# Mammoth: Gearing Hadoop Towards Memory-Intensive MapReduce Applications

Xuanhua Shi[1], Ming Chen[1], Ligang He[2], Xu Xie[1], Lu Lu[1], Hai Jin[1], Yong Chen[3], and Song Wu[1]

[1]Service Computing Technology and System Lab/ Cluster and Grid Computing Lab, School of Computer, Huazhong University of Science and Technology
[2]Department of Computer Science, University of Warwick
[3]Texas Tech University
[1]{*xhshi,mchen,xxie,hjin,llu,wusong*}*@hust.edu.cn, liganghe@dcs.warwick.ac.uk, yong.chen@ttu.edu*

**Abstract**—The MapReduce platform has been widely used for large-scale data processing and analysis recently. It works well if the hardware of a cluster is well configured. However, our survey has indicated that common hardware configurations in small- and medium-size enterprises may not be suitable for such tasks. This situation is more challenging for memory-constrained systems, in which the memory is a bottleneck resource compared with the CPU power and thus does not meet the needs of large-scale data processing. The traditional high performance computing (HPC) system is an example of the memory-constrained system according to our survey. In this paper, we have developed Mammoth, a new MapReduce system, which aims to improve MapReduce performance using global memory management. In Mammoth, we design a novel rule-based heuristic to prioritize memory allocation and revocation among execution units (mapper, shuffler, reducer, etc.), to maximize the holistic benefits of the Map/Reduce job when scheduling each memory unit. We have also developed a multi-threaded execution engine, which is based on Hadoop but runs in a single JVM on a node. In the execution engine, we have implemented the algorithm of memory scheduling to realize global memory management, based on which we further developed the techniques such as sequential disk accessing, multi-cache and shuffling from memory, and solved the problem of full garbage collection in the JVM. We have conducted extensive experiments to compare Mammoth against the native Hadoop platform. The results show that the Mammoth system can reduce the job execution time by more than 40% in typical cases, without requiring any modifications of the Hadoop programs. When a system is short of memory, Mammoth can improve the performance by up to 5.19 times, as observed for I/O intensive applications, such as PageRank. We also compared Mammoth with Spark. Although Spark can achieve better performance than Mammoth for interactive and iterative applications when the memory is sufficient, our experimental results show that for batch processing applications, Mammoth can adapt better to various memory environments and outperform Spark when the memory is insufficient, and can obtain similar performance as Spark when the memory is sufficient. Given the growing importance of supporting large-scale data processing and analysis and the proven success of the MapReduce platform, the Mammoth system can have a promising potential and impact.

**Index Terms**—MapReduce, data processing, HPC

## 1 Introduction

In recent years, large-scale data mining and machine learning have become increasingly popular. Hadoop [1] is a well-known processing platform for large data sets and widely used in many domains (e.g. Google [11], Yahoo! [5], Facebook [9], LinkedIn [3]). Most of such companies have dedicated Hadoop clusters, which are equipped with plenty of disks and memory to improve the I/O throughput. Other institutes developed special systems for processing big data applications, such as TritonSort [21]. There are increasing data processing requirements from scientific domains as well. However, many of the scientific research institutes or universities do not have dedicated Hadoop clusters, while most of them have their own high performance computing (HPC) facilities [14], [20], [12], [7], [17]. The latest survey of worldwide HPC cites conducted by IDC (International Data Corporation) also indicates that 67% of HPC systems are now performing the big Data analysis [4].

We regard the HPC system as a type of memory-constrained system, in which the memory is a bottleneck resource compared with the CPU power. This is because the nature of HPC systems requires the equipment of powerful CPUs for computation, while the memory and disks are often limited, compared with CPU power. We have surveyed the Top 10 supercomputers on the Top500 lists from June 2010 to November 2012 that published their memory configurations (these supercomputers are Titan, Sequoia, K computer, JUQUEEN, Stampede, Tianhe-1A, Jaguar Curie thin nodes, TSUBAME 2.0, Roadrunner, Pleiades, Tianhe-1, BlueGene/L, and Red Sky). Most of these HPC servers are equipped with less than 2GB memory for one CPU core, and some of them with less than 1GB.

In Hadoop, the tasks are scheduled according to the number of CPU cores, without considering other resources. This scheduling decision leads to long waiting time of CPUs, which influences the total execution time due to the performance gap between the CPU and the I/O system. In Hadoop, every task is loaded with a JVM. Every task has an independent memory allocator. A Hadoop task contains several phases that involve memory allocation: task sort buffer, file reading and writing, and application-specific memory usage. Most memory allocation is pre-set with parameters in the job configuration without considering the real tasks' demand. Besides, it does not have a memory scheduler for all the tasks in a TaskTracker. These designs will lead to the problem of buffer concurrency among Hadoop tasks. Another issue is that disk operations in Hadoop are not scheduled cooperatively. Every task reads and writes data independently according to its demand

without coordination, which potentially leads to heavy disk seeks. For instance, in the merge and shuffle phases, the overhead of uncoordinated disk seeks and the contention in accesses are so big that the I/O wait occupies up to 50% of the total time as observed, which significantly degrades the overall system performance.

To tackle the problem, we have developed a new MapReduce data processing system, named Mammoth [1], for memory-constrained systems (e.g., HPC systems). Mammoth makes the best effort to utilize the memory according to the existing hardware configuration. In Mammoth, each Map/Reduce task on one node is executed as a thread, and all the task threads can share the memory at the runtime. Thread-based tasks make sequential disk access possible. Mammoth utilizes the multi-buffer technique to balance data production from CPU and data consumption of disk I/Os, which implements the non-blocking I/O. Mammoth realizes the in-memory or hybrid merge-sort instead of simply external sorting from disks, in order to minimize the size of spilled intermediate data on disks. Mammoth also caches the final merged files output by Map tasks in memory to avoid re-reading them from disks before transferring them to remote reduce tasks. In order to coordinate various activities, these data buffers are managed by a global memory manager and an I/O scheduler on each node. The contributions of this paper are as follows:

1) We designed a novel rule-based heuristic to prioritize memory allocation and revocation among execution units (mapper, shuffler, reducer etc), to maximize the holistic benefits of the Map/Reduce job when scheduling each memory unit.
2) We designed and implemented a multi-threaded execution engine. The engine is based on Hadoop, but it runs in a single JVM on each node. In this execution engine, we developed the algorithm for memory scheduling to realize global memory management, based on which we further implemented the techniques of disk accesses serialization, multi-cache, shuffling from memory, and solved the problem of full Garbage Collection (GC) in the JVM.
3) We conducted the extensive experiments to compare Mammoth against Hadoop and Spark. The results show that Mammoth improved the performance dramatically in terms of the job execution time on the memory-constrained clusters.

The rest of this paper is organized as follows. Section 2 demonstrates the performance problem of running Hadoop in the memory-constrained situation and discuss the motivations of this research. We will then present the overall design of the system in Section 3. In Section 4 and 5, we will present the techniques for global memory management and I/O optimization, respectively. The results of performance evaluation will be presented and analyzed in Section 6. Related work is discussed in Section 7. Section 8 concludes this paper and discuss the future work.

## 2 Motivation

This section demonstrate the main performance problems when running Hadoop in the memory-constrained situations, which motivates us to develop the Mammoth system. The existing MapReduce systems usually schedule the concurrent tasks to the compute nodes according to the number of task slots on each node. This approach relies on the system administrator to set the number of task slots according to the actual hardware resources. Most existing studies about Hadoop in the literature [29], [28] choose

1. We have made Mammoth open source at the Github , and also added the patch at Apache Software Foundation. The web links are https://github.com/mammothcm/mammoth and https://issues.apache.org/jira/browse/MAPREDUCE-5605, respectively.

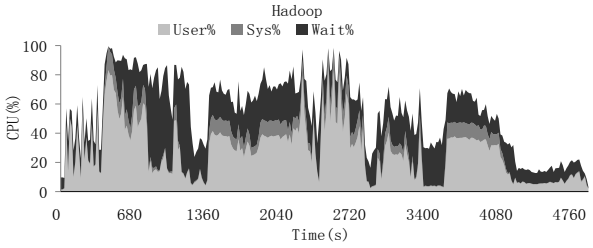



Fig. 1. CPU Utilization in Hadoop.

the number of slots according to the nodes' CPU cores, and do not take into consideration other resources such as memory or disks. This is why the actual per-node efficiency in a Hadoop cluster is often dramatically lower than the theoretical peak value [21].

