is dedicated to providing experimental results including JVM Reuse and MPI shuffling. Finally, Section 5 provides a review of related work and is followed by brief discussion and conclusion in Section 6.

## 2. Motivation

In this section, we present why Hadoop is a good choice to run MapReduce on supercomputers and then describe in details problems associated when running Hadoop MapReduce on the FX10 supercomputer.

### 2.1 Hadoop MapReduce on supercomputers

MapReduce [1] is a programming model used widely to process large datasets. It is a paradigm that makes parallelization easier and communication among processing tasks hidden to programmers to develop applications running on hundreds to thousands of nodes. MapReduce computation is split into two main phases: Mapping and Reducing. Firstly, the Mappers read and filter input data, and then write key-value pairs sorted as outputs. The output pairs of Mappers are passed to appropriate Reducers (known as shuffle phase). Finally, the Reducers sort and merge these values and apply a reducing function to obtain the final results:

- Map defines how to split data into a couple of *(key, value)*:
  *input data → list(key, value)*
- Reduce defines what results will be obtained:
  *(key, list(value)) → desired results*

Hadoop [12] is the standard of MapReduce implementation that provides easy-to-use MapReduce APIs being used widely in practice and by a large developer community. It is written in Java and TCP is its main communication protocol. Hadoop is a better choice to run MapReduce on supercomputers in the aspects of productivity and maturity rather than developing a MapReduce framework from scratch that is time-consuming. Users can reuse MapReduce source code to run on their commodity clusters or vice versa. In aspect of performance tuning, there are a lot of parameters that users can optimize, for example, cpu cores and memory of each process, spilling thresholds, and number of shuffling threads. Wu [13] provided a self-tuning model to find the best parameters on a certain cluster. Moreover, there is an ecosystem of related projects that supports Hadoop in the aspects of distributed programming [14], scheduling [15], and benchmarking [16].

### 2.2 Long start-up time of processes in iterative jobs

Hadoop provides an easy-to-use framework that is good and fit for most of MapReduce algorithms, but it does not support directly iterative computation. There are several approaches to improve iterative job performance, for example HaLoop [3], Spark [17] and Twister [18], which we discuss in Section 5.

There is a broad class of scientific algorithms that can be solved using iterative MapReduce computation, including graph ranking, clustering, and machine learning training. An iterative job is run consecutively until it meets a condition or its value converges. Figure 1 describes how an iterative job works. The results of previous MapReduce iteration is used as inputs to the next one. PageRank, a graph algorithm to rank linked documents, is a common example of iteration computation. At the initial iteration, each document has an equal rank value, $rank_D = 1$. On each next iteration at the mapping step, each document emits its rank contribution ($\frac{r}{n}$) to each outbound link, where $r$ is its current rank and $n$ is the number of out-links. At the reducing step, the new rank of a document is sum of all rank contribution that it received, $new\ rank_D = \sum c_i$ where $c_i$ is the rank contribution from an inbound link $i$. The set of new rank values is kept for next iteration computation.

In order to deploy an iterative MapReduce program on Hadoop, an iteration is considered as a small MapReduce job and run consecutively in a loop manually. A MapReduce job often consists of lots of mapping processes (Mapper) and several reducing processes (Reducer) that are newly started on distributed nodes for each job. It comes with a problem that too many processes are created during the iterative program running. Hadoop MapReduce is written in Java and its process (the JVM) is considered to have high start-up cost. Supercomputers are naturally expensive, and thus it is very costly to run iterative jobs whose duration can be several hours or a day. Long start-up time of processes is waste of resources. To examine the problem, we run PageRank program in *three* iterations and its execution is illus-

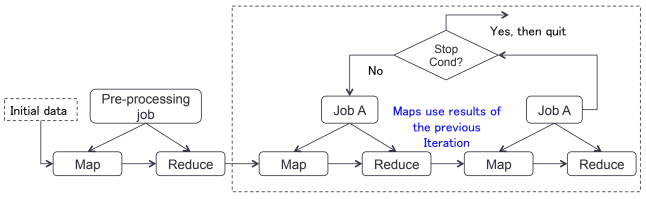


**Fig. 1** *Iterative job workflow*

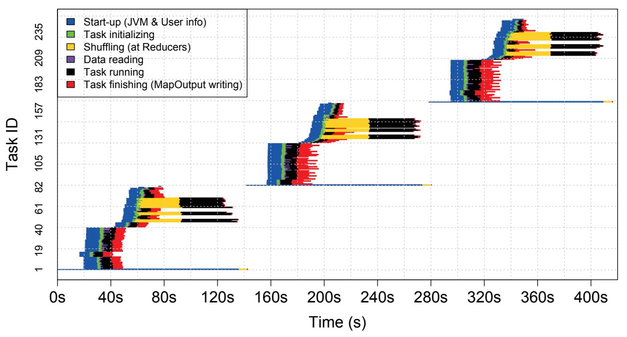


**Fig. 2** *PageRank with three iterations: the blue part denotes the start-up time of a process.*

trated in Figure 2 that reveals that the start-up time measured from requesting process creation to starting computation is relatively long in comparison with overall execution time (up to 30%).

