The University Of Tokyo

Graduate School of Information Science and Technology

Department of Creative Informatics

博士論文

Effective Deployment of Data-intensive Frameworks on Supercomputers

(スーパーコンピュータ上へのデータ集約型計算フレームワークの効果的展開)

Doctoral Dissertation of:

Thanh-Chung Dao

ダオ　タンチュン

Academic Advisor:

Shigeru Chiba

千葉　滋

Abstract

The goal of this research is to achieve a better performance of popular data-intensive frameworks, for example Hadoop and Spark, with only small modifications when running on modern supercomputers. Big data analytics applications have been becoming more important and widely being demanded to process large-scale datasets in both industry and academia. Compared with developing a new data-intensive application from scratch, using existing popular data-intensive application frameworks to develop is a better choice in aspects of productivity and maturity. Supercomputers are potentially faster than commodity clusters, such as Amazon EC2 cloud, when running data-intensive applications due to their high-performance and high-cost hardware. However, the current supercomputer design focuses more on compute-intensive applications rather than data-intensive ones, so it is hard to achieve the best performance of the hardware when running data-intensive applications on supercomputers. This is partly because it is also important to keep the original data-intensive frameworks' source code as much as possible since minimizing the cost of changes in the architecture helps increase productivity and easily upgrade to newer versions.

We observe two mismatches of the execution environment on a lack of MPI-friendly dynamic process creation and local disks when running those frameworks on supercomputers since they are designed to run on the commodity clusters, but supercomputer design is different from commodity clusters. The first mismatch raises a question of how to provide MPI-compatible fast dynamic process creation for popular data-intensive frameworks but satisfy the standard way of creating processes on supercomputers. The second mismatch brings into question of when using in-memory storage to provide virtual local disks as a replacement of physical local disks, how to deploy that in-memory storage and what deployment strategy is good on supercomputers.

To overcome the first mismatch, we propose HPC-Reuse located between YARN-like and PBS-like resource managers in order to provide better support of dynamic management with MPI. YARN, which is a key component of Hadoop, our targeted data-intensive framework, is responsible for resource management. YARN adopts dynamic management for job execution and scheduling. We identify three Ds (3D) dynamic characteristics from YARN-like management: on-Demand (processes created during job execution), Diverse job, and Detailed (fine-grained allocation). The dynamic management does not fit into typical resource managers on supercomputers, for example PBS, that are identified having three Ss (3S) static characteristics: Stationary (no newly created process during execution), Single job, and Shallow (coarse-grained allocation). Our experimental results show that HPC-Reuse can reduce execution time of iterative workloads by 26% and speed up data shuffle up to 10% by using MPI.

Regarding to the second mismatch, we report our experiments to compare various deployment strategies of memcached-like in-memory storage for our focused Hadoop framework on supercomputers, where each node often does not have a local disk but shares a slow central disk in order to find the best strategy. For the experiments, we develop our own memcached-like file system, named SEMem, for Hadoop. Since SEMem is designed for supercomputers, it uses MPI for communication. SEMem is configurable to adopt various deployment strategies and our experiments reveal that a good deployment strategy is allocating some nodes that work only for in-memory storage but do not directly perform MapReduce computation, with up to 10-13% improvement in comparison with deploying the memcached-like file system on every computation node.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor Prof. Shigeru Chiba for the unlimited and continuous support since I joined his research group (CSG) as a research student. He always gave me motivation and inspiration whenever I was stuck in my research or felt uncomfortable in my life.

I would like to thank the members of my thesis committee: Prof. Kenjiro Taura, Prof. Mary Inaba, Prof. Masayuki Inaba, Prof. Toshiya Hachisuka, Prof. Hiroshi Esaki, and Prof. Shigeru Chiba for all of their guidance and comments.

My sincere thanks also goes to my labmates for the stimulating discussion, in particular Prof. Yoshiki Sato, Maximilian Scherr, YungYu Zhuang, Kazuhiro Ichikawa, Seisei Itahashi, Joe Caldwell, and Antoine Tu.

A special thanks to Japan's Ministry of Education, Culture, Sports, Science and Technology (MEXT) for providing the scholarship during my PhD study at the University of Tokyo.

Finally, I am deeply thankful to my family, my wife, and my friends for their support.

Thanh-Chung Dao
Tokyo, Japan
August 2017

Contents

List of Figures

Listings

List of Tables

Abbreviations

Acronym	**W**hat (it) **S**tands **F**or
MPI	Message Passing Interface
RDMA	Remote Direct Memory Access
YARN	Yet Another Resource Negotiator
PBS	Portable Batch System
SSD	Solid-State Drive
HPC	High Performance Computing
HDFS	Hadoop Distributed File System
RM	Resource Manager
NM	Node Manager
JVM	Java Virtual Machine
JIT	Just-In-Time Compilation
EMR	Amazon Elastic MapReduce
HMR	Hadoop MapReduce

Chapter 1

Introduction

In the big data era, there is a demand of analyzing large-scale raw data in both academia and industry that makes data-intensive applications become more important in the roles of modelling, self-learning, and revealing meaningful information. In general, large-scale computation is divided mainly into two types of applications including compute-intensive or data-intensive ones. Compute-intensive term refers to a kind of applications that requires a lot of computation, such as biomolecular structure modeling [1] and space weather simulation [2], in which CPU is the most important resource. However, data-intensive computing focuses on how to process efficiently large volumes of data in parallel. In data-intensive applications, storage, memory, and network resources are more important than the compute one. Many data-intensive workloads, for example, healthcare data analysis [3], message mining [4], graph processing [5], machine learning [6], pre-processing of simulation and log data [7][8], and analysis of GenBank [9], are emerging as challenges.

Developing a data-intensive application from scratch is time-consuming [10][11], hence in aspects of productivity and maturity, a better choice is using existing popular data-intensive frameworks. For examples, Hadoop and Spark are the most popular implementation of MapReduce [12] that is one methodology used widely for analyzing such large-scale datasets in big data applications. Hadoop [13] and Spark [14] are well supported and used widely on commodity clusters. While Hadoop is suitable for filtering or mining terabyte and petabyte-scale datasets, Spark is designed to run iterative applications (e.g. learning machine algorithms and graph processing) both in memory and on disks. Those frameworks often have easy-to-use interfaces that allow users write distributed applications in less than 100 lines of code. Moreover, Hadoop and Spark can be deployed on thousands of nodes and support fault tolerance in restarting or continuing failed tasks.

We want to run those existing popular data-intensive application frameworks on modern supercomputers since those supercomputers are equipped with high-performance and high-cost hardware and expected to run applications faster than commodity clusters, such as the Amazon EC2 cloud [15]. A supercomputer is an 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. In a supercomputer, each computer is called a compute node. The number of compute nodes is several thousands. The total number of processor cores is millions in almost all medium size supercomputers and the total memory size is several hundred terabytes. High-speed network is designed specially for each supercomputer. It can be Infiniband switches [16] or Mesh Torus [17].

Since supercomputers are designed to run compute-intensive applications rather than data-intensive ones, it is hard to exploit the supercomputer hardware in order to achieve the best performance of those data-intensive frameworks on supercomputers. Building a balanced data and compute intensive system is not easy, requires more research, and overcomes existing shortcoming of High performance computing (HPC) systems. Compute-intensive applications require more computation cores and faster inter-node communication, whereas data-intensive ones need less CPU resource and faster storage disk I/O. HPC systems are mainly used in researches and till they are being used widely in industry, problems related to resource management [18], scheduler [19], and maintenance must be solved. Instead of building appropriate supercomputers for data-intensive applications, adapting such applications is less expensive and more efficient. Each supercomputer has different hardware and configuration, data-intensive applications need to be flexible and easy to configured. Since data is unstructured and variable, those applications are able to change to suit new properties of data. The requirement from application viewpoints always change fast, so those applications also need to be changed themselves in order to speed up software development cycle and reduce costs.

On the other hand, to increase productivity and enable easy maintenance for the data-intensive frameworks on supercomputers, it is also important to keep the original architecture and minimize source code modifications as much as possible. For instance, Hadoop and Spark, which are the popular data-intensive frameworks, have a huge codebase, changing their architecture is not easy and might affect the performance later. Moreover, they are always upgraded with new features and improvement by their large developer commodity. By keeping the same architecture and source code as much as possible, it is easy to add new features and update with improvement and maintenance.

To achieve a better performance of running the existing popular data-intensive frameworks, for example Hadoop, on supercomputers while preserving the original source

code as much as possible, we observe that we have to resolve two mismatches between the frameworks and the execution environment. The first mismatch is a lack of MPI-friendly dynamic process creation, and the second one is a lack of local disks on each compute node. The two mismatches are caused by the fact that the existing frameworks are designed mainly for commodity clusters where both dynamic process creation and local disks are usually available, and thus they assume that the execution environment must support those two characteristics. However, supercomputers are designed differently from commodity clusters, hence they do not support dynamic process creation with MPI or provide a local disk on each compute node. The network communication on supercomputers is considered ten times faster than on commodity clusters, using well supported communication protocols on supercomputers, for example MPI, is necessary. To minimize the MPI delay and avoid restriction on supercomputers, application processes should be created simultaneously at the beginning, but they are dynamically created in the data-intensive frameworks due to simple scheduling and fault tolerance. Intermediate data in the data-intensive frameworks is typically spilled out to local disks, but a shared storage on supercomputers. No local disk on each compute node helps avoid disk failure and complex hardware maintenance, but disk I/O to the shared storage is expensive.

1.1 Motivation

We describe two mismatches related to the lack of MPI-friendly dynamic process creation and local disks as our motivation of this research in more details as follows:

The first mismatch. In comparison with commodity clusters, the interconnection of supercomputers is an order-of-magnitude faster, hence we want to extend Hadoop, a popular big data analytics framework, to use MPI to speed up data exchange. While MPI is the de facto communication protocol and highly optimized for the associated interconnection on supercomputers, socket (TCP) communication is used widely in applications on commodity clusters.

Hadoop and Spark data-intensive frameworks depend on YARN [20] and Mesos [21] resource managers to execute their tasks, whereas most supercomputers use PBS [22] and Slurm [23]. Process management on supercomputers is considered static, whereas it is dynamic on the resource managers of data-intensive frameworks. There are some studies on running Hadoop and Spark as typical jobs submitted to the PBS or Slurm's job queue [24] [25] [26]. To the best of our knowledge, however, there is no study carried out to adapt and optimize YARN-like resource managers to ones on supercomputers.

To increase task diversity and resource utilization, the job execution and scheduling in YARN follow the dynamic management style. There are three *dynamic* characteristics from YARN-like management including on-Demand (creating processes on the fly), Diverse job, and Detailed (fine-grained allocation). *Dynamic process creation* means the execution process is newly created when a task is available.

The YARN-like dynamic management does not work well with resource managers of supercomputers, such as PBS and Slurm, that have three *static* characteristics including Stationary (all-or-nothing strategy), Single job, and Shallow (coarse-grained allocation). On the supercomputers, the style of static process management is less flexible, but more efficient in the aspect of system administration and maximizing performance of a job running. Dynamic management on supercomputers is supported partially and not sufficient to be integrated with YARN-like resource managers. For example, Fujitsu FX10 supercomputer [27] allows to fork a new process but MPI is not available in that process, whereas TSUBAME supercomputer [16] provides creating dynamically processes by using MPI-Spawn command [28], but its performance is slow due to a collective operator required. MPI-Spawn is a mechanism to spawn new processes on which MPI connection is still kept.

The challenge is to bridge the gap between dynamic management of the popular data-intensive frameworks requiring MPI-compatible fast dynamic process creation and static management on supercomputers adopting the all-or-nothing strategy of process creation. The popular data-intensive frameworks, for example Hadoop and Spark, want to use MPI to improve performance on supercomputers and need dynamic process creation in order to keep their current architecture and minimize source code changes.

The second mismatch. In today's top supercomputers, each compute node does not have a local disk or is equipped only with a small size of a solid state drive (SSD) due to its relative costs and high failure rate. In this thesis, we focus on supercomputers where the local disk is not available. It makes writing and reading intermediate data during job execution (e.g. Hadoop jobs) become a bottleneck since those data must be stored on the distributed central disk whose access time is considered slower five orders of magnitude than one of main memory [29]. To solve that problem, a natural approach is using in-memory intermediate data storage in order to avoid spilling to disk. We call that approach as *memcached-like* style [30] since in-memory storage could be at either a local or remote node.

How to deploy memcached-like file systems for the popular data-intensive frameworks on supercomputers is not an easily answerable question since supercomputers are different from typical clusters and the combination of memcached-like in-memory storage and Hadoop-like data-intensive frameworks is not studied well as far as we know. A feature of

supercomputers is that the number of compute nodes are often huge and supercomputer users submit jobs to a job queue in order to request a number of nodes they need. When compute nodes are given to users, should we deploy in-memory storage on every node? should we allocate dedicated nodes from the given nodes for in-memory usage only? or should we allow using memory of remote nodes?

1.2 Contributions

In this research, we aim to increase performance of the existing popular data-intensive frameworks, such as Hadoop and Spark, on modern supercomputers and avoid large source code modifications. The desired result is also to be faster the default deployment of those frameworks on supercomputers. Moreover, the result running on supercomputers should outperform on cloud services, such as Amazon EC2 [15] and Microsoft Azure [31]. An effective deployment is required to run those data-intensive application frameworks on supercomputers.

In this thesis, we define the *effective deployment* term as the standard or typical way of using something. "Effective deployment of data-intensive frameworks on supercomputers" title means that our proposed deployment achieves as good performance as running commodity clusters or even faster. In comparison with the default deployment, our proposed system should be faster as well.

To solve the problem of the lacks of MPI-friendly dynamic process creation and local disks, we have proposed HPC-Reuse and SEMem that are considered as para-virtualization of dynamic process creation and local disks, respectively. HPC-Reuse enables a process creation mechanism with MPI support and small modification for data-intensive frameworks and that mechanism is relevant to the static management on supercomputers. SEMem is about providing efficient memory-based virtual local disks instead of using the shared central storage by answering the question of which deployment strategy of in-memory storage is good and in what context it is good.

First, in order to provide better support of dynamic management with MPI, we have proposed a virtual layer located between YARN-like and PBS-like resource managers on supercomputers. It helps avoid process creation, such as MPI-Spawn, and enable MPI communication over Hadoop processes. We name it HPC-Reuse designed specially for Hadoop YARN hosting JVM-based applications, such as Hadoop MapReduce, Spark, and Storm [32] applications, running on supercomputers.

MPI integration on HPC-Reuse also gives other important advantages. For Fujitsu FX10-like supercomputers that do not allow dynamic process creation on the same node,

HPC-Reuse enables MPI on Hadoop processes and makes YARN possible to host pure MPI applications. For TSUBAME-like supercomputers on which MPI-Spawn overhead is high, HPC-Reuse helps avoid it and also host pure MPI programs more efficiently in the aspect of start-up time overhead. By virtue of fine-grained management manner in YARN, multiple workloads can be scheduled to run simultaneously and more efficiently. It allows automated and complex post processing of MapReduce and pure MPI programs. Furthermore, in such analysis workflows, data copying between MapReduce and pure MPI programs can be avoided by using in-memory cache provided on HPC-Reuse.

Compared to the original Hadoop, our experimental results show that HPC-Reuse can reduce execution time of iterative workloads by 26% on FX10 supercomputer. On TSUBAME supercomputer, compared to using MPI-Spawn to create new processes, HPC-Reuse has achieved improvement of 52% on average. In order to show MPI performance, we have designed a MPI shuffle engine to speed up data exchange. On TSUBAME and FX10 supercomputers, it reduces execution time up to 10% by using in-memory storage.

Overall, we have made four contributions for the first issue of virtualizing MPI-friendly dynamic process creation. We have identified 3D and 3S characteristics and difficulties when running YARN-like resource managers on supercomputers. 3D characteristics are used to describe dynamic resource managers, such as YARN and Mesos. We have designed and implemented HPC-Reuse that is both providing better support dynamic management on supercomputers and satisfying restrictions on the supercomputer environment. Moreover, we have showed that MPI shuffling engine is feasible to speed up data exchange and enabled YARN to host pure MPI applications efficiently.

Second, we have figured out which deployment strategy of memcached-like file systems is good on supercomputers by designing experiments of different in-memory storage deployment strategies for figuring out which one achieves a good performance when running data-intensive MapReduce applications on supercomputers. For experiments, we have designed our own memcached-like file system (or virtual local disk) named *SEMem*. Since Hadoop and Spark are good choices to run data-intensive applications even on supercomputers in aspects of productivity and maturity, we integrated our SEMem with the implementation of Hadoop. The deployment of SEMem is easily configurable and the intra-communication is through MPI, which is the de facto networking protocol on supercomputers. We examined the following deployment strategies:

1. SEMem is deployed as RAM disks where data is stored only in local memory.
2. SEMem is deployed on every node and data can be stored in remote memory.
3. SEMem is deployed only on dedicated nodes that are used only for storage.

Note that the original memcached software [33] is not a file system, does not support Hadoop directly, and uses TCP socket for data exchange. That is the reason why we developed SEMem from scratch.

Our experimental results reveal that allocating a group of nodes used only to store data in memory shows a good result in data-intensive applications with 10-13% improvement in comparison with deploying the memcached-like file system on every computation node. The benefit has been achieved in the context in which the ratio of the dedicated nodes to the total nodes is small. Note that there is no computation task running on those dedicated nodes. No computation task on memory nodes might give a chance of allocating bigger memory space to in-memory storage and makes the memory nodes less busy.

Moreover, our experiments reveal that supercomputer centers can consider in-memory storage (e.g. SEMem) instead of installing SSD storage for flexible hardware resource configuration and supercomputer users can choose in-memory storage as an alternative of SSD. Compared to the central disk and SSD storage on TSUBAME supercomputer [16], our experimental results show that SEMem reduces execution time by 25% and 5%, respectively, on average when data size is bigger than 128 GB. In this experiment, SEMem runs on dedicated nodes and for fairness of this experiment, all three configurations use the same total number of nodes. In case of SEMem, some nodes are assigned to only storing data, whereas they are used for computation in the central-disk and SSD configurations. That means SEMem uses a smaller number of computation nodes.

In this thesis, our implementation focuses on Hadoop, but our HPC-Reuse and SEMem can also work with other JVM-based frameworks, such as Spark, DataMPI [11], and Flink [34], with small changes. Our idea of deployment strategies could be applied to K MapReduce [10] and Mimir [35] (C++ MapReduce implementation), but we have not conducted experiments so far.

In summary, our scientific contributions consist of narrowing the gap between dynamic and static process management on supercomputers and showing that using dedicated nodes only for in-memory data storage has the potential to achieve a good performance.

Related technical issues solved in HPC-Reuse and SEMem, our technical contributions are listed as follows:

- We have designed and implemented HPC-Reuse to better support dynamic management on supercomputers.
- We have showed that MPI shuffling engine is faster than socket-based (TPC) communication.

- We have enabled YARN to host pure MPI applications efficiently.
- We have proposed a design of efficient data copying between Hadoop MapReduce and pure MPI applications in post processing tasks.
- We have designed SEMem that supports different deployment strategies.
- We have designed and implemented SEMem that is tightly integrated with Hadoop. It is possible to be adapted for Spark.
- We have implemented MPI communication in SEMem.
- We have minimized changes in Hadoop by providing para-virtualization implementation.

1.3 Thesis organization

While this chapter showed our motivation and contributions, the next five chapters will focus on background knowledge, our proposals, and comparison. The purposes and contents of each chapter are briefly described as follows:

Chapter 2 will give us the background of data-intensive application frameworks and the design of top supercomputers. Then, we show reasons why using supercomputers is a better choice to run data-intensive frameworks. Then, our observation and mismatch that we have found will be presented. Comparison of the existing solutions will be given in the end.

In the Chapter 3, we present HPC-Reuse and its implementation in Section 3.1 for overcoming the first mismatch. Other benefits of HPC-Reuse are discussed in Section 3.2. Section 3.3 is dedicated to providing experimental results. Finally, Section 3.4 provides a review of related work and is followed by brief discussion and conclusion in Section 3.5.

To solve the second mismatch, in the Chapter 4, we answer the question of which deployment strategy of in-memory storage is good and in what application type it is good by designing experiments in Section 4.2. The detailed SEMem architecture and several implementation issues are described in Section 4.3 and 4.4.3 including communication protocol and storage size. Section 4.5 is dedicated to show the experimental results. Finally, we review related work in Section 4.7.

In the Chapter 5, the overall performance gain of our proposals is evaluated by conducting the experiment of both HPC-Reuse and SEMem. The result will prove if we achieve our goal mentioned in Section 1.2 or not.

We conclude our thesis in Chapter 6 by showing a comparison table of our proposal and the existing solutions. Our proposals' limitations are given in Section 6.1. Final, the future work is discussed in Section 6.2.

1.4 Main publications

The contents of Chapter 3 are mainly reproduced from the paper entitled "HPC-Reuse: efficient process creation for running MPI and Hadoop MapReduce on supercomputers" [36] that was presented at 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid (CCGrid 2016). I was the first author of this paper and Prof. Chiba was a co-author.

Chapter 4 is also based on the paper entitled "SEMem: deployment of MPI-based in-memory storage for Hadoop on supercomputers" that will be presented at 23rd International European Conference on Parallel and Distributed Computing (EURO-PAR 2017). I am the first author of this paper and Prof. Chiba is a co-author.

Chapter 2

Background

Data-intensive application frameworks, supercomputer design, and mismatches when running data-intensive application frameworks on supercomputers are discussed mainly in this chapter. Big data analytics applications have been becoming important and widely used to process large-scale datasets. It is hard to process large-scale datasets in commodity clusters due to the limitation of storage and computation power, hence supercomputers and commercial HPC clusters show more advantages to process those datasets. Since supercomputers are usually equipped with high-performance and high-cost hardware, they have a potential for running faster than commodity cluster, such as laboratory servers and the Amazon EC2 cloud. On the other hand, it is important to keep the original source code of the data-intensive frameworks as much as possible due to productivity and easily upgrading, even when they run on supercomputers. These requirements cause mismatches of the execution environment. They are dynamic process creation and local disks since commodity clusters are the preferred environment of those frameworks, but the design of supercomputers and commodity clusters is not the same.

In this chapter, first we introduce background knowledge of data-intensive application frameworks and the design of top supercomputers. We are going to draw a general picture of those frameworks and supercomputers in order to locate its importance. Then, the rationale behind choosing supercomputers for processing those data-intensive frameworks is revealed. Our observations, mismatches, and related work of process management and data storage are presented in order to show the challenges that we are going to solve. Comparison of the existing solutions is presented in the end.

2.1 Data-intensive application frameworks

In the following section, we present typical data-intensive applications and review well known existing data-intensive frameworks including Hadoop, Spark, and Flink. Moreover, several data-intensive frameworks developed from research labs are also compared.

While the data-intensive *application* term refers to the real and independent software that is used directly to process and analyze data, the data-intensive *framework* describes the software used to develop applications. In this thesis, frameworks can be development libraries that enable fast development and platforms used to run applications as well.

Data-intensive (or big data) applications are one part of large-scale scientific computing. Big data analytics has become important in many scientific computing domains and MapReduce is one methodology used widely for analyzing such large-scale datasets in big data applications. Hadoop and Spark are the most popular implementation of MapReduce. Several studies call Hadoop/Spark popular data analytics tools or data processing frameworks. However, we call them *existing data-intensive frameworks*. Moreover, in this thesis, we focus mainly on Hadoop MapReduce as a concrete example of popular existing data-intensive frameworks.

2.1.1 Data-intensive applications

Research paradigm has been moving to data-intensive science recently that must reveal information and knowledge from data. That kind of science is called the fourth paradigm, the era of data science discovery [37]. The raw data itself is useless since it is unstructured and does not contain any easy-to-see information. By applying analyses, such as statistics and machine learning, on the raw data, meaningful trends and interesting patterns can be uncovered. For example, Facebook collects, stores, and analyzes all user data including 10 billion messages, 350 million uploaded pictures, 4.5 billion liking and comments on a daily basis. Facebook wants to know user preferences and behavior [38].

Data is generated and accumulated from different sources, such as sensors, telescopes, human genomics, transaction logging, and web pages. The speed of data accumulation is faster that making sense of those data. The amount of data is double every year and it makes the data analysis more complex [9]. The more data is accumulated, the more difficult storage, processing, and mining become. Those data generated from sensors or transaction logging are unstructured and most of them are stored as the raw data.

Data-intensive applications have been identified in broad science and engineering disciplines and interdisciplinary researches. Below is the list of data-intensive applications in different research fields.

- Genomics: genome and meta-genomic correlations and genome-wide association [1] are examples of data-intensive applications in which machine learning and statistical algorithms can be used to decrease computational requirements. GenBank is a database containing genomics data whose size is double every 18 months.
- Geoscience: large-scale raw geoscience data including observational data from sensors and simulation data needs to be reduced, organized and sorted in order to produce smaller data sets that is named data mining process [9] and MapReduce is a typical processing technique used to do such analysis [7]. Data mining helps convert raw data into meaningful information and a big picture.
- Astronomy: telescopes capture sky pictures, but the challenge is how to combine those images into a final one that is called *co-addition* processing when the number of images is huge. MapReduce and MPI can help distribute and process the workload in parallel [8].

2.1.2 Hadoop MapReduce

MapReduce [12] is a common programming model that is used to process large-scale datasets. Its programming paradigm is simple and many algorithms, for example PageRank [39], K cluster, and sorting, can be expressed following this model. The model helps parallel applications more easily that can be run on thousands of compute nodes by using a MapReduce frameworks, such as Hadoop. There are two main phases in the MapReduce flow: mapping and reducing. At the mapping phase, data is processed to produce (key, value) pairs. Those pairs are sorted by the same key and become input data of the reducing phase where the desire result is calculated. The two phases are summarized as follows:

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

A MapReduce PageRank example is illustrated in Fig. 2.1. It shows how raw data is processed and what result is summarized. Firstly, input data is spit into lines, then they continue to be calculated PageRank (PR) by Mapper. After that, (key, value) pairs are

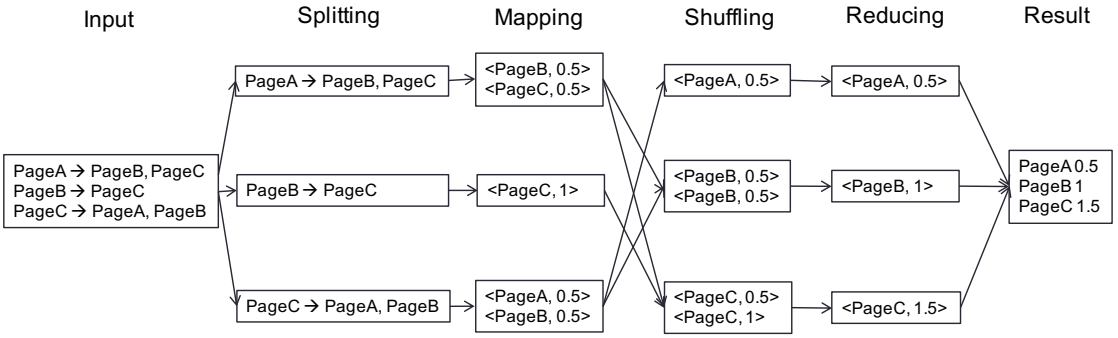


FIGURE 2.1: *Example of how MapReduce works: PageRank algorithm*

```
Function Mapper
Input  PageA → PageB, PageC
Begin
   N = outbound links
   For each outbound link
      output <Page, 1/N>
End
```

Shuffling Done automatically (Users can ignore)

```
Function Reducer
Input  <PageA, x1>, …<PageA, xn>
Begin
   rank = 0
   For each item xi
      rank += xi
   output <PageA, rank>
End
```

FIGURE 2.2: *Example of how MapReduce works: PageRank pseudocode*

shuffled to send to Reducer by the same key. At there, the total PR of each page is calculated. The pseudocode of PageRank algorithm written in the MapReduce style is shown in Figure 2.2.

Hadoop [13] is the most popular framework of MapReduce implementation that is used widely in practices and have large developer communities. It is written in Java and provides users simple usage of writing Map and Reduce functions. Moreover, it is easy to get started, even for users without parallel processing background. Nevertheless, it is designed based on TCP-based communication that is not supported well on supercomputers where MPI is considered as the de facto one. By using Hadoop MapReduce, a complex distributed application can be expressed and implemented in a hundred lines of code, for example PageRank and Breadth-first search, since it provides easy-to-use APIs. Moreover, it is possible to deploy Hadoop on thousands of nodes with fault tolerance support [40].

An example of a Hadoop MapReduce program (counting error lines) is shown in Listing 2.1 and 2.2. This example is referenced from a MapReduce tutorial [41]. The program consists of two classes: Mapping and Reducing. The mapping class is used by mapping tasks (MapTask) and the reducing class is run on reducing tasks (ReduceTask). The input data of MapTask is raw data that is split into lines that become the input data

of the *map* method. If a line is started with *ERROR*, one error is counted by calling context.write(new Text("ERROR"), one) in order to emit the output.

ReduceTask is responsible for calculating the final sum by fetching the output data emitted from MapTasks. All (key, value) pairs having the same key will be grouped and become the input data of the *reduce* method. The final sum is output by calling context.write(key, new IntWritable(sum)).

```
public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
  private final static IntWritable one = new IntWritable(1);

  public void map(LongWritable key, Text value, Mapper.Context context){
     String line = value.toString();
     if (line.startsWith("ERROR")){
         context.write(new Text("ERROR"), one);
     }
  }
}
```

LISTING 2.1: Hadoop MapReduce program example: counting error lines (Mapping class)

```
public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

public void reduce(Text key, Iterable<IntWritable> values, Context context)  {
    int sum = 0;
    for (IntWritable value : values) {
      sum += value.get();
    }
    context.write(key, new IntWritable(sum));
  }
}
```

LISTING 2.2: Hadoop MapReduce program example: counting error lines (Reducing class)

MapReduce is also used to solve problems of Graph500 [42], a supercomputer benchmark focusing on data-intensive workloads. We show an example of how BFS can be expressed in the MapReduce model in Algorithm 1. BFS is one kernel of the Graph500 benchmark.

2.1.3 Spark

Spark is a large-scale data processing framework and widely used in machine learning in which iterative computation happens, for example K-mean and logistic regression. For interactive data mining, Spark is also preferred to run multiple queries on the same data. Spark supports writing applications in Java, Scala, Python, and R and provides specific libraries for querying structured data, machine learning, graph-parallel computation,

and scalable fault-tolerant streaming computation. Spark can run on YARN and Mesos resource managers and support diverse data source, for example Hadoop distributed file system (HDFS), HBase, and S3 on Amazon web services. Note that YARN and Mesos are also used as the resource managers of Hadoop.

There is a broad class of scientific algorithms that can be solved using iterative 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 2.3 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. Note that it is possible to implement PageRank using Hadoop that is already mentioned in the previous section, but we need to write a program running mapping and reducing phases consecutively. By using Spark, however, the program is much simpler and shorter.

Spark has solved two challenges including providing explicit operators for more general data reuse and efficient fault tolerance. It lets users explicitly control and manipulate data reuse and also support broad range of applications that is not limited to MapReduce. Since data replication and checkpoint are expensive, it proposed a data model that helps easily build lost data from the original one.

while *distances to all nodes are not different from infinite* **do**
    Mapper input:
    (node_id distance adjacency list)
    Mapper:
    for *all node_id in adjacency list* **do**
        emit(node_id, distance + 1)
    end
    Reducer:
    Find the minimum of the distances to a given node
    Reducer output:
    (node_id minimum adjacency list)
end

Algorithm 1: Breadth-first search (BFS) using MapReduce with data representation (node_id distance adjacency list)

Spark proposed Resilient Distributed Dataset (RDD) abstraction used to store and manage data. RDD is a read-only and partitioned dataset and has two main aspects including persistence and partitioning. Persistence describes where data is stored, for example in memory or local disk. Since RDD is partitioned, partitioning information contains node addresses in which its data is stored.

Transformations can be applied on RDDs in order to create new RDDs, such as mapping, filtering, reducing by key, and sampling. All transformations are lazy operations when defining a new RDD and computation happens when actions are called.

An example of a Spark program (counting error lines) is shown in Listing 2.3. This example is cited from Spark research papers [43]. The first line reads text files from HDFS and data is stored in RDDs named *lines*. A filter is applied on *lines* in order to create different RDDs named *errors* that contains lines starting with *ERROR*. *error* RDDs are kept in memory by using the *persist* action. The *count* action is run on each RDDs to get the result.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
errors.count()
```

Listing 2.3: Spark program example: counting error lines

Lost RDD can be built fast from its representing information that makes Spark fault tolerant. A RDD scheme has a list of its parent RDDs, functions used to compute from parents, and data partitions.

2.1.4 Flink

Flink is another data-intensive framework that is getting more popular in industry and is started using in research. Flink's motivation is similar to Spark that brings all different

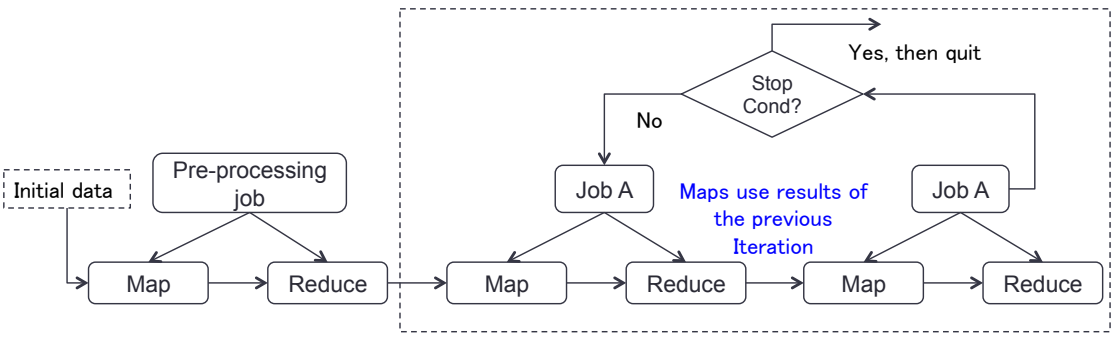


Figure 2.3: *Iterative job workflow*

kinds of processing and computation into a single platform including batch, streaming, machine learning, graph, and interactive ones. However, Flink is different from Spark in aspects of data model, memory management, and implementation.

Flink uses an abstraction named *Dataset* represented as plans in the runtime. Whereas, a RDD is represented as a Java object. Such difference enables *Dataset* to be optimized before being executed. That advantage is similar to *Dataframe* API of Spark.

Spark has used Java heap space for caching data, but Flink implemented a custom memory management located outside the heap space. The independent memory management helps avoid out of memory issues and garbage collection pauses. Moreover, resource utilization and I/O performance are improved.

Programming APIs of Flink and Spark are almost similar. While Flink is implemented using Java, Spark is written in Scala. Listing 2.4 shows an example of Flink program. Compared to the example of Spark in Listing 2.3, the difference is small and both examples are written in Scala.

```
lines = env.fromCollection(data)
errors = lines.filter(s -> s.startsWith("ERROR"))
errors.collect()
```

LISTING 2.4: Flink program example: counting error lines

2.1.5 Scratch building frameworks

Almost all common data-intensive frameworks are developed first and then widely used when they are proved useful and efficient. Developing a data-intensive framework from scratch is time-consuming and less generally used. We show two frameworks that are still being developed.

Regarding MapReduce implementation on supercomputer environments, almost all of them are under development and designed to a certain specific type of supercomputers. For example, K MapReduce [10] from Riken is developed to exploit the K supercomputer, and MapReduce MPI's experiments from Sandia [44] had been done on the Cray supercomputer. Both of them are written in C and using MPI that helps gain benefits from the beneath supercomputer hardware where Tofu and 3D-Torus interconnection are set up, respectively. However, such frameworks contain complex structures and require HPC experts to write applications based on their architecture.

2.1.5.1 K MapReduce

K MapReduce (KMR) [10] is an implementation of MapReduce programming model and its targeted supercomputer is K computer or similar descendants. Its motivation is to run large-scale MapReduce applications on supercomputers by leveraging the supercomputer's hardware including large amount of memory and fast interconnection on the K supercomputer. KMR is implemented using MPI that is the de facto communication on supercomputers. Moreover, it also provided fast file reading by exploiting the affinity between compute nodes and the data storage disks.

First, KMR is developed only for the K supercomputer, a supercomputer developed by Fujitsu using SPARC processor and Tofu interconnection, so it is not easy to deploy KMR on other supercomputers, such as ones made by Cray and IBM. KMR is optimized to used K supercomputer architecture including data accessing patterns and the communication protocol that is different from other vendors' design.

Second, KMR provided programming Application program interface (API) used to write a MapReduce program, but it did not follow any MapReduce coding style, such as Hadoop MapReduce API and Spark or Flink's API. To write a MapReduce program, users need to understand KMR's API and related parameters. Those parameters might be special configurations on the K supercomputer that users must understand clearly in order to use them. Moreover, KMR's documentation and tutorial are not detailed enough for ordinary users.

An example of a KMR program (word count) is shown in Listing 2.5. This example is cited from the KMR tutorial [45]. Mapping is done by using *kmr_map_once* method with the input data read from files. The (key, value) pair output stored in *kvs0* will be shuffled in order to group the pairs having the same key. Then, *kmr_reduce* is used to calculate the sum of each word. Note that two methods *read_words_from_a_file* and *sum_counts_for_a_word* should be written by the users and follow the KMR's specific defined syntax. Moreover, API names, such as *kmr_map_once* and *kmr_add_kv*, are not self-explanatory.