Memory is an important resource to bridge the gap between CPUs and I/O devices. However, we observed the following main problems for the memory usage in Hadoop. First, the memory management in Hadoop is rather coarse-grained. The memory size available for a Map or Reduce task is set in a static configuration file, and is fixed at runtime. Therefore, even after the Map tasks are completed, the buffers used by those tasks cannot be used by the Reduce tasks. Second, the memory mainly used by the Map tasks is the intermediate buffer. When the buffer cannot accommodate the intermediate data, the entire intermediate data will be spilled to the local disk. For example, if the intermediate data for a Map task are 100MB, but the buffer size is only 80MB, then the intermediate data will be spilled to the disk as an 80MB file and a 20MB file. As the result, the final merge-sort phase in Hadoop will be largely affected by reading an 80MB file from the disk. Finally, although different Map tasks may produce the intermediate data with different sizes (e.g., the "grep" application), Hadoop does not provide the mechanism for the concurrently running Map tasks to coordinate their memory usage with each other.

The I/O operations may also cause very inefficient usage of resources. Firstly, a merge-sort algorithm is widely used in Hadoop [19]. In this algorithm, the operations of CPU computing (sort) and disk spilling are mashed together. There are a multitude of I/O waits during this procedure. Secondly, parallel I/O is performed in Hadoop whenever possible. However, it has been shown in previous literature [21] that parallel I/O may cause vast disk seeks. Especially, the situation may become even worse when there is only one disk on a node. Finally, as mentioned above, the Reduce tasks will have to pull the output files of the Map tasks, which should be performed as early as possible in order to improve the read performance. However, the file buffer in the file system has the lowest priority of using the memory in Linux, which will cause the so called long-tail effect revealed in [30].

In order to demonstrate the performance problems in Hadoop, we have conducted the following benchmark experiments. The experiment platform includes a cluster of 17 nodes. Each node is equipped with two 8-core 2.6GHz Intel(R) Xeon(R) E5-2670 CPUs, 32GB memory and a 300GB 10,000RPM SAS disk, and running RedHat Enterprise Linux 5 (Linux 2.6.18-308.4.1.el5). The version of Hadoop used in the experiment is 1.0.1, running on a 320GB dataset created by the built-in randomtextwriter. The application we used is WordCount. The job configurations are set to have the block size of 512MB, 8 Map-slots and 8 Reduce-slots. We learned from the log files of the pre-executed jobs that the ratio of intermediate data to input data is about 1.75. We adjusted the size of the map sort buffer so that the total map output can be exactly fitted into the buffer, thereby avoiding the costly spill-merge process. We used the "iostat" command in Linux to record

the CPU-usage and I/O-wait information and plotted it in Figure 1, in which the x-axis represents the job runtime while the y-axis is the ratio of CPU to I/O wait.

As shown in Figure 1, the I/O wait time still accounts for a large proportion of the total CPU cycles, which means the I/O operations negatively impact the CPU performance. In the experiment, the number of map task waves is 5. From the Hadoop job log files, we found that map tasks in different waves needed increasingly more time to process the same sized data along with the job running. The reason is that the size of the data that the Map function produces is larger than what the I/O devices can consume at the same time. At the beginning, the generated intermediate data can fit into the memory buffer of the native file system in Linux. When the buffer cannot hold the data anymore, CPUs have to be idle and wait for the disks to finish all pending I/O operations. The situation would become even worse when the servers are equipped with multi-core CPUs but a single disk.

However, if we only consider what was discussed above, it appears that the memory usage in Hadoop has actually been adjusted to optimism: every Map task performed only one spill at the end. Then why was there so much I/O wait time? From Section 2.1, we can know that although each Map task can hold all the intermediate data in memory, the Reduce tasks must pull the Map tasks' results from the disk, that is, all the intermediate data will be read and written once. On the other hand, the Map tasks will use up at least $8 \times 512MB \times 1.75 = 7GB$ memory. After the Map tasks have finished, the Reduce tasks cannot realize there is now newly freed memory, because the reduce tasks can only use the amount of memory designated in the configuration file before the startup of a MapReduce job. This means that at least 7GB of memory is wasted. From the discussions in section 2.1, we know that this will in turn cause more I/O operations. Therefore, it can be concluded that Hadoop will cause serious I/O bottleneck on a memory-constrained platform.

Although we have adjusted the configurations to optimum in the above benchmarking experiments(e.g., the size of the sort buffer is adjusted so that all Map output can be accommodated), the CPUs still spent a considerable amount of time waiting for the completions of I/O operations. This is because the native Hadoop cannot utilize the memory effectively, and as the consequence has to perform a large number of unnecessary I/Os. Therefore, a more efficient strategy to utilize the memory is desired to improve the overall system performance. Motivated by these observations, we have designed and implemented a memory-centric MapReduce system, called Mammoth. The system adopts the global memory management strategy and implements a number of I/O optimization techniques to achieve the significant performance improvement on the memory-constrained platform. In the remainder of this paper, a overall system architecture of Mammoth is introduced in Section 3. Then Section 3.4 and 4.3 present the memory management strategies and the I/O optimization techniques, respectively, implemented into the Mammoth architecture.

## 3 System Design

This section presents the system architecture of Mammoth and how to reform the MapReduce execution model. To improve the memory usage, a thread-based execution engine is implemented in Mammoth.

### 3.1 System Architecture

Figure 2 shows the overall architecture of the Execution Engine in Mammoth. The execution engine runs inside a single JVM

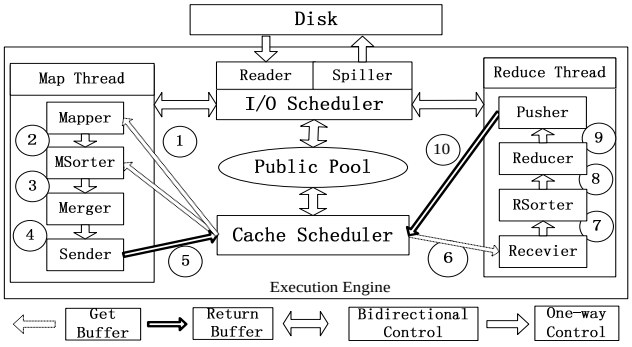



Fig. 2. Architecture and Workflow of Mammoth.

(Java Virtual Machine). In Mammoth, all Map/Reduce tasks in a physical node run inside the execution engine, and therefore in a single JVM, which is one of the key architectural differences between Mammoth and Hadoop. Mammoth retains the upper cluster management model in Hadoop, i.e. the master JobTracker manages the slave TaskTrackers through heartbeats. When the TaskTracker obtains a new Map/Reduce task, the task is assigned to the execution engine through RPC (Remote Procedure Call) and the execution engine informs the TaskTracker of the task's real-time progress through RPC too. In Hadoop, a Map/Reduce task runs in a separate JVM. In the execution engine, the Map/Reduce tasks interact with the Cache Scheduler, which is responsible for processing the memory allocation and revocation requests from the tasks (i.e., the control flow for acquiring and returning the buffer) and more importantly, coordinating the memory demands among the memory management components through the Public Pool. The I/O Scheduler is responsible for reading the data from the disk (through the Reader) and spilling the data from the Public Pool to the disk (through the Spiller). Both a Map task and a Reduce task contain 4 functional components which will be discussed in detail in next subsection.

Mammoth implements the global memory management through the Public Pool. The Public Pool contains a few fundamental data structures which enable the effective memory management. The Public Pool consists of three types of memory areas: Element Pool, Cache Pool and Multi-buffer, each of which has a different data structure. The Element Pool consists of the Elements. An Element is a data structure consisting of *index*, *offset*, $< key, value >$ contents, which is similar as the data structure in Hadoop. The Elements store the unsorted intermediate $< key, value >$ pairs collected by the user-defined Map function (i.e., the Mapper in a Map Task in Fig. 3). The default size of an Element is 32K. The Cache Pool consists of the *cache units*. A *cache unit* is a byte array with the default size of 2M. Different from the Element Pool, the Cache Pool is used to hold the sorted intermediate data generated by other functional components in a Map/Reduce Task in Fig. 3. Multi-buffer is used by the I/O Scheduler to buffer the data to be read from and spilled to the disk. This is also why Figure 2 depicts the interaction between the I/O scheduler and the Public Pool. The data structure of the Multi-buffer will be discussed in detail when we discuss the I/O optimization in Section 5.

Similar as in Hadoop, the execution of a Map/Reduce job in Mammoth also has three phases: Map, Shuffle, and Reduce. However, in order to realize the global memory management through the above data structures and achieve better performance, the detailed execution flow in each phase is different, which is presented in the next subsection.

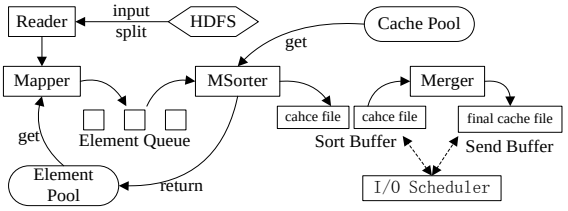



Fig. 3. The Map phase in Mammoth.