The JVM stands for Java Virtual Machine and its instance works the same as a process that executes a computer program compiled in Java bytecode. When users run a command, for example *"java A"*, the process is initialized at operating system level first. Then, a class loader subsystem including class loading and linking (verification & static field initializing) is called to invoke *main()* method of the *A* program. Finally, execution engine of the JVM often optimized by a JIT compiler executes *A* instructions. Figure 3 shows the working flow.

## 2.3 MPI communication on the FX10 supercomputer

MPI is the de facto communication on our supercomputer (FX10) and required for high-speed communication. However, MPI is not available over Hadoop processes on the FX10 supercomputer. The reason comes from Hadoop architecture that requires dynamic process creation to start MapTask and ReduceTask processes, but dynamically created processes on FX10 cannot use MPI. That is the specification existing on the FX10 that makes only slow TCP/IP available among newly created processes.

The FX10 [8] is a supercomputer at the University of Tokyo and developed by Fujitsu. Its compute nodes consist of 50 racks of PRIMEHPC FX10 with a peak of 1.13 PFLOPS. One rack contains 96 computing nodes equipped with SPARC64 IXfx processors [19]. K supercomputer [20] at RIKEN is also using the PRIMEHPC FX10 rack, but its processor is SPARC64 VIIIfx, a predecessor of IXfx.

MPI stands for Message Passing Interface that is used widely on supercomputer environment rather than TCP/IP. Figure 4 shows the throughput between MPI and TCP on the FX10 whose difference is tenfold. Figure 2 also shows bottlenecks at the shuffling phase and writing MapOutput in which:

- Long shuffling (fetching data) that can be improved by being replaced with MPI data exchange.
- Slow MapOutput writing to disks that can be speeded up using parallel MPI I/O.

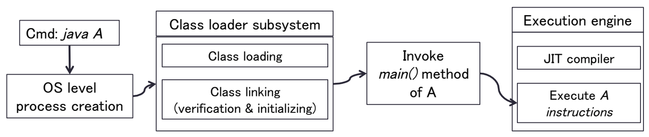


**Fig. 3** *JVM start-up flow of a program A*

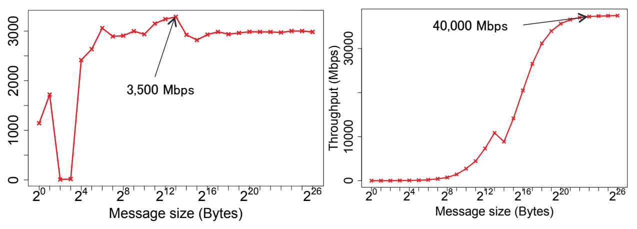


**Fig. 4** *TCP Throughput (left) and MPI Throughput (right) on the FX10 supercomputer*

## 3. JVM Reuse

The above-mentioned iterative jobs often consist of many short running processes whose start-up time is relatively long. To avoid starting and terminating processes, the JVM keeps a process running and when it is free, a new program's instruction is loaded on top of its. In this section, we firstly describe the idea of reusing and show how JVM Reuse addresses the discussed problems. Section 3.2 illustrates implementation's features in details.

### 3.1 Idea of Reusing

In the Hadoop version 2.x [11], YARN is used for resource managing and task scheduling that consists of *Resource* and *Node Manager* running on master and slave nodes, respectively. A Hadoop cluster often has one master node and many slave nodes. When *Node Manager* receives a MapTask or ReduceTask request, it creates a totally new JVM process to run that task by using the process builder of Java. The creation flow is described in Figure 5 (left): firstly, a process builder is used to call a shell script created in advance; after exporting environment variables in the shell script, the Java program that contains a MapTask or ReduceTask class is executed. When computation finishes, the MapTask and ReduceTask processes (Mapper and Reducer) are terminated.