```
static int
sum_counts_for_a_word(const struct kmr_kv_box kv[], const long n,
    const KMR_KVS *kvs, KMR_KVS *kvo, void *p)
{
    long c = 0;
    for (long i = 0; i < n; i++) {
        c -= kv[i].v.i;
    }
    struct kmr_kv_box nkv = {
        .klen = kv[0].klen,
```

```
        .k.p = kv[0].k.p,
        .vlen = sizeof(long),
        .v.i = c};
    kmr_add_kv(kvo, nkv);
    return MPI_SUCCESS;
}

    // Mapping
    KMR_KVS *kvs0 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_INTEGER);
    kmr_map_once(kvs0, 0, kmr_noopt, 0, read_words_from_a_file);

    // Shuffling
    KMR_KVS *kvs1 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_INTEGER);
    kmr_shuffle(kvs0, kvs1, kmr_noopt);

    // Reducing
    KMR_KVS *kvs2 = kmr_create_kvs(mr, KMR_KV_OPAQUE, KMR_KV_INTEGER);
    kmr_reduce(kvs1, kvs2, 0, kmr_noopt, sum_counts_for_a_word);
```

Listing 2.5: KMR program example: word count

2.1.5.2 DataMPI

DataMPI [11] is a communication library used to extend MPI to support Hadoop-like computing on high performance computing (HPC) clusters. Since there is a lack of efficient communication support, for example MPI, for Hadoop-like big data frameworks including Spark, Dryad, S4, and Twister. DataMPI is developed to provide the first class communication protocol, MPI, on HPC clusters instead of using traditional techniques, such as HTTP and RPC.

DataMPI can be easily configured for different supercomputers, but its programming API is complex and not intuitive. Although it is a Hadoop-like library, its programming API is not similar to Hadoop MapReduce's one. In order to develop MapReduce programs, users need to understand the coding style of DataMPI including how to send and receive data.

2.2 Supercomputers

As the data size is getting bigger and bigger, the data cannot fit into small and commodity clusters in aspects of storage and computation. Supercomputers and commercial HPC clusters are more relevant to processing those large-scale data. In the following section, we introduce the general design of supercomputers consisting the first and the current most powerful ones. Then, the architecture of some well known supercomputers

is introduced. We also compare commercial HPC clusters, such as Amazon EC2 and Microsoft Azure.

In this thesis, we define *supercomputers* having thousands of identical high-performance nodes and the node allocation and deallocation has been managed by a PBS-like [22] or Slurm-like [23] resource manager. Most of supercomputers from Cray, IBM, and Fujitsu fit into our definition. *Supercomputers* and *HPC clusters* terms are sometimes used interchangeably, but a HPC cluster should be smaller than a supercomputer in aspects of number of nodes and users. Although a HPC cluster is also equipped with high-speed network and high-performance compute and memory resources, but its scale is still small. A HPC cluster often has a local disk on each compute node that makes HPC clusters close to commodity clusters. We focus more on supercomputers in this research.

2.2.1 Supercomputers versus Commodity clusters

Supercomputer design has a long research and is changed significantly during development of microprocessor, RAM technology, network communication, and data storage. The first supercomputer was developed by Cray [46] in 1964 when vacuum tubes were mainly used in large computers and its peak performance was 3 million floating point operations per second (flops). In the TOP500 list, the current fastest supercomputer has achieved 93 petaflops including more than 10 million processor cores [47]. The size of supercomputers is also reduced from the first one that can fill four cabinets to small racks containing thousand of nodes. Performance is the most important motivation when building a new supercomputer. Since CPU clock is limited, the only way to increase computation speed is adding more processors and more nodes. It is hard to combine thousand of compute nodes and it requires careful design of I/O and inter-node communication.

There are two main differences between supercomputers and commodity clusters: network communication and data storage. Commodity means using low-performance and low-cost hardware (components) that is typically standardized and manufactured by multiple vendors. Commodity clusters can be a group of simple personal computers and hardware on each computer is different from each other. By contrast, since each supercomputer is designed with uniqueness in order to achieve targeted performance, high-cost and identical hardware is used for every compute node.

There are three main components on supercomputers: compute nodes, data storage servers, and high-speed switches (or I/O nodes). The data storage servers consist of distributed hard disks and their total size is several petabytes. We call it the *central*

disk for later use. Compute nodes and the central disk are connected to each other through high-speed network and I/O nodes.

Network communication. Both compute- and data-intensive applications require fast internode communication in order to make parallel tasks faster. Compute nodes are connected to each other through special connections having high throughput and low latency, such as 6D Mesh/Torus or QDR Infiniband. By contrast, disk I/O from compute nodes to the central disk is considered slow since it is piped through I/O devices and the central disk is a distributed storage consisting of thousand of HDDs.

MPI is the de facto communication protocol on supercomputers and optimized to exploit the high-speed interconnection of supercomputers. By contrast, commodity clusters use traditional techniques, such as HTTP, RPC, and TCP. On almost all supercomputers, Remote Direct Memory Access (RDMA) is supported to enable direct memory copying without involving of operating system. MPI also leverages RDMA for large data messages.

Data storage. In today's top supercomputers, it is typical that there is no local disk on each compute node, but SSD is sometimes equipped [16]. The size of a local disk is not scalable and is difficult to upgrade for all compute nodes. The local disk on each node can be a point of failure and when it happens, it is difficult to fix. Moreover, since physical space is limited on each compute node, equipping a local disk is not feasible. Instead of using local disks, a central disk (or storage servers) is deployed for the sake of scalability, fault tolerance, easy maintenance, and compactness. SSD can be integrated in each compute node or gathered as a shared storage. Note that the integrated SSD should be a temporary storage where data will be deleted after job execution.

2.2.2 Examples of supercomputers

There are several main supercomputer vendors in the world including Cray, IBM, Fujitsu, NEC/HPE, and Lenovo. In the Top500 List (November 2016) [48], Cray Inc. is the most popular vendor with four supercomputers in the top ten. In Japan, Fujitsu is the main vendor that made the K computer and Oakforest-PACS supercomputer. TSUBAME is ranked second in the most energy-efficient supercomputers of the Green500 list (November 2015).

The general design of supercomputers including compute nodes, data storage servers, and high-speed switches (or I/O nodes) is the same for almost all vendors, but each vendor has its own interconnection and data storage design. In this section, we introduce

two popular supercomputers in Japan: TSUBAME and Fujitsu FX10, and other vendors including Cray and IBM.

2.2.2.1 TSUBAME supercomputer

At TSUBAME supercomputer [16] of Tokyo Institute of Technology, each compute node is equipped with Intel Xeon X5670 2.93GHz (12 cores), 54GB memory, and 3 GPU NVIDIA Tesla K20X. Total memory size of all nodes is 80.55 TB. Regarding performance, the peak processing speed is 17 PFlops when combining both CPU and GPU. It has 1292 compute nodes in total.

Each node is connected with Infiniband device Grid Director 4700 and the maximum throughput of that interconnection is 80 GB/s. It supports non-blocking and the full bisectional bandwidth is available. An optimized MPI library is provided for writing applications.

TSUBAME has a SSD local storage on each compute node, but its size is just 120 GB. SSD is a temporary storage on each node and any running job on the compute node can write and read data. Data written during job running should be deleted manually after the job is finished. It raises security problem if the data is not removed after job running. Its central disk has six DDN SFA10000 developed by Data Direct Network Inc. The size of central disk is 7.13 PB in total and the central disk uses Lustre parallel file system.

2.2.2.2 Fujitsu FX10 supercomputer

PRIMEHPC FX10 (called FX10 for short) is a supercomputer brand of Fujitsu. Its first generation is K computer that is the fastest supercomputer in 2011 [49] and FX10 supercomputer at The University of Tokyo is the second generation. A FX10 system includes compute nodes, a central disk, and interactive and login nodes.

At The University of Tokyo, a FX10 node is equipped with SPARC64 IXfx 1.848 GHz processor (16 cores) and 32GB main memory. It has 50 racks and each rack has 96 compute nodes and 6 I/O nodes. Its peak performance is 1.13 PFlops and there is 150 TB of memory available in total.

Computing nodes are connected with each other through Tofu interconnection developed by Fujitsu on which MPI's maximum throughput is 10 GB/s, and they access the central storage through InfiniBand network. On each compute node, the SPARC processor is

connected directly to a dedicated interconnect controller (ICC). The ICC can be connected to 10 other ICCs. The Tofu interconnect includes Tofu network router (TNR), Tofu network interface (TNI), and Tofu barrier interface (TBI). TNR is responsible for packet transferring between ICCs. TNI controls packet communication and also provides RDMA communication that does not require intervention of the operating system on the destination node. TBI provides collective communication. Tofu interconnection is equipped with six-dimensional (6D) mesh/torus network [17]. When users request compute nodes, they can specify the one-dimensional (1D), 2D, or 3D shape that is corresponding to the network topology among compute nodes. Communication performance can be improved through that configuration. MPI communication is optimized to used Tofu interconnection.

The FX10 does not have a local disk for each computing node, conversely a central disk used. The size of central disk is 2.1 PB. Lustre file system is deployed on the central disk. The supercomputer also provides a temporary disk that is used for job staging.

2.2.2.3 Cray Titan supercomputer

Cray has a long history of building supercomputers and Titan, a Cray XK7 supercomputer, is one of them that is located at Oak Ridge National Laboratory [50]. Its peak performance is more than 20 PFlops that contains 18,688 physical compute nodes. Each compute node is equipped with a 2.2 GHz AMD Opteron 6274 processor (16 cores) and 32 GB of memory. The total size of memory is 598 TB. Each two compute node uses one Gemini interconnect router, the Cray custom interconnection. Moreover, each node contains an GPU NVIDIA Kepler with 6 GB of memory. Titan is currently ranked third in the top 10 [48].

Titan's interconnection uses a 3D torus network that is able to handle tens of millions of MPI messages per second. A compute node is connected to Gemini using HyperTransport 3.9 technology, a direct architecture that avoids PCI bottlenecks that is inherent in commodity networks. The interconnection's peak bandwidth is 20 GB/s.

Titan's compute node does not contain a local disk and it has only a central disk in which the OLCF's center-wide Lustre parallel file system is deployed. The storage size if 32 PB including 20,000 disk drives. The throughput of disk I/O on Titan's data storage is 1.4 TB/s considered as the fastest storage in 2013 [51].

2.2.2.4 IBM Sequoia supercomputer

IBM is also a big vendor of building supercomputer and Sequoia is an example of the Blue Gene/Q supercomputer constructed by IBM that is located at Lawrence Livermore National Laboratory (LLNL). It is ranked fourth in top 10 of Top500 list (November 2016) and was the fastest supercomputer in June, 2012 with the LINPACK performance is 17.17 PFlops. Sequoia has 98,304 compute nodes contained in 96 racks. Each compute node has a PowerPC A2 processor (16 cores) and 16 GB of DDR3 memory. The total size of memory is 1.5 PB.

The interconnection of Sequoia is 5D torus topology that contains ten 2-Gbytes/s bidirectional links and one 2-Gbytes/s link for I/O nodes [52]. The interconnection's peak bandwidth is 8.75 GB/s. I/O nodes are used to connect compute nodes and the file system.

There is no local disk on each compute node and Sequoia uses Lustre file system for its central disk. Its data storage size is 55 PB. There is no SSD integrated on each compute node.

2.2.3 Commercial HPC clusters

In comparison with academic supercomputers, commercial HPC clusters provided by Amazon and Microsoft are described in order to show drawback and strong points. While supercomputers often give users physical compute nodes, users prefer virtual compute nodes on cloud services. The physical instance is also provided on Amazon EC2, namely *Dedicated Hosts*, but its price is more expensive, up to 16 times higher.

2.2.3.1 Amazon HPC cluster

Amazon Web Services (AWS) is a pay-as-you-go cloud computing platforms that provides both commodity and high performance computing servers. It is flexible to provide on-demand servers for specific compute-intensive and data-intensive applications. Users do not need a large capital investment to build a HPC computer since they can use real physical HPC servers from AWS [53]. Moreover, users are able to scale out their applications across hundreds of nodes. C3 and C4 instances (4xlarge or 8xlarge) are considered as high performance compute nodes. C4 instances on AWS is equipped with Intel Xeon V3 Haswell (18 cores) and up to 60 GB of memory.

AWS network is not as fast as the real supercomputers since its network architecture is built for both commodity and HPC servers. The maximum throughput is 1.25 GB/s to

each instance. Moreover, there is no MPI optimization for its network hardware. Each type of instances has a different maximum bandwidth and expensive instances, such as c3.8xlarge and c4.8xlarge , should have a faster network throughput.

The C4 instance is backed by Amazon Elastic Block Store (EBS) that is considered as a central disk. Users can choose SSD-backed or HDD-backed storage. The size of storage can be easily scale out or in on demand. SSD storage on Amazon EC2 should be remote disks and does not locate in the same compute node with the processors and memory since it helps increase the efficiency of storage allocation and easy maintenance.

Since running MapReduce on the cloud is demanded, Amazon provides Elastic MapReduce (EMR) that enables easy, fast, and automated deployment of Hadoop clusters. Users can scale out the cluster on the fly, even when MapReduce applications are running. Amazon EMR enables the effective deployment in the aspect of fast and easy cluster deployment.

2.2.3.2 Microsoft Azure HPC cluster

Microsoft Azure is another cloud platform that also supports HPC servers for large-scale applications by using Microsoft HPC Pack [54]. Users can choose run compute-intensive and data-intensive applications on Windows of Linux servers. *A8/A9* instances of Azure has up to 16 cores and 112 GB of memory.

Those HPC instances are connected to each other through 5GB/s high-speed network that provides RDMA technology for MPI applications. Azure provides Microsoft MPI (M-MPI), a Microsoft implementation of MPI.

HPC instances are backed by Azure storage blobs that is a kind of central disks. The size of storage can be easily scaled out or in on demand. Users can choose SSD-backed or HDD-backed storage to run an instance.

In comparison with Amazon EMR, Microsoft provides Azure HDInsight that provisions Hadoop MapReduce clusters on demand. Azure HDInsight is an elastic platform in which users can easily monitor performance and cost.

2.2.4 System comparison

Table 2.1 shows the comparison of above introduced systems in aspects of how fast the interconnection is, if SSD is available or not, how big memory size is, and how easy to use those systems. Each criterion is rated high (✓✓✓), fair (✓✓), or low (✓). x denotes not being supported.

	Interconnection	SSD-backed	Memory size	Easy-to-use
TSUBAME	✓✓✓	✓✓	✓✓✓	✓
Fujitsu FX10	✓✓✓	x	✓✓	✓
Cray Titan	✓✓✓	x	✓✓	✓
IBM Sequoia	✓✓✓	x	✓✓	✓
Amazon HPC	✓	✓✓✓	✓✓✓	✓✓
Microsoft Azure HPC	✓✓	✓✓	✓✓✓	✓✓

TABLE 2.1: Comparison of the introduced systems

The rating of interconnection on supercomputers is high since they are equipped with connections having high throughput and low latency, such as Torus or QDR Infiniband. The Amazon HPC cluster is rated low since the maximum throughput is just 1.25 GB/s. Microsoft Azure provides a better one with as much as 5GB/s. Most supercomputers do not have a local disk on each compute node. TSUBAME has SSD storage up to 120 GB. Commercial HPC clusters offer large sizes of SSD storage that could be either local or remote disks. The memory size on cloud services is also larger than supercomputers' one, for example TSUBAME and Fujitsu FX10. In the aspect of ease of use, Amazon EC2 and Microsoft Azure have better interfaces to submit and control jobs. The command line interface is common on the supercomputer environment.

2.3 Where data-intensive frameworks should be run

A question is that users should use supercomputers or commodity clusters, such as Amazon EC2 and Microsoft Azure, to run their data-intensive applications. Our hypothesis is that supercomputers are potentially faster than cloud services. Accessing the supercomputer service is becoming easier as cloud ones. While cloud services (e.g. Amazon EC2 and Microsoft Azure) provide the pay-as-you-go payment model, supercomputers offers block computations, for example you must buy 3000 computation units for each payment that costs 129,900 Yen or 1,171 US dollars [55]. Those units is equivalent to 3156 hours of computation.

2.3.1 Choosing supercomputers

In comparison with commodity clusters, such as the Amazon EC2 or Microsoft Azure cloud, supercomputers have more advantages in the aspect of first-class hardware equipped in compute nodes and interconnection, hence choosing supercomputers is better choice. Compute nodes are connected to each other through special connections having high throughput and low latency, such as 6D Mesh/Torus or QDR Infiniband. While the maximum bandwidth on most of top supercomputers is 8 GB/s, the typical network bandwidth on lab clusters and cloud services is 1 GB/s. Moreover, it is typical that users can use physical compute nodes on supercomputers, but they are given virtual ones on Amazon EC2 and Microsoft Azure.

To verify the performance of data-intensive frameworks on supercomputers, we conducted an experiment of a Hadoop MapReduce benchmark, Tera-sort, running on the TSUBAME supercomputer and Amazon EMR. The size of input dataset is 128 GB and we keep the problem size unchanged. We change the number of compute nodes and measure the execution. Every measurement is done twice. On the TSUBAME supercomputer, a compute node has Intel Xeon X5670 2.93 GHz (12 cores), 54 GB memory, and 120 GB of SSD. The name of Intel CPU microarchitecture on TSUBAME is Westmere released in 2010 [56]. On Amazon EMR, we choose two instance types: m3.2xlarge and c3.8xlarge.

The m3.2xlarge instance has 8 vCPUs, 30 GB of memory, and 160 GB of SSD. The m3 instance can be based on Intel Xeon E5-2670 v2 (Ivy Bridge) or Intel Xeon E5-2670 (Sandy Bridge) processors. Each vCPU on Amazon EC2 is equivalent to a hardware hyper-thread. The m3 instance is considered to have a balance of CPU, memory, and network resources [57]. The c3.8xlarge instance has 32 vCPUs, 60 GB of memory, and 640 GB of SSD. The name of Intel CPU microarchitecture of c3.8xlarge is Intel Xeon E5-2680 v2 (Ivy Bridge). C3 is considered as high performance computing instances. Moreover, the c3.8xlarge instance is equivalent to a physical node [58].

In the aspect of hardware configuration, the m3.2xlarge instance is a bit weaker than the TSUBAME compute node, but its CPU microarchitecture, Sandy Bridge, is newer than the Westmere of TSUBAME nodes. The c3.8xlarge instance is more powerful than the TSUBAME compute node. While input data is stored in the central disk on TSUBAME, S3 storage is used on Amazon EMR. SSD storage on Amazon EMR should be located on the same node with compute and memory resources. However, that SSD storage can be provisioned from a pool of SSD disks.

Figure 2.4 presents Hadoop MapReduce performance running on the TSUBAME supercomputer and Amazon Elastic MapReduce (EMR). TSUBAME shows faster execution

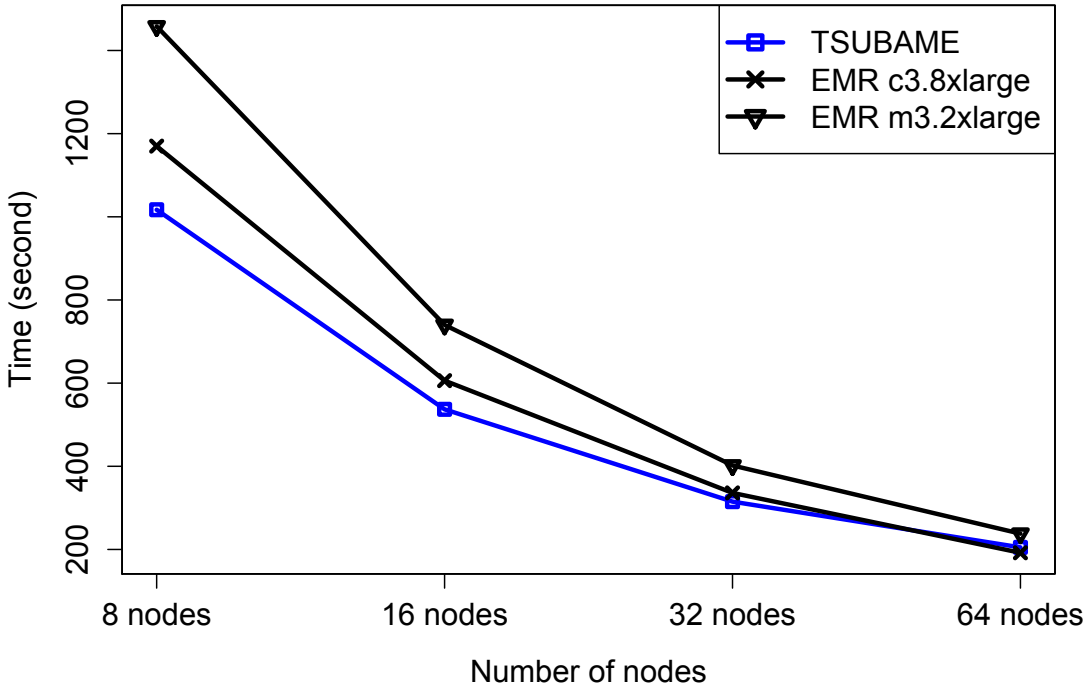


FIGURE 2.4: *Hadoop Performance on TSUBAME and Amazon EMR*

time when the number of compute nodes is less than 32 nodes. Compared Amazon EMR m3.2xlarge and c3.8xlarge, the execution on TSUBAME is faster 26% and 10%, respectively, on average. When there are 64 compute nodes, the performance of all three test cases is almost the same. The m3.2xlarge is a bit slower since its compute and memory resources is weaker. The c3.8xlarge cluster is running a little faster since it is using Ivy Bridge CPU microarchitecture that is newer and faster 17% and 4% than Westmere and Sandy Bridge, respectively [59]. When the number of compute nodes increases, the improvement on TSUBAME is small since data size is smaller and the compute resource on Amazon EMR is stronger. When the size of input data is large, there might be bottlenecks at S3 storage and remote SSD that make the performance of Amazon EMR is not fast as TSUBAME where SSD is located in the same node with compute and memory resources.

To check whether the price of supercomputer service is expensive and affordable or not. We compare the price of three test cases by hours. On Amazon EC2, users can choose dedicated on-demand instances that are immediately provisioned and spot instances that allow to bid spare Amazon EC2 resources. The price of spot instances can be cut up to 90% in comparison with the on-demand instances. The disadvantage of spot instances is that users might not bid instances or wait longer time to be provisioned. The spot

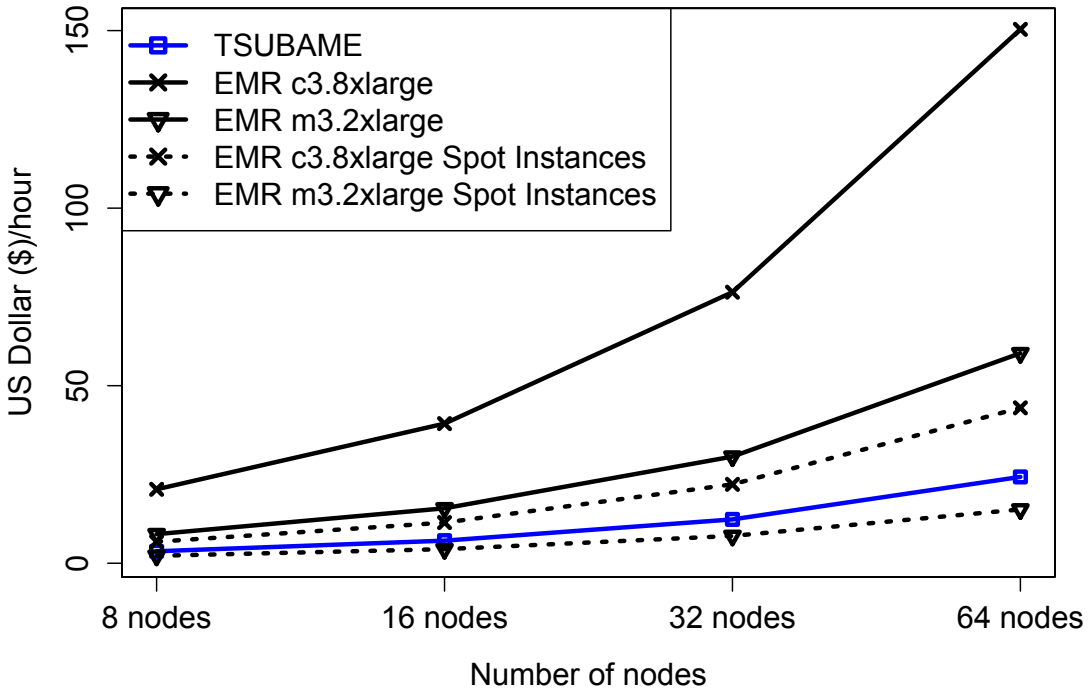


FIGURE 2.5: *Compute node price on TSUBAME and Amazon EMR*

instances are close to the supercomputer service since users must submit applications to the supercomputer's job queue and wait until the requested resource is available. In the aspect of expense, the TSUBAME supercomputer offers more reasonable hourly price than Amazon EMR. In Figure 2.5, compared to Amazon EMR m3.2xlarge and c3.8xlarge instances requested on demand, the hourly price of TSUBAME is cheaper 58% and 83%, respectively. The Amazon EMR m3.2xlarge spot instance is cheaper a bit than TSUBAME.

2.3.2 Performance is not satisfactory

The first-class hardware on supercomputers does not seem to help achieve the best performance of the popular data-intensive frameworks since the design of supercomputers aims to serve traditional compute-intensive applications. Although TSUBAME showed better performance than Amazon EMR m3.2xlarge and c3.8xlarge, but its improvement is not satisfactory as our expectation.

In this thesis, our targeted supercomputer is a large-scale multiple-node machine without a physical local disk on each compute node. We mention several supercomputers equipped with local disks, for example Cray CS300 Hadoop [60] and IBM ASC Purple

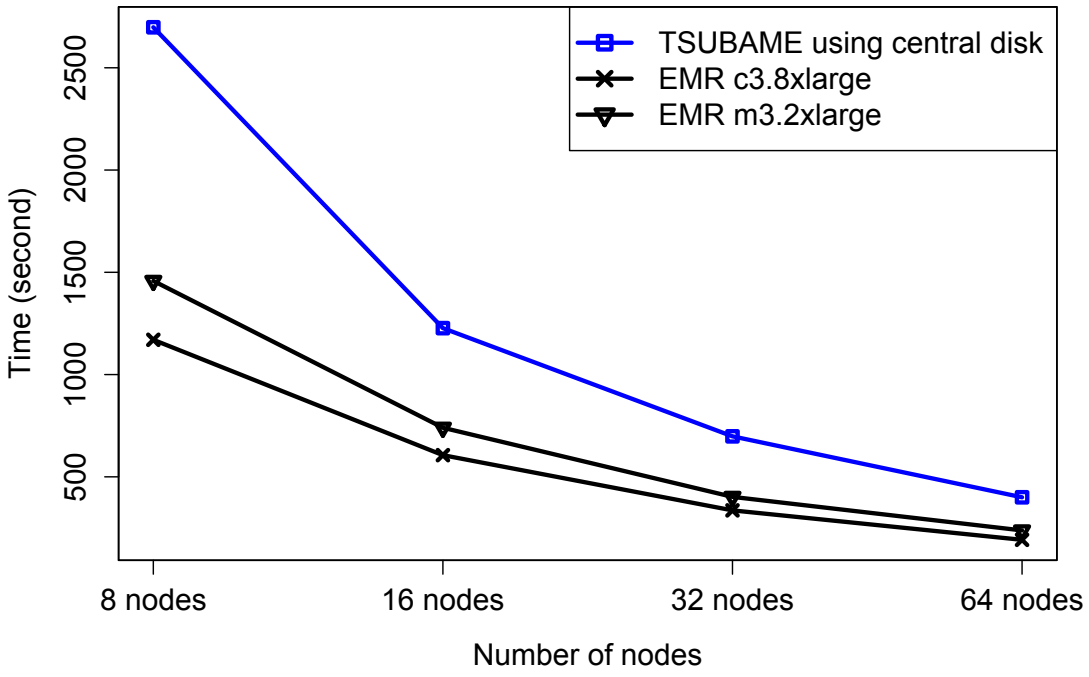


FIGURE 2.6: *Hadoop Performance on TSUBAME using its central disk and Amazon EMR*

[61], in Section 2.3.2.1, but they are different from our target. Note that the current top's supercomputers including Oakforest-PACS, K supercomputer, Cray Titan, and IBM Sequoia do not have local disks. The supercomputers described below are small-scale or already replaced by next generations.

Supercomputers are considered super-fast, but the performance of TSUBAME did not outperform Amazon EMR in the Figure 2.4, even being slower when using 64 compute nodes. The TSUBAME supercomputer has a faster interconnection and its central disk might be faster than S3 storage [62] on Amazon EC2 in which the input and output data of experiments is stored. Moreover, the SSD storage on TSUBAME is located on the same node with compute and memory, but on Amazon EC2, SSD storage might be a remote disk that is provisioned from a pool of SSD disks.

When the central disk is used to store the intermediate data, it is slow on TSUBAME as shown in Figure 2.6. In order to check the performance of the central disk on TSUBAME, we conducted an experiment by changing where the intermediate data is stored. Instead of using SSD as Figure 2.4, the central disk is used. Note that the central disk is a shared and remote storage. Figure 2.6 shows that it is 2 times slower than the EMR c3.8xlarge cluster. The reason is that disk I/O to the central disk is slower than using

the local SSD. The local SSD is not typical on the supercomputing environment and TSUBAME is a special case. In general, we should consider that the central disk is the de facto storage on supercomputers.

2.3.2.1 Data-intensive supercomputers

Although most supercomputers are designed to run compute-intensive applications, there are several ones proposed to run data-intensive applications. Some of them have local disks on compute nodes, such as the Cray CS300 Hadoop and DASH. However, local disks are fragile and will be a point of failure. It is hard to maintain local disks on every compute node, especially in large clusters having up to thousands of nodes. That reason prevents such clusters from scalability and satisfactory performance. We describe Cray CS300 Hadoop and DASH clusters in more details as follows:

The Cray CS300 Hadoop [60] provides a fast and optimized cluster served to run Hadoop applications efficiently and allows users to combine supercomputing technologies and data analysis of Hadoop. The Hadoop solution integrated in the CS300 system includes workload management and storage hierarchy. The Hadoop performance on the optimized CS300 cluster might be faster than on other supercomputers, but the hardware configuration is more complex and not efficient, for example in order to avoid using the central disk, the local disk on each node is equipped. Infiniband is supported on the cluster, but it is not clear how Hadoop can exploit MPI on such high-speed network.

DASH [29] is a prototype of data-intensive supercomputers that is equipped with SSD located between main memory and its central disk. SSD helps decrease latency when spinning to hard disks. However, this storage architecture is not scalable and still a small-scale prototype. Moreover, it is unknown that Hadoop and Spark can achieve a good performance on such platform.

2.3.2.2 Hadoop/Spark on supercomputers

There are several studies related to Hadoop deployment on supercomputers, but most of them are just using the default deployment based on TCP protocol and local disks. Note that Hadoop is one of the most popular data-intensive frameworks. We survey its deployments at some supercomputer centers as follows:

The *Scaling Spark on HPC* [63] paper reported scaling Spark on Cray XC systems. The paper considered the difficulty of using Lustre to handle the intermediate data due to the overhead in **fopen**. Its deployment solved the problem and improved scalability. The

paper also showed that there is more room to improve performance of data-intensive frameworks, such as Hadoop, on supercomputers.

Spark4MN [64] deployed Spark on the MareNostrum supercomputer without using Hadoop YARN or Mesos resource manager. The paper analyzed several deployment configurations and examined them. Its approach was only to show how to deployment, but not to exploit advantages of high-speed communication and avoid the central disk.

YARN with Lustre and RDMA [65] proposed a design of YARN that can leverage both Lustre file system and RDMA on HPC clusters. The paper considered using Lustre without local disks and RDMA-based data shuffle of intermediate data. Due to using the central disk, the overhead of reading and writing is still significant. Moreover, the existing architecture must be changed much.

HPCHadoop [66] helps run Hadoop workloads on supercomputers, especially on Cray X-series. The paper raised the problems of resource management handled by PBS [22] or Slurm [23] and no local data storage on each node. The main goal of the paper is to deploy Hadoop on supercomputers and it is a naive deployment that does not consider supercomputing resources and designs, for example how to leverage the high-speed network and avoid disk I/O to the central disk.

myHadoop [24][67] developed at the San Diego supercomputer center has aimed to deploy a private Hadoop cluster on top of the supercomputer by submitting a job script to the job queue. The Hadoop cluster is launched on allocated compute nodes. This is just a naive default deployment of running Hadoop on supercomputers. The Hadoop applications cannot exploit supercomputing resources, such as network and large memory. The default deployment still has a performance bottleneck at accessing the central disk.

2.3.2.3 MPI-based MapReduce frameworks

We survey studies attempting to build MapReduce frameworks on top of MPI as below. Note that since MapReduce is the widely used parallel programming model in data-intensive processing, it is worth investigating existing MapReduce frameworks.

Mimir [35] proposed a MPI-based in-memory MapReduce framework running on supercomputers. This framework was written in C/C++ and MPI that leverages much the supercomputing network. It also allows executing workloads in memory that helps avoid using the central disk and achieve significantly better performance. In the aspect of execution time performance, Mimir is satisfactory, but users will not satisfy in productive and mature performance since it does not provide easy-to-use programming APIs.

Moreover, Mimir was designed to support narrow applications and platforms. Users will satisfy more if there is an effective deployment of popular data-intensive frameworks, such as Hadoop and Spark, on supercomputers.

FT-MRMPI [68] is a MapReduce library written in C and MPI supporting fault tolerance. As Mimir, it could exploit MPI communication on supercomputers, but it did not provide any mechanism to avoid using the central disk. FT-MRMPI is considered difficult to write applications and it just supports narrow applications and platforms.

2.3.3 Our observations

According to our observation, the major causes of unsatisfactory performance of Hadoop-like data-intensive frameworks on the supercomputer environment are the lack of MPI-friendly dynamic process creation and local disks on each compute node. Supercomputer programs should be run simultaneously in order to minimum the network communication delay, for example MPI, but the resource manager of the existing popular data-intensive frameworks require dynamic process creation for simple scheduling and fault tolerance. However, there is no efficient dynamic process creation mechanism supporting MPI. Moreover, disk I/O to the central disk is expensive, but data-intensive applications require data accessing to the disk in order to store the intermediate data. A lack of local disks causes compute nodes to heavily access a slow central disk. Note that those Hadoop-like frameworks can still run properly on our targeted type of supercomputers, but the performance is not satisfactory due to using TCP communication and a lack of local disks.

2.4 Mismatch: MPI-friendly dynamic process creation

In this section, we present the first mismatch related to MPI-friendly dynamic process creation. Our observation is that MPI-compatible fast dynamic process creation required in the data-intensive frameworks is not supported well and not the first citizen on supercomputer environment. The data-intensive frameworks, for example Hadoop and Spark, want to use MPI, the de facto communication protocol on supercomputers, to improve performance on supercomputers and need dynamic process creation in order to keep their current architecture and minimize source code changes. The challenge is to support better dynamic process creation with MPI for the data-intensive frameworks on the supercomputer environment. The *MPI-friendly* term means the used mechanism must be compatible with MPI and fast. *MPI-Spawn* is an example of MPI-compatible mechanisms but slow.

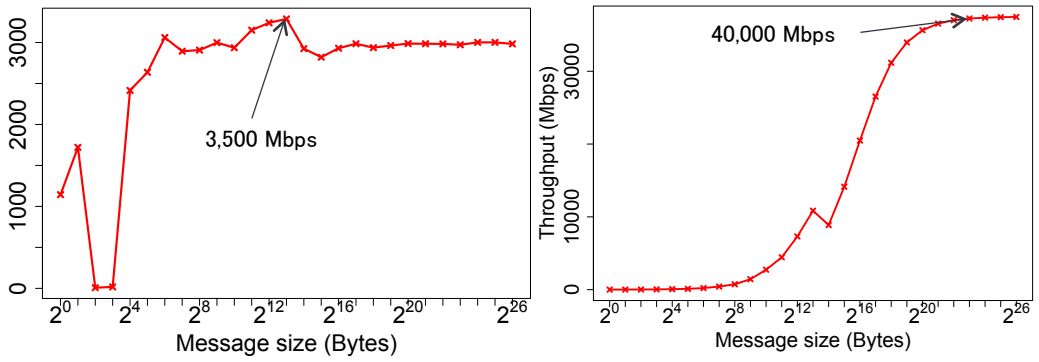
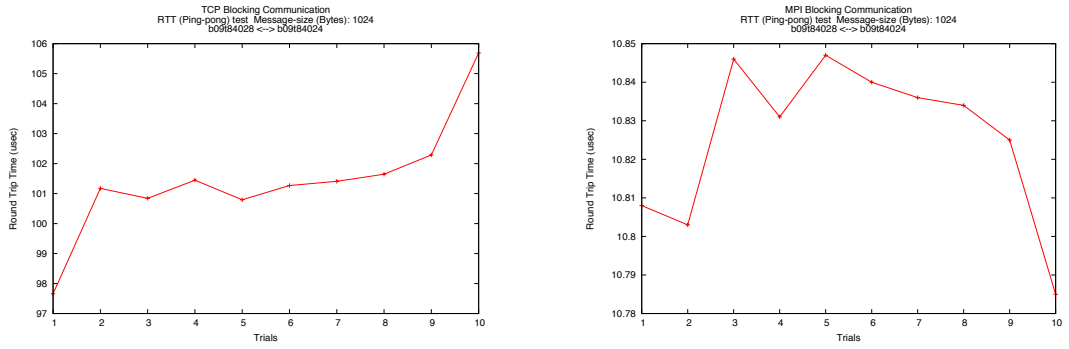


FIGURE 2.7: *TCP vs. MPI throughput on the FX10 supercomputer*



(A) TCP (B) MPI

FIGURE 2.8: *TCP vs MPI latency on the FX10 supercomputer*

To avoid confusion later between jobs on supercomputers and Hadoop YARN resource manager, which is the substrate of Hadoop and Spark, we use the *job* term for ones submitted to a supercomputer job queue and *workload/application* term for ones submitted to Hadoop YARN resource manager.

2.4.1 MPI for data-intensive frameworks on supercomputers

MPI should be used in Hadoop-like data-intensive frameworks to speed data exchange and node I/O. Socket (TCP) communication is used widely in the popular data-intensive processing frameworks, for example Hadoop, Spark, and Flink since TCP is the de facto communication on commodity clusters where those frameworks are typically deployed. However, TCP communication is slower than MPI.

On the supercomputer environment, MPI implementation is faster socket (TCP) communication. MPI should have a larger throughput and shorter latency than TCP. Figure 2.7 and 2.8 have showed that MPI throughput is 10 times larger than TCP and its latency is shorter than TCP.

As we have already seen, there are several studies related to using MPI to fill the gap between data-intensive analytics tools and scientific computation. For MapReduce over MPI approaches, MPI-based data-intensive frameworks can be found in K MapReduce [10], Mimir [35], and FT-MRMPI [68], but all of them are not popular frameworks. For the deployment of popular frameworks, such as Hadoop and Spark, *high performance design of YARN* [65] proposed how to use remote direct memory access (RDMA) to speed up data exchange and *accelerating Spark with RDMA* [69] showed a lot of potential of using RDMA. Note the RDMA is a lower layer than MPI and turned on automatically in some MPI configurations.

2.4.2 Hadoop YARN resource manager

In this section, we analyse the behaviour of Hadoop, a popular data-intensive frameworks, running on YARN resource manager. Note that YARN is also the resource manager of other data-intensive frameworks, such as Spark and Flink. Hadoop has got significant performance improvement and more flexible execution when evolving from version 1.x into 2.x by adopting YARN, a new resource manager and key feature in the second generation. YARN is responsible for allocating and de-allocating resources, managing and monitoring jobs, invoking execution engine, and ensuring security controls. We identify three *dynamic* characteristics from YARN-like management as follows:

on-Demand. Execution processes are created on demand. A job often consists of several or many e.g. mapping and reducing tasks each of which is run on a separate process container that is newly created when the task is started. Number of processes and CPU time are subject to change during job execution that depend on size of input and output data.

Diverse. It supports running simultaneously multiple jobs with fine-grained scheduling. Those jobs are scheduled to share resources on the same nodes, such as CPU time and memory, in order to maximize overall utilization. Job types are not limited to MapReduce since YARN can host Spark and Storm as well.

Detailed (fine-grained). Resources are managed by number of process containers running on each node based on amount of memory. The resource manager is responsible for scheduling, process executing, and handling failures. In MapReduce execution, when number of mapping and reducing processes is bigger than limitation of allowable container number on the cluster, it lets some processes run first, and the remainder must wait.

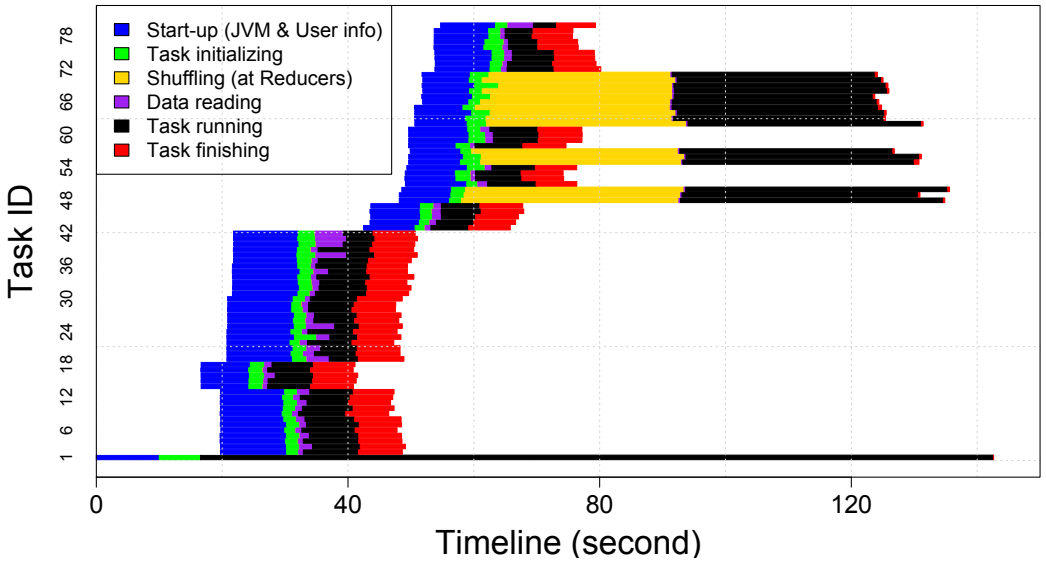


FIGURE 2.9: *MapReduce PageRank running on YARN resource manager with 64 MapTasks and 16 ReduceTasks*

Figure 2.9 shows an execution timeline of PageRank running on Hadoop and YARN resource manager with 64 MapTasks and 16 ReduceTasks: the cluster has 8 nodes; the maximum simultaneous process on each node is set to 6; reducing processes have longer running time; and the first process is a master process (namely AppMaster in YARN). Mapping processes are run first and when the first MapTask finishes, ReduceTasks will be started. Mapping processes are started at different timestamps since 42 (our configuration) is the maximum number of processes that can run at once. Each process is spawned during job execution.