### 3.2 The Map phase

Figure 3 shows the details of the Map phase in Mammoth. The **Reader** component reads the input split from HDFS into its Multi-buffer. **Mapper** requests an empty Element (through the Cache Scheduler) from the Element Pool, and reads the data from the Multi-buffer and collects the translated $< key, value >$ pairs into the new Element. When the Element is full, the $< key, value >$ pairs in the Element is sorted. The Element holding the sorted data is then added to the Element Queue. The Element Queue grows as more elements are added. When the size of the Element Queue reaches a threshold (10M by default), **MSorter** requests a corresponding number of *cache units* from the Cache Pool, and then merge-sorts the Element Queue and store the sorted data into those *cache units*. This batch of cache units is called a *cache file*. After the Element Queue is stored into the cache file, the corresponding Elements are recycled to the Element Pool. A Sort Buffer is designed to hold these *cache files*. The maximum size of the Sort Buffer is determined by the Cache Scheduler. After **Mapper** and **MSorter** have finished their work (i.e., the input data of the Map task has all been converted to the form of the *cache files*), **Merger** merges all intermediate *cache files* in the Sort Buffer to a *final cache file* in the Send Buffer. The Send Buffer holds the data to be consumed in the Shuffle phase. When the Sort Buffer or the Send Buffer cannot hold the intermediate data, it will have to be spilled to the local disk and be read into the buffer later. The I/O requests are managed by the I/O Scheduler.

### 3.3 The Shuffle phase

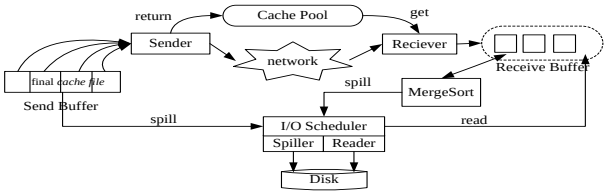



Fig. 4. The Shuffle phase in Mammoth.

In Hadoop, the reduce task starting time is decided by the parameter "mapreduce.reduce.slowstart.completed.maps", which is set to be 0.05 by default. It effectively means that the Reduce tasks will be launched shortly after the Map tasks begin. In most cases, it brings more benefits by launching reducers early rather than later. This is because i) it can overlap the shuffle network I/O with the map computation, and ii) if Reduce tasks start early, then it is more likely that they can pull the newly written data from the page cache, reducing the I/O data at the Map side. This is why in the default setting of Hadoop the Reducers are launched shortly after the Mappers begin. From the perspective of the Mammoth design, it is also more beneficial by launching the reducers shortly after the mappers start. This is because Mammoth manages the data buffer in the task execution engine, and tries to send the buffered data to the Reduce tasks as soon as possible.

There is a major difference in the Shuffle phase between Hadoop and Mammoth. In Hadoop, the data generated by the Map tasks is written to the local disk, and the Reduce tasks pull the data from the local disk in the Map side to the Reduce side, which we call *passive pulling*. In Mammoth, the intermediate data produced by the Map tasks (i.e., the *final cache file* in the send buffer) are *actively pushed* by the Map task to the Reduce tasks. When the memory is insufficient, the send buffer may only be able to hold part of the intermediate data and therefore some intermediate data have to be written to the disks. The sender on the map side knows the status of the buffer and the intermediate data in real time, and therefore it can push the intermediate data to the reduce side before they are written to disks. If pulling is used, the reduce side has no idea about the status of the intermediate data (in disks or in buffer) and may not be able to pull the data before they are written to disks (therefore cause more I/O operations).

The design details of the Shuffle phase in Mammoth are illustrated in Figure 4. As shown in Figure 4, there is a Sender and a Receiver in the Execution Engine on a node. Sender sends the *file cache file* in the send buffer to the Reduce tasks through the network. More specifically, The *final cache file* contains the data that need to be sent to different Reduce tasks. Sender consists of a group of send threads, each of which is responsible for sending the portion of data destined to a particular Reduce task. After **Sender** finishes sending the *final cache file*, its *cache units* are recycled to the Cache Pool.

At the Receiver side, Receiver also consists of a group of receive threads, called *subreceiver*. When **Receiver** receives a data-transfer request originated from a Send thread, it distributes the request to the corresponding Reduce task. Then, the Reduce task distributes the request to one of its *subreceiver*. The *subreceiver* requests a *cache file* from the Cache Pool to store the received data. The newly allocated *cache file* becomes a part of the Receive Buffer. The process described above implements the active pushing approach for shuffling, which differentiates the passive pulling approach in Hadoop. The data in the Receive Buffer will be processed next by the Reduce phase.

The maximum size allowed for the Receive Buffer (and for the Send Buffer) is decided by the Cache Scheduler at runtime, which will be discussed in Section 4 in detail. The Receiver can predict whether the Receive Buffer is not sufficient to hold all data being sent by the Map tasks. If it is not sufficient, the corresponding amount of existing data in the Receive Buffer will be merge-sorted and spilled to the local disk through the Spiller.

As shown in Fig. 5, the *final cache file* in the Send Buffer is also written to the disk through Spiller. But the writing is for the purpose of fault tolerance, and it occurs concurrently with Sender/Receiver sending/Receiving data. Therefore, the writing does not stall the progress of the job. After **Spiller** finishes writing data, the Map task is deemed to be completed successfully. Note that Mammoth follows the same fault tolerance design as Hadoop. Firstly, the map tasks' output data are written to disks for fault tolerance. Secondly, when a machine is down, Mammoth will re-execute the tasks scheduled to run on the machine.

### 3.4 The Reduce phase

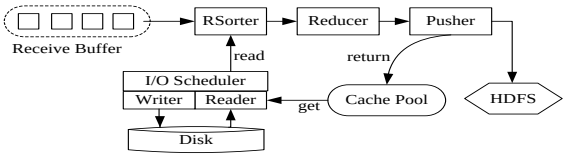



Fig. 5. Reduce phase in Mammoth.

Figure 5 shows the details of the Reduce phase. In typical cases, all data to be reduced are in memory (i.e., the Receive Buffer). The Reduce phase starts with RSorter using the merge-sort to sort the data in the Receive Buffer. However, for some data-intensive applications or in case that the cluster nodes have small memory sizes, the size of the total intermediate data may be more than that of the Cache Pool. Under this circumstance, some intermediate data must be spilled to the disk. Therefore, in Figure 6, the sources of the data to be sorted by RSorter may be hybrid, i.e., from both memory and disk.

After **RSorter** has sorted the intermediate data, **Reducer** begins to process the aggregated $< key, values >$ pairs using the Reduce Function. **Pusher** pushes the final output to HDFS. The cache units holding the final output are then returned to the Cache Pool for recycling. Both **Reader** and **Pusher** works in a non-blocking way to overlap CPU and I/O. **Reader**, **RSorter-Reducer** and **Pusher** forms three stages of a pipeline for processing the Reduce tasks. Note that we regard RSorter and Reducer together as one stage. This is because RSorter merge-sorts the sorted intermediate files so that the <key, value> pairs with the same key can be aggregated during the sort procedure and can feed the Reducer simultaneously. Therefore, our view is that RSorter and Reducer are synchronized and should be treated as one pipeline stage together. The granularity of the pipeline is one *cache unit*, that is, after one *cache unit* is processed by a stage in the pipeline, it is handed over to the next stage for processing. After all these procedures are completed, the Reduce task is completed.

## 4 MEMORY MANAGEMENT

In this section, we will first summarize the usage of the memory in Mammoth, describing various rules about sharing the global memory. And then we will present the two key components of the memory management: Cache Scheduler and Cache Pool. Finally we will describe the scheduling algorithms that are designed to achieve the efficient sharing of the global memory.

### 4.1 Memory Usage Analyses in Mammoth

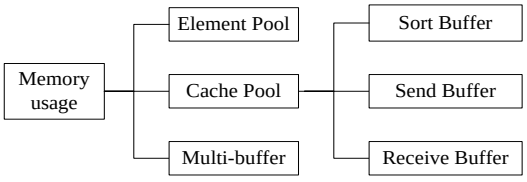



Fig. 6. Memory usage types

In Mammoth, in order to maximum the memory utilization, we design an execution engine, JVM, to manage most part of a node's memory in the application level. As discussed in section 3, there are three types of memory usage: Element Pool, Cache Pool and Multi-buffer (as illustrated in Figure 6). The Element Pool is used for the Map Phase, presented in section 3.2. The Muliti-buffer is used in the I/O Scheduler, to be described in Section 5.2. The Cache Pool consists of three types of buffer: Map tasks' Sort buffer, Map tasks' Send buffer and Reduce tasks' Receive Buffer.