Instead of creating newly a process and terminating it after fin-

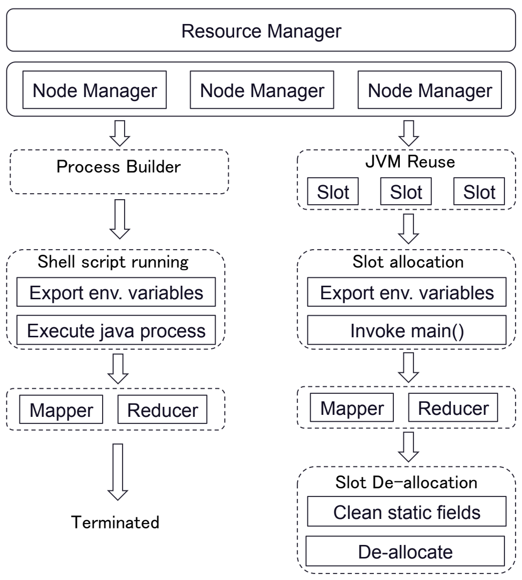


**Fig. 5** *Process builder vs. JVM Reuse workflow*

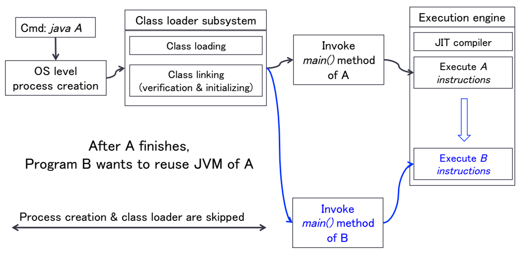

Cmd: *java A*
OS level process creation
Class loader subsystem
Class loading
Class linking (verification & initializing)
Invoke *main()* method of A
Execution engine
JIT compiler
Execute *A* instructions
Execute *B* instructions
After A finishes, Program B wants to reuse JVM of A
Process creation & class loader are skipped
Invoke *main()* method of B

**Fig. 6** *JVM Reuse of a program B*

ishing, we create a pool of empty JVM processes in advance and do allocation and de-allocation, respectively. Figure 5 (right) illustrates how JVM Reuse is used in Hadoop MapReduce workflow. *Node Manager* sends a request containing the shell script path to the pool of JVM processes and a slot in the pool is delegated to run the shell script file and invoke the Java program of Mapper or Reducer. When it completes, the slot is cleaned and given back to the pool. By virtue of reusing the JVM, not only start-up time but also MPI communication can be achieved.

#### 3.1.1 JVM Reuse shortens start-up time

As discussed in the previous section, the JVM start-up flow is divided into four main steps: OS-level creation, class loading, main() invoking, and instruction execution. JVM Reuse skips the first two step since the process does not need to be created newly and related classes are already loaded and linked together. Figure 6 illustrates our description. When *A* program finishes, its JVM process will be kept running to invoke *main()* method of another program called *B*.

If *A* and *B* instructions are identical, JVM Reuse might take advantage of compilation technology, such as Just-in-time (JIT) compilation and adaptive optimization that are designed to improve execution performance. In this paper, however, we do not evaluate effectiveness of JIT compilation on Hadoop MapReduce performance.

#### 3.1.2 JVM Reuse enables MPI communication on the FX10

MPI communication is established among processes started at the beginning when their job is submitted. JVM Reuse keeps those processes running without terminating, so MPI connection is always available during job execution time. When a new process is spawned on a node, MPI connection is not created between the new process and the running processes that is discussed in the previous section.

To avoid the requirement of dynamic process creation, the Mapper and Reducer JVM processes should be started in advance, but we cannot create too many processes and the number of required processes is unknown. Our JVM Reuse creates a limited number of JVM processes in advance, keeps it, and provides a mechanism to invoke a new program on top of the existing JVM processes.

### 3.2 Process pool

To implement JVM Reuse, we create a process pool running on each node at the beginning when Hadoop cluster is started. The pool is initialized with empty JVM processes. Figure 7 shows the architecture of the pool inside one node. *Pool Manager* is an independent process and responsible for slot allocation and deal-