2.4.3 Issue of resource management

There are three Ss (3S) *static* characteristics of typical resource managers on supercomputers that is contrary to the above dynamic management as follows:

Stationary. It is typical to start all necessary processes at the beginning of job running and there is no newly created process during execution (e.g. MPI programs). The number of processes and CPU time should be specified at the job submission.

Single. It is typical to run jobs one by one on allocated nodes. Simultaneous jobs can affect jobs' performance because resource managers only support coarse-grained manner in which each job must handle resources that it is using.

Shallow (coarse-grained). Resources are often requested by number of nodes and the whole resource on a node is allocated to a user's job. Since the resource manager does

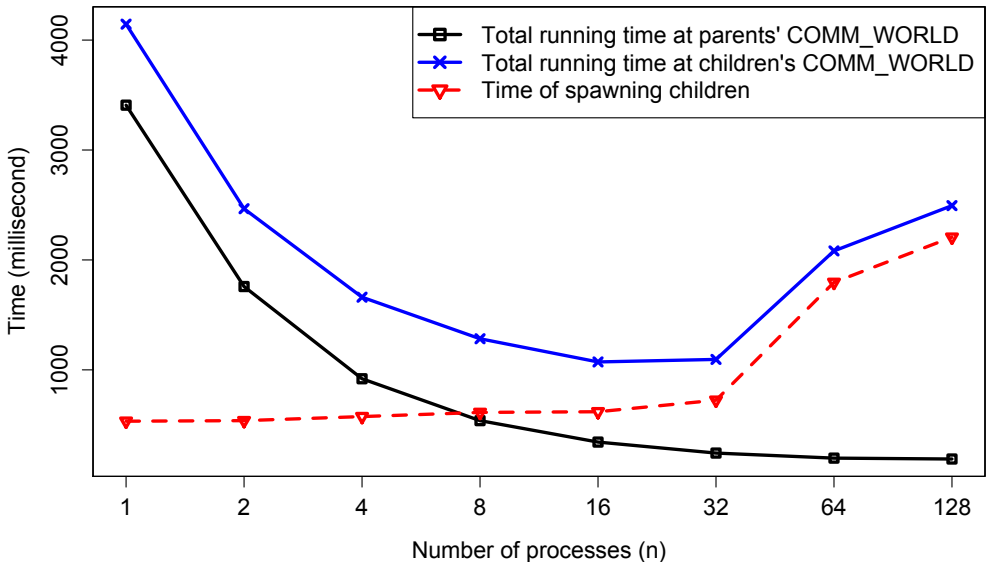


FIGURE 2.10: *MPI application (Prime count) running on parent processes versus spawned child processes on Tsubame supercomputer*

not control processes running on a node, users decide how many processes run on those nodes, how they are executed, and how to handle failures.

PBS, Slurm, and their variant are widely used on supercomputers, whereas YARN and Mesos are common on Hadoop and Spark, popular data-intensive frameworks. For example, Portable Batch System (PBS) is a popular resource manager that many supercomputer centers use as a base project to develop their own one [18]. Users must create a PBS script file in which number of nodes, MPI processes (mpi-procs), CPU core number (ncpus), and maximum memory are typically specified in advance. The PBS-like resource managers are created to support MPI style models and running coarse-grained applications. That style of static management is less flexible, but more efficient in the aspect of system administration (usage accounting and resource allocation) and maximizing performance of a job running on the supercomputers.

Dynamic management on supercomputers is supported only partially and not sufficient to be integrated with resource managers, such as Hadoop YARN. For instance, the *fork* mechanism is available to create a new process on FX10 supercomputer, but MPI will be lost on that process. By contrast, on TSUBAME supercomputer, MPI can be kept when spawning a new process by using MPI-Spawn call, but it is a collective operator that is considered slow. MPI-Spawn is a MPI call invoking new processes and MPI connection is still available in the child processes. FX10 does not allow using MPI-Spawn to spawn new processes on the same node. Figure 2.10 shows a MPI application, Prime count, running on 32 Tsubame nodes and using OpenMPI 1.6.5. In the first test case, we run the application on the *COMM_WORLD* communicator and measure total running time. In

the second one, we spawn child processes from the *COMM_WORLD* communicator and run the same program on those processes. It reveals that the spawning time is relatively long in comparison with total execution time, and it takes 0.5 second to spawn a process.

2.4.4 MPI-friendly dynamic process creation

On the supercomputer environment, MPI-compatible dynamic process creation is not the first citizen, since it can delay high-speed communication. The mismatch is that MPI-compatible dynamic process creation is needed in the dynamic management of YARN and Mesos which are the substrate of Hadoop-like data-intensive frameworks, but it is not recommended or restricted on the PBS resource manager of some supercomputers. Processes that are dynamically created on demand is called *Dynamic process creation*. Running all processes simultaneously is required in order to minimum communication delay. It is also called all-or-nothing scheduling strategy. Resources are often requested by number of nodes and the whole resource on a node is allocated to a user's job. Since the resource manager does not control processes running on a node, users decide how many processes run on those nodes, how they are executed, and how to handle failures.

On several supercomputers, it is restricting resizing running jobs due to scheduling fairness. It is typical to start all necessary processes at the beginning of job running and there is no newly created process during execution (e.g. MPI programs). The number of processes and CPU time should be specified at the job submission. It is typical to run jobs one by one on allocated nodes. Simultaneous jobs can affect jobs' performance because resource managers only support coarse-grained manner in which each job must handle resources that it is using.

Regarding the mismatch of MPI-compatible dynamic process creation, we survey related work that is having potential to solve the mismatch or indirect connection. In Section 2.1.5.1, we introduced K MapReduce that is written in C and MPI and using MPI-Spawn to create new processes. MPI-Spawn performance is slow and might increase communication delay in MPI. DataMPI mentioned in Section 2.1.5.2 is another framework implemented in pure MPI that is using both MPI-Spawn and process reusing.

Gerbil [70] showed a design of hosting unmodified MPI applications on YARN in order to avoid copying data from a cluster dedicated to running Hadoop MapReduce to another cluster used for pure MPI applications (HPC clusters). Their approach is to start MPI process containers by using MultiPurpose Daemon [71].

In the Hadoop v1 (without YARN), MapTasks can be executed in a JVM process in sequence (*mapreduce.job.jvm.numtasks*) [72]. However, this JVM process is only used

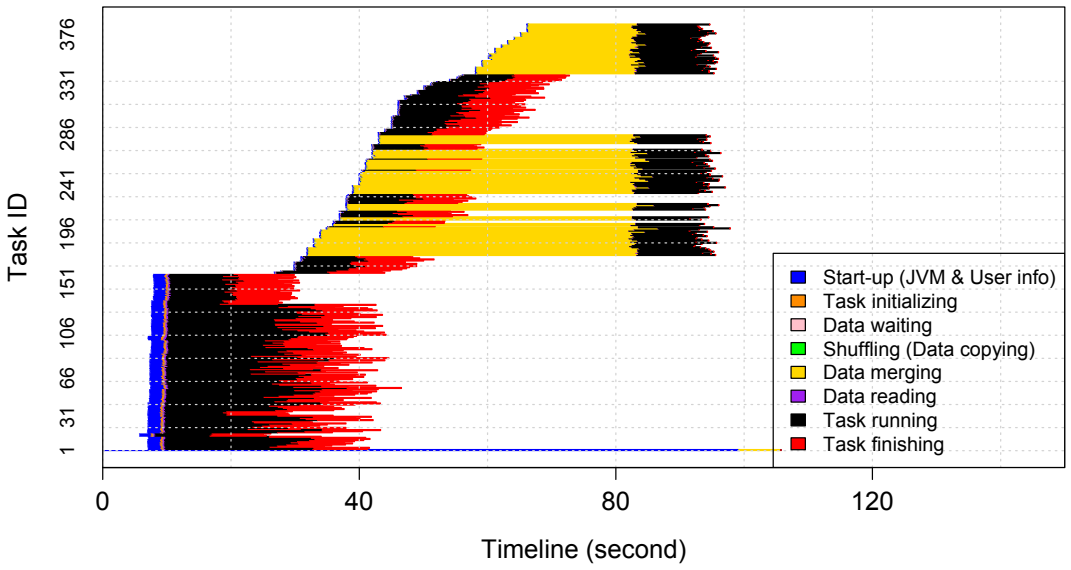


FIGURE 2.11: *Tera-sort running on TSUBAME supercomputer using its central disk: shorter running tasks are mappers, the longer ones are reducers.*

for a single workload, and it will be terminated when MapTasks finish. Hadoop v1 does not provide any mechanism to keep that JVM running to execute tasks of other jobs.

2.5 Mismatch: local disk on a compute node

The second mismatch is that data-intensive frameworks depend on local disks to spill out the intermediate data when its in-memory buffer is full, but a local disk is not available on each compute node in today's top supercomputers. Instead of using fragile local disks, a central disk with large capacity is typically equipped to store input, output, and even intermediate data if the temporary storage is not available. The challenge is to figure out the best deployment strategy of in-memory storage on supercomputers when virtual local disks are implemented using memory (DRAM).

2.5.1 Expensive disk I/O

When running Hadoop MapReduce, a popular data-intensive framework, on supercomputers, writing/reading intermediate data is a performance bottleneck since it must be stored on the central disk. There are two main phases in Hadoop MapReduce workflow: mapping and reducing. Mapping tasks generate intermediate data that is written to a

node's local disk and deleted after being sent to reducers. On supercomputers, the central disk is used to store that intermediate data instead of local disks. Figure 2.11 shows execution timeline of tera-sort application on TSUBAME supercomputer consisting of 256 mapping tasks and 128 reducing tasks. It reveals that in mapping tasks, writing time of intermediate data to the central disk (red color) is relatively long in comparison with the total execution time. In the figure, mapping tasks have shorter execution time than reducing ones. Using SSD helps improve writing/reading performance, but it is not always available.

2.5.2 In-memory approach

In-memory storage is a natural approach that helps avoid disk spilling by keeping intermediate data in memory [43] [73][74]. It is well known to implement virtual local disks by using memory (DRAM). A part of memory is allocated to store data and this storage can be considered as a virtual local disk on each compute node. That approach is sometimes called memcached-like style [30]. Memcached [33] is a distributed memory cache software that is used widely in web applications to speed up database accessing. A typical deployment of Memcached is installing its daemon on dedicated nodes used only for in-memory storage.

2.5.2.1 When in-memory storage is useful

First, as mentioned earlier, when I/O to a hard disk or shared storage is expensive and the size of data is not big, in-memory processing is more efficient. A shared storage consists of distributed disks and is considered slow due to overhead of data management and the bottleneck of connection between the storage and compute resources. Keeping data in memory requires more handling and processing, but the improvement from avoiding spilling out data to hard disks is much more that overhead. In the aspect of expense, the price of DRAM is getting close to SSD.

When a super fast I/O is required, for example in relational database management system (RDMS) and real-time analytics, in-memory processing has advantages. DASH paper [29] showed that accessing time of local DRAM memory is five-order-of-magnitude faster than to hard disks. If SSD is used, DRAM memory is still three-order-of-magnitude faster. It is not feasible for real-time database and analytics if data is spilled out to hard disks.

There are many challenges of current in-memory designs including how to deploy in-memory storage, how to make it scalable, how to fit large-scale datasets into memory,

and how to use the next storage level when memory is full. M3R [73] is an in-memory Hadoop implementation, but it requires the input dataset must be fitted into memory that makes this approach difficult to scale up. Spark is also an in-memory framework, but it will spill data to disk when there is not enough memory space. Therefore, performance can still be degraded when writing to disk happens. How big memory size is allocated to the memory space and each task process is also another research question. Each process running tasks should be allocated with big enough memory in order to not slow down computation. Therefore, the size of the memory space is subject to change when number of tasks running on a node increases.

2.5.2.2 Examples of in-memory storage implementation

Memcached software [33] is a popular in-memory implementation used widely in web and database applications. It can be installed as distributed in-memory storage [30]. In the distributed mode, there are memcached servers and clients. Any node can be a memcached client by deploying the memcached daemon on that node. Memcached software is using mainly TCP-based communication.

Redis is another in-memory storage implementation and has more features than Memcached. Redis supports complex data structures, whereas it is static and plain data in Memcached. Redis also provides distributed deployment with master and slave nodes. When we need powerful features, such as different data structures and data partition, Redis is a good choice. By contrast, if we want to keep in-memory organization simple, Memcached is a better option.

It is possible to manually create an in-memory storage in Linux by using *mount ramfs/tmpfs* [75]. There are two types of RAM disks including *ramfs* and *tmpfs*. In the *ramfs* file system, the capacity is the same as memory size and out of memory can happen at any time. By contrast, *tmpfs* can specify the size of in-memory space and this storage can be seen as a partition.

2.5.3 Deployment of in-memory storage

It is not easy to figure out the best deployment way of memcached-like file systems on supercomputers when applications are MapReduce. We focus on MapReduce rather than other application types since it is widely used in big data analytics. Whether installing in-memory storage on each compute node affects the performance of computation tasks or not. Using memory space of other nodes might cause extra overheads of management

and monitoring. Moreover, if we allocate dedicated nodes for in-memory usage only, is there any context or configuration in which it shows a good performance?

Deployment of in-memory storage is important since it can affect the performance of in-memory storage and cause the bottleneck of data movement. Installing distributed in-memory storage on the same node with computation tasks might slow down the task performance since the in-memory storage consumes compute and memory resources that depends on implementation techniques. Our hypothesis is that the overhead of in-memory storage is significant enough. Moreover, data movement among nodes by using remote in-memory storage can be also a bottleneck, especially at the nodes whose in-memory storage is accessed more frequently.

Related to answer the above question, some related work has proposed mechanisms of how to store in-memory data and choosing preferred locations. Also, there are several proposals of using in-memory storage in Hadoop, but they did not clearly describe and evaluate deployment strategies including location of in-memory instances and storage size. M3R [73] is an in-memory Hadoop engine implemented using X10 programming language and M3R instances running on each node is responsible for in-memory storage by providing a shared heap-state. Although X10 supports where data is stored through *places* and *activities* operators, but the paper did not mention it explicitly and also have any evaluation.

HaLoop [74] provides caching preferences, such as reducer input and output cache and mapper input cache, but intermediate data is only shared on the same node between jobs and deployment strategies are not relevant in this context. HaLoop does not provide in-memory storage since the intermediate data is stored at the local disk by providing efficient hash algorithms for reading and writing.

Spark [43] is a data-intensive framework and uses in-memory storage to improve performance compared with Hadoop. It proposed a programming model based on Resilient Distributed Dataset (RDD) and intermediate data is built and generated from RDDs. It is possible to choose a location for a RDD through *preferredLocations()* operator, but there was no evaluation of RDD deployment in the paper. Moreover, it does not provide deployment strategies in general.

2.6 Comparison of existing solutions

We have just introduced existing data-intensive frameworks in Section 2.1 and related work in Section 2.4 and 2.5, but there is no comparison and overview in aspects of mismatches. For that purpose, we compare advantages and drawbacks of the existing frameworks and approaches that are proposed or having potential to solve the mismatches of dynamic process creation and local disks on supercomputers. First, we define criteria for comparison as follows:

Dynamic process creation is compared by two aspects:

- Process creation: refers to which process creation mechanism is used in the frameworks. Process *fork()* is a system call to create a new process, for example ProcessBuilder in Java and pid_t pid = fork() in C. MPI-spawn call helps spawn new processes at run-time in which MPI connection is kept in those new processes. JVM reuse is a technique to reuse a process in order to avoid newly created processes in Java.
- MPI support: refers to if a framework supports MPI or not and MPI connection is kept in newly created process or not. *Might* means that framework is supposed to have MPI support, but it is not clear or under development.

The local disk mismatch is determined by two factors:

- In-memory storage: refers to whether a framework supports in-memory storage or not. Note that buffering at data writing to the disk is not considered as in-memory storage.
- Deployment strategy: refers to if any deployment strategy of in-memory storage is applied, for example how big in-memory storage size on each node is allocated and which node in-memory data is stored.

Although dynamic process creation and local disk are two main concerns to compare related work, there are other criteria used to evaluate whether a framework is good, bad, or neutral in the general usage in academia and industry as follows:

- Productivity: refers to how easy developers can write applications by using that framework. Moreover, if the framework follows API coding standards.
- Maturity: refers to how long the framework has been developed so far, how large the developer community is, and how often the framework is maintained and upgraded.

- Compatibility: refers to whether the application written by that framework can run on different supercomputers and clusters without major installation and modification or not.
- Scalability: refers to if the framework can be deployed on large clusters and easy to scale out or not.

Hadoop is written in Java and using ProcessBuilder to spawn new processes. The original Hadoop implementation does not support MPI and its process creation mechanism cannot keep MPI connection. The original Hadoop is not providing in-memory storage and deployment strategy for caching or storing intermediate data.

For the general usage, its overall rating is good since Hadoop's productivity and maturity is high. Hadoop provides ease-to-use MapReduce APIs that enable to write distributed applications less than a hundred lines of code. The initial release of Hadoop was six year ago and development status is still active with a large developer community. Hadoop application is written in Java, so it needs to be compiled once and run anywhere. Hadoop can be run cloud services, such as Amazon EC2 and Microsoft Azure, laboratory or company clusters, and supercomputers. Hadoop supports fault tolerance and scaling-out to thousand of compute node [40].

K MapReduce is written in C and MPI, so MPI_Comm_spawn is used to spawn new MPI processes that are able to communicate with parent MPI processes. There is no in-memory storage and deployment strategy implemented in K MapReduce, so intermediate data should be written to the central disk.

It is rated bad when K MapReduce is used for the general usage in aspects of productivity and maturity. K MapReduce is optimized and designed mainly for the K supercomputer [76] developed by FUJITSU. Therefore, regarding compatibility, the framework might be fast on the FX10 supercomputer, the second generation of the K supercomputer. The framework was evaluated on the K supercomputer, so it is scalable.

DataMPI is developed in Java and C with the upper layer having Hadoop-like interfaces and the lower layer using JNI in order to connect the upper one to native MPI libraries. DataMPI is using both MPI-spawn to spawn new processes for a communicator and process reusing to run different tasks. MPI communication is available among all DataMPI processes. There is no in-memory storage and deployment strategy implemented in DataMPI.

We rate DataMPI as bad as K MapReduce, since its productivity and maturity is low. DataMPI has a mixture of Java and MPI syntax that is not easy to use. The developer community of DataMPI is small. Although users can write applications in Java, but in

	Hadoop	K MapReduce	DataMPI	Gerbil	Hadoop v1.0	M3R	HaLoop	Spark	Memcached software
Dynamic process creation									
Process creation	Fork	MPI-spawn	MPI-spawn & reuse	MPI-spawn-like	JVM reuse	JVM reuse	Fork	Fork & JVM reuse	none
MPI support	No	Yes	Yes	Yes	No	Might	No	No	No
Local disks									
In-memory storage	No	No	No	No	No	Yes	No	Yes	Yes
Deployment strategy	No	No	No	No	No	No (data affinity)	No	No (data affinity)	No
Other criteria									
Overall	Good	Bad	Bad	Neutral	Neutral	Neutral	Neutral	Good	Good
Productivity	High	Low	Low	None	High	High	High	High	None
Maturity	High	Low	Low	High	High	Fair	Fair	High	High
Compatibility	Fair	Low	Fair	Fair	Fair	Low	Fair	Fair	Fair
Scalability	High	Fair	Fair	Fair	Fair	High	High	High	High

Table 2.2: Comparison of existing solutions

order to use MPI, they need to compile mpiJava [77] for their environment. DataMPI can achieve linear scalability.

Gerbil enables pure MPI applications on YARN and spawns MPI processes by using MultiPurpose Daemon [71], a MPI-spawn-like mechanism. MPI communication is available on newly created processes. There is no in-memory storage and deployment strategy managed by Gerbil.

Gerbil is an extension of the YARN resource manager, so its maturity is high and compatible with Hadoop, Spark, and Flink. Scalability of MPI applications on Gerbil has not been evaluated.

Hadoop v1.0 supports running many tasks in the same JVM that is called JVM reuse technique. However, this JVM process is only used for a single workload, and it will be terminated when MapTasks finish. There is no MPI communication, in-memory storage, and deployment strategy available in this version.

Hadoop v1.0 support is still active, so its maturity is high. The syntax of MapReduce applications in this version is the same as the current Hadoop version, so its productivity is considered high as well. Hadoop v1.0 does not use any good resource manager, so the size of the cluster in which Hadoop v.10 can be deployed is limited. Its scalability is considered neutral. Overall, we rate it neutral as well.

M3R is using JVM reuse technique to run mapping and reducing tasks in the same JVM. MPI communication is possible in M3R, but it is not clear that how MPI is supported. M3R is written in X10 and has a Hadoop-like API syntax. M3R enables in-memory storage in the heap space of JVM and X10 supports where data is stored through *places* and *activities* operators that we call *data affinity*.

M3R has high productivity since it provides a Hadoop MapReduce-like syntax. M3R is developed by IBM and still active, but its developer community is small. Since M3R is written in X10 and an independent Hadoop implementation, the clusters and supercomputers where M3R is deployed must support X10. That is why we rate it low in the aspect of compatibility. The framework is scalable on thousands of commodity nodes.

HaLoop is based on the Hadoop engine, so it is also using ProcessBuilder and does not support MPI communication. HaLoop provides caching preferences, such as reducer input and output cache and mapper input cache, but intermediate data is only shared on the same node between jobs and deployment strategies are not relevant in this context.

HaLoop keeps the same the Hadoop MapReduce syntax, so its productivity is high. However, HaLoop is relevant to iterative MapReduce applications, so its community is small. HaLoop is as scalable as Hadoop.

Spark depends on YARN and Mesos resource managers to spawn new applications and processes. Besides using ProcessBuilder to spawn a new executor for tasks, multiple tasks can be also run on the same JVM. MPI communication is not available on Spark. Spark offers in-memory storage for its RDD data model and it is possible to choose a location for a RDD through *preferredLocations()* operator that we call *data affinity*.

The overall rating for Spark is a good framework due to its productivity, maturity, and scalability. Spark applications can be written in Java and Scala and its syntax is flexible and ease to use. The lines of code in Spark applications should be shorter than the Hadoop MapReduce applications.

Memcached software does not have any process management since it only provides in-memory storage. Memcached uses TCP-based communication and RDMA-based Memcached [78] is an extension that speeds up internode communication by using RDMA. MPI communication is not supported in Memcached software. It does not support deployment strategies clearly, but users can choose explicitly where can install the memcached daemons.

The rating for maturity and scalability is high the software is still well supported from industry and developers. The Memcached software is proved highly scalable at Facebook by being deployed on a thousand of servers and million of users [79].

2.7 Effective deployment

The comparison table 2.2 has showed that in aspects of productivity and maturity, Hadoop and Spark are good choices to run data-intensive applications on supercomputers in comparison with developing application frameworks, for example a new MapReduce library, from scratch. Both frameworks are used widely on commodity clusters and in industry. However, the performance of the default deployment is not satisfactory due to mismatches on dynamic process creation and local disks. For more effective deployment of the data-intensive frameworks, such as Hadoop, those mismatches have to be mitigated, in particular, with minimum source code modification of the frameworks.

Effective deployment is required to run the popular data-intensive application frameworks on supercomputers in order to exploit supercomputers' advantage, reduce burdens on hardware upgrade, and keep the original codebase of those frameworks as much

as possible. In this thesis, process management (dynamic process creation) and data storage (local disks) are chosen to be optimized because of the reasons of keeping fast inter-connection and avoiding expensive disk I/O on the supercomputer design.

The effective deployment should help exploit supercomputers' advantage including fast interconnection and large memory size and avoid the drawback of no local disks on almost all supercomputers. The data-intensive frameworks depend on mainly TCP-based communication protocol, but that protocol is not supported well on the fast interconnection on supercomputers. The effective deployment should enable supported protocols on supercomputers, such as MPI and RDMA. Due to no local disk, the effective deployment should prevent disk I/O as much as possible.

The effective deployment should help minimizing the cost of changes in the architecture of data-intensive frameworks and keep the original codebase as much as possible. Hadoop and Spark have a huge codebase, changing their architecture is not easy and might affect the performance later. The effective deployment should provide a para-virtualization environment to run those data-intensive frameworks.

The effective deployment could reduce the start-up time of cluster deployment on supercomputers. On a commodity cluster, when Hadoop is deployed, the cluster is always ready to run applications. However, on supercomputers, Hadoop cluster must be deployed from scratch.

Chapter 3

Virtualizing Dynamic Process Creation

Regarding the first observation of MPI-friendly dynamic process creation on supercomputers, the challenge is that how to provide MPI-friendly dynamic process creation for popular data-intensive frameworks, such as Hadoop and Spark, but satisfy the standard way of creating processes on the PBS-like resource manager of supercomputers.

We want to extend those popular frameworks to use MPI, the de facto communication protocol on supercomputers, to improve performance. Those Hadoop-like data-intensive frameworks need dynamic process creation in order to keep their current architecture and minimize source code changes, but MPI-compatible fast dynamic process creation is not available. To overcome that difficulty, in this section, we propose HPC-Reuse, an efficient process creation for running MPI and our focused Hadoop MapReduce on supercomputers. Evaluation and discussion are shown at the end of this chapter.

3.1 HPC-Reuse: virtualizing dynamic process creation

To provide better support of dynamic management with MPI, we propose a virtual layer located between YARN-like and PBS-like resource managers on supercomputers. In order to avoid process creation and satisfy resource specification at the beginning of job running, we create a bunch of processes in advance and then allocate them to the dynamic resource manager when it requests process execution. These processes are cleaned and de-allocated back in order to be used next time when they finish. We use a process pool to implement this mechanism. We name it HPC-Reuse designed specially for Hadoop YARN hosting JVM-based applications, such as MapReduce, Spark

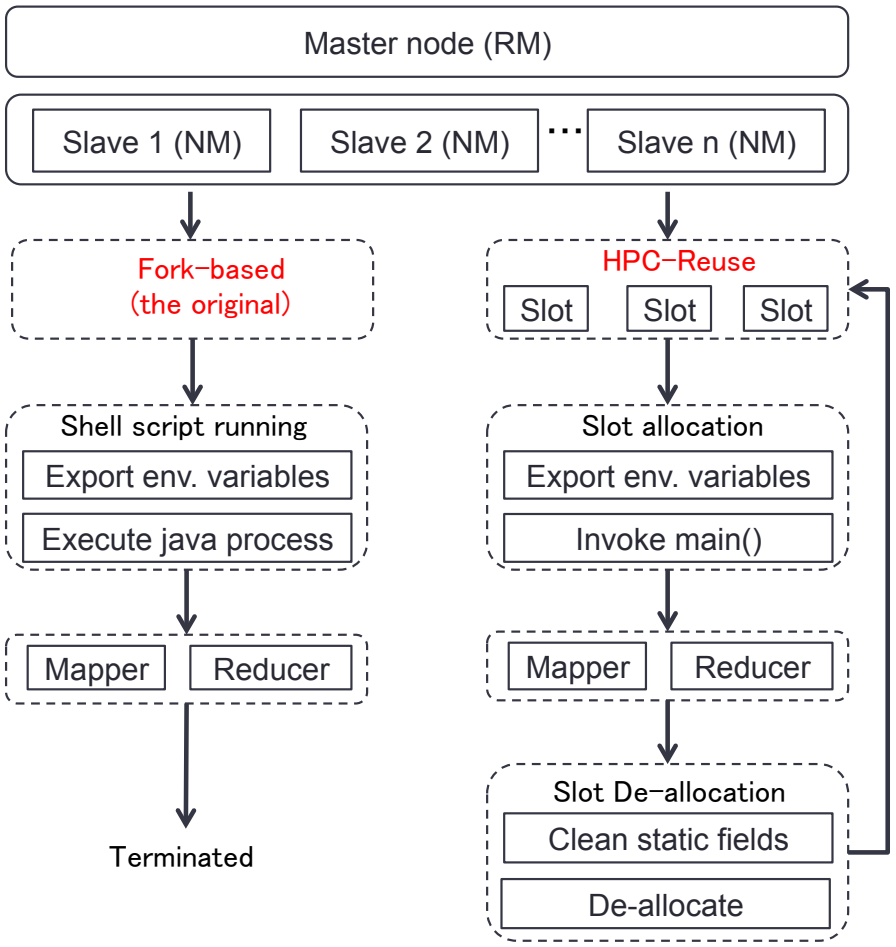
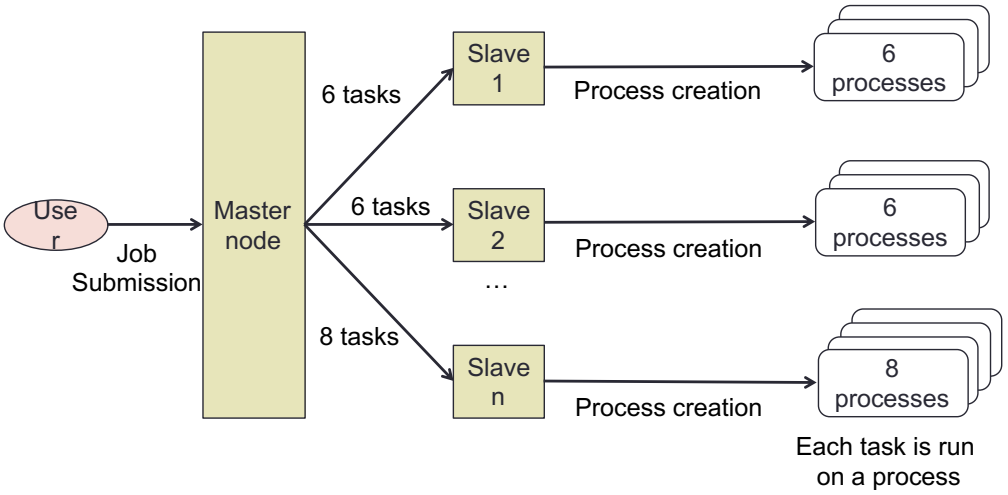


FIGURE 3.1: *Fork-based vs. HPC-Reuse workflow: slots denote JVM processes; RM and NM stand for ResourceManager and NodeManager, respectively.*

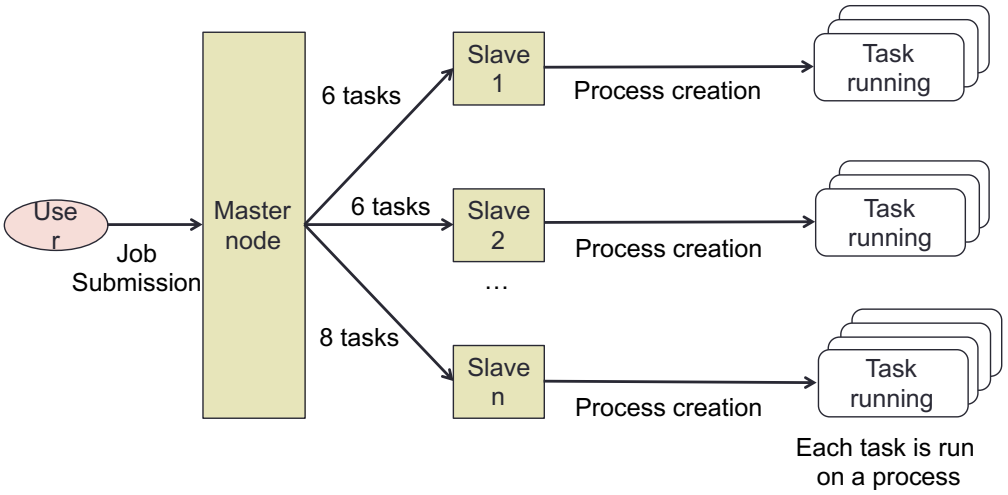
and Storm, running on supercomputers. HPC-Reuse plays a role as a virtual layer in providing dynamic resource management over static one on supercomputers. In this section, first we describe the idea of reusing applied to YARN and show its design. Technical issues are followed.

3.1.1 Idea of Reusing

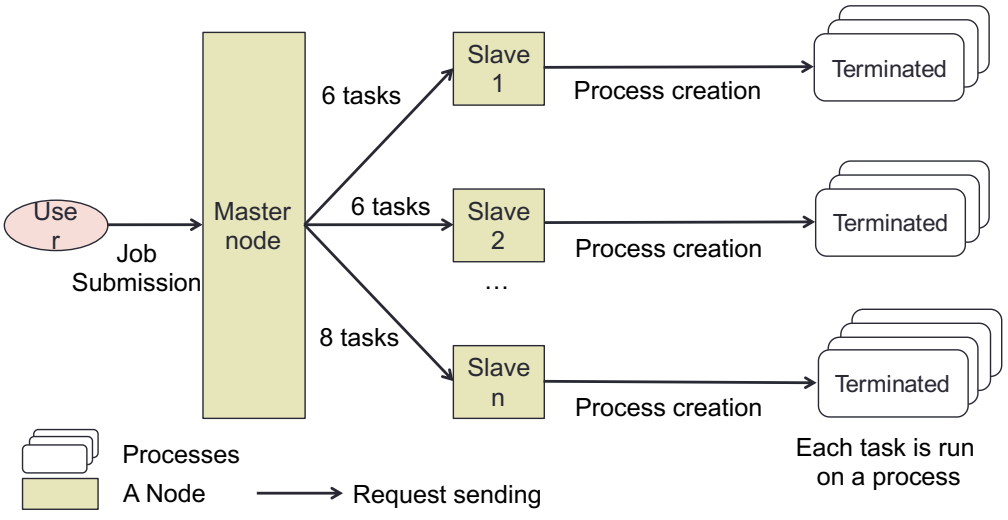
YARN creates process containers on demand when it receives requests from workloads. *Container* is used as abstraction for resource allocation. It has two main components: *ResourceManager* (RM) and *NodeManager* (NM) running on master and slave nodes, respectively. RM is responsible for allocating resources to workloads and scheduling, and NM is responsible for executing containers and monitoring them on a node. It is typical to have one RM. NM uses the fork-based mechanism to spawn new JVM process containers when it receives task running requests (e.g. MapTasks and ReduceTasks)



(A) Process creation



(B) Task running



(C) Task finishing and process terminated

FIGURE 3.2: *How processes and tasks are run in the original design*

from a workload. Figure 3.1 (left) shows the creation flow. First, a shell script containing a MapTask or ReduceTask is prepared in advance. Then, it is invoked to export environment variables and run the process container (called Mapper or Reducer). When the assigned task is completed, the process container is terminated.

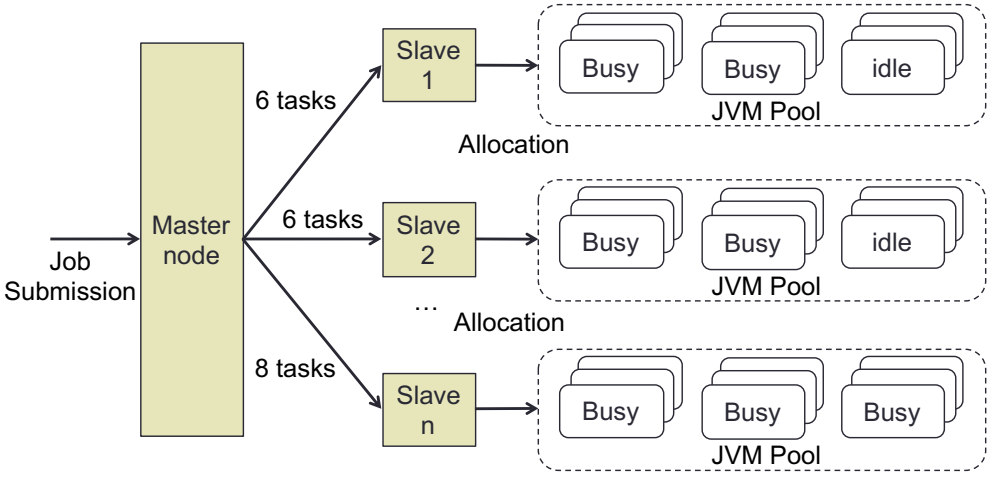
To avoid process creation, we keep JVM process containers running without being terminated. A pool of empty JVM process containers is responsible for container allocation and de-allocation to each execution request. Figure 3.1 (right) illustrates this approach (called HPC-Reuse). When NM receives a container request, it sends that request containing the shell script path to HPC-Reuse. Then, a slot is assigned to run the shell script and invoke the *main()* of the Java program. When the assigned task is completed, that slot is cleaned and returned back to the pool.

Figure 3.2 shows how processes and tasks are run in the original design. The slave daemon running on each node first spawns new processes (Figure 3.2a) and tasks will be executed on processes (Figure 3.2b). Then, when tasks finish, the processes will be terminated (Figure 3.2c). Figure 3.3 shows our idea of how processes will be created and tasks are run. On each slave, a pool of processes is created in advance. When tasks are available, processes from the pool will be allocated to the tasks (Figure 3.3a). Then, they are run on the given processes (Figure 3.3b). When tasks finish, the processes are kept running and cleaned in order to reset to the original condition. All reset processes are set idle and ready to execute other new tasks (Figure 3.3c).

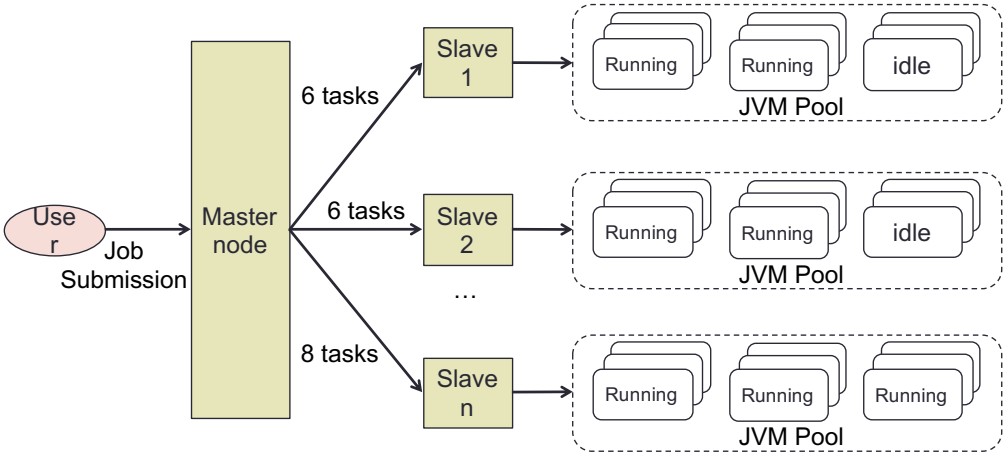
3.1.2 Process pool

We use process pool approach to implement HPC-Reuse. A pool containing empty JVM processes (namely slots) is created on each node at the beginning when the Hadoop cluster is deployed on the supercomputer. Note that the number of slots is fixed and unchanged during job execution. Figure 3.4 shows the pool architecture on a slave node where a NodeManager is running. Container requests are forwarded from NM to *Pool Manager* instead of calling a process builder (fork-based approach). A flag array is used to store slot status in the pool. We use round-robin scheduling to choose an empty slot. A container request is held until there is a slot available.

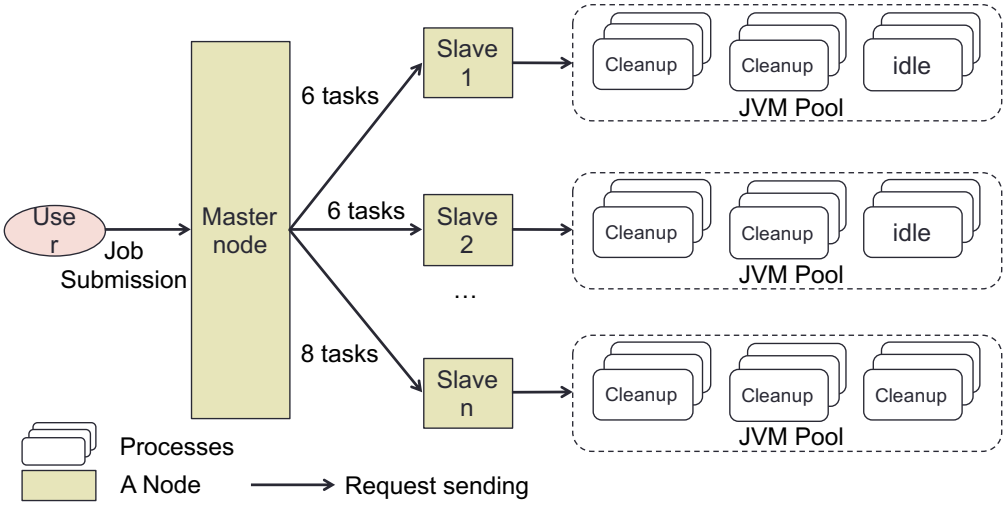
When a slot is assigned to a container request, first, environment variables are exported. Then, a new class loader is created to load user classes. Finally, *main()* method is invoked to run the associated task. De-allocation including static field clean-up and slot resetting will be called after the task is completed.



(A) Process creation



(B) Task running



(C) Task finishing and process cleaned

FIGURE 3.3: *How processes and tasks are run in the HPC-Reuse design*

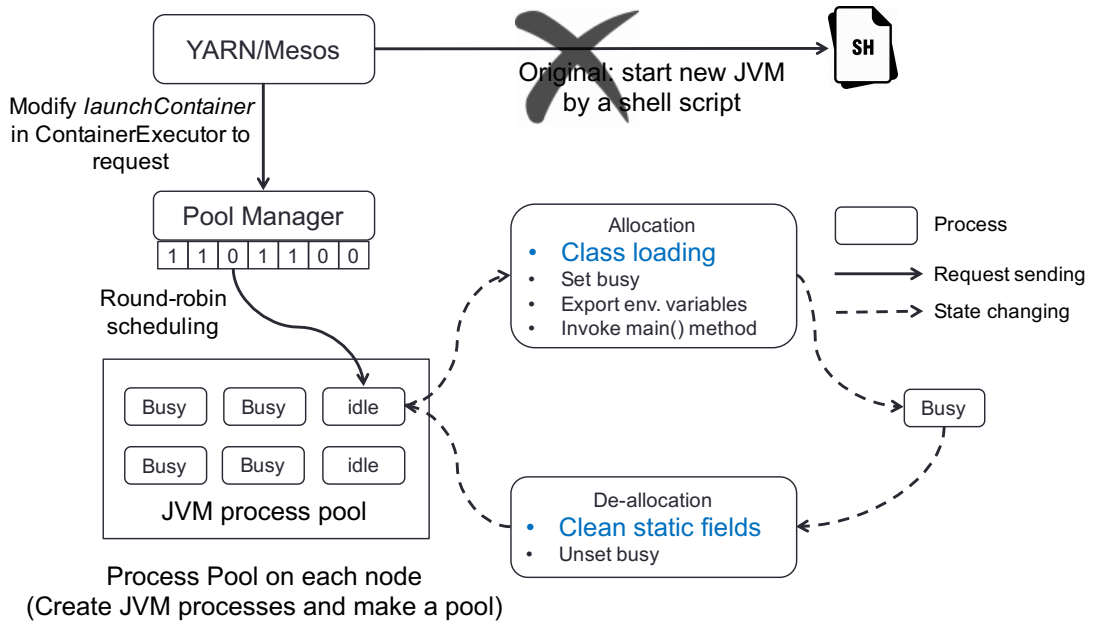


FIGURE 3.4: *Pool architecture in a node: one represents an occupied state and zero represents an empty state*

3.1.3 Technical issues

There are three issues discussed in this section including class loading, clean-up, and the number of slots in the pool. In order to make allocation and de-allocation possible and safe, there are two issues: how to load user classes and clean a slot. Note that the two issues are problematic for JVM-based applications running on YARN, such as MapReduce, Spark, Flink, and Storm.

3.1.3.1 Class loading

We use a new class loader to load user classes when a user submits a MapReduce workload to the Hadoop cluster. In the original flow of process creation (fork-based approach), *CLASSPATH* containing user classes is exported before container execution, so the container can find and load such classes. *CLASSPATH* is defined before a JVM process is started and classes in the *CLASSPATH* are loaded by the *application* classloader. In HPC-Reuse flow, however, slots in the pool are started before the *CLASSPATH* is exported and thus the user classes are not found.

Figure 3.5 shows how classes in Java are loaded at run-time. The top class loader is the *bootstrap* one that is the parent of all class loaders. This class loader is responsible for loading standard Java class files from rt.jar, for example String.class and Integer.class. Then, other extended classed located at *JRE/lib/ext* are loaded by the *extension* class loader. Hadoop classes declared in *CLASSPATH* are loaded by the *application* class

loader. When the Hadoop cluster is deployed and the pool on each node is initialized, the *application* class loader starts loading all necessary Hadoop classes. Note that a class is actually loaded only when it is declared and used at run-time. A class already loaded will not be loaded again. When a class is not found, it will be looked for in rt.jar first, and if the class is not there, the *extension* class loader starts finding in *JRE/lib/ext* directory. The next class loader in the Figure 3.5 will look for when the class is not found in the parents.

User classes are not included in the *CLASSPATH*, so they will be loaded during run-time when users run their workloads. When the user submits *workload1*, a new class loader named *classload1* is declared and will load classes of *workload1*. This class loader is a child of the *application* class loader and inherits all classed loaded by the parent class loader. When the user submits another workload, namely *workload2*, a different new class loader named *classload2* is declared and will load classes of *workload2*. Note that the classes of *classload1* are not available in *classload2*.

At the step of slot allocation in HPC Reuse workflow, a new class loader is created to load the user classes before invoking its *main()* method. If the user submits the same class or package that exists in the previous workload, an error will not happen. A class

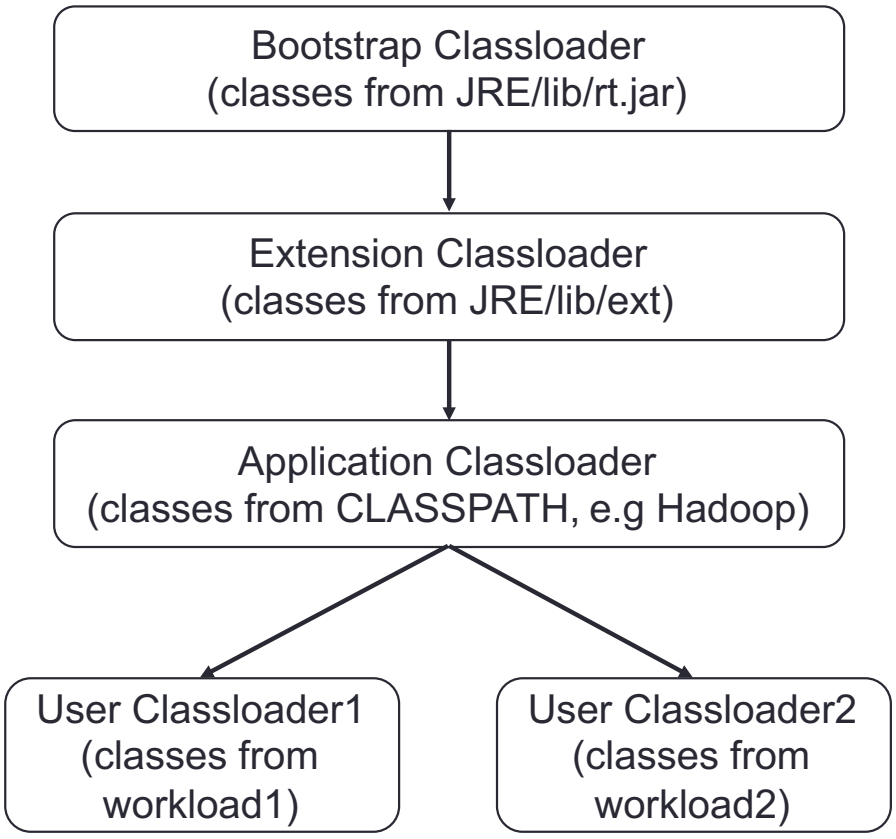


FIGURE 3.5: *How user classes are loaded*

loader is created newly for each workload. However, note that we do not reload all Hadoop classes, and only the user classes are reloaded. That helps reduce class loading time and exploit compilation technology in JVM. User classes are often small, so it does not take a long time to reload user classes. They should be mapping and reducing classes in the Hadoop MapReduce framework.

The current design of class loaders is working for Hadoop MapReduce, but the mechanism should be the same for Spark and Flink since they are also JVM-based frameworks. Spark is written in Scala that also provides ClassLoader.scala to declare a new class loader.

Another possible approach not requiring different class loaders is using a Java bytecode manipulation tool and just check whether the new workload contains any the same class name or not. If the same class name is detected, the old one can be replaced at run-time.

3.1.3.2 Clean-up

In order to reuse a JVM process and run the next task correctly and efficiently, the JVM process must be cleaned up after de-allocation and returning to the pool. HPC-Reuse may have problems of security and performance due to reusing the JVM in which static fields for the previous workload are already set. In Java, a static field is a variable and located in a class, but it is not in an instance of the class. Although there are many instances of the class created, there is only one static field existing in the class. There are two kinds of static fields including *static field* and *final static field*. While *final static field* is used to define constants in the class, *static field* is used to share values among instances. A static field can be accessible from any class of the scope.