As mentioned in Section 3.1, an Element is a data structure which contains the index, the offset arrays, and the content in the $< key, value >$ pairs. The additional information is mainly used by quick-sort, which can reduce the overhead of CPU because the swapping of $< key, value >$ pairs can be replaced by the simply swapping of their indices. However, since memory is constraint in many cases, So we make size of Element Pool relatively small and merge-sort the quick-sorted contents in Elements in the following steps. We have optimized in-memory merge-sort which will be demonstrated in section 5.3. For sharing more fairly between the Map tasks and recycling faster, the size of an Element will be relatively small too, which is 32KB by default.

The size of the Element Pool can be calculated using Eq. 1.

$$ElementPoolSize = SpillMemSize \times MapSlots \quad (1)$$

In Equation (1), *ElementPoolSize* represents the total size of the Elements in the Element Pool, *SpillMemSize* represents the threshold memory size that a map task could use in the Element Pool, which is 10M by default, *MapSlots* represents the number of the Map slots in a TaskTracker node. Note that the size of the Element Pool is fixed at runtime.

The multi-buffer is mainly used by **Reader** and **Spiller** to overlap the I/O operations. However, it is likely that the CPU computation or the disk I/O operations become the performance bottleneck when both CPU computation and I/O operations are being performed. For example, if the CPU produces the data faster than the disk I/O, the disk I/O becomes the performance bottleneck and the data will accumulate in the buffer queue. Otherwise, the CPU computation is the bottleneck. This is the reason why every queue in **Spiller** or **Reader** has a threshold memory size of 10MB by default. Once the size of the accumulated data exceeds the threshold of the queue, the CPU will be blocked until there is the spare space in the queue. Once there are the data added to the queue, it will be written to the disk soon. The situation for **Reader** is the same, and therefore its discussion is omitted.

In the following, we will give a typical and concrete example of the memory usage. Suppose each node has 16 CPU cores with 1GB memory per core. The execution engine JVM is configured with the heap of size 15 GB, 2/3 of which is used for the intermediate buffer, i.e. 10GB. The number of the Map slots for each node is 12, while the number of the Reduce slots is 4. The Block size in HDFS is 512MB.

As for the Element Pool, since the parameters, *SpillMemSize* and *MapSlots*, are configured in the configuration files and fixed at runtime, *ElementPoolSize* is relatively small and fixed, e.g. $10MB \times 12 = 120MB$. After analysing the entire workflow in Mammoth, we find that a task will not issue more than two I/O operations once. Thus, the memory used for multi-buffer will not exceed $16 \times 2 \times 10MB = 320MB$, and is usually much lower than this maximum size. The rest of the memory buffer is used by the Cache Pool, which is $10GB - 120MB - 320MB = 9560MB$. Therefore, the main task of memory management in such a system is the management of the Cache Pool.

For many data-incentive applications, e.g. Sort, PageRank, etc., the size of the intermediate data is the same as that of the input data. Therefore, since there are 12 Map tasks in one wave and the block size in HDFS is 512MB, the size of the intermediate data generated in one wave will be $12 \times 512MB = 6GB$. Usually, the Shuffle phase can run simultaneously with the Map process and is the bottleneck for data-incentive applications. Consequently, the intermediate data generated in different waves of Map tasks will accumulate. The strategies we adopt to share the Cache Pool among the Sort Buffer, Send Buffer and Receive Buffer will be discussed in section 4.3.

### 4.2 Memory Management Structure

In Mammoth, the Cache Scheduler is responsible for the memory management, which mainly operates on the data structure, Cache Pool, which consists of a linked list of Cache Units, is the main data structure that the memory management operations have to operate on.

#### 4.2.1 Cache Scheduler

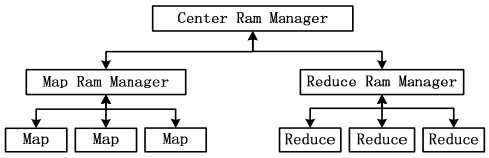



Fig. 7. Cache Scheduler Architecture.

In Mammoth, the component handling the global memory management is the Cache Scheduler, whose role has been illustrated in Figure 7. The tasks executing in the execution engine interact with the Cache Scheduler for memory allocation and release. The Cache Scheduler executes as a single model (i.e., a single instance) in the execution engine. A simple design is to design a single component that is responsible for all memory allocations and revocations. However, such a design will cause the Cache Scheduler to become a performance bottleneck at runtime. This is because the cache scheduler must interact with the map/reduce tasks at runtime to assign and recycle memory, and the interactions can be very frequent. Therefore, we designed a hierarchical structure for the Cache Scheduler, which is shown in Figure 7.

There are two layers in the Cache Scheduler. The top layer is called the Central Ram Manager (CRM), and the middle layer is called Ram Manager (RM). CRM uses a global memory management algorithm to dynamically adjust the memory proportions among RMs, based on the conditions of the global memory usage (The detailed algorithm outline will be described in the supplementary file). There are two types of RMs: Map Ram Manager (MRM) and Reduce Ram Manager (RRM). MRM is responsible for managing the memory used by the Map tasks, while RRM is for managing the memory used by the Reduce tasks. The tasks (Map tasks or Reduce tasks) request or free the memory directly from RMs (MRMs or RRM). When RMs allocate the memory to the tasks, it comes down to allocating the memory to the three types of buffers: Sort, Send and Receive. RMs will update CRM of the value of their parameters at runtime, which will be used by CRM to make the global memory management decisions. With this hierarchical cache scheduler, adjusting the quotas is conducted by the central ram, while interacting with tasks is performed by the map/reduce ram manager, which helps prevent the Cache Scheduler from becoming a performance bottleneck at runtime.

RMs allocate the memory to the tasks based on the memory status at runtime. However, the following two general rules are used by the Cache Scheduler for memory allocations.

a) the priority order of the three types of buffer is: Sort $>$ Send $>$ Receive (the reason for this priority setting will be explained in Section 4.3. This means that when CRM distributes the memory between MRM and RRM , MRM is deemed to have the higher priority than RRM (i.e., the Sort buffer and the Send buffer have the higher priority than the Receive buffer), and that when MRM allocates the memory to the Sort buffer and the Send buff, the Sort buffer has the higher priority than the Send buffer.

b) Memory revocation is conducted using the priority order that is opposite to that in memory allocation. This means that when the memory is insufficient, Mammoth may spill the data to the disk based on the order of Receive $>$ Send $>$ Sort. But different from the allocation, the memory can be recycled unit by unit.

#### 4.2.2 Cache Pool

Since there are frequent memory allocations and revocations in Mammoth, there will be the big pressure on Garbage Collection (GC) if we simply rely on JVM [18] for GC. Another problem is that if we rely on the file system's buffer to cache the intermediate data for shuffle. It is very likely that the shuffled data are still in memory while the intermediate data produced later is flushed to the disk, which will cause the long-tail effect. Our solution is to design an application-level mechanism to reclaim the unused cache space [26]. The Cache Pool is responsible for this purpose. Moreover, an data structure, called *cache file*, is designed for the Cache pool to control the management of the data cache.

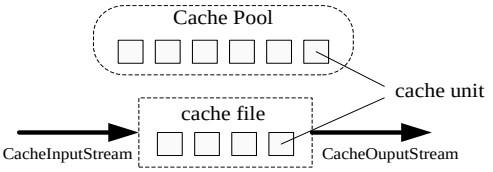



Fig. 8. Data Structure for Memory Management.

As shown in Figure 8, the Cache Pool is made of a list of *cache units*. When an operation requests a memory block, a *cache file*, composed of a queue of *cache units*, will be returned. There are the java classes, *CacheInputStream* and *CacheOuputStream*, associated with a *cache file*. Therefore, accessing a cache file is just like accessing a typical file in Java. In doing so, the way of accessing the memory is unified and consequently the file buffer can be managed in the application level. Memory allocation and revocation in the Cache pool is performed by inserting and deleting the Cache Units in the Cache pool. Since the Cache Units are organized as a linked list in Cache pool. The overhead of Memory allocation and revocation in Cache pool is rather low. The pressure for garbage collection is therefore greatly reduced. Further, Although the memory is allocated in the unit of a cache file, the memory can be recycled by cache units in order to accelerate the memory circulation, i.e., if any part of a cache file is not needed, the corresponding cache units can be freed.

### 4.3 Memory Management Strategies

This section discusses the memory management algorithms of the Cache Scheduler. In Mammoth, all tasks are executed as the threads in a execution engine to achieve the global memory management. The following issues need to be addressed in order to optimize the efficiency of memory usage.

1) Since multiple types of buffer may be used simultaneously in the system, the memory has to be allocated properly among these different types of buffer.