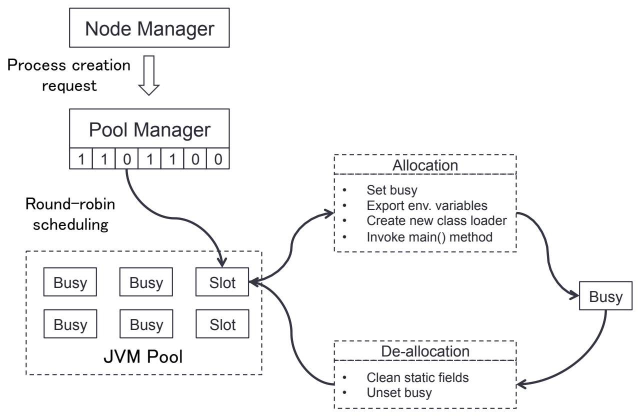


**Fig. 7** *Pool architecture in a node*

location in the pool. Process creation requests in *Node Manager* are forwarded to *Pool Manager (Mgr)* instead of calling a process builder. *Pool Mgr* uses a flag array to store which JVM process slot in the pool is busy and free and chooses an empty slot based on round-robin scheduling. In the Fig. 7, *one* denotes the occupied state and *zero* represents the empty state. MPI is used as inter-process communication.

When a process slot is allocated, its flag is set *one* and then environment variable exporting, creation of a new class loader, and *main()* method invoking are executed consecutively. When the Mapper or Reducer occupied on the slot is finished, de-allocation including static field clean-up and busy-flag reseting is called.

### 3.3 Technical issues

To make allocation and de-allocation possible, we employ *Reflection* and run *clean-up* before de-allocating.

#### 3.3.1 Reflection

Reflection is used to solve the problem of class loading that happens when a user submits a MapReduce job to the Hadoop cluster. In the original flow of process creation using a process builder, at the step of exporting environment variables, *CLASSPATH* that contains the user's MapReduce classes is declared, so Mapper or Reducer process can find and load such classes. By contrast, JVM processes in the pool are started before the above *CLASSPATH* is exported and thus the user's defined classes are not found during execution time.

Refection is a technique that allows to examine and modify a program at runtime. Java reflection makes it possible to load and invoke classes and methods at runtime whose name is unknown at compile time. At the step of slot allocation in JVM Reuse workflow, a new class loader is created to load the user's MapReduce classes before invoking its *main()* method. Since a new class loader is used to load the user's MapReduce classes, although the user submits the same class and package that exists in the previous job, an error will not happen.

An iterative application consists of many small MapReduce jobs and the class loader is created newly for each job (changing in job context). Note that we do not reload all Hadoop classes and only the user's classes is reloaded.

#### 3.3.2 Clean-up

The JVM Reuse may have a problem of security due to reusing the JVM in which static fields for the previous job are already set. For instance, a UserGroup static field is initialized when a user submits a job, and it is kept unchanged whenever its value is not *null*. Therefore, the later users will use the same key as the first one who set it. Although the Hadoop cluster is used by only one user on supercomputer environment, but clean-up of all static fields is necessary in general. At the current design, we only do a simple clean-up by reseting only static fields containing user information and job configuration.

#### 3.3.3 Number of slots in the pool

The pool containing many slots may affect node performance since empty JVM processes consume too much CPU time and memory. On the contrary, the pool with a few slots causes inefficient resource utilization. We set number of slots equal to maximum containers defined in the Hadoop Yarn configuration file by default. If that parameter is not declared, we set it equal to number of processor cores.

## 4. Experimental Evaluation

All our experiments are conducted on the FX10 supercomputer. A FX10 node is equipped with SPARC64 IXfx 1.848 GHz processor (16 cores) and 32GB main memory. The FX10 does not have a local disk for each computing node, conversely a central storage used. Computing nodes are connected with each other through Tofu interconnection [21] on which MPI's maximum throughput is 5 GB/s, and they access the central storage through InfiniBand network.

JVM Reuse can be integrated with any Hadoop version 2.*x*, but we use Hadoop v2.2.0 (a stable version) in our evaluation. In order to adapt to JVM Reuse, Hadoop source code is changed with the below ratio:

- Line of code / total of Hadoop: 1100 / 1,851,473
- Number of classes / total of Hadoop: 9 / 35142

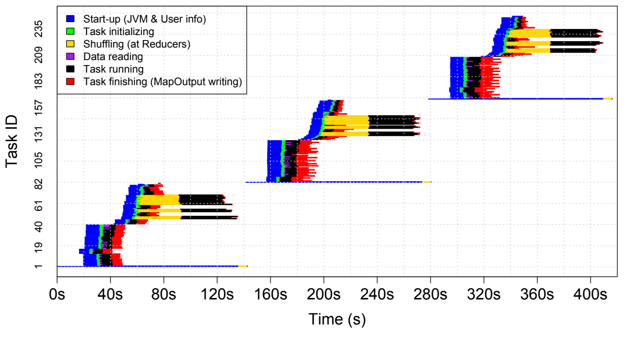
In all our experiments, one node (master) is dedicated to run both Resource Manager and NameNode of Hadoop Distributed File System (HDFS). On other nodes (slaves), we also run both Node Manager and DataNode of HDFS. The maximum MapTasks and ReduceTasks can be run simultaneously on a node is *six*. Since the local disk on each node does not exist, HDFS is run on the central storage. Although it is possible to make Hadoop access directly the central storage (Lustre-based file system), but it is not our evaluation target.