```
/**
 * Information about the logged in user.
 */
private static UserGroupInformation loginUser = null;
private static String keytabPrincipal = null;
private static String keytabFile = null;

/**Environment variable pointing to the token cache file*/
public static final String HADOOP_TOKEN_FILE_LOCATION =
"HADOOP_TOKEN_FILE_LOCATION";
```

LISTING 3.1: An example of static field usage in UserGroupInformation of Hadoop

An example of static field usage in UserGroupInformation class of Hadoop [80] in Listing 3.1 is that when the container is executed, UserGroup static field is initialized. That field is kept unchanged whenever its value is not *null*. Therefore, the same login static field is used for other users. Although on the supercomputer, Hadoop cluster is used by only

one user at a time, but clean-up of all static fields is necessary in general. Moreover, the configuration of two different workloads might be not changed since the static fields of Configuration are set. That will cause wrong behavior and worse performance.

At the current implementation, we just do a simple clean-up by resetting only static fields containing user information and workload configuration. First, we manually check which classes are loaded and which static fields are set when a workload is executed. Then, we record those static fields and will reset them to null value when a new workload is executed. The drawback of this approach is that we cannot detect all static fields since the workload might invoke different classes. Moreover, when upgrading Hadoop and Spark, the checking of static fields must be repeated. Note that only static fields changed during loading the previous workload will be reset to null value.

Another possible approach is to detect all static fields associated with the execution container by using Java bytecode manipulation tools, for example Javassist [81]. By using those tools, all static fields can be detected during at run-time and reset automatically.

3.1.3.3 Number of slots in the pool

Node performance could be affected if the pool has lots of slots since idle JVM processes still consume CPU time for waiting requests from *Pool Manager* and memory for Hadoop classes. 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 the number of processor cores.

When the Hadoop cluster is deployed and the pool is initialized, the number of slots in the pool is kept unchanged during the cluster running. That drawback limits the scalability of our proposed HPC-Reuse when we want to increase the number of parallel tasks that are able to run on a node. Although increasing the number of slots helps improve parallelism, but the performance can be degraded since compute, memory, network bandwidth resources are limited.

3.1.4 Performance benefit

As mentioned in the motivation section, MPI-Spawn is a mechanism to spawn a new MPI process, but its performance is slow. Note that nested MPI jobs are not allowed on most of implementation of MPI on supercomputers. Figure 3.6 illustrates MPI-Spawn's collective operation: rounded rectangles represent spawning call; rectangles denote processes; dotted rectangles show waiting states. When spawning is invoked on

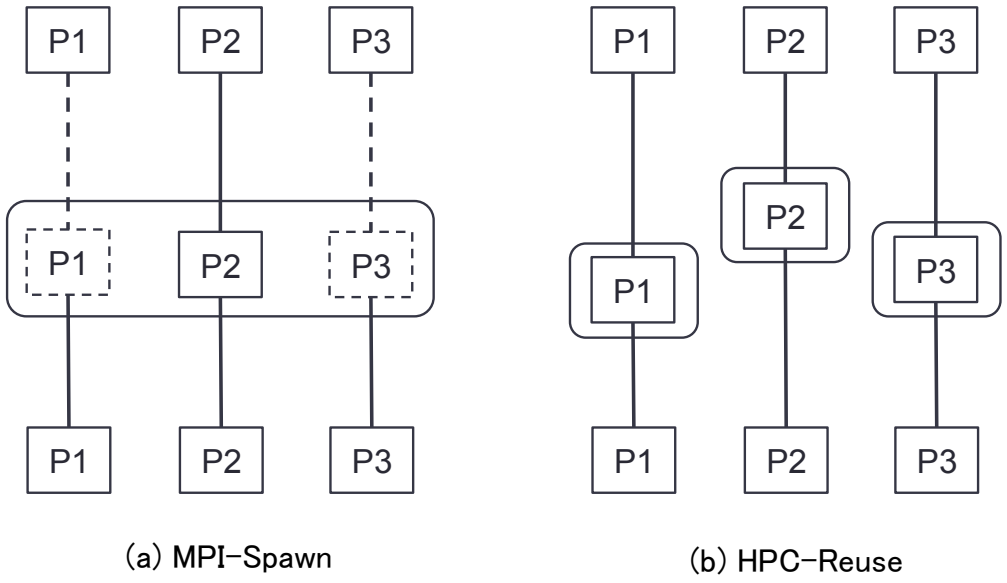


FIGURE 3.6: *MPI-Spawn vs. HPC-Reuse in process creation: rounded rectangles represent spawning call; rectangles denote processes; dotted rectangles show waiting states.*

the *COMM_WORLD* communicator at a certain process, it requires all other processes must call spawning with the same parameters (Fig. 3.6(a): P1 and P3 must stop their execution while P2 is calling). Each spawning call must be serialized. Dotted rectangles show the waiting state. Conversely, HPC-Reuse supports multiple threads and does not use MPI-Spawn, so each process can request creating as many new processes as at any time (Fig. 3.6(b)). HPC-Reuse keeps those processes running without terminating, so MPI connection is always available during job execution time.

Not only helping avoid MPI-Spawn, HPC-Reuse also shortens JVM start-up time. The JVM start-up flow is divided into four main steps: OS-level creation, class loading, main() invoking, and instruction execution. HPC-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 3.7 illustrates the flow of JVM start-up: JVM process executing A program is reused to run B program. Moreover, HPC-Reuse takes advantage of compilation technology, such as Just-in-time (JIT) compilation and adaptive optimization that are designed to improve execution performance.

Iterative MapReduce workloads can benefit much from HPC-Reuse since it helps shorten start-up time. An iterative workload is run consecutively until it meets a condition or its value converges. Figure 2.3 of the previous chapter describes how an iterative workload works. The result of previous MapReduce iteration is used as the input to the next one. For each iteration, mapping and reducing phases repeated cause lots of processes created. HPC-Reuse reduces the start-up time of those processes.

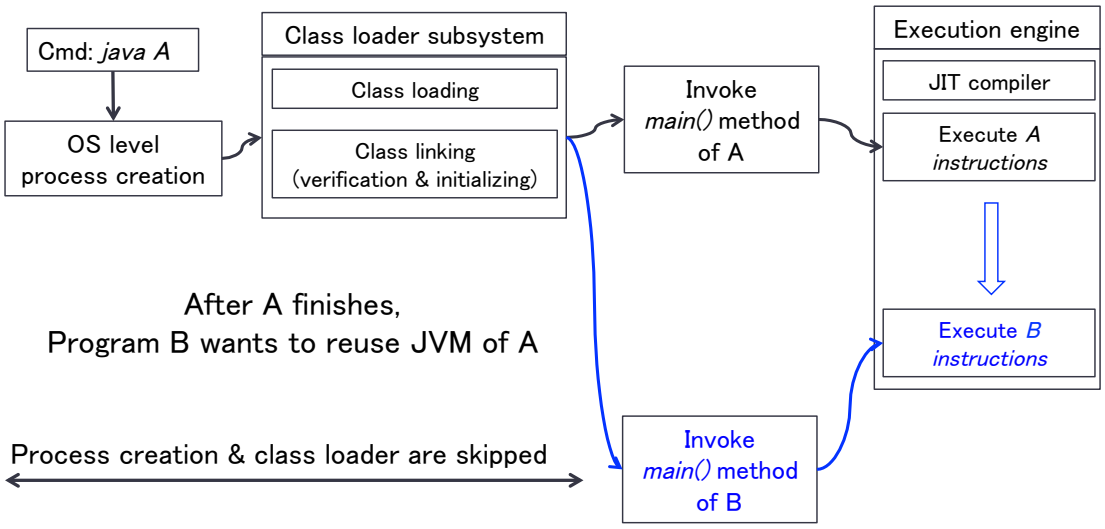


FIGURE 3.7: *JVM Reuse of a program*

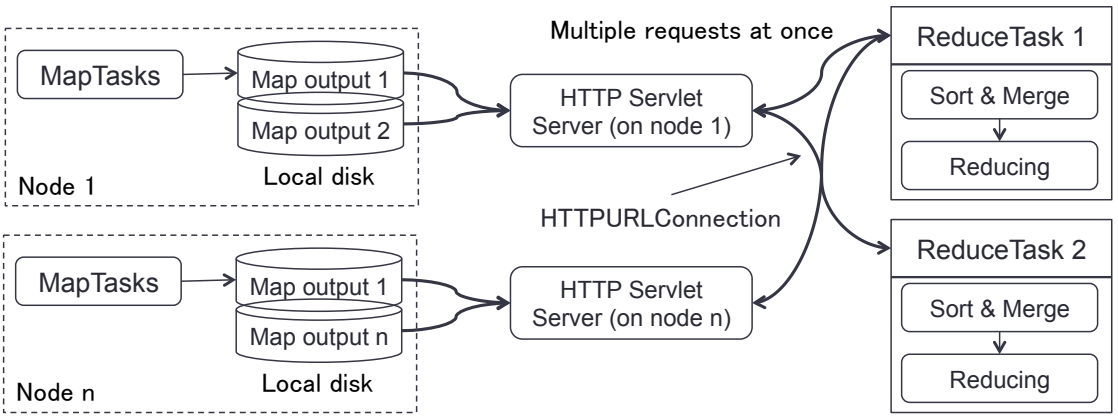
3.2 Other benefits of HPC-Reuse

HPC-Reuse not only solves the problem of dynamic management and shortens start-up time, but MPI integration on HPC-Reuse also gives other important advantages. For Fujitsu FX10-like supercomputers that do not allow dynamic process creation (MPI-Spawn) on the same node, HPC-Reuse enables MPI on Hadoop processes and makes YARN possible to host pure MPI applications. For TSUBAME-like supercomputers on which MPI-Spawn overhead is high, HPC-Reuse helps avoid it and also host MPI programs more efficiently in the aspect of start-up time overhead. By virtue of fine-grained management manner in YARN, multiple workloads can be scheduled to run simultaneously and more efficiently. Furthermore, in rich data analysis workflow, post processing after MapReduce tasks enables YARN to host pure MPI applications and efficient data copying between MapReduce and pure MPI applications.

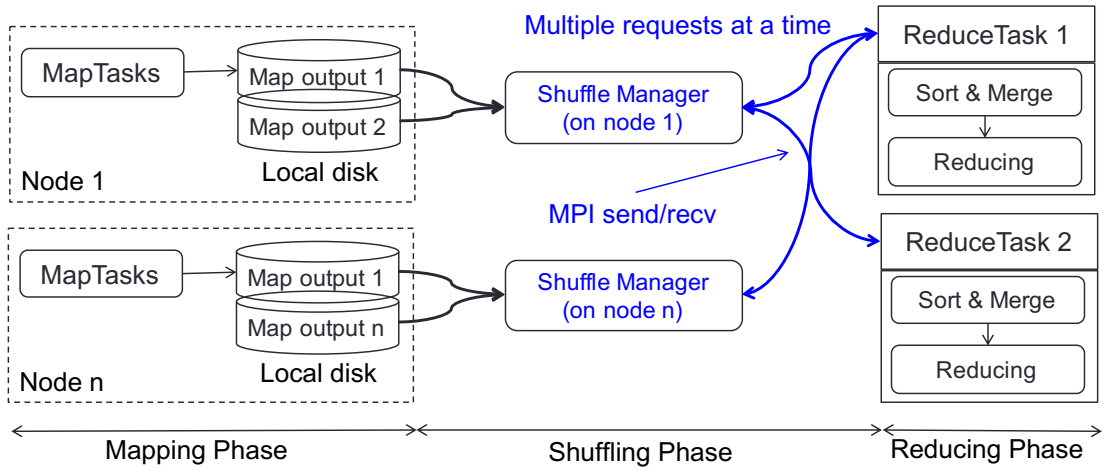
3.2.1 MPI communication over Hadoop processes

MPI connection is always available among processes in the pool since HPC-Reuse keeps those processes running without terminating. MPI communication is established among processes at the beginning when the Hadoop cluster is deployed. In the pool, we create an intra-communicator for all execution slots. Pool Manager and Node manager do not belong to that intra-communicator, but COMM_WORLD of course.

Data shuffle performance in HMR is slow since it is using TCP-based communication and JVM's transport protocols [82]. It is specially slow in workloads having high volume of data exchange, such as tera-sort and self-join. Its performance can be improved by



(A) TCP/IP-based shuffling



(B) MPI-based shuffling

FIGURE 3.8: *TCP/IP vs. MPI shuffling*

using MPI instead of TCP. In order to show benefit of MPI communication, we have implemented a MPI shuffling engine to bypass using the original shuffling based on TCP communication.

Figure 3.8 shows comparison between the TCP-based shuffle engine and MPI-based shuffle engine. HTTP servlet server is replaced with our *Shuffle Manager* to handle receiving requests and sending MapOutput data. *Shuffle Manager* process on each node belongs to COMM_WORLD communicator. Rank of each *Shuffle Manager* is stored in the pool. HTTP servlet server can handle multiple HTTPURLConnections from Reducers at once (non-blocking type), and our *Shuffle Manager* can also receive multiple requests at a time. The detail design of our *Shuffle Manager* is described in Section 4.4.1 of Chapter 4 since it requires in-memory storage.

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. 3.8a). Each slave node has one HTTP servlet server that can handle multiple HTTPURLConnections from Reducers at once (nonblocking type). HTTPURLConnection is a library class based on TCP/IP. In the same way, our shuffle engine does receive requests from Reducers, then read MapOutput files, and send back to the Reducers. Nevertheless, we use MPI connection instead of HTTPURLConnection.

3.2.2 Post processing after MapReduce tasks

After MapReduce tasks, post processing, for example MPI-based Support Vector Machine (SVM) training, helps increase job diversity and avoid data copying between MapReduce and post processing tasks. We present two benefits as follows:

3.2.2.1 Enabling YARN to host pure MPI applications efficiently

Pure MPI applications are typical on supercomputers, especially those that require compute-intensive computation, such as simulation, modelling, and machine learning. Although MPI applications can be submitted as typical jobs to the job queue, running them on YARN makes resource utilization more efficient since YARN is a fine-grained resource manager. Moreover, the combination of Hadoop MapReduce and pure MPI applications also enables rich data analysis workflows. So far, Gerbil [70] is a solution to run both pure MPI and MapReduce on YARN, but its start-up overhead is significant for short running applications.

A YARN application often consists of three main components: YarnClient, AppMaster, and YarnChild that are shown in Figure 3.9. First, YarnClient sends a request including execution contexts and application classes to *ResourceManager* (RM) in order to create a new application and start AppMaster. Then, RM leaves the control of the application to its AppMaster. After that, AppMaster requests execution containers from RM and asks assigned *NodeManagers* to start those containers. YarnChild is a wrap of the application that is executed in containers.

We create MpiClient, MpiAM, and MpiChild, respectively, for running pure MPI applications. MpiClient is responsible for replacing the COMM_WORLD communicator with our YARN_COMM_WORLD by using a bytecode manipulation tool. It then packages the application's classes and sends to RM. After that, MpiAM is started. MpiAM requests containers with a special flag that denotes a MPI application that *Pool Manager* will use to create an intra-communicator for assigned slots. We use MPI_Comm_split(color, key) to create that intra-communicator. Its key is the current rank and its color is 1

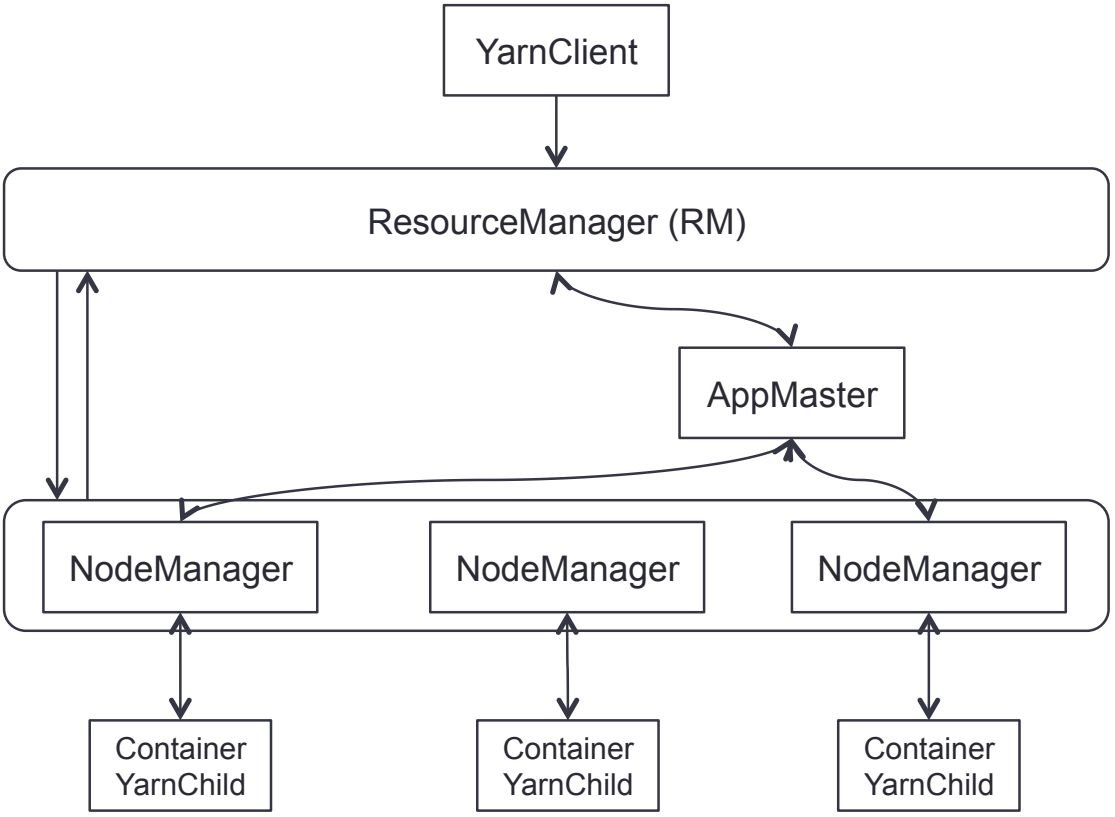


FIGURE 3.9: *A YARN application often consists of three main components: YarnClient, AppMaster, and YarnChild.*

for assigned slots, but otherwise 0 for the remainder. We assign that communicator to YARN_COMM_WORLD.

Since the MPI application is executed on JVM processes, so our assumption is that the application is written in Java using a MPI binding. Moreover, due to creation of an intra-communicator for each MPI application, our design can host only one MPI application at a time.

Protection of communication is required when MPI program is running on HPC-Reuse because all processes in the pool belong to the same COMM_WORLD. We avoid it at the step of replacing the COMM_WORLD communicator with our YARN_COMM_WORLD. This COMM_WORLD only contains processes allocated to the MPI program.

3.2.2.2 Efficient data copying

Using both MapReduce and pure MPI models is required for complex data analysis workflows in which first large-scale input data is filtered by MapReduce, and then its output is processed by a pure MPI application. For example, eye tracking research uses eye positions to study the human visual system, psychology, human computer

interaction, and user center design. In eye tracking data analysis, eye positions are captured by using an eye tracker that has high frequency, e.g. 300Hz (300 positions detected every second). If recording time is several months, the number of eye positions is billions. In order to detect fixations and saccades, MapReduce is employed to analyze the data [83]. Detected fixations and saccades are used to create features for SVM training in order to predict gaze-based interaction [84].

Output data of Hadoop MapReduce is typical to be written to Hadoop file system (HDFS), whereas MPI applications read data from shared disks (or local disks if staging-in is done in advance). Time of moving data from HMR to pure MPI applications is significant, since output is written to HDFS and then copied to local disks or shared storage where MPI will read later. Note that reading-writing I/O on supercomputers is expensive.

We add a key-value in-memory cache to HPC-Reuse in order to keep output data from MapReduce in memory. Since pure MPI applications are able to run on HPC-Reuse, they can access MapReduce output from the cache. In order to minimize data moving overhead, MPI processes are scheduled running on pool slots in which ReduceTasks have used before. Pool Manager is responsible for the scheduling.

3.3 Experimental evaluation

Our experiments are conducted on the 33 TSUBAME nodes or 33 FX10 compute nodes. A TSUBAME node (thin type) is equipped with Intel Xeon X5670 2.93GHz processor (12 cores) and 54GB main memory. All compute nodes are connected using the Infiniband network (Grid Director 4700). Its maximum throughput is 80 Gbps. Each node has 120GB of local SSD storage and a parallel central disk are also provided. A FX10 node is equipped with SPARC64 IXfx 1.848 GHz processor (16 cores) and 32GB main memory. Computing nodes are connected with each other through Tofu interconnection [17]. FX10 does not have a local disk for each computing node, conversely a central disk used.

HPC-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 HPC-Reuse, Hadoop source code is changed with the below ratio:

- Line of code / total of Hadoop: 336 / 1,851,473
- Number of classes / total of Hadoop: 8 / 35142

Each Hadoop cluster has one master node dedicated to run both Resource Manager and NameNode of Hadoop Distributed File System (HDFS), the remainder are slave nodes. Node Manager and DataNode are run on each slave node. A Hadoop cluster is deployed by using a MPI program. When the cluster is ready, MapReduce workloads are submitted to YARN. Note that Hadoop deployment and workload/application submission are run in the same supercomputer job.

The maximum MapTasks and ReduceTasks that can be run simultaneously on a node is *four*. Each execution container is configured to use 4096 MB of memory at most. The central disk are used for Hadoop Distributed File System (HDFS) storage with 256MB block size.

We use OpenJDK 7 and OpenMPI 1.6.5 on both TSUBAME and FX10. In order to use MPI from Java, we use a MPI binding [85] 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 TSUBAME and FX10. On FX10, experiments are run with the MCA parameter *plm_ple_cpu_affinity = 0* to disable CPU binding to each MPI process.

We use three test cases: fork-based YARN, MPI-Spawn YARN, and HPC-Reuse YARN. The original YARN is called fork-based YARN. We replace process fork mechanism of the original YARN with using MPI-Spawn command (at DefaultContainerExecutor class) that we call MPI-Spawn YARN. HPC-Reuse YARN is our proposal. On fork-based YARN, MPI is not available among processes (the below table).

Test case	TCP/IP	MPI
Fork-based YARN	✓	×
MPI-Spawn YARN	✓	✓
HPC-Reuse YARN	✓	✓

To evaluate HPC-Reuse performance, we compare the total execution and the start-up time of Tera-sort and iterative PageRank in three test cases. In order to show MPI performance on Hadoop MapReduce, we show shuffling time between MPI- and TCP/IP-based communication. We also evaluate further our HPC-Reuse by testing on Puma benchmark suite [86]. Pure MPI application running is evaluated based on execution time. We use eye tracking data from [84] to run MapReduce and SVM training to evaluate copying data overhead.

3.3.1 Benefit of HPC-Reuse

This experiment is aimed to show the performance of HPC-Reuse YARN is as good as the original fork-based YARN in general and HPC-Reuse approach shortens start-up

time in iterative workloads. We run Tera-sort workload of different input size up to 128GB on TSUBAME. We calculate the average execution and start-up time for each input size. We run iterative PageRank on FX10 and measure its total execution and start-up time. Each experiment of a data size or an iteration is run twice. Note that MPI-Spawn YARN does not work on FX10 because it is not allowed to spawn new processes on the same node. For fair comparison, the original shuffle engine (TCP/IP) is used for three test cases. Number of Reducers is set equal to number of Mappers in each experiment.

Figure 3.10 shows results of Tera-sort running on TSUBAME. In Figure 3.10a, HPC-Reuse outperforms the MPI-Spawn approach when data size is bigger than 32GB and its performance is the same as the fork-based one, even better with 128GB data size. Compared to MPI-Spawn YARN and Fork-based YARN, HPC-Reuse YARN reduces the workload execution times by 52% and 6%, respectively, on average. Regarding start-up time (Figure 3.10b), it achieves improvement of 82% and 30%, respectively, on average.

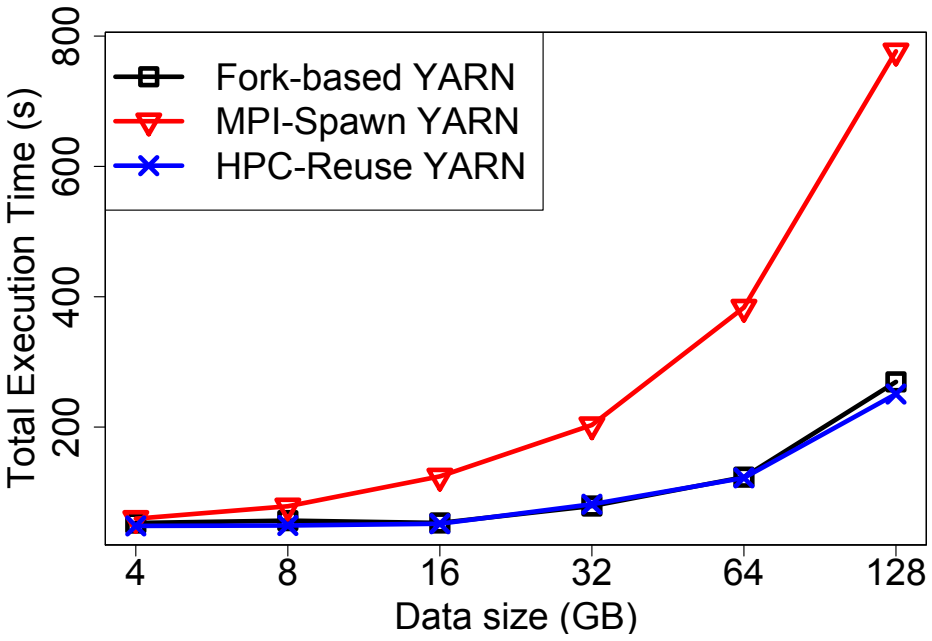
Compared to Fork-based YARN in the aspect of total execution time, the improvement of HPC-Reuse YARN is modest since its computation time is significantly longer than start-up time as shown in 3.11a. By contrast, when using MPI-Spawn YARN, up to 60% execution time is used for process creation in case of 128GB input size. The more input data increases, the more number of MapTasks is required.

Another observation shown in Figure 3.10b is that when average start-up time increases significantly when data size grows. This is because that each process container is spawned in sequence when using MPI-Spawn command and each spawning call takes nearly 0.5 second on TSUBAME (shown in Figure 2.10). Note that number of process containers increases when data size grows.

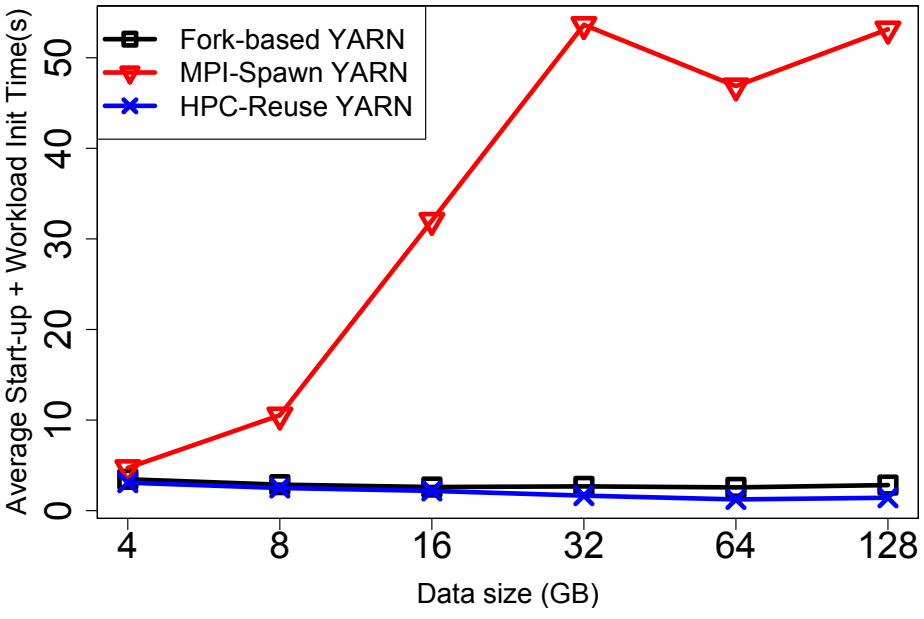
Figure 3.12 shows results of iterative PageRank running on FX10. In Figure 3.12a, when the number of iterations increases, HPC-Reuse approach shows more reduction, up to 26% at 32 iterations. This is because the more number of iteration is executed, the more JVM process containers are created that makes start-up time of Fork-based YARN longer. HPC-Reuse helps eliminate overhead of JVM start-up.

In Figure 3.12b, HPC-Reuse reduces start-up and workload initialization time by 57% on average. Moreover, Figure 3.13a reveals the ratio of start-up time in comparison with overall execution time is small in case of HPC-Reuse YARN. Workload initialization time is reduced because of connection and file caches, some un-reset static fields, and JIT compilation.

Figure 3.14 shows how the start-up time is shorten in the iterative PageRank when using HPC-Reuse. At the top figure 3.14a, three iterations have the same start-up time



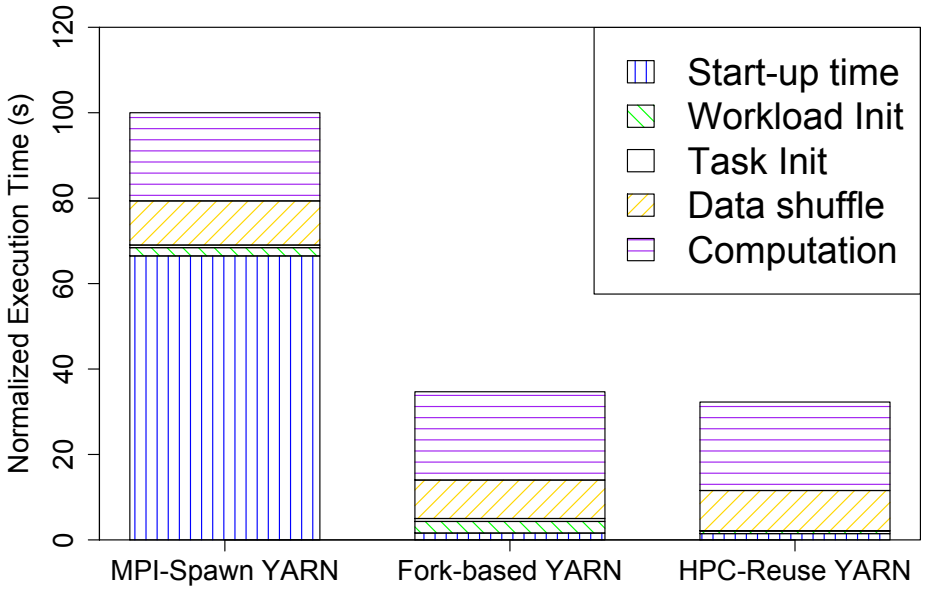
(A) Total execution time



(B) Average start-up time + workload init

FIGURE 3.10: *Benefit of HPC-Reuse (Tera-sort on TSUBAME)*

that marked with blue color. At the bottom figure 3.14b, however, the second iteration has shorter start-time than the first one. The third iteration is also shorter. In this PageRank applications, HPC-Reuse achieves 22% improvement at the second iteration and as much as 25% after the third iteration.



(A) 128GB

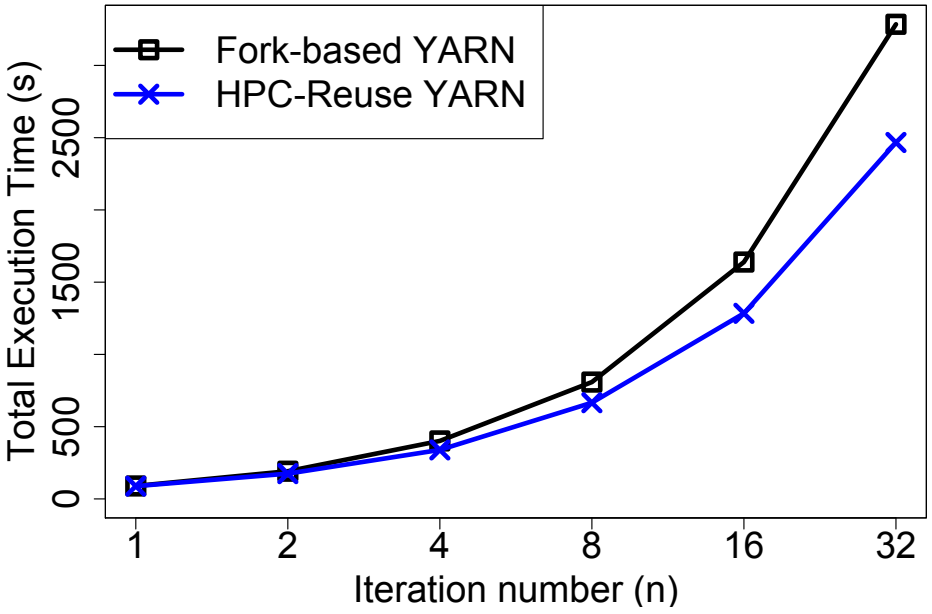
FIGURE 3.11: *Execution time breakdown Tera-sort on TSUBAME*

3.3.2 Benefit of MPI

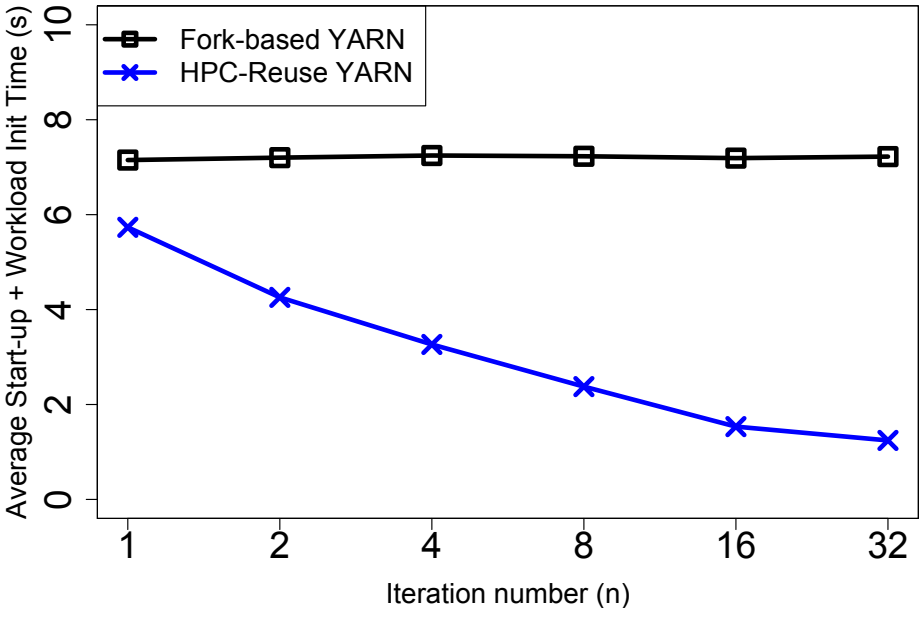
This experiment is conducted to show how fast MPI shuffle engine runs in comparison with the original TCP/IP engine. Tera-sort workload is chosen for the comparison and only HPC-Reuse YARN is used. In our MPI shuffle engine, since TSUBAME and FX10 do not support multiple thread-safe communication, we set number of fetcher threads running on each Reducer to *one* (*mapreduce.reduce.shuffle.parallelcopies = 1*). The number of Reducers is set to 64.

Figure 3.15 reveals when data size is 32GB, execution time starts decreasing by using MPI shuffle engine, but its improvement is modest: by up to 7% and 8% on TSUBAME and FX10, respectively.

Our current design of MPI shuffle engine is immature and straightforward: only one fetcher thread is used for each Reducer, and MPI *Shuffle Manager* can handle one request at a time (blocking). Moreover, we do not use any special techniques, such as in-memory cache [73] and pre-fetching [82]. We show more improvement of using MPI in Chapter 4 where in-memory storage is implemented that helps leverage MPI communication.



(A) Total execution time

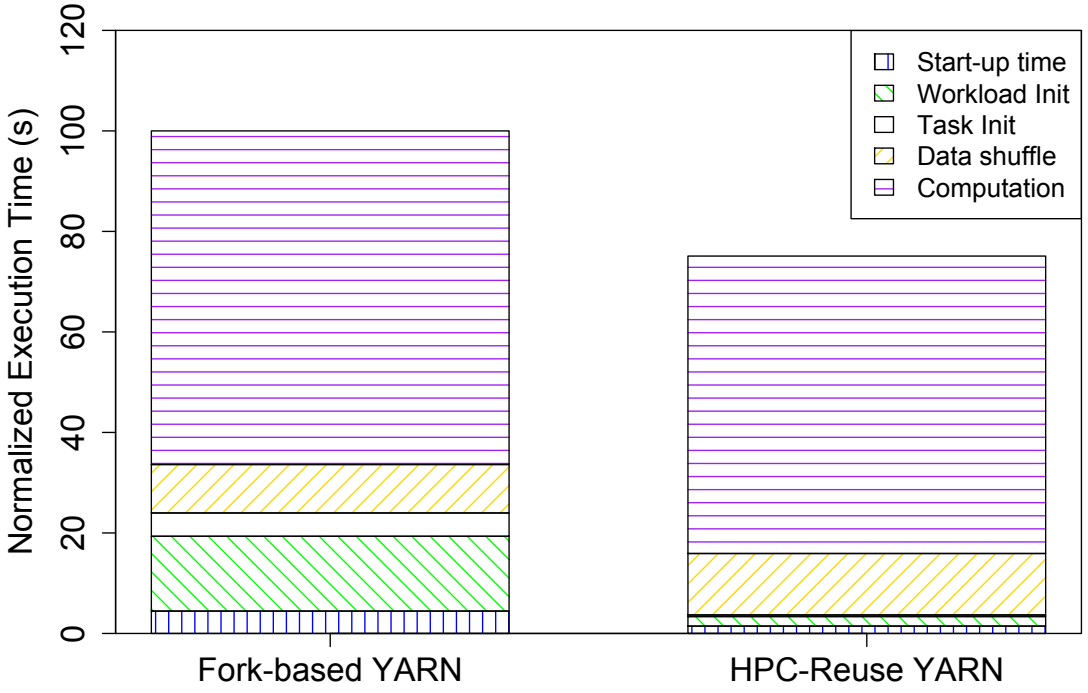


(B) Average start-up time + workload init

FIGURE 3.12: *Benefit of HPC-Reuse (Iterative PageRank on FX10)*

3.3.3 Purdue benchmark suite

To evaluate HPC-Reuse with other Hadoop MapReduce workloads, we use Purdue benchmark suite [86]. Kmeans and Classification workloads use a 15GB movie dataset, and the remainder use 30GB wikipedia data. All workloads are run on TSUBAME. Each experiment of a workload is run just once. The number of Reducers is set to 32.



(A) 32 iterations

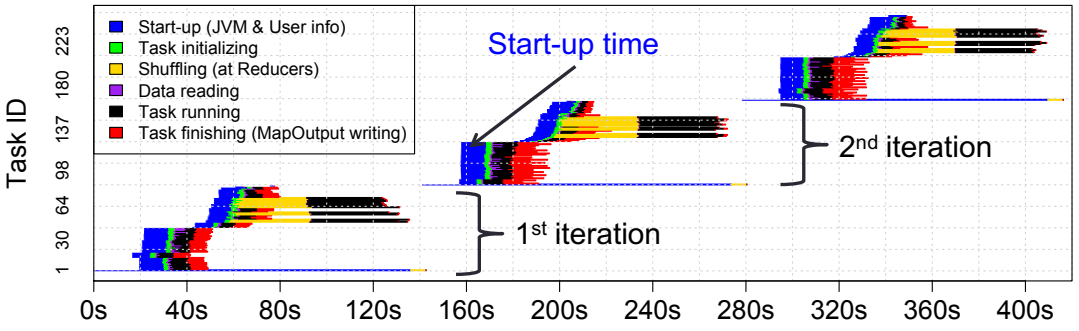
FIGURE 3.13: *Execution time breakdown of iterative PageRank on FX10*

Figure 4.9a shows execution time of those workloads. In comparison with MPI-Spawn YARN, HPC-Reuse YARN achieves improvement of 48% on average. At Grep and Classification workloads, since computation time is short, execution time is reduced by 80%.

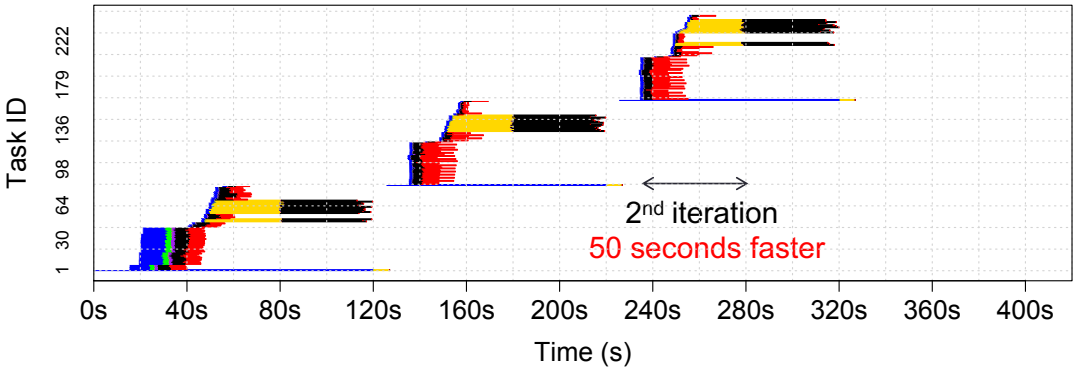
Compared to Fork-based YARN, HPC-Reuse reduces execution by 6% on average, especially 10% in case of Kmeans. This is because Kmeans workload has iterative computation that HPC-Reuse can exploit complication technology in JVM. Our MPI shuffle engine shows bad performance in Kmeans and Classification since the input size is not large.

3.3.4 Pure MPI application running

The purpose of this experiment is to show overhead of running pure MPI applications on HPC-Reuse YARN in comparison with on the supercomputer's job queue. First, we run a MPI program as a supercomputer job. Then, we run the same program on HPC-Reuse by submitting to YARN. We run this experiment on TSUBAME. MPI programs are written in Java.



(A) Iterative PageRank



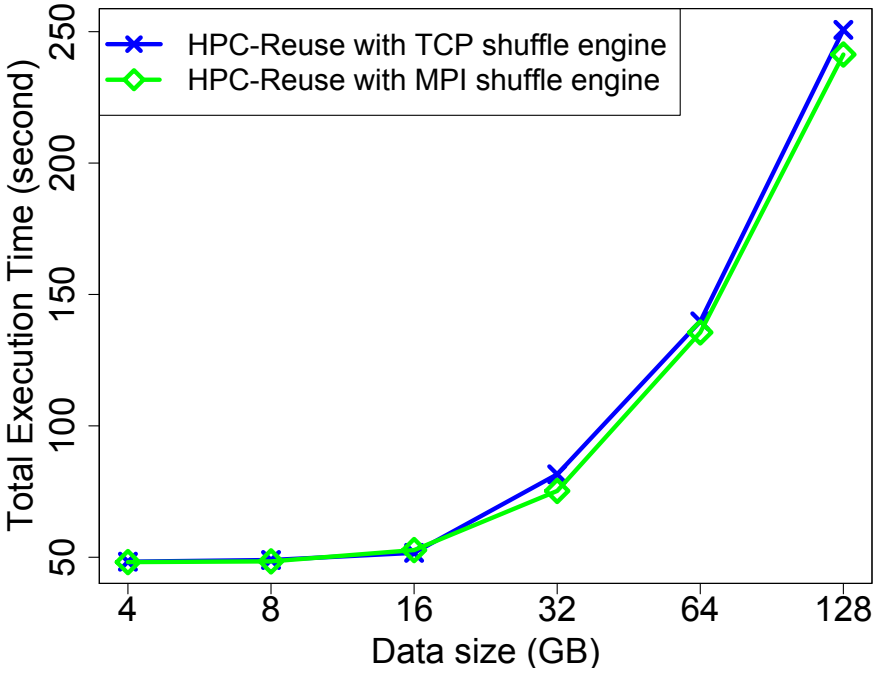
(B) Iterative PageRank + HPC-Reuse

FIGURE 3.14: *Comparison in iterative PageRank*

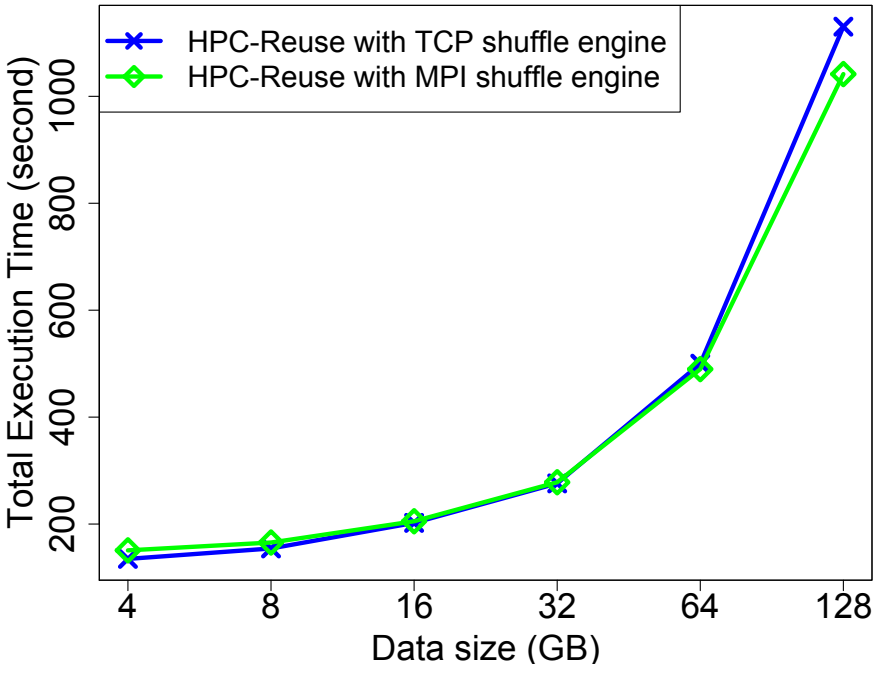
We evaluate three simple MPI applications: prime count, Pi estimation, and numerical integration. In the Figure 3.17, compared to MPI jobs submitted to the supercomputer's job queue, execution time running on YARN is 3.5 seconds longer on average. This is because it takes time to start new YARN AppMaster and process containers. 3.5-second for start-up time is reasonable in long running programs, for example applications in the Purdue MapReduce benchmark. The MPI applications used in the experiment is simple and short running. It is our future work to run more complex and long running MPI applications on HPC-Reuse YARN.

3.3.5 Data copying overhead

To show data copying overhead between MapReduce and pure MPI applications, we conduct an experiment with an eye-tracking analysis workflow. Eye tracking research uses eye positions to study the human visual system, psychology, human computer



(A) On TSUBAME



(B) On FX10

FIGURE 3.15: *Benefit of MPI*

interaction, and user center design. First, we use MapReduce to compute fixations, saccades, and 13 eye movement features [84] from over 167,000 eye positions. Then, we run SVM training as an embarrassingly parallel pure MPI application. The output from MapReduce is input of this pure MPI application.