2) Memory allocation has to be adjusted dynamically. This is because the memory demands will be different in different execution stages of MapReduce jobs.

This section aims to tackle the above two issues.

From last section, we know that the memory needed by the Element Pool and the Multi-buffer is relatively small. So our attention is mainly focused on how to manage the Cache Pool.

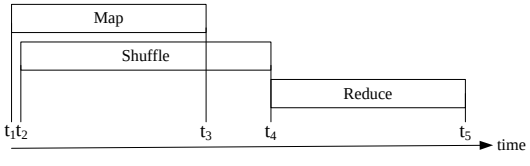



Fig. 9. Execution flow for MapReduce.

In order to allocate the memory properly among different types of buffer, i.e., Sort Buffer, Send Buffer and Receive Buffer, we first analyze their usage pattern over time. For a MapReduce Job, there are some important time points in its life cycle. They are illustrated in Figure 9. The Map phase begins at $t_1$. The Shuffle

phase begins shortly after at $t_2$. After Map ends at $t_3$, Shuffle is performed for another period of time and ends at $t_4$. Once Shuffle ends, the Reduce phase begins, and it finally ends at $t_5$. In different phases of the lifecycle, there are different memory usage patterns. The Sort Buffer is only used before $t_3$; The Send Buffer is used between $t_2$ and $t_4$; The utilization of the Receive Buffer occurs within the time period from $t_2$ to $t_5$. These features give us the chance to assign the three types of buffers with different priority at runtime.

Every intermediate <key, value> pair flows from Sort Buffer to Send Buffer and then from Send Buffer to Receive Buffer, which forms a buffer stream. If the upstream buffer does not have higher priority, the memory will gradually be accumulated towards the downstream buffer and eventually, the upstream buffer will end up having no memory to use. For example, if the Sort Buffer does not have higher priority than the Send Buffer, then when one wave of map tasks finish, the memory they use will be transferred to the Send buffer and the memory in the Sort Buffer will be recycled. When the next wave of tasks request the memory for the Sort Buffer, they cannot force the Send Buffer to be recycled and the intermediate data generated by them have to be spilled to the disk. Therefore, more disk I/Os will be generated. The situation will also happen with the pair of Send Buffer and Receive Buffer.

Our aim is that only the intermediate data that exceed the total size of the physical memory on one node should be spilled and read once. As shown in Figure 3, after the Map phase is completed at $t_3$, the Sort Buffer is not needed anymore, and no new data (i.e., the final cache file) will be produced (by the Merger component in Figure 3 and consume the Send Buffer. In order to accelerate the Reduce phase, the memory allocated to the Map tasks (through MRM) is gradually returned to the Cache Pool over the time duration between $t_3$ and $t_4$, so that it can be reused by the Reduce tasks (through RRM). In doing so, the received data that have been spilled to the disk can be read back to the Receive Buffer in an overlapping way, which has been described in section 3.3. We can know that only between $t_2$ and $t_3$, may the Sort, Send and Receive phase request the memory simultaneously. In order to optimize performance as discussed above, we set the priorities of the three types of buffers in the following order: Sort Buffer $>$ Send Buffer $>$ Receive Buffer.

The following equations will be used by the Cache Pool to determine the sizes of different types of buffer at runtime, in which *MaxSize* is the maximum size of memory that can be used for intermediate data, *EMPoolSize* is the total size of the Element Pool, *MultiBufSize* is the total size of memory used by I/O Scheduler for the I/O buffer, *TotalSize* is the total size of Cache Pool, *MRMSize* is the size of memory that MRM can use to allocate among Map tasks, *RRMSize* is the size of memory RRM that can use among Reduce tasks, *SortBufSize* is the size of Sort Buffer, *CurSendBufSize* is the current size of Send Buffer, *MapNum* is the number of Map tasks that are running currently, *CurSortBufSize* is the current size of Sort Buffer, *MapSortPeak* is the peak size of each map task's sort buffer (initialized to be 0), i.e., the maximum size of the sort buffer in the running history of the MapReduce job. By using the MapSortPeak, Mammoth can reserve enough space for map tasks and prevent frequent memory recycling from the send or receive buffer when the memory is constrained.

$$
\begin{aligned}
&TotalSize = MaxSize - EMPoolSize - MultiBufSize \\
&RRMSize = TotalSize - MRMSize \\
&MRMSize = Min(SortBufSize + CurSendBufSize, TotalSize) \\
&SortBufSize = \begin{cases} MapSortPeak \times MapNum & MapSortPeak \neq 0 \\ CurSortBufSize & MapSortPeak = 0 \end{cases}
\end{aligned} \tag{2}
$$

When one Map task is launched or completed, it will register or unregister to MRM. In this way, MRM can know the value of *MapNum*. *MergeSort* reserves (i.e., requests) the *cache files* for Sort Buffer from MRM, and Sender unreserves (i.e., returns) the *cache units* that have been shuffled from the Send Buffer and returns them to MRM. This is how MRM counts *SortBufSize* and *CurSendBufSize*. Every Map task calculates its *MapSortPeak* at runtime, and just before it is completed, it reports to MRM its *MapSortPeak*. When the values of these variables change, MRM informs CRM and CRM decides the heap sizes for both MRM and RRM. Similar to Map, when a Reduce task begins or ends, it registers or unregisters to RRM, and RRM then divides *RRMSize* evenly among the executing Reduce tasks. Receiver reserves the memory for the Receive Buffer from RRM, while Pusher unreserves the memory from the Receive Buffer and returns it to RRM. RRM informs CRM of its memory usage condition, which is used to determine when memory is a constraint for the Map tasks. During the procedure of the whole workflow, the memory usage for RMs will change and their memory quotas will be adjusted dynamically. When the new heap size for RM is larger than before, there is enough memory for the tasks to use. On the contrary, when the new heap size is smaller than before, it suggests that the memory is not sufficient and certain parts of the buffer have to be spilled to the disk promptly. The detailed algorithm outline for CRM scheduling memory between MRM and RRM will be presented in the supplementary file.

## 5 I/O Optimization

In Mammoth, in addition to optimizing memory management as presented above, we have also developed the strategies to optimize the I/O performance: 1) designing a unified I/O Scheduler to schedule I/O requests at runtime; 2) tackling the problem that parallel I/O causes massive disk seeks; 3) optimizing the Merge-sort in Hadoop.

### 5.1 I/O Scheduler

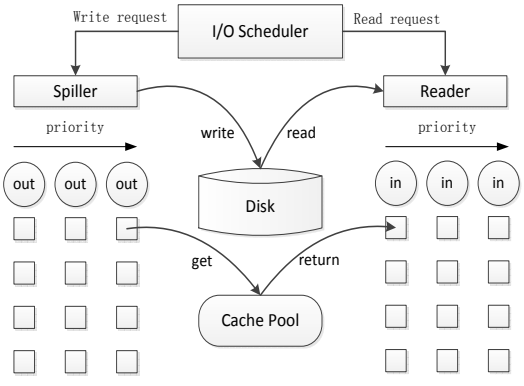



Fig. 10. I/O Scheduler architecture.

In Hadoop, because every task is executed in an individual JVM and does not interact with each other, parallel I/Os that are performed can cause massive disk seeks, and therefore decrease the I/O performance (this phenomenon is also observed in the literature [21]). To implement sequential I/O and overlap the CPU

computing and disk I/O, we have developed an I/O scheduler in Mammoth, whose architecture is shown in Figure 10. There are two components: **Spiller** and**Reader**, responsible for write and read operations respectively. Both components have request buffer queues, with each queue corresponding to one I/O request. These buffers are called multi-buffer. Each buffer queue has a priority, used by **Spiller** or **Reader** to reorder the write/read operations.

In Mammoth, the I/O operations are divided into two types: passive I/O and active I/O. An I/O operation is called passive I/O when the I/O operation is generated because the intermediate data can not be hold in the buffer and they have to be spilled to the disk temporally. Other I/O operations are called active I/O, such as reading the input data from HDFS, writing the final results to HDFS, and writing the Map tasks' intermediate results for fault tolerance. Active I/O has higher priority than passive I/O, because Active I/O is more important for the job progress and should be performed as promptly as possible. For passive I/Os, the operations that operate on the type of buffer with higher allocation priority have the lower spill priority and the higher read priority. For example, since the Sort buffer has the higher allocation priority than the Send buffer. When the system is short of memory, the data from the Send Buffer has the higher spill priority to generate spare memory. If both types of buffer have spilled the data, the Sort buffer has the higher read priority. The detailed algorithm outline for spilling in the I/O scheduler will be presented in the supplementary file.