We use OpenJDK 7 and OpenMPI 1.6 optimized by Fujitsu. For the sake of using MPI from Java, we use a MPI binding [22] that has been included in OpenMPI 1.7.5 or later. We made a minor modification to integrate this MPI binding with OpenMPI on the FX10. All experiments are run with the MCA parameter *plm_ple_cpu_affinity = 0* to disable CPU binding on each MPI process.

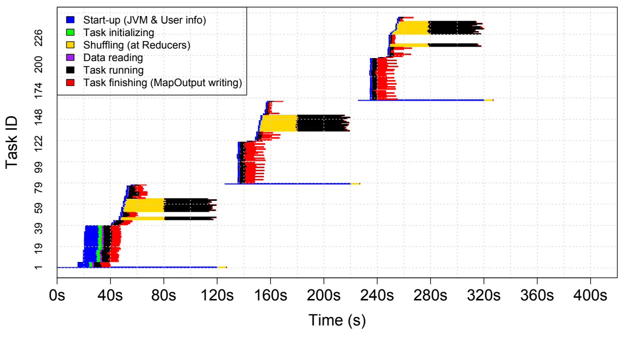
To evaluate JVM Reuse performance, we compare start-up time of our JVM Reuse and the original Hadoop. In order to show that MPI is fast on Hadoop, we design a MPI shuffle engine and show shuffling speed between MPI- and TCP/IP-based communication.

### 4.1 Start-up time

We run PageRank algorithm on 8 FX10 nodes (1 master and 7



(a) Original Hadoop



(b) JVM Reuse-based Hadoop

**Fig. 8** *JVM Reuse shortens start-up time (PageRank with three iterations)*

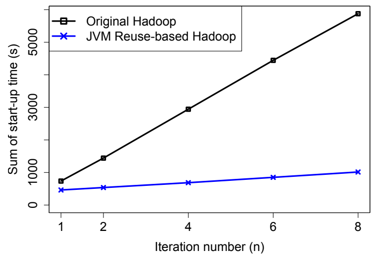
slaves) to evaluate start-up reduction performance and a dataset of 400 GB en-wiki is employed. Wiki data is parsed to build an adjacency list first and then its iteration is executed. A 6-slot pool is configured to run on each slave node and the maximum heap memory is 4GB (-Xmx4096m) for each JVM process. HDFS file block size is 128MB.

#### 4.1.1 PageRank performance

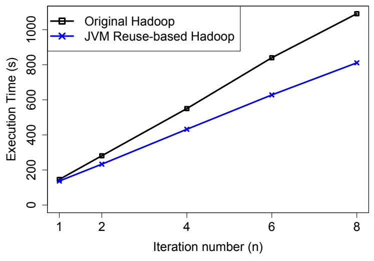
Figure 8 illustrates PageRank execution time in three iterations. The blue part denotes the start-up time of Mapper and Reducer including the JVM start-up itself and initialization of user information. In this experiment, we do not reset static fields. The longer running time represents Reducer processes. Each iteration is divided into two batch of running since the maximum slot is 42. At JVM Reuse chart (bottom), the start-up time starts decreasing from the second batch and is reduced much from the second iteration. After three iteration, total execution time is shortened 25%.

With more iterations, overall execution and start-up time is shown in Figure 9. When the number of iterations increases, JVM Reuse approach shows more reduction. In Fig. 9a, contrary to the start-up time of the original Hadoop increasing linearly with number of iterations, one of JVM Reuse-based Hadoop is a little changed. Regarding overall execution time (Fig. 9b), at the 8-iteration job, performance improvement is 35%. Moreover, Fig. 9c reveals the ratio of start-up time in comparison with overall execution time is small in case of JVM Reuse Hadoop.