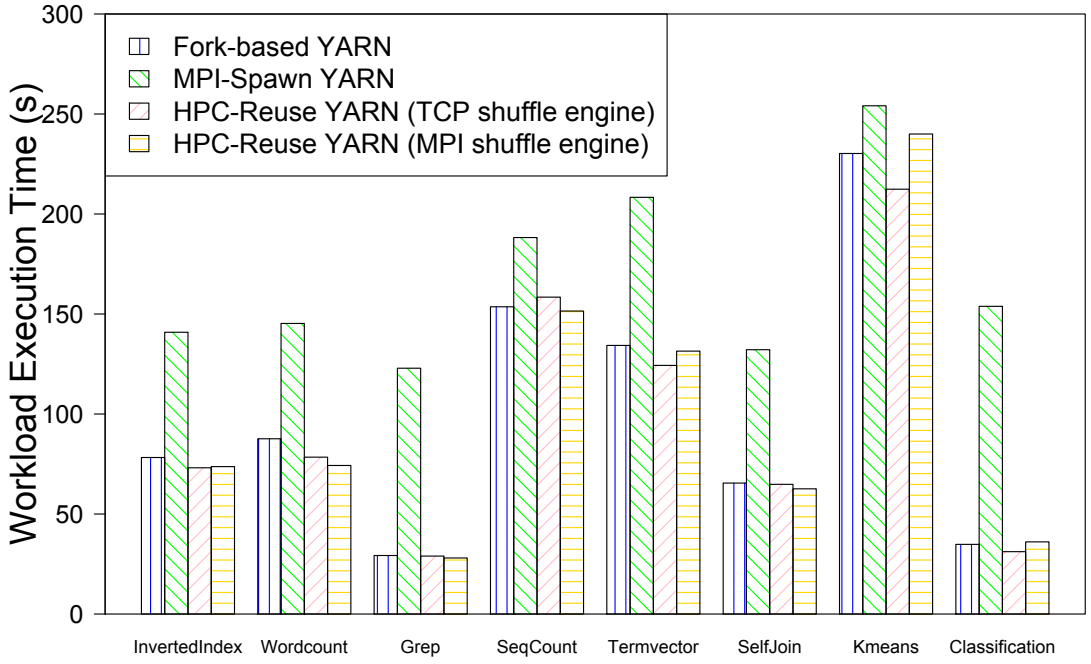


FIGURE 3.16: *Purdue MapReduce benchmark*

There are two test cases of how to run the pure MPI application. In the first case (HMR + MPI on the supercomputer's job queue), the pure MPI application is submitted to the supercomputer's job queue as a typical job, and it reads MapReduce output from the central disk. In the second one (HMR + MPI on HPC-Reuse YARN), the pure MPI application is run on HPC-Reuse YARN, and MapReduce output is read from key-value

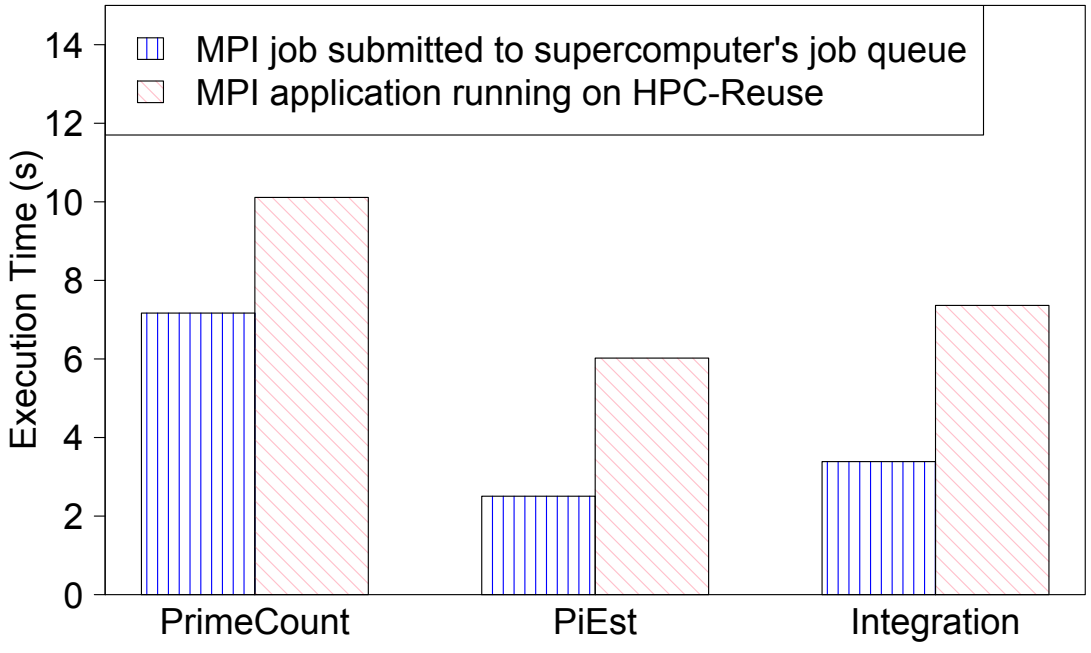


FIGURE 3.17: *Pure MPI applications running on HPC-Reuse*

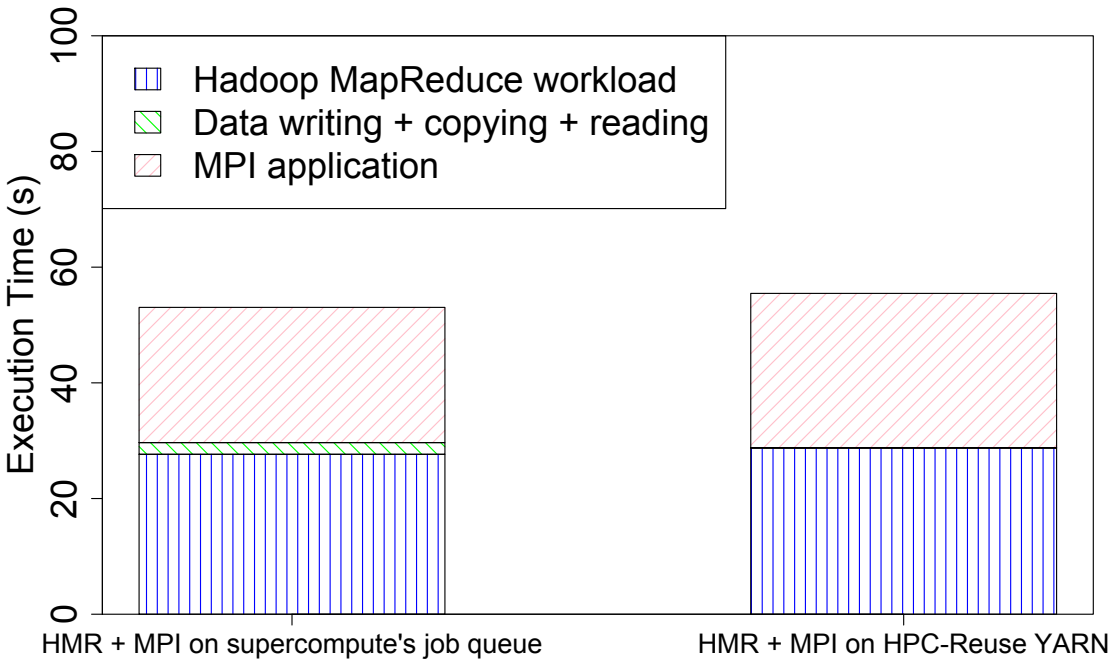


FIGURE 3.18: *Data copying overhead*

in-memory storage.

This experiment is run on TSUBAME. Figure 3.18 shows that data copying time is reduced by 90% when the MPI application is run directly on HPC-Reuse YARN. If the pure MPI application is run on the supercomputer's job queue, it takes two seconds for data copying and reading. However, execution time of the MPI application is longer on YARN. This is because it takes 3-4 seconds to start a MPI application running on HPC-Reuse YARN.

Another advantage of running HMR and pure MPI on HPC-Reuse YARN is the pure MPI application can be immediately started after MapReduce is finished when both applications run on HPC-Reuse YARN. By contrast, we must manually check whether data is available in order to submit the MPI application to the supercomputer's job queue.

3.4 Related work

JVM Reuse. M3R [73] used X10 language and JVM Reuse to implement HMR engine and run in-memory MapReduce. However, technical issues were not provided, for example class loading and static field clean-up. Also, there was no specific evaluation of JVM Reuse, such as start-up time reduction. In our HPC-Reuse, we provide optimization to

use JVM Reuse more efficiently, and its effectiveness on iterative workloads has been evaluated. Moreover, we keep the original HMR engine with minimum changes.

MPI on YARN. Gerbil [70] is already presented in Section 2.4.4. While their approach is to start MPI process containers by using MultiPurpose Daemon [71], in our approach, MPI process containers are allocated from a process pool. Moreover, in their approach, we are aware that for short running MPI applications, start-up overhead is significant in comparison with total execution time. For performance comparison, it takes 9 seconds for YARN AppMaster start-up and container allocation in Gerbil. Their experiment was conducted on nodes equipped with two Intel Quad-core Xeon E5462 2.8 GHz processors and AppMaster container was configured to use 3072 MB of memory. Our HPC-Reuse approach needs just 3.5 seconds in order to start MPI programs on TSUBAME nodes equipped with a Intel Xeon X5670 2.93 GHz processor. Each pool slot is allocated with 4096 MB of memory.

Data shuffle. JVM-Bypass [82] proposed C-based shuffle engine integrated with RDMA in order to speed up the data shuffle phase. They were aware that JVM-based shuffle engine is slow due to Java's deep stack of transport protocol. While they focused on evaluating effectiveness of bypassing JVM and using RDMA, we aim at using MPI over Hadoop processes and evaluating how fast MPI shuffle engine can speed up.

Resource management. Slurm++ workload manager [87] is an extended implementation from Slurm that aims to support mixture of applications, such as traditional HPC applications (MPI), ensemble runs, and many-task computing at exascale. It uses multiple controllers to manage partitions of compute nodes, but its current design only supports running coarse-grained workloads. Our HPC-Reuse YARN can host various kinds of application in fine-grained management manner.

Wasi et al. [65] proposed high performance design of YARN running on HPC clusters. YARN is redesigned to utilize more efficiently Lustre file system and Remote Direct Memory Access (RDMA). While their main objective is to evaluate combination of Lustre and RDMA, we focus on using MPI over Hadoop processes and hosting pure MPI applications.

Hadoop on Demand (HoD). The first HoD (mentioned in [20]) was developed at Yahoo!, and Torque [18] and Maui [88] were used to allocate a Hadoop cluster on a pool of shared nodes. Torque/Maui was also responsible for starting Hadoop's JobTracker and TaskTrackers. myHadoop [24], another HoD, used this style to deploy a Hadoop cluster on supercomputers. However, there are several shortcomings of HoD: resource utilization and application diversity. For example, it does not support MPI communication, hosting

pure MPI applications, exploiting huge main memory, or avoiding I/O to the central disk.

3.5 Discussion and Summary

We have proposed HPC-Reuse to better support dynamic management with MPI on supercomputers. It helps avoid process creation mechanism, such as MPI-Spawn, and enable MPI communication over Hadoop processes. We have also designed a MPI shuffle engine to speed up data exchange. Our HPC-Reuse YARN can host pure MPI applications that helps improve the resource manager's efficiency. In addition, combination of MapReduce and pure MPI workflow as post processing is optimized by using in-memory caching on our HPC-Reuse. Compared to other JVM Reuse implementation, our design considers optimization including class loading and clean-up. Moreover, our approach only makes minimum changes of the original Hadoop. Our experimental results show effectiveness of HPC-Reuse with improvement of 26% in iterative workloads and 8% in data shuffling.

HPC-Reuse design can be applied to other fine-grained resource managers, for example Mesos, an offer-based resource manager. Implementation of HPC-Reuse using a process pool can work independently, and it is easy to be integrated with outside components. HPC-Reuse is providing a para-virtualization layer, so the upper frameworks need to be modified in order to adapt to HPC-Reuse, but the modification should be small as discussed in Section 3.3.

Although there is some implementation using JVM reuse, but all of them did not evaluate JVM reuse and provide any technical issues when using JVM reuse. In our JVM reuse, we have considered issues including class loading used to load users' classes and clean-up that helps to reset to the original condition.

Overhead by adding HPC-Reuse layer on the supercomputer environment is small since compute and memory resources used for keeping slots of HPC-Reuse are not significant. Those slots should be in the sleep state when there is no task running. Benefits from HPC-Reuse is much bigger than such overhead.

Through experiments in Section 3.3, if we compare HPC-Reuse with the original Hadoop, the major performance gain comes from reducing start-up time, especially in iterative workloads, such as PageRank. At the current implementation and datasets, the improvement from MPI is less than 8% in the tera-sort workload. The performance gain is not much, but we have already proposed a mechanism to use MPI for data shuffling.

The trend showed that more improvement could be achieved for larger datasets that is discussed in Chapter 4 where in-memory storage is used.

Chapter 4

Efficiently Virtualizing Local Disks

Regarding the second mismatch on expensive disk I/O to the central disk on supercomputers, we focus on supercomputers where the local disk is not available and we want to run the popular data-intensive frameworks on such supercomputers having only the central disk.

We solve the expensive disk I/O mismatch by using in-memory storage to provide efficient virtual local disks instead of using the central disk. Although this approach is not new, the challenge is to choose the best deployment strategy of in-memory storage (or virtual disks) in the context of MapReduce, our targeted application type. In this chapter, we present our own memcached-like in-memory storage for Hadoop (our focused framework), namely SEMem, that supports different deployment strategies including being deployed as RAM disks, deployed on every node and data can be stored in remote memory, and deployed only on dedicated nodes that are used only for storage.

We have conducted experiments and found the dedicated-node strategy showed a good performance in some benchmarks including Puma and Tera-sort although that strategy does not intuitively seem good due to fewer nodes used for computation tasks. Moreover, SEMem is tightly integrated with the implementation of Hadoop, hence the users of Hadoop with SEMem can easily configure the system to select an appropriate deployment strategy. Its intra-communication is through MPI, which is the de facto fast networking protocol on supercomputers.

4.1 Virtualizing local disks

In-memory storage is a natural approach of virtualizing local disks that helps avoid disk spilling by keeping intermediate data in memory [43][73][74]. We choose in-memory storage to provide virtual local disks since the size of memory on each compute node of supercomputers is large enough to serve both computing and data storage. Although typical in-memory storage for MapReduce frameworks use the local memory space of each node, the location of in-memory storage is not limited to the local memory space but it can be the remote memory of other compute nodes. We identify that this in-memory storage can be seen as memcached-like style [30].

We developed our memcached-like in-memory storage for Hadoop so that Hadoop can efficiently run even on supercomputers that have only a slow central disk but no local disk per node. We named this storage system SEMem. Unlike Memcached, SEMem is integrated into Hadoop with HPC-Reuse in Chapter 3, hence it can use MPI for communication. It supports multiple deployment strategies on which memory spaces are used as local storage, local memory, remote memory, or their mixture. The specified memory spaces are managed by a SEMem daemon as in-memory storage and provided as a local disk for mapper tasks and reducer tasks of Hadoop, which communicate to a SEMem daemon shown in Figure 4.1. The users of Hadoop with SEMem can easily change deployment strategies to investigate an appropriate strategy for their applications.

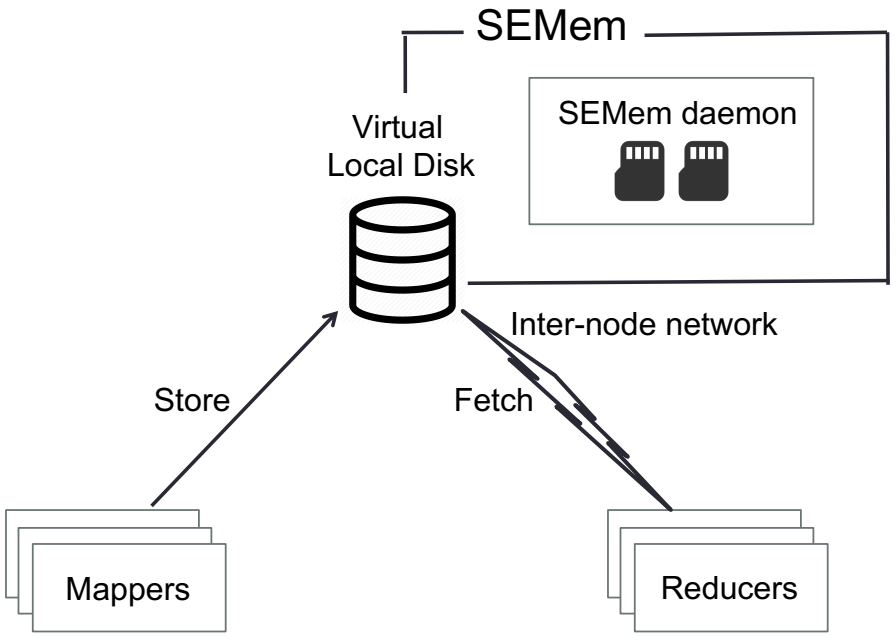


FIGURE 4.1: *SEMem providing virtual local disks*

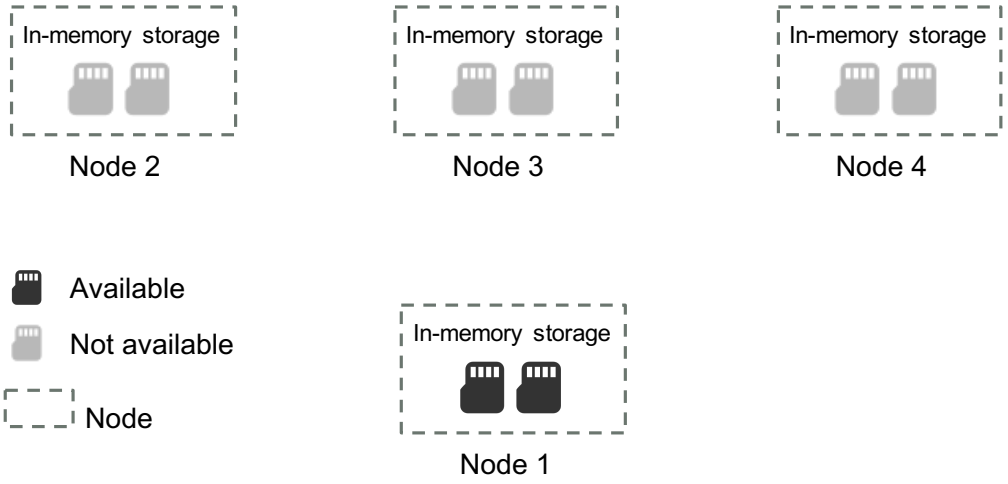


FIGURE 4.2: *RamDisk: deployed as RAM disks where data is stored only in local memory*

4.2 Deployment strategies

SEMem supports three deployment strategies: RamDisk, Every node, and Dedicated nodes. We below present these three strategies.

RamDisk: in-memory storage is deployed as RAM disks where data is stored only in local memory. Memory size on each node is limited, data might not fit into memory. When out of memory happens, the task must be stopped or restarted on other nodes. For large-scale datasets, this approach is not feasible. Our expectation is this deployment strategy is fast in small-scale datasets.

Figure 4.2 shows how in-memory storage is used in the RamDisk strategy. There are four computation nodes in the figures and on each node, a memory space is allocated to store in-memory data. Since there are computation tasks running on the node, the allocated memory space is limited. Out of memory will happen on a node when its in-memory storage is full. Note that the in-memory storage on each node is independent and node #1 cannot use the in-memory storage of node #2, 3, or 4. The design of in-memory storage in RamDisk is simple and the overhead of running in-memory storage is small.

Every node: in-memory storage is deployed on every computation node and data can be stored in remote memory. We call nodes that are responsible for running applications *computation nodes*. Computation nodes are used to run a job's tasks, e.g. Mapping or Reducing tasks in Hadoop MapReduce. The SEMem daemon is run on all computation nodes and helps share a node's in-memory storage with other nodes. This strategy solves the problem of out of memory when data size is bigger than a node's in-memory storage size.

Figure 4.3 shows how in-memory storage is deployed on four nodes following the every-node strategy. On node #1, when its in-memory storage is full, it will request other nodes to send data to those nodes. In-memory storage on every node can be shared and gathered in order to create a in-memory pool. This in-memory pattern is quite similar to the partitioned global address space (PGAS) programming model, but we focus more on data organization and movement.

Compared to RamDisk, a more complex in-memory storage design is required since each node must handle data from other nodes and manage its own in-memory storage efficiently. On each node, a waiting daemon should be used to receive data from other nodes. To balance data among in-memory storage on each node, we can use the round robin algorithm to store data on remote nodes. The size of in-memory storage on each node should be fixed in order to not affect the performance of computation tasks running on the same node.

Regarding data movement, there is more data sent and received among nodes that makes network busier, especially in unbalanced input data. Data movement happens when in-memory storage on a node is full and it must send data to other nodes. In network-intensive applications, data movement can be a performance bottleneck. A balanced data distribution of in-memory storage is required to avoid the bottleneck that might happen on certain nodes where too much data is stored.

Dedicated nodes: in-memory storage is deployed only on dedicated nodes that are used only for storage. We allocate a group of nodes that is only used to store data in their memory. No computation task is run on that group of nodes. We call those nodes *memory nodes* and their memory *external memory*. The SEMem daemon is run only on each memory node. In this deployment strategy, we trade computation resource for data

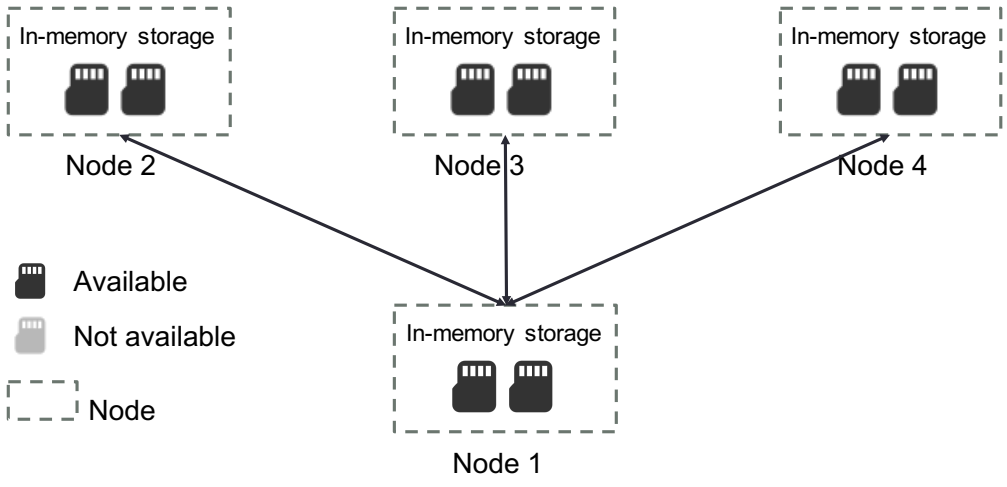


FIGURE 4.3: *Every-node: deployed on every node and data can be stored in remote memory*

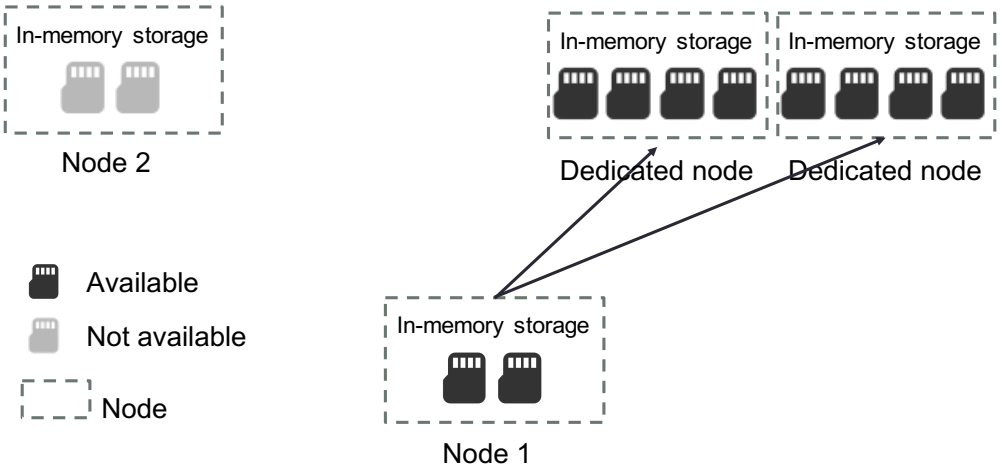


FIGURE 4.4: *Dedicated-node: deployed only on dedicated nodes that are used only for storage*

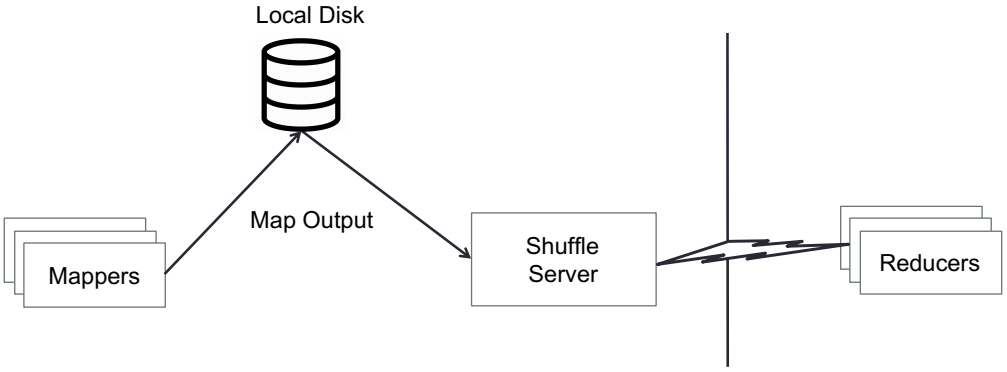
storage by using memory nodes only for keeping data. A question is that whether this strategy is a complete waste of resources or not. No computation task on memory nodes gives a chance of allocating bigger memory space to SEMem and makes the memory nodes less busy.

Figure 4.4 shows how in-memory storage is deployed on four nodes following the dedicated-node strategy. There are two nodes used for in-memory storage called *dedicated node*. Node #1 can send its data to those dedicated nodes when its in-memory storage is full. Note that the in-memory storage of node #2 cannot be used by node #1. Those dedicated nodes are used only for in-memory storage without computation task running on those nodes. Due to no computation task, bigger in-memory storage size can be allocated on those nodes.

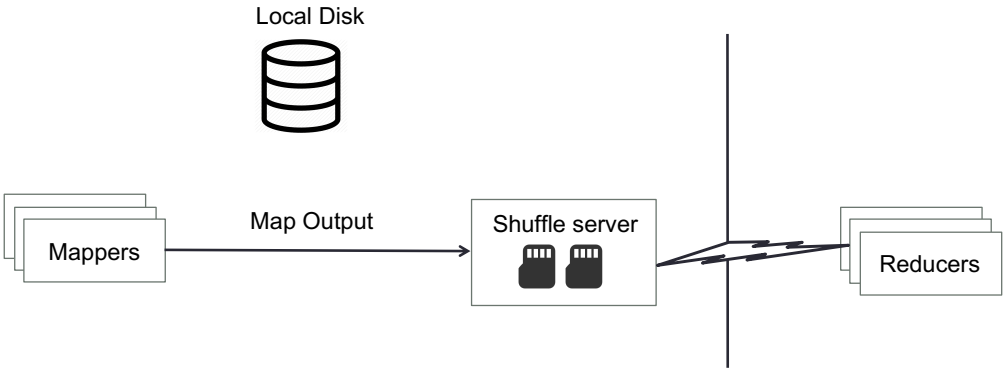
Compared to Every-node, the in-memory storage design on each computation node is simpler since it does not need to handle receiving data from other nodes. The design is as simple as RamDisk's one. On each dedicated node, a waiting daemon is deployed to receive data from computation nodes. We can use the round robin algorithm to balance data stored on dedicated nodes.

The amount of data sent and received among nodes is the same as Every-node, but the performance bottleneck will happen only at memory nodes. On a computation node, it sends and receives its own data, so there is no network bottleneck happening. The round robin data placement on memory nodes helps avoid unbalanced data distribution of in-memory storage.

In the case that the dedicated-node strategy is chosen, the total performance might be slower than the every-node's one since the number of computation nodes is smaller.



(A) The original Hadoop



(B) RamDisk deployment

FIGURE 4.5: *SEMem configuration for RAMDisk deployment strategy: a shuffle server is integrated into NodeManager of Hadoop framework.*

If the application is compute-intensive and in-memory storage is not used much, the dedicated-node strategy could show its disadvantage. However, in this thesis, we focus on data-intensive applications with large-scale intermediate data. The number of memory nodes is one of our technical issues.

4.3 SEMem architecture

In the original Hadoop MapReduce workflow (Figure 4.5a), intermediate data generated by mapping tasks is written to a node's local disk. In case of supercomputer, instead of local disks, the central disk or SSD is used to store that data. Figure 4.5a shows how map output is copied from mappers to reducers. At the mapping side, map output is buffered in memory, and then when its size reaches buffer size, it is spilled to the local disk. The spilled files are merged into a final map output at the end of the mapping phase. There is a *shuffle server* (thread) running on each computation node waiting for fetching request from reducers. When a request comes, the shuffle server will look up

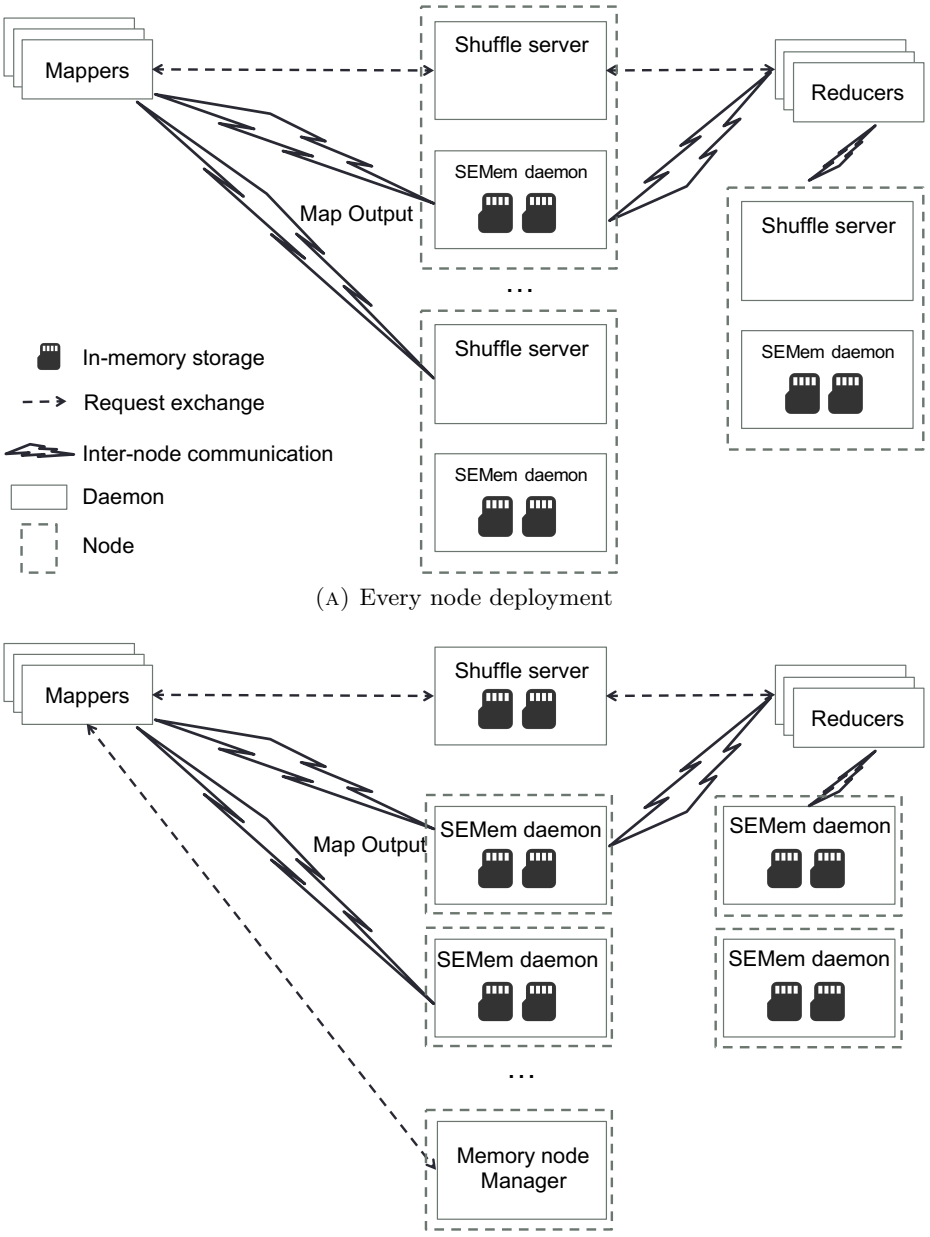
map output metadata, read again those data from the local disk, and send back to the reducer.

In the RamDisk deployment (Fig. 4.5b), map output is stored totally in memory to avoid writing to and reading from the local disk. Map output is kept in memory instead of spilling to the local disk when the buffer is full. After merging, map output is sent to the shuffle server. A memory space is created on each shuffle server to keep map output in memory. When a fetching request comes, the shuffle server looks up the memory space and is able to send back immediately to the reducer. The size of the memory space on each shuffle server is limited, so out of memory might happen during data fetching. For example, suppose that the memory space size is 8 GB and there are 20 map tasks running on the node. If each input data is 512 MB and running tera-sort application, the total size of map output is 10 GB. Those data cannot fit into the memory space. Although the map output can be deleted right after it is sent to the corresponding reducer, that out of memory error can happen at any time.

In Every-node deployment (Fig. 4.6a), SEMem daemon is run on the same node with the shuffle server. In contrast to RamDisk where data is stored at a shuffle server, data can be kept at any node's in-memory storage (memory daemon). Round-robin data affinity is used to store in-memory data to remote memory since it makes data more distributed and helps speed up data fetching later. This kind of data affinity does not use a certain remote memory for storage as long as possible since unbalanced intermediate data might be a performance bottleneck.

Dedicated-node deployment (Fig. 4.6b) is to create a group of nodes (called *memory nodes*) used only for storing map output and there is no map or reduce task running on those nodes. On the supercomputer environment, those nodes are requested on the same job as the nodes used to run Hadoop applications. In Figure 4.6b, the group of memory nodes consists of one node manager (master) and slave nodes (just called memory nodes). The node manager is responsible for data placement and monitoring memory left on each memory node. Since there is no task running on memory nodes, the size of the memory space can be allocated bigger.

In comparison with RamDisk design, data exchange among nodes of SEMem in Every-node and Dedicated-node is more complex. First, the mapper requests its shuffle server to send map output. If the shuffle server still has empty space, data is sent then. If the shuffle server's memory space is full, the mapper requests the memory node manager to send map output. The node manager will find a relevant place and send back to the mapper. Then, the mapper starts sending map output to the specified memory node. When finishing sending map output, the mapper also informs the shuffle server about the location of map output on memory nodes.



(A) Every node deployment

(B) Dedicated node deployment

FIGURE 4.6: *SEMem configuration for every-node and dedicated-node deployment strategies. Memory daemon (or node) is a separate process running on each node of the every-node and dedicated-node strategies and responsible for in-memory storage. In the every-node strategy, the memory daemon and shuffle server are available on the same node.*

What is the best strategy of storing map output in external memory? Should we distribute map output to as many memory nodes as possible or just store on a memory node and when it is full, the next memory node will be used? The former helps increase fetching throughput since reducers can request to more memory nodes. The latter is useful if map output is packaged and sent once to a reducer. In the current design, we have implemented a round robin scheduler on the memory node manager. Data will be distributed on all memory nodes.

Compared to using the central disk and SSD to store intermediate data, SEMem helps avoid spilling out intermediate data to disks that is considered slow. SEMem might outperform the central disk since disk I/O to the central disk is expensive as shown in Section 2.5.1. SSD is often equipped in the same node with compute (CPU) and memory resources, so disk I/O to SSD is faster than the central disk's one. However, reading and writing from/to the memory are faster than SSD. Using remote memory might be also faster then SSD due to high-speed interconnection among compute nodes on supercomputers.

Although total memory size is still limited that depends on the number of memory nodes, SEMem can be scalable by adding new memory nodes on the fly. In the current design, that feature is not available yet because it is prohibited from combining two different jobs. The load balance of memory usage might be also better than other in-memory implementations since the round robin algorithm is used to put data on SEMem's memory nodes. The intra- and inter-communication is through MPI. Data sending and receiving from/to SEMem are fast since SEMem is designed to exploit high-speed network on supercomputers. In the design of the central-disk-based Hadoop, data from compute nodes is read/written to the disk through Infiniband switches (sometimes controlled by I/O nodes [27]). It is typical that compute nodes are connected to each other through special interconnection, such as 6D Mesh/Torus [17] and QDR InfiniBand switch. The inter-node communication of SEMem is faster one between nodes and the central disk (through I/O nodes). We detail SEMem's communication protocol in the next section.

4.4 Technical issues

In this section, we show details of SEMem's implementation mainly in aspects of communication protocol and storage size. SEMem is a storage architecture using compute nodes for storage only where there is no computation task running. Since we trade computation resource for data storage, SEMem must be designed to leverage other resources and minimize number of memory nodes. As discussed in the previous section, SEMem exploits high-speed network on supercomputers in order to reduce latency of putting and fetching data. We use MPI for inter-node communication and the number of memory nodes is estimated based on the size of input data.