### 5.2 Parallel I/O

We have conducted an experiment about three I/O types: parallel, sequential, and interleaved on a cluster node. The node is equipped with one disk and 8 CPU cores, running CentOS Linux 5 and its memory is almost used up by the applications. The program is coded in java and executed with jdk-1.6.0. For each of the three types of I/O, $8 \times 200MB$ files are spilled three times, each with different sizes of write buffer. The completion time of each spill is recorded and the average results are plotted in Figure 11, in which the x-axis is the size of write buffer, while the y-axis is the completion time of the spill, representing the spill speed. What we can observe from the results are as follows: Firstly, in most cases, the order of the completion time is: parallel $>$ interleaved $>$ sequential; Secondly, when the buffer gets bigger, the speed of parallel I/O decreases rapidly.

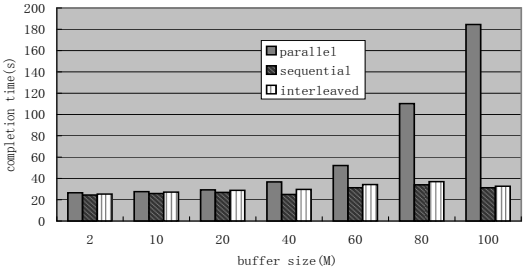



Fig. 11. Comparison among different I/O types

In theory, parallel I/Os will cause overhead due to massive disk seeks, while sequential I/Os do not have this problem. When there is sufficient memory, Linux file system can buffer the I/O requests and solve this problem effectively through a series of optimizations. However, for data intensive applications, there will be little memory left for the file system, and the inefficiency of parallel I/Os will appear. In this case, sequential I/Os can access the disk more effectively (without massive disk seeks). However, sequential I/Os causes the problem of unfairness. When the granularity of interleaved I/O is set appropriate, we can achieve a balance between efficiency and fairness. Since most memory is used as a sequence of cache units in Mammoth, the granularity is set as one *cache unit*. When spilling a *cache file*, the data is added to the corresponding I/O queue *cache unit* by *cache unit*. For the **Reader**, when a *cache unit* is full, it will pop out from the queue and be read.

### 5.3 Merge-Sort

In Mammoth, the merge-sort operations are mainly performed on the data stored in the memory, which we call in-memory merge-sort. In the conventional Hadoop, however, the sort algorithm is mainly performed over the data stored in the disks, which we call the external sort. As the result, the CPU-bound sort instructions and disk I/Os are interleaved in executions. Consequently, after the buffer is full of the sorted data, the CPU must block and wait. With the multi-buffer described above, we can implement non-blocking I/Os. As for the **Spiller**, the units of a *cache file* are added to the **Spiller**'s buffer queue. After a *cache unit* is written to the disk, it will be returned to the Cache Pool immediately.

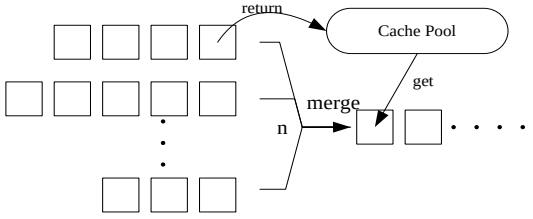



Fig. 12. Merge-sort in Mammoth.

As we know, the space complexity of the traditional merge-sort algorithm is relatively high. In Mammoth, the cached data has a common data structure, i.e., the *cache file*, which is composed of multiple cache units. As shown in Figure 12, assume that there are $n$ sorted cache files to be merged, that the size of one cache unit is $U$, and that the $i$-th ($1 \leq i \leq n$) cache file contains $L_i$ cache units. Then the space complexity of the traditional merge-sort is $\sum_{i=1}^{n} L_i \times U$. In Mammoth, however, after a cache unit is not used anymore, it can be returned to the Cache Pool. Its space complexity is $n \times U$.

## 6 Evaluation

In this section, performance evaluation is presented to verify the effectiveness of the Mammoth system. The settings of the experiments are described in corresponding sections. The performance of Mammoth is compared against Hadoop of version 1.0.1.

### 6.1 Intermediate Data Size

As discussed in Section 2, the size of intermediate data has a big impact on performance of Hadoop. Three typical built-in benchmark applications in Hadoop are used in these experiments: *WordCount without Combiner*, *Sort* and *WordCount with*

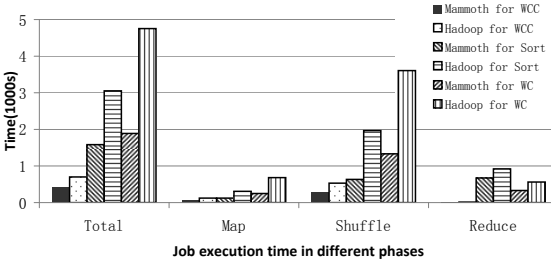



Fig. 13. The performance relative to intermediate data size.

TABLE 1
Configuratons of Hadoop

| Parameter | Values |
|---|---|
| mapred.tasktracker.map.tasks.maximum | 8 |
| mapred.tasktracker.reduce.tasks.maximum | 8 |
| mapred.reduce.child.java.opts | 2GB |
| mapred.map.child.java.opts | 2.4GB |
| io.sort.factor | 100 |
| io.sort.mb | 1.7GB |
| io.sort.spill.percent | 0.9 |
| io.sort.record.percent | 0.5 |
| dfs.block.size | 512MB |

*combiner*. These three benchmarks represent different relations between intermediate data and input data. *WordCount without combiner*, *Sort*, and *WordCount with combiner* represent the cases where the size of intermediate data is larger than, equal to and smaller than the size of input data, respectively.

WordCount is a canonical MapReduce application, in which the Map function translates every word in the input data to a $< word, 1 >$ pair in the intermediate data and the Reduce function sums the word's occurrences and transmits a $< word, N >$ pair. *WordCount without combiner* refers to this version of WordCount. Suppose that the average length of the words is $x$ bytes. In the intermediate data, the value of the $< key, value >$ pair is an integer and its length is 4 bytes in Java. Then the ratio between the size of the intermediate data and the size of the input data is $\frac{x+4}{x}$. The *WordCount with Combiner* refers to WordCount application with a combiner function. The combiner function aggregates the Map task's results by summing up the word's occurrences and transmitting a $< word, N >$ pair for a map task. Based on this function, the intermediate data will be $< word, N >$ pairs, which will be smaller than the input words. Sort is the useful measurement of MapReduce performance, in which the MapReduce framework will sort the data automatically, and the Map function just transmits all the input $< key, value >$ items as the intermediate items.

17 nodes are used in this set of experiments. Each node is equipped with two 8-core Xeon-2670 CPUs, 32GB memory and one SAS disk. One node works as the master and the rest 16 nodes work as slaves. We will compare the performance of Mammoth and Hadoop in terms of i) job execution time, ii) CPU utilization, and iii) I/O utilizations. The input dataset of Sort is produced by randomwriter (native in Hadoop). The size of the dataset for Sort is 320GB, 20GB for each slave node. The input dataset for WordCount is produced by randomtextwriter (native in Hadoop too), and the size is also 320GB with 20GB for each slave node. Each job is run independently for three times and the collected results are then averaged. The job configurations of Hadoop are shown in Table 1. This is the best configuration for Hadoop according to our analysis and testing. The configurations of Mammoth are the same as Hadoop, except that the whole memory that all tasks are sharing the heap of the same JVM. The heap size of the execution engine JVM in Mammoth is 32GB, which is the total memory size in a node. Note that in Table 1, the block size is set to be 512MB. This is because the work in [19] shows that setting the bloack size to be 512MB in Hadoop (instead of the common configuration of 64MB) can reduce the map startup cost and increase the performance. Therefore, we are not comparing Mammoth against Hadoop with the disadvantaged configurations in the experiments.

Figure 13 shows the total execution times of three different applications as well as the breakdown of the execution times in three different execution phases, where WCC is WordCount with combiner and WC is WordCount without combiner. It can be seen from the figure, when running WC, Mammoth reduces the job execution time significantly compared with Hadoop. The execution time of WC in Hadoop is 4754s, while the execution time in Mammoth is 1890s, which amounts to a speedup of 2.52x in job execution time. The reasons why Mammoth achieves the performance improvement are as follows:

1) Mammoth reduces the number of disk accesses. Although in Hadoop the 1.7GB intermediate buffer in the Map tasks can hold all the intermediate data in memory, which minimizes the I/O operations for intermediate data, Hadoop still writes the intermediate data once and read once for the Shuffle phase. Mammoth pushes the intermediate data to the Reduce tasks directly from memory and performs fault-tolerance spilling in an asynchronous manner. Therefore, there are no I/O waits in the shuffle phase. The decrease in disk I/Os greatly improves the Map tasks' performance. As shown in Figure 13, the average time taken by the Map phase of Mammoth is 249s, compared to 624s of Hadoop.