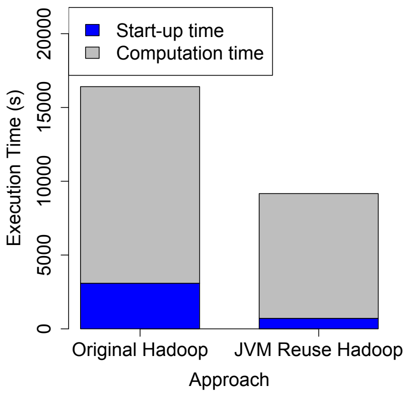
We do not provide evaluation of static file clean-up impact and different number of slots in the process pool. They are considered as our future work.



(a) Start-up time



(b) Overall execution time



(c) Ratio between start-up and computation time

**Fig. 9** PageRank with more iterations

## 4.2 MPI versus TCP/IP shuffling

As discussed in Section 2.3, shuffling in Hadoop MapReduce is slow in jobs having high volume of data exchange, such as tera-sort and self-join, and its performance can be improved by using MPI instead of TCP. In order to show benefit of MPI communication, first we present a MPI shuffling engine to bypass using the original shuffling based on TCP communication. Our evaluation on the tera-sort workload shows proofs of MPI advantage.

### 4.2.1 MPI shuffling design

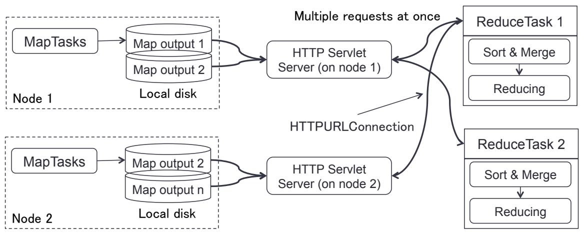
Shuffle engine of the original Hadoop MapReduce is removed, and we replace its HTTP servlet server with our *Shuffle Manager* to handle receiving requests and sending MapOutput data. Figure 10 shows comparison between the Hadoop shuffle engine and ours. There are three main phases in Hadoop MapReduce: mapping, shuffling, and reducing. Mapping and reducing run users' MapTask and ReduceTask, respectively. The shuffling phase is located in the middle in which a Reducer fetches MapOutput data from MapTasks that is written to local disk through HTTP servlet servers (Fig. 10a). Each slave node has one HTTP servlet server that can handle multiple HTTPURLConnections from Reducers at once (nonblocking type). HTTPURLConnection is a library based on TCP/IP.

In the same way, our shuffle engine does receive requests from Reducers, then read map MapOutput files, and send back to the Reducers. Nevertheless, we use MPI connection instead of HTTPURLConnection. Since number of requests is unknown and the requesting timestamp of each Reducer is different, it is impossible to use nonblocking MPI. Therefore, our shuffle engine can only handle one request at once (blocking type).

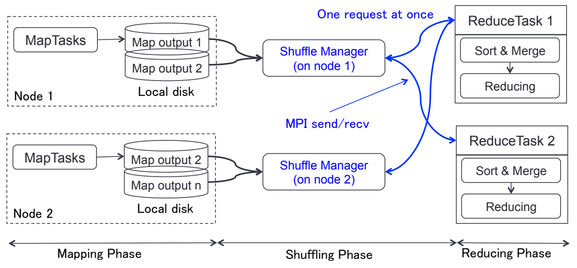
### 4.2.2 MPI shuffling performance

Tera-sort is a common MapReduce benchmark to measure shuffling performance since amount of exchange data is big and almost equal to its input size. We run Tera-sort on 32 FX10 nodes with a 4-slot pool, -Xmx4096m heap memory for each JVM process, and 256MB of HDFS file block size. Since OpenMPI on FX10 does not support multiple thread-safe communication, we set number of fetcher threads running on each Reducer to *one* (*mapreduce.reduce.shuffle.parallelcopies = 1*).

Figure 11 shows that improvement from MPI shuffling is up to 5% and 10% in comparison with nonblocking and blocking TCP one respectively. During the experiment, we notice that when input size is less than 4GB, advantage of MPI shuffling is unclear.