```
while true do
    if req == null then
        req = MPI.iRecv
    end
    if there is a new request then
        Add req to sendingPool's waitList;
        Reset req = null
    end
    for slot in sendingPool's slots do
        if data reading finishes then
            MPI.iSend to the client
        end
        if iSend finishes then
            free the slot
        end
    end
    Assign req in waitList to free slots;
end
```

Algorithm 2: Multiplexing non-blocking MPI on the memory nodes

4.4.1 Communication protocol

Compute nodes are connected to each other through special connections having high throughput and low latency, such as 6D Mesh/Torus or QDR Infiniband. By contrast, disk I/O from compute nodes to the central disk is considered slow since it is piped through I/O devices and the central disk is a distributed storage consisting of thousand of HDDs.

We use MPI as communication protocol since MPI is fast and optimized to exploit the high-speed interconnection of supercomputers. Memcached is an alternative implementation of SEMem and its communication is based on TCP. However, MPI is the de facto communication protocol on supercomputers. While some other variants of Memcached use Remote Direct Memory Access (RDMA) [78] to speed up data exchange among remote memory, MPI also leverages RDMA for large data messages. In the following sections, we describe how MPI works on SEMem and direct buffer memory in Java when sending/receiving MPI messages.

4.4.1.1 Non-blocking MPI on memory nodes

Non-blocking MPI is required to handle multiple requests of fetching and putting data. It helps decrease latency for each request. It is possible to use multiple threads to process each request of fetching or putting data. However, calling MPI from multiple threads is

more complex in order to avoid race condition or the MPI implementation must support *MPI_THREAD_MULTIPLE* mode.

To multiplex non-blocking communication, SEMem runs a dedicated thread on every node to handle it. Note that most of the supercomputers we know, for example TSUBAME and Fujitsu FX10, do not support *MPI_THREAD_MULTIPLE* mode. This approach helps avoid calling MPI send/recv in different threads.

On each memory node, we run a waiting daemon to listen incoming requests. The algorithm implemented in the daemon is shown in Algorithm 2. We use progress strategy to check iRecv and iSend status and data is queued as well. A loop is called to check if there is any coming message or not (req variable). When there is a new request, it is added to a queue (*waitList*). This waiting list contains requests that have been processed. A sending pool, namely sendingPool, is created to process queued requests. A slot of the sending pool is responsible for reading data and calling iSend. When data is read to the sending buffer, iSend is called to send data to the client. Data reading must be also implemented to use an asynchronous mechanism, for example AsynchronousFileChannel in Java. The progress strategy helps prevent calling MPI.iSend multiple times simultaneously. The Algorithm 3 shows how data is received at the client, for example at reducers in Hadoop MapReduce. Fetching data is repeated until all MapOutputs are requested and received. There are several MapOutputs needing to be fetched on each host, so they are requested at once. Data receiving from the *shuffle servers* will be stored in the heap space of the JVM process.

4.4.1.2 Direct memory in Java

We have chosen to implement multiplexing non-blocking MPI in Java since Hadoop, Spark, and Flink are JVM-based frameworks. Moreover, MPI is run on each JVM-based processes of Hadoop, so it is more convenient to implement in Java. Java is not considered as a well supported programming language on supercomputers and typical