In this experiment, the number of the Map task waves is 5 ($\frac{20GB}{512MB\times 8} = 5$). In Mammoth, Shuffle is performed from Map tasks' memory to Reduce tasks' memory, and the Shuffle phase is performed concurrently with the Map Phase. Therefore, the Shuffle phase in Mammoth is not the bottleneck in the MapReduce model. As shown in Figure 13, the shuffle time is 1331s , which is almost equal to the total Map tasks' time ($5 \times 249s = 1245s$). However, the Hadoop Shuffle time is 3606s, which is larger than Map tasks' total time ($604s \times 5 = 3020s$).

At the Reduce side of Hadoop, the heap size for one Reduce task JVM is 2GB and a Reduce task can use up to 70% of its heap size (i.e. 1.4GB), called *maxsize*, to hold the intermediate data from the Map tasks. When the size of the intermediate data grows to 66.7% of maxsize, the Reduce task will write all the intermediate data into the disk. The size of the intermediate data for a Reduce task is about 4GB. So a Reduce task will have to spill about $1.4GB \times 66.7\% \times \lfloor \frac{4G}{1.4GB\times 66.7\%} \rfloor = 3.7GB$ intermediate data into the disk and read the intermediate data from the disk for the Sort and Reduce operations. As for Mammoth, the heap size is 32GB. Mammoth only needs to spill $4GB \times 8 - 32 \times 70\% = 9.6GB$ intermediate data into the disk, compared to $3.7GB \times 8 = 29.6GB$ in Hadoop. This is why Mammoth gains 70% performance improvement over Hadoop in the Reduce Phase.

2) The second reason why Mammoth achieves the shorter job execution time is because we improved the I/O efficiency for data spilling. As mentioned in section 5.1 and section 5.2, a large number of parallel I/Os will incur massive disk seeks, and the sequential or interleaved I/O is adopted, which improves the read/write speed. CPU computing and disk accessing are also overlapped. In Hadoop, merge-sort is widely used. The merge-sort operations cause the CPU to wait until the disk writes finish. Mammoth utilizes a Multi-buffer technique to implement the asynchronous I/O and achieves the optimized in-memory merge-sort. Consequently, CPU does not need to wait for the completion of I/O operations, which improves the resource utilization.

From Figure 13, we can also see that the performance improvements of Mammoth for the other two applications (Sort and WCC) are not as significant as for WC (a speedup of 1.93x and 1.66x in job execution time for Sort and WCC, respectively). The reason for this is as follows. The size of the output data produced by a Map task is 850MB for WCC, 512MB for Sort and 30MB for WC, respectively. As the size of the intermediate data decreases, the problem of I/O waits becomes less serious. The CPU power becomes a more critical element for the job execution time.

### 6.2 Impact of memory

We conducted the experiments to evaluate the impact of memory by examining the performance of both Mammoth and Hadoop under different size of physical memory on each node. In the experiments, 17 nodes are used and WC is run in the same way as in Section 6.1. The experimental results are shown in Figure 14. In these experiments, each node is equipped with 64GB physical memory in total, and we modify the grub configuration file of the Linux OS to set the runtime available memory volume. In order to get the highest performance, the heap size in Mammoth and the size of map sort buffer in Hadoop were tuned adaptively according to the physical memory capacity. The following observations can be made from Figure 14.

First, as the memory size decreases, the job execution time of both Mammoth and Hadoop increases. This suggests that the memory resource has the inherent impact on the performance of data intensive applications.

Second, when the memory is sufficient(the cases of 64GB and 48GB in this figure), both Hadoop and Mammoth gains good performances in terms of job execution time. In this case, Mammoth achieves about 1.9x speedup against Hadoop. When the memory becomes a constraint (i.e., 32GB and 16GB in the figure), the job execution time in Hadoop increases much more sharply than that in Mammoth. Under these circumstances, Mammoth gains up to 2.5x speedup over Hadoop.

The reason for this can be explained as follows. When the memory becomes constrained, both Hadoop and Mammoth will spill intermediate data to the local disk. As explained in section 6.1, however, Hadoop performs some unnecessary disk accesses and the coarse-grained memory usage causes the problem of memory constraint even more serious. The experimental results suggest that Mammoth can adapt to different memory conditions and achieve consistent performance improvement.

### 6.3 Scalability

We have also conducted the experiments on an increasing number of nodes to examine the scalability of Mammoth in terms of jobs' execution time. The experimental results are shown in Figure 15. In these experiments, two applications, WC and Sort, are run on 17, 33, 65 and 129 nodes, with one node as the master and the rest as slaves. Each node is equipped with 32GB memory and the bandwidth of the network interconnecting the nodes is 1Gbps. For each application, there are 20GB data on every slave node.

As shown in Figure 15, the execution time of WC increases under both Hadoop and Mammoth as the number of nodes increases. We carefully profiled the processing of the application. We find that the reason for the increase in the execution time of WC is the high straggling effect occurring in the execution of WC. The WC application is to count the occurrence frequency of the words in the input data. Since the occurrence frequencies of different words could be very different, the straggling effect is

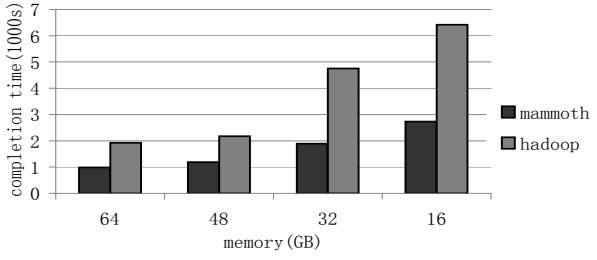



Fig. 14. Performance relative to available memory.

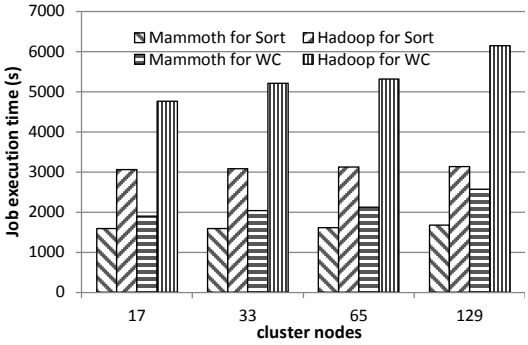



Fig. 15. Performance relative to scalability

easy to occur. And as the input data becomes bigger, the likelihood of having a longer tail is higher, which causes the increases in the execution time. Our observation and profiling of the WC running support this analysis, because we found that the sizes of the input data to different reduce tasks varied greatly. This phenomenon was also observed in [22], which is summarized as the problem of partitioning skew in [22].

Mammoth uses the task speculative execution, which is inherited from Hadoop, to deal with stragglers. Also, Mammoth can coordinate the resource usage between tasks, and tries to allocate more memory to the tasks which demand more memory and consequently reduce the amount of data that have to be spilled out. This is helpful in mitigating task straggling.

The trend of the execution time of the Sort application is different from that of WC. It can be seen from the figure that the execution time of Sort is stable under both Mammoth and Hadoop as the number of nodes increases from 17 to 129. This is because in Sort the data partitioning among tasks is fairly even and therefore there is no partition skewness seen in WC. The stable execution time over an increasing number of nodes suggests that Mammoth has the similar scalability as Hadoop.

It can be seen in Figure 15 that Mammoth outperforms Hadoop for both WC and Sort. This is because Mammoth can coordinate the memory usage among tasks. This result verifies the effectiveness of Mammoth.

### 6.4 Experiments with Real Applications

CloudBurst [23] is a MapReduce implementation of the RMAP algorithm for short-read gene alignment, which is a common task in bioinformatics. We evaluated the performance of running CloudBurst with the lakewash_combined_v2 data set obtained from the University of Washington [15]. Pegaus [16] is a MapReduce library for the graph algorithms written in Java. We also conducted the experiments with four frequently used graph computing applications: 1) PageRank, a graph algorithm that is widely used by the search engines to rank web pages; 2) Concmpt, an application for connected component detecting, often used as the basic building block in the complex graph mining applications; 3) Radius, an application for graph radius computing; 4) DegDist, an application for vertex degree counting.

We ran all these applications except CloudBurst with the same dataset on 17 nodes, with the 20GB data for each node. Each node is equipped with 32GB memory. We ran CloudBurst on one node. The reason for this is because the lakewash_combined_v2 data set of CloudBurst is too small to be run on multiple nodes. The results are shown in Table 2.