(a) TCP/IP-based shuffling



(b) MPI-based shuffling

**Fig. 10** *TCP/IP vs. MPI shuffling*

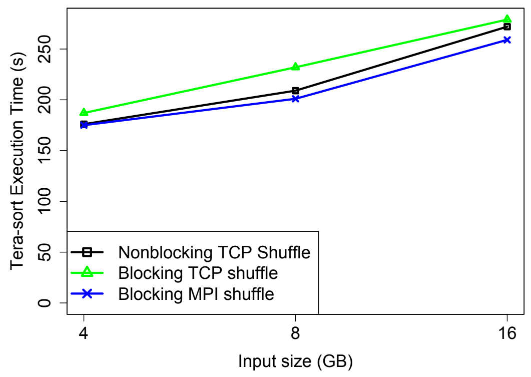


**Fig. 11** *MPI shuffling performance*

## 5. Related work

**JVM Reuse.** M3R [4] proposed a Hadoop MapReduce (HMR) engine written in X10 language [23] that proved that significant performance gain can be obtained by totally implementing HMR APIs in a main-memory implementation using JVM Reuse. Experiments of their implementation of M3R showed advantages of in-memory implementation in aspects of proportion of remote shuffle and running time. Although one of M3R's contribution that we are aware of is to identify the sources of performance gain in main-memory implementation including input/output cache and avoiding disk-based out-of-core shuffling and JVM start-up, they do not provide any specific evaluation of JVM start-up time reduction by using JVM Reuse. Also, there was no optimization for JVM Reuse provided. In this paper, we provide a mechanism to employ JVM Reuse more efficiently and evaluation in details on how the original HMR engine benefits from JVM Reuse that M3R omitted. Moreover, our approach keeps the original HMR engine with minimum changes.

In the Hadoop v1, users can set number of tasks that are executed in a JVM process in sequence (*mapreduce.job.jvm.numtasks*) [24]. This parameter is useful to avoid overheads of JVM start-up in a single job, but the JVM itself will be terminated after its job completed. Hadoop v1 does not provide any mechanism to keep that JVM running to execute tasks of other jobs.

**Data shuffle.** JVM-Bypass [25] has the same purpose of speeding up shuffling phase. They figured out JVM-based shuffling engine is slow because of its deep stack of transport protocols, so they implemented their own C-based shuffling engine that also supported RDMA. While their main objective is to evaluate effectiveness of bypassing JVM, we focus on using MPI over Hadoop processes and how fast MPI data exchange speeds up fetching of MapOutput files.

**Iterative computation.** For the sake of improving iterative MapReduce computation performance, HaLoop [3] provides a mechanism to cache and reuse mapper and reducer input data, and reducer outputs. HaLoop helps make all iterations running in only one MapReduce job, but MapTask and ReduceTask processes still need to be created newly. Spark [17] is designed to benefit iterative algorithms by using resilient distributed immutable datasets (RDD) in which parallel operators (*map, filter, reduce, and collect*) can be performed. It uses Mesos resource manager to allocate new containers with which our JVM Reuse can be also integrated.

**MapReduce on supercomputers.** Regarding MapReduce implementation on the supercomputer environment, it is often designed to a certain specific type of supercomputers. For example, K MapReduce [26] from Riken is developed to exploit the K supercomputer, and MapReduce MPI's experiments from Sandia [27] had been done on the Cray supercomputer. Both of them are written in C and using MPI in order to gain benefit from beneath supercomputer hardware where Tofu and 3D-Torus interconnection are setup respectively. However, in aspects of productivity and maturity, Hadoop MapReduce takes more advantage.

## 6. Discussion and Conclusion

We propose JVM Reuse approach to improve Hadoop MapReduce performance on the FX10 supercomputer to run iterative jobs, such as PageRank and K-Means, more efficiently. JVM Reuse helps shorten start-up time and enable MPI communication among Hadoop processes on the FX10. Compared to other JVM Reuse implementation, our design considers optimization including *Reflection* and *Clean-up*. Moreover, our approach only makes minimum changes of the original Hadoop. Our evaluation shows effectiveness of JVM Reuse with improvement up to 35% obtaining from both start-up time reduction and MPI shuffling.

**JVM Reuse Drawbacks.** Performance of long running and CPU-bound tasks in a reused JVM process can be affected since we do not carry out a full clean-up including garbage collection, reseting all static fields, and checking nonstop running threads. Thus, it comes to slow down the JVM process. Furthermore, idle JVM processes in the pool might also consume a certain amount of CPU time that we will examine in the future work.

**JVM Reuse for other frameworks.** JVM Reuse idea is not limited to Hadoop. In addition, its design can be applied to other frameworks, for instance, Spark [17] using Mesos resource manager. Our implementation of JVM Reuse, the process pool, can work independently, so it is easy to integrated with outside components in the way of allocation and de-allocation.

Our future work is to add improvement for JVM Reuse, such as full clean-up including static fields and heap memory.

### References

[1] Dean, J. and Ghemawat, S.: MapReduce: simplified data processing on large clusters, *Communications of the ACM*, Vol. 51, No. 1, pp. 107–113 (2008).

[2] White, T.: *Hadoop: The definitive guide*, " O'Reilly Media, Inc." (2012).