```
while any MapOutput do
    Wait for (host, MapOutputs) ;
    for each MapOutput do
        MPI.Send to the host ;
        if MPI.Recv from the host then
            Data in heap ;
        end
    end
end
```

Algorithm 3: Receiving data at reducers of Hadoop MapReduce

compute-intensive applications should be written in C or Fortran that are the de facto programming languages. Java performance has been being improved significantly by using just-in-time compilation (JIT) [89] and other optimizations, for example adaptive optimizing and garbage collection. Therefore, there is no reason why Java should not be used on supercomputers. Lots of data-intensive frameworks, such as Hadoop and Spark, are already written in Java and run on JVM.

Most current MPI implementations are written in C, but those popular data-intensive frameworks and our implementation are in Java. Our approach is using a MPI binding for Java [85]. That MPI binding helps call MPI, for example MPI.iSend and MPI.iRecv, from Java and Scala. For the detail of implementation, we are using a MPI binding that is included in OpenMPI 1.7.5 [90]. Using a MPI binding might degrade communication performance due to JNI overhead and memory copying.

Memory copying between JVM and the memory space of C program is a bottleneck when using MPI to send data stored in Java objects. In order to avoid using the heap memory of JVM, we use direct memory, for example direct ByteBuffer, to send and receive data in the MPI binding library. Through JNI libraries, MPI written in C can execute sending and receiving. Direct memory is located outside the heap space of JVM and it needs to be allocated explicitly by users. Objects in direct memory will automatically clean up their buffers after using or when there is no reference to the objects. The cleaning is one part of Java Garbage Collection (GC).

4.4.1.3 Reusing JVM processes

In order to keep MPI connection among processes, we use HPC-Reuse [36] proposed in the previous chapter to avoid MPI-Spawn or other methods that are used to start new processes in which MPI connection to the running processes is still available. HPC-Reuse is keeping MPI connection among compute nodes that will benefit SEMem when sending and receiving data at shuffle servers and memory nodes. Note that by using HPC-Reuse, the start-up time of process creation is also shortened.

4.4.2 Storage size in dedicated-node deployment

The storage size of SEMem depends on number of memory nodes. In order to reduce the waste of CPU resource on memory node, it is necessary to minimize number of memory nodes, but out of memory must not happen and performance is good enough. The number of memory nodes should be scaled out on demand, but the current design does not support that feature. The number of memory nodes must be fixed when SEMem is

started. When the total number of compute nodes is fixed, it will be problematic if the ratio of the dedicated nodes to the total nodes is large. In this scenario, the number of nodes used for running tasks is small and it can cause slow performance.

Suppose we have a fixed number of compute nodes (denoted by N_{comp} = Constant). N_{mem} denotes number of memory nodes (that used only for storing data). $Exec$ denotes execution time when number of memory nodes is N_{mem}. We need to find a relevant N_{mem} where $Exec$ is fastest as follows:

$$Min(Exec(N_{comp}, N_{mem}))$$

In our current design, the number of memory nodes is estimated roughly based on the size of input data. Our assumption is that total size of intermediate data is smaller than one of input data. Therefore, the number of memory node is chosen that satisfies a condition that total size of memory space on shuffle servers and memory nodes must be bigger than size of input data. It might not achieve the best performance of data fetching, but it helps prevent out of memory error. That error is more serious if it happens since the whole task will be restarted.

The storage size issue does not happen in other deployment strategies. In the RamDisk strategy, the storage size is the total memory used for in-memory storage on all nodes. In the every-node strategy, the storage size is also the same as the RamDisk.

4.4.3 Minimizing changes in Hadoop

Keeping the original source code (e.g. Hadoop or Spark) unchanged as much as possible is important to increase SEMem's capability in order to be integrated easily to other different data-intensive frameworks. In our implementation of Hadoop integration, Hadoop source code is changed with the below ratio as follows:

- Line of code / total of Hadoop: 443 / 1,851,473
- Number of classes / total of Hadoop: 8 / 35,142

4.5 Experimental results

In this section, first we compare three in-memory deployment strategies including *RamDisk*, *Every-node*, and *Dedicated-node* that are described in Section 4.2. We use three workloads in Puma benchmark suit [86]: WordCount, InvertedIndex, and SequenceCount.

Second, we evaluate how fast SEMem (using the dedicated-node deployment strategy) is in comparison with the central-disk-based and SSD-based storage. There are three test configurations in this experiment: *Central-disk*, *SSD*, and *SEMem* storage. Tera-sort application is mainly used in comparison.

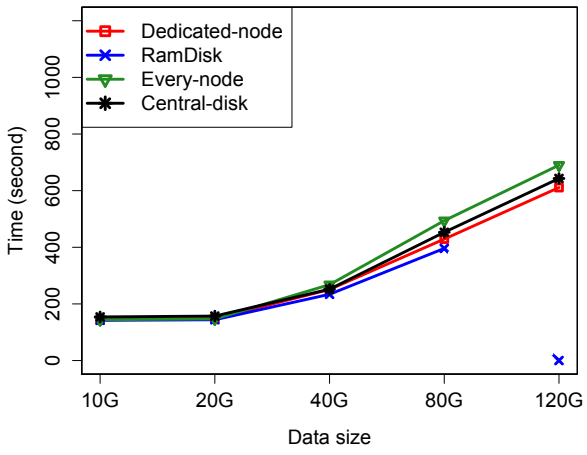
Our experiments are conducted on Fujitsu FX10 supercomputer at The University of Tokyo [27] and TSUBAME supercomputer at Tokyo Institute of Technology [16]. A FX10 node is equipped with SPARC64 IXfx 1.848 GHz (16 cores) and 32GB memory. FX10 nodes are connected with each other through Tofu interconnection [17]. The maximum throughput of that interconnection is 80Gbps. A TSUBAME node has less cores (Intel Xeon X5670 2.93GHz - 12 cores) and 54GB memory. Each node is connected with Infiniband device Grid Director 4700. Another feature of TSUBAME is that 120GB of SSD storage is available on each node.

SEMem has been implemented using Java since it helps integrate more easily with Hadoop, Spark, or Flink frameworks. We use Hadoop v2.2.0 for experiments, but any Hadoop version *2.x* can be integrated with SEMem. Spark can also use SEMem as an intermediate storage. Each slave node can run at most four MapTasks or ReduceTasks simultaneously and maximum heap size of a JVM is 4096 MB of memory. Block size of Hadoop Distributed File System (HDFS) is 256 MB. In our experiments, all input data is stored in HDFS running on the central disk.

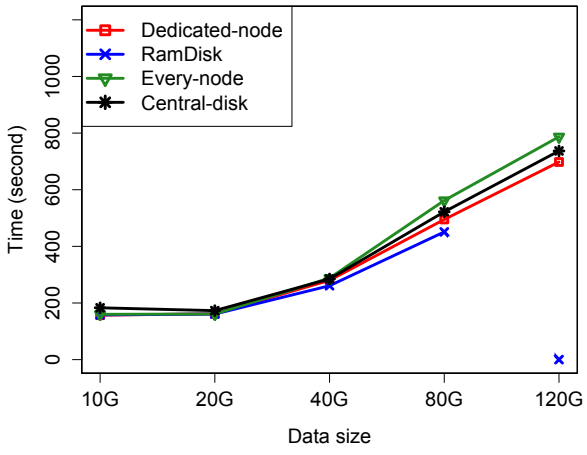
4.5.1 The performance of deployment strategies

This experiment is aimed to show which deployment strategy of in-memory storage shows a better performance in data-intensive applications. We compare three strategies on Fujitsu FX10 supercomputer: RamDisk, Every-node, and Dedicated-node. Moreover, for comparison of in-memory storage and spilling data to the central disk, we compare three strategies with another configuration called *Central-disk*. The largest size of input data is 120 GB. The number of reducers is set to 128 for all datasets. Note that running time is 0 (zero) denoting that out of memory happens and the corresponding job is stopped.

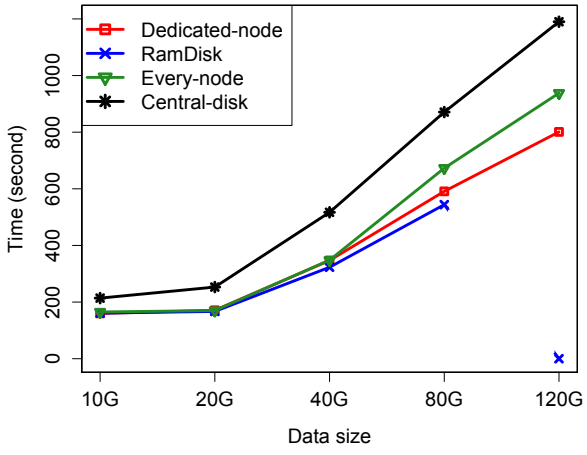
Regarding fairness of this experiment, all three configurations use the same total number of nodes. In case of dedicated-node, some nodes are assigned to only storing data, but they are used for computation as well in Ramdisk and every-node. In this experiment, while we use 32 nodes for computation and 4 nodes for SEMem storage, 36 nodes are allocated for computation in Ramdisk and every-node configurations.



(A) WordCount



(B) InvertedIndex



(C) SequenceCount

FIGURE 4.7: *Running time on different deployment strategies: zero at 120GB at RamDisk denotes that out of memory happens and the corresponding job is stopped.*

Figure 4.7 reveals that RamDisk is the fastest strategy, but out of memory happens when data size is bigger than 120GB. RamDisk is the fastest deployment strategy since data is stored at local memory and no inter-node communication. Moreover, there is no complex distributed in-memory storage installed on the RamDisk strategy and intensive data movement among nodes. However, due to uneven distribution of input data, intermediate data is not fit into memory at some nodes. That is why out of memory happens.

Dedicated-node shows a better performance than every-node when data size is bigger than 40GB. It is 10% improvement on average in both WordCount and InvertedIndex, and it is 13% on average faster in SequenceCount. When the size of dataset is bigger, the dedicated-node strategy shows a bit higher improvement than the every-node strategy, for example 12% and 14% for 80 GB and 120 GB datasets, respectively, in SequenceCount. The dedicated-node strategy has achieved better performance than the every-node one since there is no complex distributed in-memory storage running on computation nodes and the bottleneck of data fetching at those nodes. In the every-node deployment, the more complex SEMem daemon disturbs computation tasks at several places, for example network bandwidth, CPU time, and memory, to a certain degree. On the other hand, the computation nodes' resources are used mainly for computation tasks and in-memory management is offloaded to dedicated nodes (or memory nodes) in the dedicated-node strategy. A network bottleneck could happen at memory nodes, but the high-speed network combining with using MPI and RDMA on supercomputers could speed up sending and receiving data between memory and computation nodes. In case of SequenceCount, since the size of intermediate data is larger than in WordCount and InvertedIndex, there is more improvement.

Compared with central-disk configuration, dedicated-node is always faster, especially 32% improvement in SequenceCount workload, whereas every-node is slower in WordCount and InvertedIndex, but faster in SequenceCount 26% on average. In WordCount and InvertedIndex workloads, every-node is slower than central-disk configuration since more communication is required and computation nodes might be busier when SEMem daemon is run. By contrast, central-disk is the slowest configuration in SequenceCount because the size of intermediate data is larger.

Every-node performance for the dataset less than 80 GB should be similar to the RamDisk, but our experiment showed that it is slower since the configuration of every-node is more complex and more communication required. We run one more daemon on each node to handle in-memory storage in the every-node deployment. That daemon is responsible for receiving and sending the intermediate data from mappers and reducers. However, in the RamDisk deployment, those data are stored directly by the *shuffle*

server. A separate daemon makes the *shuffle server* less busier and helps organize in-memory storage more efficiently, but the entire compute node's performance might be affected. Moreover, in the every-node deployment, the size of in-memory storage must be fixed in order to ensure that out of memory will not happen since we know how big the size of in-memory storage can be allocated. Data can be stored on the local node as long as the total used size is smaller than the maximum size. We set 8 GB in the experiment. In the RamDisk deployment, data can be written to the in-memory storage as long as possible, but out of memory can happen at any time. From the experimental results, the overhead is significant for small data size, but when the data size is bigger, every-node is faster than the central disk.

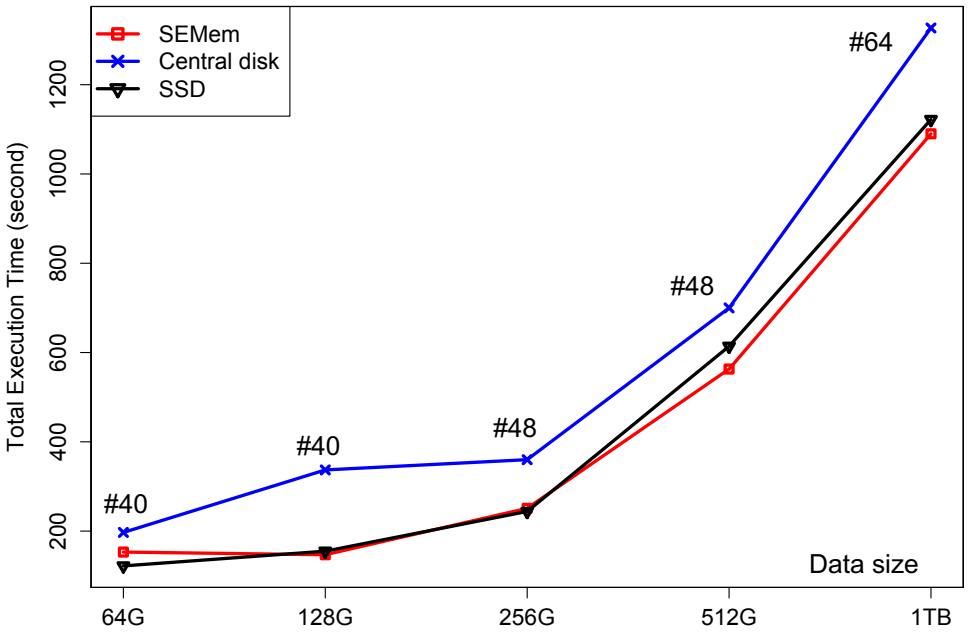
4.5.2 SEMem (dedicated-node) vs. central-disk (HDD) and SSD

This experiment is aimed to show how fast SEMem is in comparison with the central-disk-based and SSD-based approaches. We compare three test configurations on TSUBAME: the central-disk-based Hadoop, SSD-based Hadoop, and our approach (SEMem). Tera-sort is used for comparison and input data is generated by tera-gen. The largest size of input data is 1 Terabyte. We run each experiment of a data size twice and calculate the average execution time. The number of reducers is fixed to 128 for all data sizes. The number of nodes (#) is denoted in the figure.

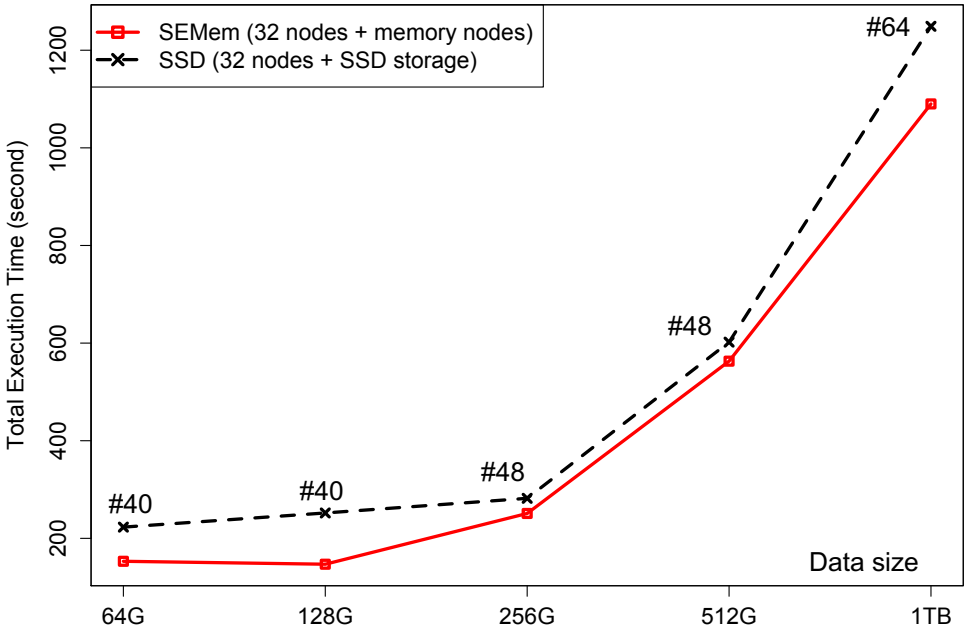
In this experiment, SEMem runs on dedicated nodes and for fairness of this experiment, all three configurations use the same total number of nodes. In case of SEMem, some nodes are assigned to only storing data, whereas those nodes are used for computation in the central-disk and SSD configurations. That means SEMem uses a smaller number of computation nodes.

Figure 4.8a shows that as the size of input data increases, SEMem is faster than central-disk-based storage, and close to SSD-based storage. Compared to central disk and SSD storage, SEMem reduces execution time by 20% and 5%, respectively, on average when data size is bigger than 512 GB. SSD storage has better performance when the data size is less than 256 GB.

Choosing SSD or SEMem storage? at the supercomputer centers, when they design a new supercomputer, a question is that whether each node should be equipped with a SSD storage or not. SSD storage is considered difficult for maintenance and it can be a point of failure. For example, data on SSD storage of TSUBAME must be deleted manually after each job. If they do not want to install SSD, a question is that SEMem can be an alternative. Moreover, if SSD is available and also become a paid resource on supercomputers, whether supercomputer users should buy SSD storage or choose



(A) SEMem vs. Central disk and SSD

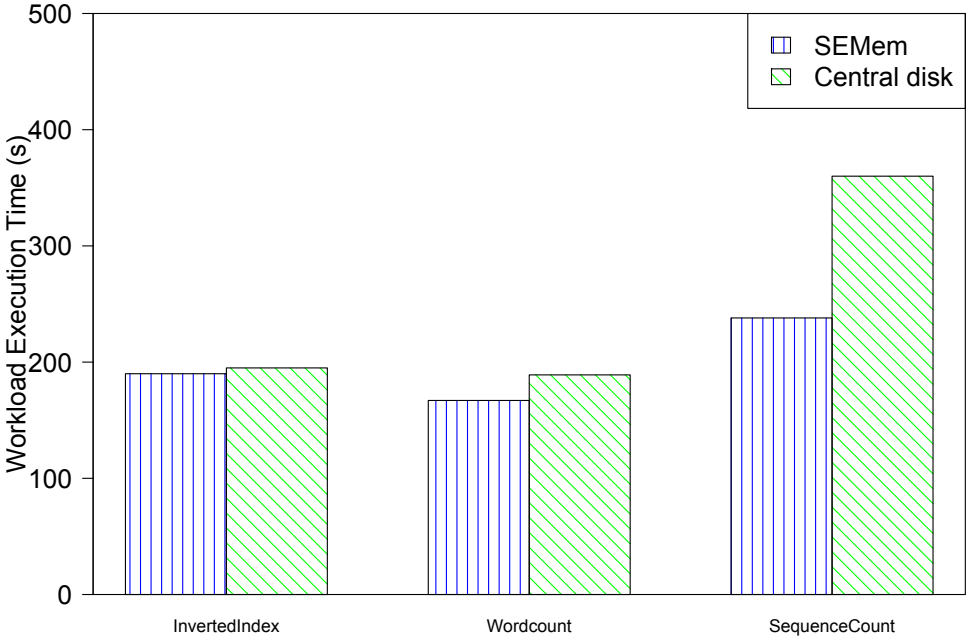


(B) Choosing SSD or compute nodes?

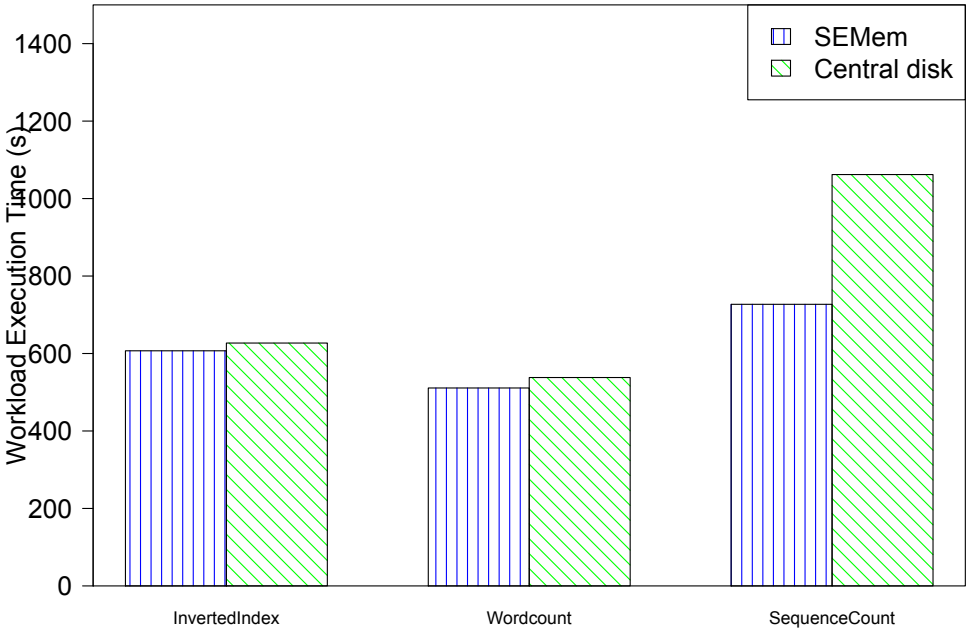
FIGURE 4.8: *SEMem vs. Central disk and SSD*

to increase number of computation nodes in order to create SEMem. We design an experiment to answer those questions. This experiment is conducted on TSUBAME.

We keep the same number of computation nodes for both SEMem and SSD test configurations. We assume that SEMem and SSD are two types of resources that users can choose. SEMem has a defined number of nodes used for storage. Tera-sort is used for comparison and maximum input data is 1 Terabyte.



(A) On TSUBAME



(B) On Fujitsu FX10

FIGURE 4.9: *Purdue MapReduce benchmark*

Figure 4.8b reveals that SEMem is always faster than SSD storage, especially 41% improvement at 128 GB of input data. When input size is 1 Terabyte, the improvement is 13% in comparison with SSD storage, but SEMem helps decrease the total execution time (1249 seconds) by 159 seconds. The figure shows that SEMem is feasible to become an alternative of SSD storage.

4.5.2.1 Other benchmarks

To evaluate SEMem with other Hadoop MapReduce workloads, we use Purdue benchmark suite [86]. InvertedIndex, Wordcount, and SequenceCount workloads use 120GB Wikipedia data. The number of reducers is set to 128 for all experiments.

Figure 4.9a shows performance of three workloads using SEMem and the central disk on TSUBAME supercomputer. InvertedIndex and Wordcount show a modest improvement (less than 10%) when using SEMem since the size of intermediate data is small, whereas SEMem reduces execution time of SequenceCount by 33%.

4.5.2.2 Results on Fujitsu FX10 supercomputer

In order to evaluate our approach on other different supercomputers, we run SEMem on Fujitsu FX10 where SSD is not available. When SSD is not available, SEMem can become more useful. Note that FX10 does not has large memory size that makes SEMem have to require more memory nodes. We run tera-sort application with different input sizes and the same micro benchmark as on TSUBAME.

Figure 4.10 shows the results of tera-sort running Fujitsu FX10. Compared with the central disk configuration, SEMem achieves improvement of 25% on average. Although the input sizes are small, SEMem shows a good improvement. In the Figure 4.9b of

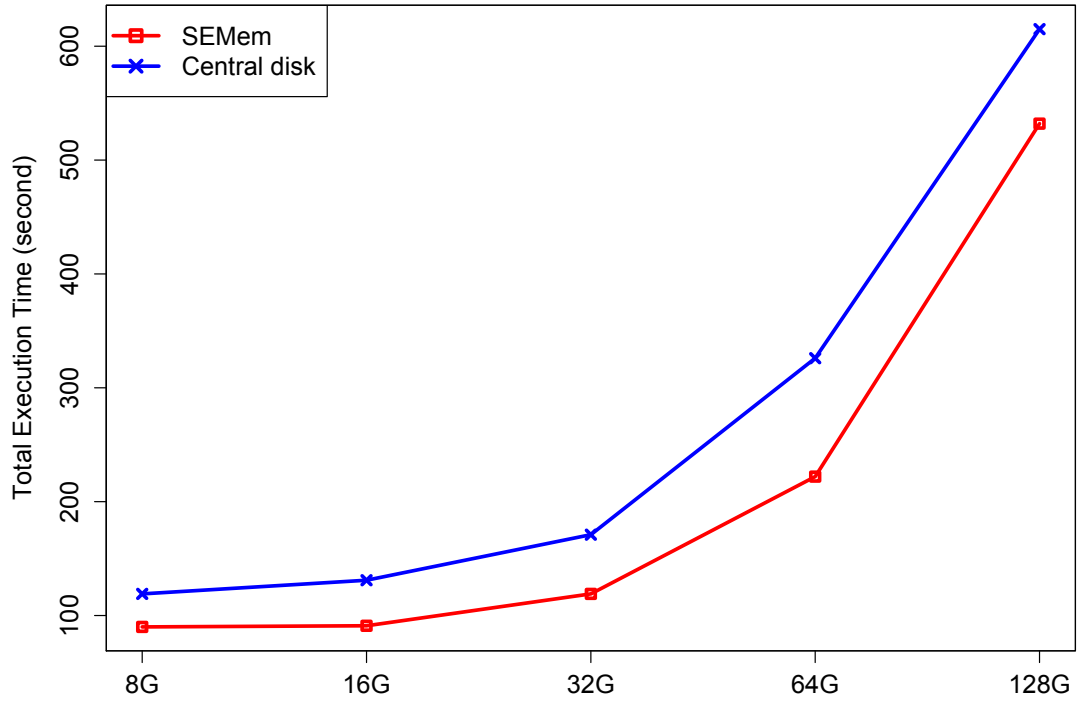


FIGURE 4.10: *SEMem vs. Central disk on Fujitsu FX10*

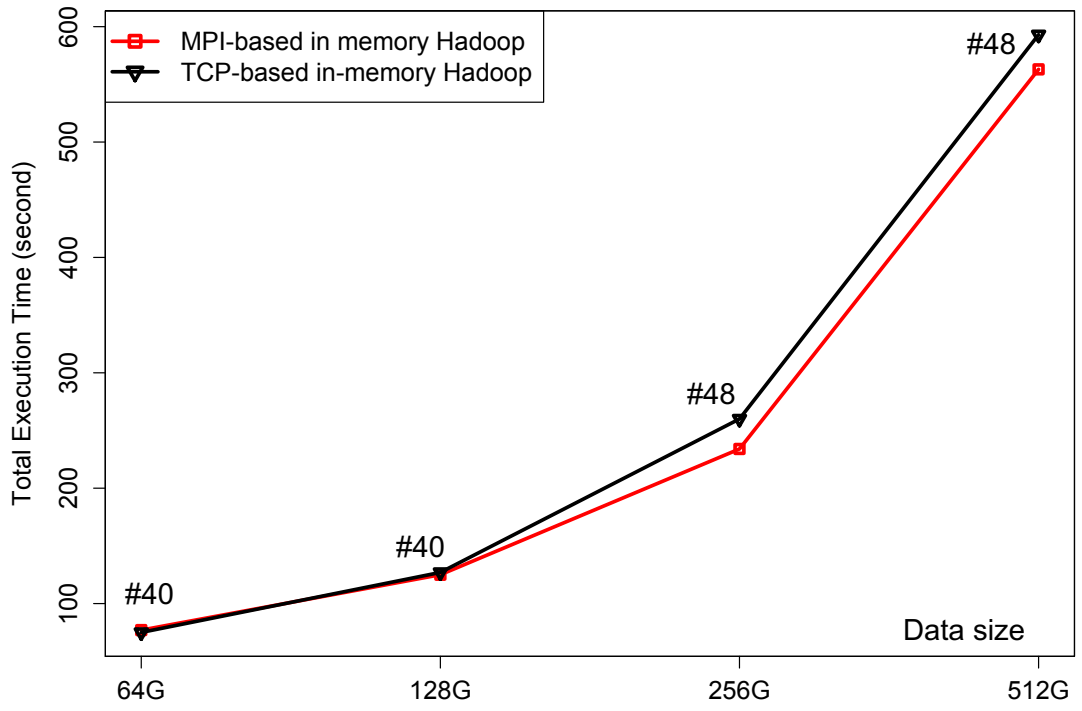


FIGURE 4.11: *MPI-based vs. TCP-based protocols*

micro benchmark running on FX10, reduction is small in InvertedIndex and Wordcount, but SEMem outperforms the central disk in SequenceCount workload.

4.5.3 Communication protocol

This experiment is aimed to check how fast our MPI implementation on SEMem is in comparison with TCP communication and whether the improvement achieved by using SEMem comes mainly from in-memory storing or fast MPI communication. We run the original Hadoop in memory and compare it with SEMem-based Hadoop. The original Hadoop uses TCP communication for data exchange between reducers and shuffle servers. This comparison is fair because SEMem is an in-memory storage, but use MPI for data exchange. Tera-sort workload is used for comparison and the number of nodes (#) is denoted in the figure.

Figure 4.11 shows there is no difference when the size of input data is smaller than 128 GB. However, MPI-based Hadoop is faster 10% and 5%, respectively, when the input sizes are 256 and 512 GB. Compared with Figure 4.8a at the data size of 256 GB, MPI communication contributes less than 20% to the performance improvement by using SEMem. This experiment has proved that the main source of improvement (80%) comes from in-memory storing on memory nodes of SEMem. Moreover, in comparison

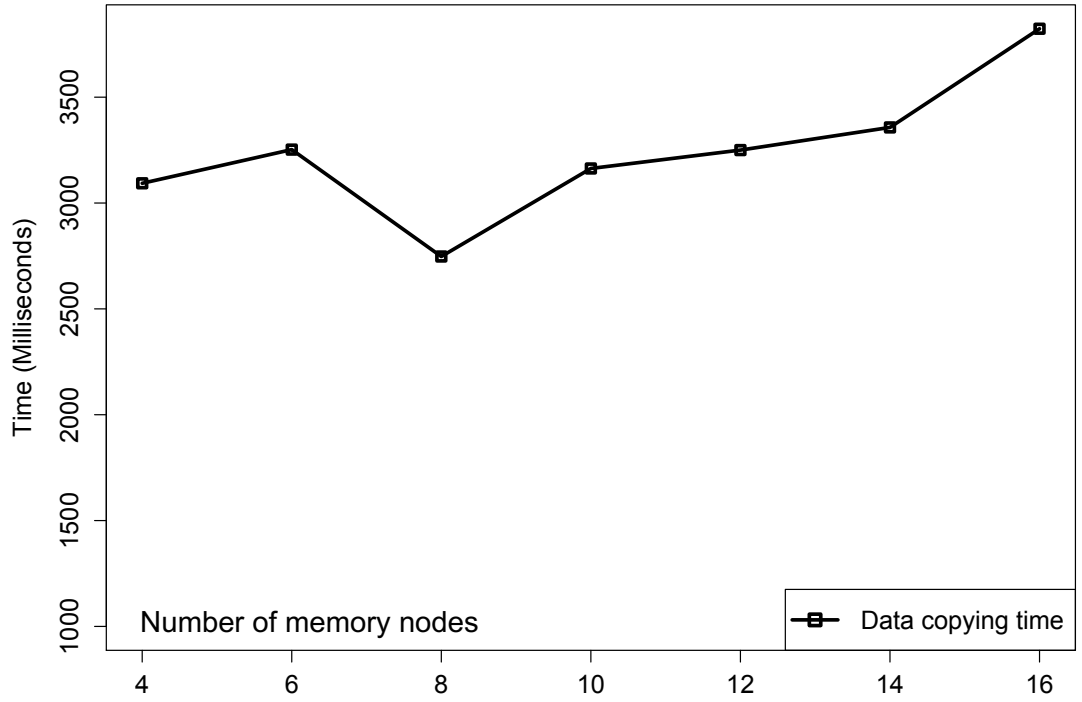


FIGURE 4.12: *How many memory nodes are enough*

with the MPI experiment in Section 3.3.2 of Chapter 3, the improvement is increased to up to 10% since the bottleneck of disk I/O in our MPI communication design is removed.

4.5.4 Storage size of SEMem

This experiment is designed to measure performance impacts of storage size. SEMem's memory capacity can be configured based on number of memory nodes. We have conducted an experiment by keeping the same number of computation nodes and changing number of memory nodes. The purpose of this experiment is to check whether our current approach of finding the number of memory nodes is effective or not. We always run tera-sort application on 32 computation nodes. The number of memory nodes ranges from 4 to 16. The size of input data is 64 GB generated using tera-gen application. According to our estimation, the required number of memory nodes should be $4 \sim 8$ nodes. We measure data copying time (between shuffle servers and memory nodes and reducers) rather than total execution time. Figure 4.12 shows that when the number of memory nodes is 8, data is copied fastest. However, if increasing the number of memory nodes from 8 to 16, the copying time is slower due to complexity in node management.

In order to show when out of memory happens if we change the number of memory nodes. Figure 4.13 reveals that for 64 GB of the problem size, out of memory occurs when the number of memory nodes is two.

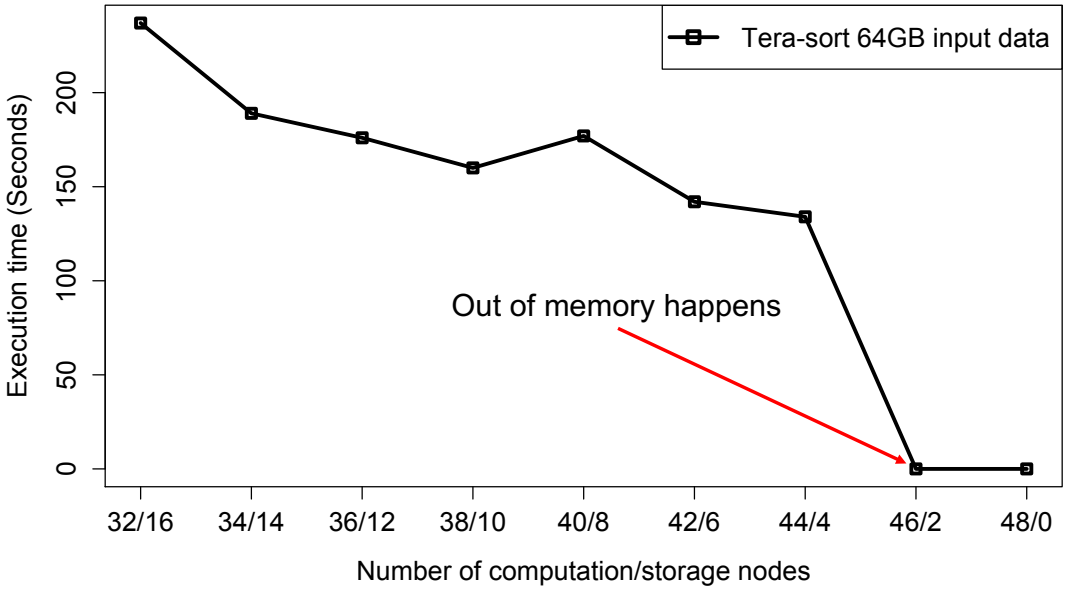


FIGURE 4.13: *When out of memory happens*

4.6 Related work

Memcached software [33] is close to our implementation of SEMem since it is a distributed caching system. The combination of Hadoop and Memcached is a study topic. However, to the best our knowledge, there is no study on using Memcached to store intermediate data implicitly in Hadoop MapReduce workflow. We also evaluated advantages of that combination SEMem and Hadoop. Memcached uses TCP-based communication and RDMA-based Memcached [78] is an extension that speeds up internode communication by using RDMA. In our SEMem, we use MPI for communication among memory nodes. RDMA could be also enabled automatically in MPI communication on supercomputers.

DASH [29] has the same motivation of decreasing latency gap between main memory and hard disks. It proposed an storage architecture exploiting SSD and DRAM to fill that latency gap. DASH is a remote distributed in-memory storage and equipped with SSDs. However, such storage architecture is not scalable since total storage size is fixed. The storage size of our approach, SEMem, can be extended by adding more memory nodes.

HydraDB [91] proposed an in-memory layer supported by RDMA that is located between computation frameworks, such as Hadoop and Spark, and storage. It mainly focuses on

reading and writing from/to HDFS. However, SEMem is based on MPI communication and aimed to store intermediate data.

FARM [92] is a distributed main memory platform that enables to process large-scale datasets in memory with fast network communication, RDMA, in order to improve latency and throughput. Nevertheless, our SEMem approach uses memory on computation nodes exclusively for data storage. Moreover, SEMem has exploited MPI supported by RDMA to speed up data exchange.

4.7 Discussion and Summary

We have examined in-memory storage deployment strategies including RamDisk, Every-node, and Dedicated-node. For experiments, we have designed our own memcached-like file system called SEMem. Dedicated-node shows a good result in the Puma benchmark with up to 13% improvement of the SequenceCount workload in comparison with Every-node strategy. Dedicated-node is motivated by a feature of supercomputers that the number of compute nodes are often huge and supercomputer users can request a number of nodes they need. Computation tasks are not run on those nodes.

By answering the question of which deployment strategy of in-memory storage is good, our SEMem provides efficient virtual local disks that help reduce total execution time as much as 32%. Moreover, SEMem is easily configurable for different deployment strategies. Our own in-memory storage is tightly integrated with the implementation of Hadoop and using MPI for its communication protocol, but it can be used by other popular data-intensive frameworks, such as Spark and Flink. The benefit of dedicated-node has been achieved in the context in which the ratio of the dedicated nodes to the total nodes is small. If the number of dedicated node is bigger than the number of nodes used for computation, the application should run slow due to the bottleneck of running computation tasks.

Chapter 5

Performance benefit of both HPC-Reuse and SEMem

In this chapter, we combine HPC-Reuse and SEMem and conduct experiments in order to evaluate the overall performance improvement in comparison with the original data-intensive frameworks running on supercomputers. Although SEMem is already using HPC-Reuse to enable MPI communication in Chapter 4 in which all experiments of its evaluation were run on HPC-Reuse for the fairness, the experimental results in this chapter will give us a conclusive proof of our research by comparing the original TCP-based Hadoop without running on HPC-Reuse and SEMem, only using HPC-Reuse, and combination of HPC-Reuse and SEMem. Note that Hadoop is our targeted framework for experimental evaluations.

5.1 Experimental results

We show the experimental results of two benchmarks. One is Tera-sort that was already used in comparison of the TSUBAME supercomputer and Amazon EMR, and SEMem and the central disk. Another one is the PUMA benchmark [86] including several workloads. We compare the original implementation of Hadoop without modification and the obtained results from the previous chapters.

5.1.1 Tera-sort benchmark

The purpose of this experiment is aimed to show how fast the combination of HPC-Reuse and SEMem deployed in the dedicated-node strategy can achieve in comparison with a

naive deployment of the unmodified original data-intensive frameworks on supercomputers when running workloads having large-scale intermediate data, for example Tera-sort. We compare four test cases on TSUBAME including the original Hadoop, HPC-Reuse-central-disk Hadoop, HPC-Reuse-SSD Hadoop, and HPC-Reuse-SEMem Hadoop that are described as follows:

- The original Hadoop: refers to the unmodified original implementation that is used as the baseline for comparison.
- HPC-Reuse-central-disk Hadoop: refers to using HPC-Reuse and writing intermediate data to the central disk.
- HPC-Reuse-SSD Hadoop: refers to using HPC-Reuse and writing intermediate data to local SSDs.
- HPC-Reuse-SEMem Hadoop: refers to using both HPC-Reuse and SEMem.

In this experiment, the four test cases have the same Hadoop configuration and the total number of compute nodes. However, since SEMem requires dedicated nodes used for in-memory storage, the number of compute nodes in HPC-Reuse-SEMem Hadoop is smaller depending on how many memory nodes are used. The number of memory nodes is estimated based on the discussion in Section 4.4.2. The cluster chosen for the comparison has 32 nodes and the problem size is 128 GB.

In Figure 5.1, compared with the original Hadoop, HPC-Reuse-SEMem Hadoop, our combination of HPC-Reuse and SEMem, is faster 61% at the 128 GB of problem size. The intermediate data of the original Hadoop is written to the central disk, so HPC-Reuse-central-disk Hadoop is the main baseline. HPC-Reuse-central-disk Hadoop achieved 12% improvement, HPC-Reuse-SSD Hadoop showed 59% faster.

5.1.2 PUMA benchmark

In this experiment, we show the performance comparison of the unmodified original Hadoop and different deployment strategies using both HPC-Reuse and SEMem. Compared with the experiment in Section 4.7, the original Hadoop of this experiment is not based on HPC-Reuse. There are five test cases as follows:

- The original Hadoop: is the same as in the Tera-sort benchmark.
- Central-disk Hadoop: refers to using HPC-Reuse.

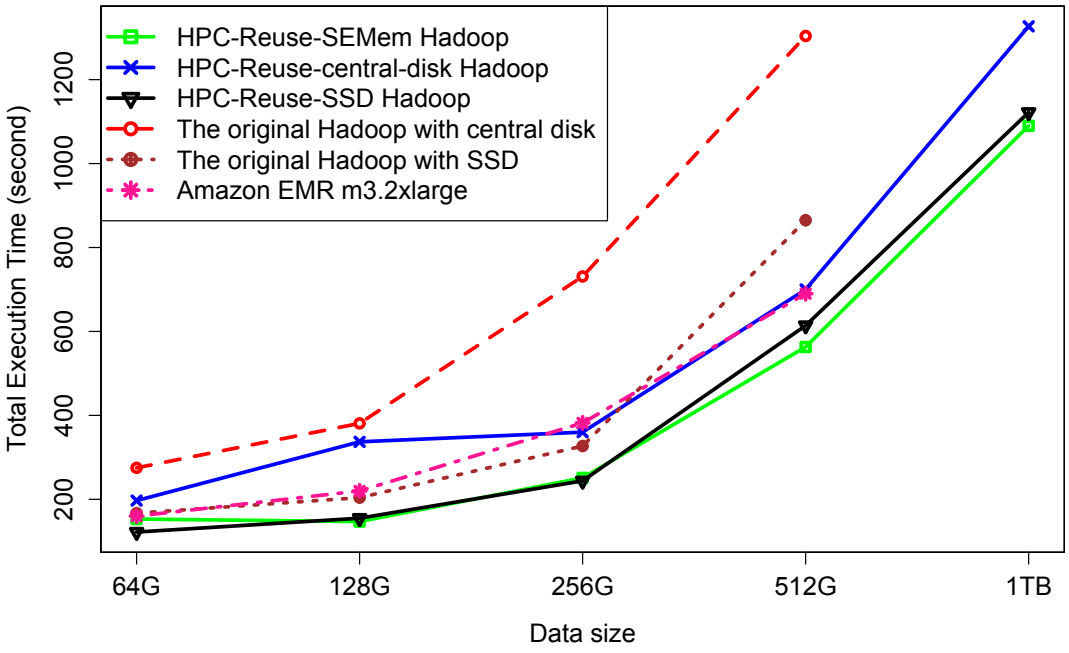


FIGURE 5.1: *Performance of our approaches: four test cases*

- Dedicated-node Hadoop: refers to using both HPC-Reuse and SEMem deployed in the dedicated-node strategy.
- RamDisk Hadoop: refers to using both HPC-Reuse and SEMem deployed in the RamDisk strategy.
- Every-node Hadoop: refers to using both HPC-Reuse and SEMem deployed in the every-node strategy.

We use the PUMA benchmark including WordCount, InvertedIndex, and SequenceCount workloads. The cluster has 32 nodes and the problem size ranges from 10 GB to 120 GB.