[3] Bu, Y., Howe, B., Balazinska, M. and Ernst, M. D.: HaLoop: efficient iterative data processing on large clusters, *Proceedings of the VLDB Endowment*, Vol. 3, No. 1-2, pp. 285–296 (2010).

[4] Shinnar, A., Cunningham, D., Saraswat, V. and Herta, B.: M3R: increased performance for in-memory Hadoop jobs, *Proceedings of the VLDB Endowment*, Vol. 5, No. 12, pp. 1736–1747 (2012).

[5] Page, L., Brin, S., Motwani, R. and Winograd, T.: The PageRank citation ranking: Bringing order to the web. (1999).

[6] Gara, A., Giampapa, M., Heidelberger, P., Singh, S., Steinmacher-Burow, B., Takken, T., Tsao, M. and Vranas, P.: Blue Gene/L torus interconnection network (2005).

[7] Ajima, Y., Takagi, Y., Inoue, T., Hiramoto, S. and Shimizu, T.: The tofu interconnect, *High Performance Interconnects (HOTI), 2011 IEEE 19th Annual Symposium on*, IEEE, pp. 87–94 (2011).

[8] FX10: http://www.cc.u-tokyo.ac.jp/system/fx10/index-e.html (2015).

[9] MPISpawn: https://www.open-mpi.org/doc/v1.8/man3/MPI Comm spawn.3.php (2015).

[10] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S. and Stoica, I.: Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center., *NSDI*, Vol. 11, pp. 22–22 (2011).

[11] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S. et al.: Apache hadoop yarn: Yet another resource negotiator, *Proceedings of the 4th annual Symposium on Cloud Computing*, ACM, p. 5 (2013).

[12] White, T.: *Hadoop: the definitive guide: the definitive guide*, " O'Reilly Media, Inc." (2009).

[13] Wu, D. and Gokhale, A.: A self-tuning system based on application Profiling and Performance Analysis for optimizing Hadoop MapReduce cluster configuration, *High Performance Computing (HiPC), 2013 20th International Conference on*, IEEE, pp. 89–98 (2013).

[14] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P. and Murthy, R.: Hive: a warehousing solution

over a map-reduce framework, *Proceedings of the VLDB Endowment*, Vol. 2, No. 2, pp. 1626–1629 (2009).

[15] Islam, M., Huang, A. K., Battisha, M., Chiang, M., Srinivasan, S., Peters, C., Neumann, A. and Abdelnur, A.: Oozie: towards a scalable workflow management system for hadoop, *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, ACM, p. 4 (2012).

[16] Ahmad, F., Lee, S., Thottethodi, M. and Vijaykumar, T.: Puma: Purdue mapreduce benchmarks suite (2012).

[17] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. and Stoica, I.: Spark: cluster computing with working sets, *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pp. 10–10 (2010).

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

[19] Yoshida, T., Maruyama, T., Akizuki, Y., Kan, R., Kiyota, N., Ikenishi, K., Itou, S., Watahiki, T. and Okano, H.: Sparc64 X: Fujitsu's New-Generation 16-Core Processor for Unix Servers, *Micro, IEEE*, Vol. 33, No. 6, pp. 16–24 (2013).

[20] Yokokawa, M., Shoji, F., Uno, A., Kurokawa, M. and Watanabe, T.: The K computer: Japanese next-generation supercomputer development project, *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, IEEE Press, pp. 371–372 (2011).

[21] Ajima, Y., Sumimoto, S. and Shimizu, T.: Tofu: A 6D mesh/torus interconnect for exascale computers, *Computer*, Vol. 11, No. 42, pp. 36–40 (2009).

[22] Vega-Gisbert, O., Roman, J. E. and Squyres, J. M.: Design and implementation of Java bindings in Open MPI (2014).

[23] Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O. and Grove, D.: X10 language specification (2011).

[24] Stewart, R. and Singer, J.: Comparing fork/join and MapReduce, Technical report, Citeseer (2012).

[25] Wang, Y., Xu, C., Li, X. and Yu, W.: Jvm-bypass for efficient hadoop shuffling, *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, IEEE, pp. 569–578 (2013).

[26] Matsuda, M., Maruyama, N. and Takizawa, S.: K MapReduce: A scalable tool for data-processing and search/ensemble applications on large-scale supercomputers, *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, IEEE, pp. 1–8 (2013).

[27] Plimpton, S. J. and Devine, K. D.: MapReduce in MPI for large-scale graph algorithms, *Parallel Computing*, Vol. 37, No. 9, pp. 610–632 (2011).