The Figure 5.2, 5.3, and 5.4 show that the central-disk Hadoop using HPC-Reuse is faster 12%, 11%, and 25% on average than the original running in WordCount, InvertedIndex, and SequenceCount, respectively. Regarding SEMem deployment strategies, in the WordCount workload, RamDisk, dedicated-node, and every-node achieve 36%, 16%, and 10% on average, respectively. In the InvertedIndex workload, RamDisk, dedicated-node, and every-node achieve 34%, 14%, and 9% on average, respectively. In the SequenceCount workload, RamDisk, dedicated-node, and every-node achieve 61%, 48%, and 45% on average, respectively.

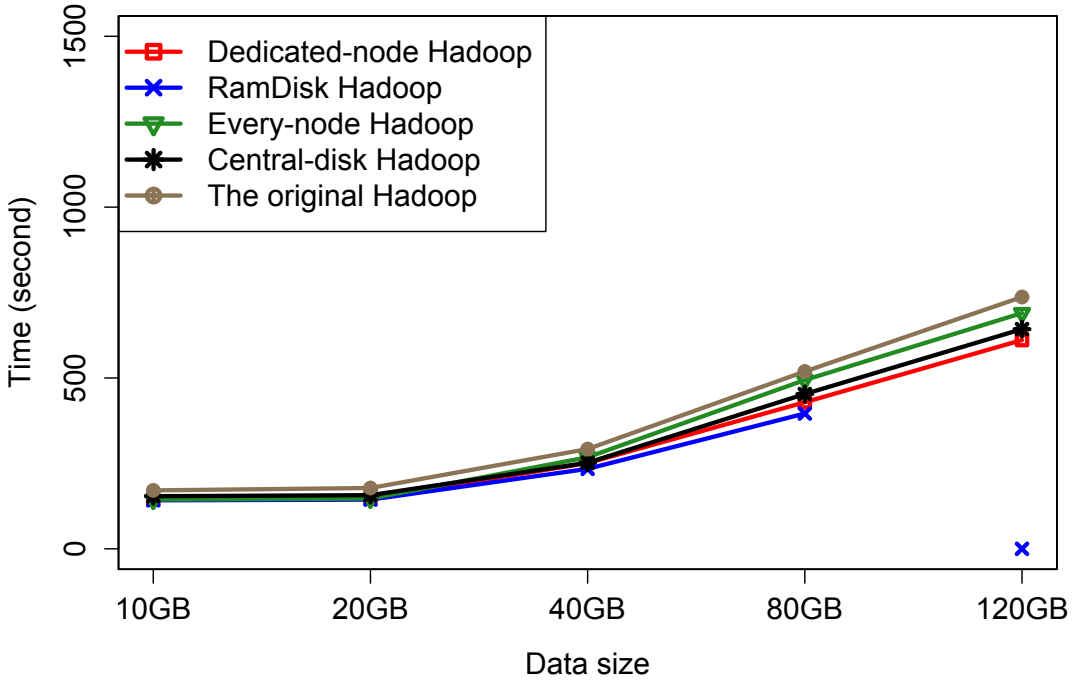


FIGURE 5.2: *PUMA benchmark: WordCount*

5.2 Hadoop's source code changes

The goal of achieving a better performance without large source code modifications on supercomputers is feasible since the source code changes are not much in comparison with the size of Hadoop codebase. Figure 5.5 shows how many lines of code and classes we modified in the original Hadoop source code. In Hadoop v2.2 [93], the total lines of code (LOC) is more than 1.8 millions and there are around 35,142 class files. In order to work with HPC-Reuse, we modified mainly **ContainerExecutor** class with 336 LOC. **MapTask** and **Fetch** classes are changed to communicate with SEMem, and the total changes are 443 LOC.

5.3 Discussion

The overall improvement obtained by using HPC-Reuse and SEMem is 61% faster than the original Hadoop without modification. HPC-Reuse is making up 12% and SEMem contributes 49% improvement on average in the tera-sort workload at the problem size of 128 GB. When SSD is not available, the performance benefit from SEMem is large. If

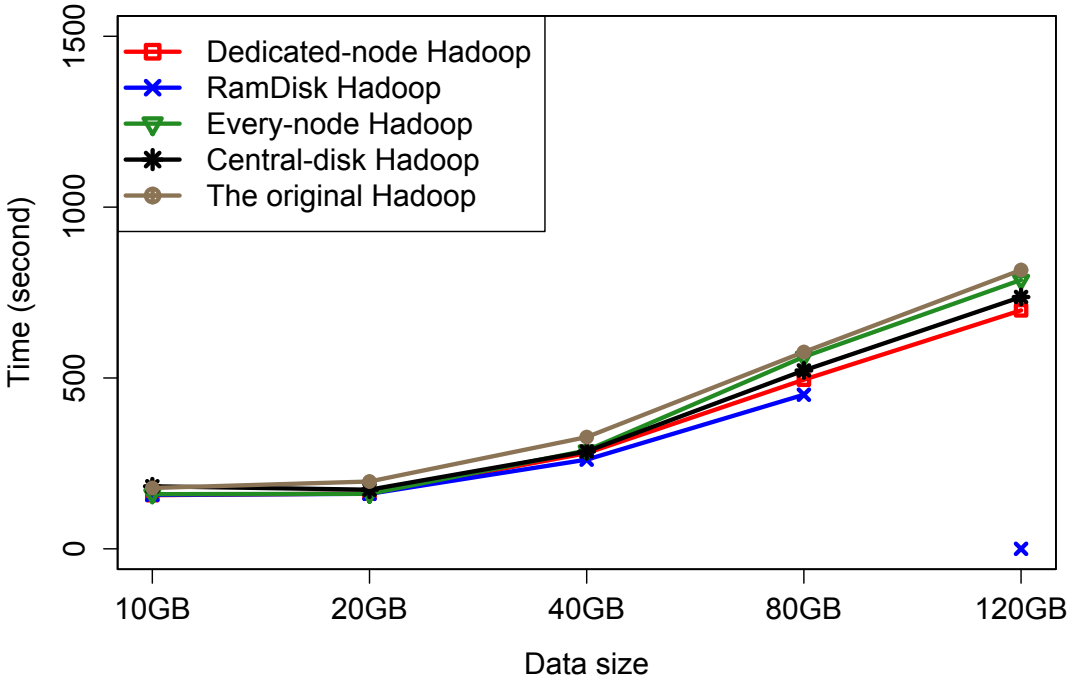


FIGURE 5.3: *PUMA benchmark: InvertedIndex*

SSD is installed on supercomputers, the improvement is only significant with large-scale datasets, for example 512 GB and 1 TB.

In the Figure 5.1, we have only experimental results of the original Hadoop at 64 GB and 128 GB. However, the trend showed that our proposals are outperforming the original Hadoop deployed on supercomputers. For the larger datasets, the performance of the original Hadoop should be much slower.

In the WordCount and InvertedIndex workloads, while HPC-Reuse contributes 11% faster on average, SEMem shows a modest improvement with as much as 5%. However, in the SequenceCount workload, HPC-Reuse and SEMem achieve 25% and up to 23%, respectively. The data size in PUMA benchmark is not big that is why HPC-Reuse showed more improvement.

5.3.1 Scalability

Regarding the scalability in our experiments, it has showed that our proposal is scalable up to 128 nodes since we deployed our HPC-Reuse and SEMem Hadoop on up to 128 compute nodes and on each compute, we run up to 8 task processes in the experiments. Those experiments run stably on that configuration. We expect that our system can

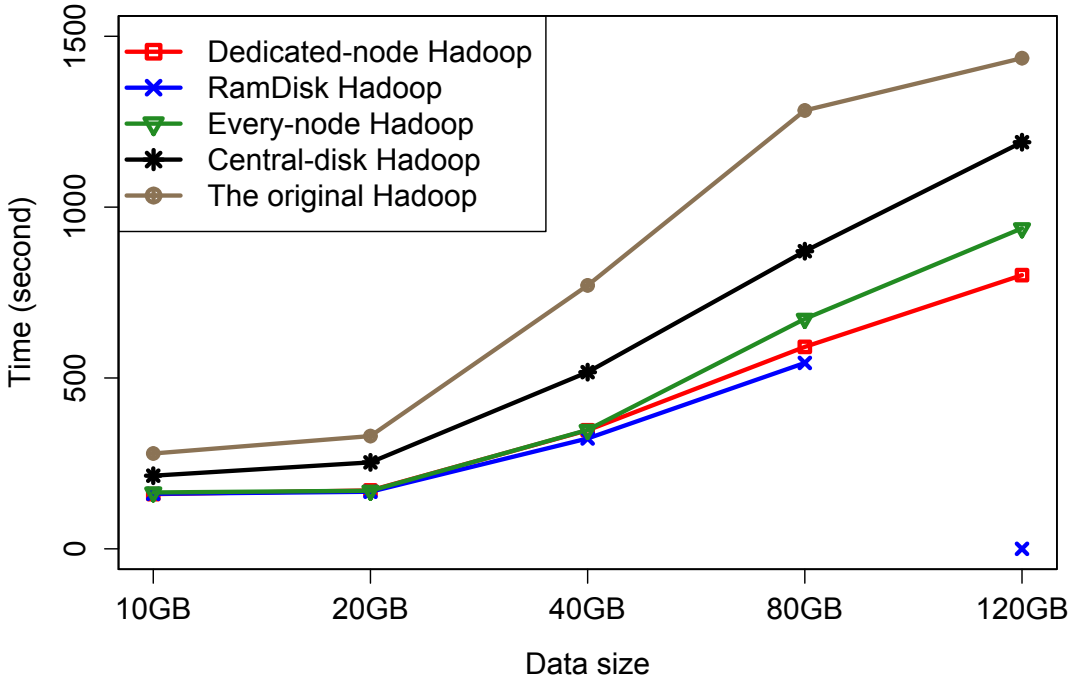


FIGURE 5.4: *PUMA benchmark: SequenceCount*

execute workloads stably having up to 1024 task processes. If we consider that one process is equivalent to one compute node, we expect that our system could scale on the cluster containing 1024 nodes. Compared to typical commodity clusters configured at the laboratory or cloud services, the number of nodes is big enough.

We expect that our system can be scaled out to handle bigger datasets since the performance could be improved more and the trend can be seen as shown in Section 3.3.1, 4.5.2, and 5.1.1. The biggest data size in our experiments was one terabytes and the figures have showed that when the size of datasets increases, the performance improvement is going up gradually. It means that our proposal could be efficient for large-scale datasets.

We also expect that HPC-Reuse and SEMem Hadoop is as scalable as the original Hadoop that was already deployed on thousands of nodes at Yahoo [94]. HPC-Reuse and SEMem are just a para-virtualization layer that do not affect much on the scalability of Hadoop. While the HPC-Reuse daemon is just monitoring a node, the SEMem daemon on the dedicated-node strategy is responsible for memory nodes whose ratio to the total number of nodes is small. Our HPC-Reuse does not support fault tolerance that is the drawback of our system when the number of nodes is large. That is still our future work.

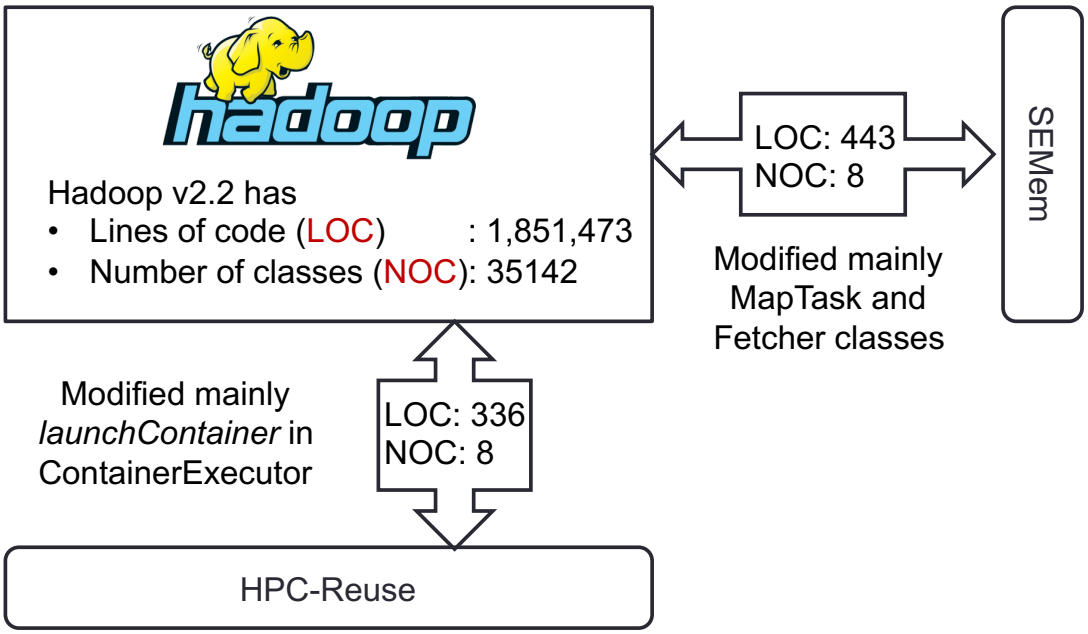


FIGURE 5.5: *Hadoop's source code modifications*

Regarding the strong and weak scaling, there are several results in our experiments showing such measurement. In the Figure 5.1, the number of computation nodes was adjusted in the small range, and the problem size is increased. The figure's trend has showed a constant improvement as the weak scaling. The PageRank experiments in Figure 3.14 have presented our system's strong scaling by showing that when the number of nodes increases from 32 to 128 computation nodes, the more improvement was achieved.

5.3.2 Effectiveness

We show that our proposal has satisfied the requirements of an effective deployment defined in Section 2.7 as follows:

- HPC-Reuse enables using MPI communication in the effective way by starting all processes simultaneously and SEMem enables a fast virtual local disk based on memory that helps avoid expensive disk I/O to the central disk.
- The changes in Hadoop is small when using on the top of HPC-Reuse and SEMem as shown in Section 5.2. Minimizing modifications helps increase productivity and ease of upgrade.
- We did not evaluate how long the deployment of Hadoop cluster takes, but it should be short in comparison with the total execution time of applications in our experiments.

Chapter 6

Conclusions

This thesis is about proposing an effective deployment of popular data-intensive frameworks, such as Hadoop and Spark, with minimum source code modifications when they are run on modern supercomputers rather than on commodity clusters where they are typically widely used. By observing two mismatches related to lack of MPI-friendly dynamic process creation and local disks, we show the inefficient way of using network communication and performance bottleneck of the central disk in the default deployment on supercomputers. Commodity clusters are the targeted environment for the data-intensive frameworks, but supercomputers designed differently from commodity clusters cause those mismatches. The naive default deployment of those frameworks on supercomputers is considered not effective. Note that our targeted framework in implementation and evaluations is Hadoop MapReduce. As shown through the experimental results in Chapter 3, 4, and 5, our proposal of HPC-Reuse and SEMem have resolved the two mismatches and made the deployment of Hadoop-like popular data-intensive frameworks effective on the modern supercomputers.

Table 6.1 shows the comparison of our proposal, HPC-Reuse and SEMem, and other approaches mentioned in Section 2.6. Our proposal is using JVM Reuse that helps reduce start-up time and supporting MPI communication among processes following an effective mechanism that does not require MPI-Spawn or the process *fork()*. Our SEMem enables in-memory storage and supports different deployment strategies. Moreover, it also provides simple data affinity. Our proposal is as good as Hadoop in aspects of productivity and maturity, but it is more compatible since it can be deployed on supercomputers in an effective way. Its scalability is as high as Hadoop since HPC-Reuse and SEMem can be deployed on a large number of nodes.

Regarding the data size that our system can handle, since HPC-Reuse-SEMem Hadoop is as strong as the original Hadoop, we expect that it is possible to handle and process

	HPC-Reuse-SEMem Hadoop (our proposal)	Hadoop	K MapReduce	DataMPI	M3R	Spark
Dynamic process creation						
Process creation	JVM reuse	Fork	MPI-spawn	MPI-spawn & reuse	JVM reuse	Fork& reuse
MPI support	Yes	No	Yes	Yes	Might	No
Local disks						
In-memory storage	Yes	No	No	No	Yes	Yes
Deployment strategy	Yes	No	No	No	No	No
Other criteria						
Overall	Good	Good	Bad	Bad	Neutral	Good
Productivity	High	High	Low	Low	High	High
Maturity	High	High	Low	Low	Fair	High
Compatibility	High	Fair	Low	Fair	Low	Fair
Scalability	High	High	Fair	Fair	High	High

TABLE 6.1: Our proposal vs. existing solutions

large-scale datasets. In our experiments, although 1 TB is the biggest dataset we have tested, the trend in the Figure 4.8 and 4.11 showed that it could be scalable for larger datasets.

Our proposals of HPC-Reuse and SEMem help achieve a better performance of data-intensive frameworks when running on supercomputers in general. From our experiments, sorting algorithm and iterative MapReduce applications are getting much benefit. HPC-Reuse enables faster process creation, so it is good for iterative MapReduce applications, for example PageRank and k-means clustering, or any application to create many processes. HPC-Reuse enables also MPI communication, so it is good for

	HPC-Reuse-SEMem Hadoop (our proposal)	Hadoop	K MapReduce	DataMPI	M3R	Spark
Dynamic process creation						
Process creation	JVM reuse	Fork	MPI-spawn	MPI-spawn & reuse	JVM reuse	Fork & reuse
•Slot resizing	No	Yes	Yes	No	Yes	Yes
MPI support	Yes	No	Yes	Yes	Might	No
•Fault tolerance	No	Yes	No	Yes	Might	Yes
Local disks						
In-memory storage	Yes	No	No	No	Yes	Yes
•Data affinity	Partial	No	No	No	Partial	Yes
•Multiple levels	No	Yes	No	No	No	Yes
Deployment strategy	Yes	No	No	No	No	No
•Preferred locations	No	No	No	No	Might	Yes

TABLE 6.2: Limitations of our proposal

network-intensive applications as well, such as Tera-sort. Regarding SEMem, it enables efficient virtual local disks, so it is good for applications having large intermediate data.

6.1 Limitations

Regarding process creation, our HPC-Reuse cannot resize the number of slots in a node. M3R supports adding more slots, namely *place* in X10 [95], on the fly [96]. Spark also provides a mechanism in order to resize the number of slots called *executor* in Spark [97]. That drawback limits the scalability of our proposed HPC-Reuse when we want to increase the number of parallel tasks that are able to run on a node. Although

increasing the number of slots helps improve parallelism, but the performance can be degraded since compute, memory, and network bandwidth resources are limited.

Since MPI does not support fault-tolerance in its implementation on supercomputers, our HPC-Reuse process pool will be terminated when any process in the pool fails due to errors. As our future work, it is possible to implement a checkpoint mechanism for HPC-Reuse in order to restart its state when an error happens. DataMPI supports both MPI and fault tolerance since DataMPI is a pure MPI framework. It is also not clear that if M3R is supporting MPI fault tolerance or not.

SEMem partially supports data affinity, for example users can configure how to place data into the in-memory storage including round-robin style and storing on a node as much as possible. M3R supports where data is stored through *places* and *activities* operators, but they are implicitly controlled through X10. In Spark, it is possible to choose a location for a RDD through *preferredLocations()* operator.

When in-memory storage is full, out of memory will happen and our system cannot extend to store in the next level of storage, for example SSD and the central disk. Although the original Hadoop supports that feature, but there is no connection of our implementation of in-memory storage and that configuration. Spark is fully supporting multiple storage levels, such as memory, local disks, SSD, and the central disk.

Our SEMem does not provide exactly which node data will be stored that is called *preferred location*. That feature is useful when users want to exploit data locality for running tasks, for example running reducing tasks on the same nodes where intermediate data is available. Spark is fully supporting that feature by putting the preferred location information into its RDD data model. M3R supports partially and implicitly through the *place* of X10 programming language.

6.2 Future work

HPC-Reuse: our future work is to add more features and improvement, such as full clean-up including static fields and heap memory and a slot resizing mechanism mentioned in the previous section. We also have a plan to redesign our MPI shuffle engine in order to support fault tolerance.

Estimating number of memory nodes: the number of memory nodes affects waiting time in the job queue on supercomputers and job performance as well. The size of intermediate data is always changed that depends on the type of applications. In our current design, we estimate roughly the number of memory nodes based on the size

of input data. That way does not achieve the best performance. We are going to use machine learning to study sample data first and then run the real application later. Moreover, one more straight solution is to add more memory nodes on the fly by submitting another job to the job queue. However, it is only possible if the supercomputer job queue allows to combine different running jobs, but the waiting time on the job queue is another problem that needs to be considered.

Topology of memory nodes: on some supercomputers (e.g. Fujitsu FX10 using Tofu interconnection), compute nodes can be allocated by a shape of 2D or 3D. A relevant node topology can improve job performance since it helps increase throughput and decrease latency among nodes. Note that we used the default shape in our current experiments.

Bibliography

[1] Allan Snavely, Gwen Jacobs, David A. Bader, and Ananth Grama. Workshop report: Petascale computing in the biological sciences. 2006.

[2] The National and Science Foundation. Workshop Report : Petascale Computing in the Geosciences. 2011.

[3] Wullianallur Raghupathi and Viju Raghupathi. Big data analytics in healthcare: promise and potential. *Health information science and systems*, 2(1):3, 2014.

[4] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, et al. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1071–1080. ACM, 2011.

[5] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[6] Xiao Cai, Feiping Nie, and Heng Huang. Multi-view k-means clustering on big data. In *IJCAI*, pages 2598–2604, 2013.

[7] Zhenlong Li, Chaowei Yang, Baoxuan Jin, Manzhu Yu, Kai Liu, Min Sun, and Matthew Zhan. Enabling big geoscience data analytics with a cloud-based, mapreduce-enabled and service-oriented workflow framework. *PloS one*, 10(3): e0116781, 2015.

[8] Keith Wiley, Andrew Connolly, Jeff Gardner, S Krughoff, Magdalena Balazinska, Bill Howe, Y Kwon, and Yingyi Bu. Astronomy in the cloud: using mapreduce for image co-addition. *Publications of the Astronomical Society of the Pacific*, 123 (901):366, 2011.

[9] Alexander Szalay and Jim Gray. 2020 computing: Science in an exponential world. *Nature*, 440(7083):413–414, 2006.

[10] Motohiko Matsuda, Naoya Maruyama, and Shin'ichiro Takizawa. K mapreduce: A scalable tool for data-processing and search/ensemble applications on large-scale supercomputers. In *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[11] Xiaoyi Lu, Fan Liang, Bing Wang, Li Zha, and Zhiwei Xu. Datampi: extending mpi to hadoop-like big data computing. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 829–838. IEEE, 2014.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[14] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[15] Amazon EC2. https://aws.amazon.com/ec2/.

[16] TSUBAME. http://tsubame.gsic.titech.ac.jp/en/top. *User's Guide*, 2016.

[17] Yuichiro Ajima, Shinji Sumimoto, and Toshiyuki Shimizu. Tofu: A 6d mesh/torus interconnect for exascale computers. *Computer*, 11(42):36–40, 2009.

[18] Garrick Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 8. ACM, 2006.

[19] Ke Wang, Xiaobing Zhou, Hao Chen, Michael Lang, and Ioan Raicu. Next generation job management systems for extreme-scale ensemble computing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 111–114. ACM, 2014.

[20] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[21] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[22] Albeaus Bayucan, Robert L Henderson, James Patton Jones, Casimir Lesiak, Bhroam Mann, Bill Nitzberg, Tom Proett, and Judith Utley. Portable batch system openpbs release 2.3 administrator guide, 2000.

[23] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[24] Sriram Krishnan, Mahidhar Tatineni, and Chaitanya Baru. myhadoop-hadoop-on-demand on traditional hpc resources. *San Diego Supercomputer Center Technical Report TR-2011-2, University of California, San Diego*, 2011.

[25] Manisha Gajbe, Kalyana Chadalavada, Gregory Bauer, and William Kramer. Benchmarking and performance studies of mapreduce/hadoop framework on blue waters supercomputer.

[26] Troy Baer, Paul Peltz, Junqi Yin, and Edmon Begoli. Integrating apache spark into pbs-based hpc environments. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, page 34. ACM, 2015.

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

[28] MPISpawn. https://www.open-mpi.org/doc/v1.8/man3/mpi comm spawn.3.php, 2015.

[29] Jiahua He, Jagatheesan, et al. Dash: a recipe for a flash-based data intensive supercomputer. In *Proc. of the 2010 ACM/IEEE SC'10*, 2010.

[30] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004.

[31] Microsoft Azure. https://azure.microsoft.com/en-us/.

[32] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[33] Memcached. https://memcached.org. *A caching system*, 2017.

[34] Paris Carbone, Stephan Ewen, Haridi, et al. Apache flink: Stream and batch processing in a single engine. *Data Engineering*, 2015.

[35] Tao Gao, Yanfei Guo, Boyu Zhang, Pietro Cicotti, Yutong Lu, Pavan Balaji, and Michela Taufer. Mimir: Memory-efficient and scalable mapreduce for large supercomputing systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 1098–1108. IEEE, 2017.

[36] Thanh-Chung Dao and Shigeru Chiba. Hpc-reuse: Efficient process creation for running mpi and hadoop mapreduce on supercomputers. In *16th IEEE/ACM CC-Grid*, 2016.

[37] Tony Hey, Stewart Tansley, Kristin M Tolle, et al. *The fourth paradigm: data-intensive scientific discovery*, volume 1. Microsoft research Redmond, WA, 2009.

[38] Avantika Monnappa. How facebook is using big data - the good, the bad, and the ugly. 2015.

[39] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[40] Yahoo Hadoop cluster. https://wiki.apache.org/hadoop/poweredby#y, 2017.

[41] MapReduce Tutorial. https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapreducetutorial.html#source_code.

[42] Graph500: large-scale benchmarks. http://graph500.org.

[43] Matei Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the 9th USENIX NSDI*, 2012.

[44] Steven J Plimpton and Karen D Devine. Mapreduce in mpi for large-scale graph algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[45] RIKEN AICS. http://pf-aics-riken.github.io/kmr-manual, 2016.

[46] The history of supercomputers. https://www.extremetech.com/extreme/125271-the-history-of-supercomputers.

[47] TOP 10 Sites for June 2017. https://www.top500.org/lists/2017/06/.

[48] Top500 List. https://www.top500.org/list/2016. *Top500 List*, 2016.

[49] Japan's "K Computer" is Now the Fastest Supercomputer in the World. https://www.forbes.com/sites/alexknapp/2011/06/20/japans-k-computer-is-now-the-fastest-supercomputer-in-the-world.

[50] Titan Cray XK7. https://www.olcf.ornl.gov/computing-resources/titan-cray-xk7/. *User's Guide*, 2016.

[51] James Plafke. Titan supercomputer will have world's fastest storage at 1.4tb/s. 2013.

[52] Dong Chen, Noel Eisley, Philip Heidelberger, Robert Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David Satterfield, Burkhard Steinmacher-Burow, and Jeffrey Parker. The ibm blue gene/q interconnection fabric. *IEEE Micro*, 32(1):32–43, 2012.

[53] Amazon web services. https://aws.amazon.com/hpc/. *High Performance Computing*, 2017.

[54] Microsoft. https://docs.microsoft.com/en-us/azure/batch/big-compute-resources. *Microsoft HPC*, 2017.

[55] TSUBAME 2.5. http://tsubame.gsic.titech.ac.jp/en/paid-services, 2017.

[56] Stephen L. Smith. 32nm westmere family of processors. 2009.

[57] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types, 2017.

[58] Amazon EC2 Dedicated Hosts Pricing. https://aws.amazon.com/ec2/dedicated-hosts/pricing/, 2017.

[59] Ivy Bridge Sandy Bridge Intel five generation IPC test: Broadwell, Haswell and Nehalem. https://us.hardware.info/reviews/6215/2/intel-five-generation-ipc-test-broadwell-haswell-ivy-bridge-sandy-bridge-and-nehalem-results, 2015.

[60] Cray Cluster Supercomputers for Hadoop. http://www.cray.com/assets/pdf/products/cs/cs300hadoopbrochure.pdf.

[61] ASC Purple. https://computing.llnl.gov/tutorials/purple/.

[62] Amazon S3. https://aws.amazon.com/s3/, 2017.

[63] Nicholas Chaimov, Allen Malony, Shane Canon, Costin Iancu, Khaled Z Ibrahim, and Jay Srinivasan. Scaling spark on hpc systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 97–110. ACM, 2016.

[64] Ruben Tous, Anastasios Gounaris, Carlos Tripiana, Jordi Torres, Sergi Girona, Eduard Ayguadé, Jesús Labarta, Yolanda Becerra, David Carrera, and Mateo Valero. Spark deployment and performance evaluation on the marenostrum supercomputer. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 299–306. IEEE, 2015.

[65] Md Wasi-ur Rahman, Xiaoyi Lu, Nusrat Sharmin Islam, Raghunath Rajachandrasekar, and Dhabaleswar K Panda. High-performance design of yarn mapreduce on modern hpc clusters with lustre and rdma. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 291–300. IEEE, 2015.

[66] Scott Michael, Abhinav Thota, and Robert Henschel. Hpchadoop: A framework to run hadoop on cray x-series supercomputers. *Cray USer Group (CUG)*, 2014.

[67] Mahidhar Tatineni, Jerry Greenberg, Richard Wagner, Eva Hocks, and Christopher Irving. Hadoop deployment and performance on gordon data intensive supercomputer. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*, page 45. ACM, 2013.

[68] Yanfei Guo, Wesley Bland, Pavan Balaji, and Xiaobo Zhou. Fault tolerant mapreduce-mpi for hpc clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 34. ACM, 2015.

[69] Xiaoyi Lu, Md Wasi Ur Rahman, Nusrat Islam, Dipti Shankar, and Dhabaleswar K Panda. Accelerating spark with rdma for big data processing: Early experiences. In *High-performance interconnects (HOTI), 2014 IEEE 22nd annual symposium on*, pages 9–16. IEEE, 2014.

[70] Luna Xu, Min Li, and Ali R Butt. Gerbil: Mpi+ yarn. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 627–636. IEEE, 2015.

[71] Ralph Butler William. A scalable process-management environment for parallel programs. In *In Euro PVM/MPI*. Citeseer, 2000.

[72] Robert Stewart and Jeremy Singer. Comparing fork/join and mapreduce. Technical report, Citeseer, 2012.

[73] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3r: increased performance for in-memory hadoop jobs. *Proc. of the VLDB Endowment*, 2012.

[74] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. Haloop: efficient iterative data processing on large clusters. *Proc. of the VLDB Endowment*, 2010.

[75] The Difference Between a tmpfs and ramfs RAM Disk. https://www.jamescoyle.net/knowledge/951-the-difference-between-a-tmpfs-and-ramfs-ram-disk.

[76] Mitsuo Yokokawa, Fumiyoshi Shoji, Atsuya Uno, Motoyoshi Kurokawa, and Tadashi Watanabe. The k computer: Japanese next-generation supercomputer development project. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, pages 371–372. IEEE Press, 2011.

[77] mpiJava. http://www.hpjava.org/mpijava.html, 2007.

[78] Jithin Jose, Hari Subramoni, Luo, et al. Memcached design on high performance rdma capable interconnects. In *2011 Int'l Conf. on Parallel Processing*, 2011.

[79] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *nsdi*, volume 13, pages 385–398, 2013.

[80] UserGroupInformation Class. https://hadoop.apache.org/docs/r2.4.1/.

[81] Shigeru Chiba. Javassist—a reflection-based programming wizard for java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, volume 174, 1998.

[82] Yandong Wang, Cong Xu, Xiaobing Li, and Weikuan Yu. Jvm-bypass for efficient hadoop shuffling. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 569–578. IEEE, 2013.

[83] Thanh-Chung Dao, Roman Bednarik, and Hana Vrzakova. Heatmap rendering from large-scale distributed datasets using cloud computing. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 215–218. ACM, 2014.

[84] Roman Bednarik, Hana Vrzakova, and Michal Hradis. What do you want to do next: a novel approach for intent prediction in gaze-based interaction. In *Proceedings of the symposium on eye tracking research and applications*, pages 83–90. ACM, 2012.

[85] Oscar Vega-Gisbert, Jose E Roman, and Jeffrey M Squyres. Design and implementation of java bindings in open mpi. 2014.

[86] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and TN Vijaykumar. Puma: Purdue mapreduce benchmarks suite. 2012.

[87] Ke Wang, Xiaobing Zhou, Kan Qiao, Michael Lang, Benjamin McClelland, and Ioan Raicu. Towards scalable distributed workload manager with monitoring-based weakly consistent resource stealing. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 219–222, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3550-8. doi: 10.1145/2749246.2749249. URL http://doi.acm.org/10.1145/2749246.2749249.

[88] David Jackson, Quinn Snell, and Mark Clement. Core algorithms of the maui scheduler. In *Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer, 2001.

[89] Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M Parikh, and James M Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *ACM SIGPlAN notices*, volume 33, pages 280–290. ACM, 1998.

[90] Open MPI: Version 1.7.5. https://www.open-mpi.org/software/ompi/v1.7, 2007.

[91] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. Hydradb: a resilient rdma-driven key-value middleware for in-memory cluster computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 22. ACM, 2015.

[92] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[93] Hadoop 2.2 Release Notes. http://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-common/releasenotes.html.

[94] Yahoo. Yahoo: We run the whole company on hadoop. *Yahoo: We Run the Whole Company on Hadoop*, 2014.

[95] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification, 2011.

[96] Elastic X10. http://x10-lang.org/documentation/practical-x10-programming/elastic-x10.html, 2017.

[97] Dynamic resource allocation in Spark. http://spark.apache.org/docs/latest/configuration.html#dynamic-allocation, 2017.