In PageRank and Concmpt, both Map and Reduce Phases are I/O intensive. Therefore the improvements of Mammoth over Hadoop are high for these two applications. Radius is also an

TABLE 2
Performance comparison

| *Application* | *Running Time* | | *Speedup* |
|---|---|---|---|
| | *Mammoth* | *Hadoop* | |
| PageRank | 2760s | 13090s | 4.74x |
| Concmpt | 2380s | 12364s | 5.19x |
| DegDist | 438s | 737s | 1.68x |
| Radius | 3365s | 12250s | 3.64x |
| CloudBurst | 1395s | 2313s | 1.66x |

I/O-intensive application, but the intensity is not as big as that of PageRank and Concmpt. Therefore, its speedup is consequently smaller than that of PageRank and Concmpt. CloudBurst's intermediate data are big. But it is also a CPU-intensive application. Therefore, the extent of I/O bottleneck is not very big. That is why the improvement for CloudBurst is relatively small. DegDist has relatively small size of intermediate data and therefore is not an I/O-intensive application, which explains why its achieved performance improvement is also relatively small.

### 6.5 Comparing Mammoth and Spark

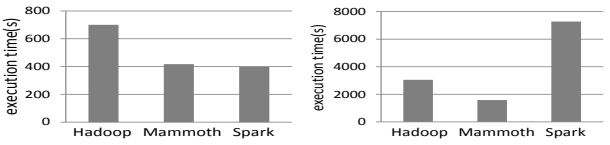
We conducted two sets of new experiments to compare the performance of Mammoth, Spark (Version 0.9.0) and Hadoop. We understand that Spark can achieve the excellent performance for iterative and interactive applications (such as Pagerank) when the memory is sufficient, which has been demonstrated in the literature [31] and is also discussed in Subsection 7.1. We believe that Spark is also able to outperform Mammoth in those circumstances. Therefore, the two sets of new experiments were conducted in the settings beyond the Sparks "comfort zone", i.e., i) not for iterative or interactive applications and ii) when the memory is insufficient.

In the first experiment, Mammoth and Hadoop execute the WordCount with Combiner (WCC) application, while Spark implements the WordCount application with the reduceByKey operator. Under these configurations, the intermediate data generated by Spark, Mammoth and Hadoopare are small (which will be discussed in more detail in the analysis of the experimental results). Therefore, the memory in the running platform is sufficient. However, WC is a batch processing application, not an iterative and interactive application, since the intermediate data generated by the Map tasks will be taken directly as input by the Reduce tasks to produce the final results.

In the second experiment, Mammoth, Spark and Hadoop all execute the Sort application, which is not an iterative and interactive neither. During the running of the Sort application, a large amount of intermediate data will be generated by Map tasks, which effectively cause the situation where there is insufficient memory in the running platform.

In the two sets of new experiments, the cluster configuration and the input dataset are exactly the same as those in Section 6.1. Spark was deployed with the standalone mode. The original input dataset and the final results were stored in the HDFS. For WordCount, the input datas, 20GB for each node, were generated using the built-in program in Hadoop, RandomTextWriter. For Sort, the input data, 20GB per node too, were generated using the built-in program in Hadoop, RandomWriter. In order to make the fair comparison, we optimize the software configuration parameters of Spark according to the official documents [6]. We set the parameter spark.serializer to be org.apache.spark.serializer.KryoSerializer and set the parameter spark.shuffle.consolidateFiles to be true.

Figure 16(a) demonstrates the performances of the three systems for running WordCount. It can be seen from the figure that Mammoth and Spark obtain almost the same performance,




(a) WordCount (b) Sort

Fig. 16. Performance comparison among Spark, Hadoop and Mammoth

which is about a 1.7x performance speedup over Hadoop. The three systems aggregate the intermediate data in the similar way, i.e., the <key, value> pairs with the same key will be summed up to one <key, value> pair. For example, two <word, 1> pairs are transformed to one <word, 2> pair. In this way, the quantity of the intermediate data will be reduced significantly. Our experimental records show that the size of the intermediate data generated by Mammoth and Hadoop in the experiment is 926M and that by Spark is 103.4M. Consequently the memory was sufficient. In Mammoth, most intermediate data are processed in the memory except when spilling the Map tasks results to the disks for fault tolerance. Spark processes shuffling based on the disks, and uses the hash table instead of the sort to implement data aggregation. Both of them utilize the memory better than Hadoop, which is the reason for their performance improvements over Hadoop. This result indicates that for non-iterative and non-interactive job like WordCount, Mammoth can achieve similar performance as Spark, even when the memory is sufficient.

Figure 16(b) shows the performance comparison of the three systems for running Sort. It can be seen that Mammoth obtains the best performance, while surprisingly Spark achieves the worst performance, even worse than Hadoop. We have carefully investigated the reason for this phenomenon. For the Sort application, the intermediate data cannot be aggregated and therefore, their size cannot be reduced by optimistic algorithms, which is different from WordCount. In Hadoop and Mammoth, the quantity of the intermediate data is the same as that of the input dataset, that is $20GB \times 16 = 320GB$. Mammoth implements a global memory manager which can minimize the quantity of disk I/O, and a global I/O manager which can minimize the overhead of I/O operations. Through the execution log, we found that most of the intermediate data were processed in the memory. In Spark, however, its serializer expands the intermediate data, and the compression rate of the intermediate data is not high because the data are randomized. As the result, the quantity of the intermediate data reaches as high as 663GB. Firstly, the intermediate data were written and read from the disks in the shuffle phase. Secondly, on the reduce side, the whole data cannot fit into the memory, and therefore Spark has to implement the external merge sort. That is why the performance of Spark is so poor. As for Hadoop, it does not depend on the memory size as much and therefore its performance is relatively stable. This result suggests that Mammoth can achieve much better performance than Spark for batch processing applications, such as Sort, when the memory is not sufficient. In summary, Spark can achieve better performance than Mammoth and Hadoop for interactive and iterative applications when the memory is sufficient. For batch processing applications, Mammoth can adapt better to various memory environments and can obtain similar performance as Spark when memory is sufficient, and can outperform Spark when the memory is insufficient.

## 7 Related Works

Extensive studies have been carried out in the near past to improve Hadoop or other MapReduce implementations. This section dis-

cusses the closely related work from the following two aspects, leaving other broadly related work to the supplementary file.

### 7.1 In-memory data analytics

PACMan [8] makes the underlying HDFS cache the input data blocks of the MapReduce jobs in the nodes' main memory. It uses a per-job "all-or-nothing" cache replacement strategy to accelerate small interactive jobs, especially the jobs with only one-wave map tasks, on the data centers shared by multiple jobs.

Similar to Mammoth, Storm presented in [2] also uses a thread to run a task and the tasks on a node can share the same JVM memory. However, STORM leaves the multi-task memory management completely to the JVM. In fact, Storm has different design goals from Mammoth. Storm is designed for real-time streaming processing, and such low-latency streaming applications require the support for the fast task bootstrap based on JVM reuse. The main challenges and novelty of our work are to carry out the insightful observations and analyses for the characteristics of the disk accessing behaviors and the memory usages in the MapReduce phases as well as the relation between these characteristics and the potential opportunities for I/O performance optimization, and exploit these characteristics to design and implement memory scheduling for buffer spilling avoidance and disk accessing optimization. As the result, our work is able to utilize the system resources more intelligently rather than just opportunistically. These designed optimization measures are especially important (and therefore the benefits of these measures are even more prominent) when the resources are restricted.

Several newly proposed systems [10], [31], [25] store all the job data in memory if possible, which significantly reduces the response time of many interactive queuing and batched iterative applications. All of these systems aim at making full use of the rich memory resources of modern data center servers, but do not address the associated problems when total memory demand of the job exceeds the physical memory size and different datasets compete the limited capacity. Mammoth is designed for the servers equipped with moderate memory sizes. Our heuristic memory scheduling algorithm tries to maximize the holistic benefits of the MapReduce job when scheduling each memory unit. Since Spark is a new data processing system that is gaining in huge popularity nowadays [31], we would like to discuss Spark and compare it with Mammoth in more detail in the following.

Compared with Hadoop, Spark can make better use of memory and achieve up to 100x performance improvement, especially for iterative and interactive applications [31]. However, the improvement of up to 100x is achieved in the condition that the running applications can make the best use of the Spark framework and there is the sufficient memory in the system for the running application. The condition does not always hold. Spark requires a large amount of memory for running Spark itself and the applications. It may well be the case that there is not sufficient memory in the running platform for the applications to be processed. The aim of Mammoth is to improve the usage efficiency of the memory in various circumstances, especially when there is not sufficient memory in the supporting platform. Although both Mammoth and Spark execute the tasks with threads and manage the memory in the application level, Mammoth differs from Spark in the following aspects.

First, the ways of memory management in Mammoth and Spark are rather different. Mammoth is based on MapReduce. We have carefully analyzed the characteristics of memory usage in different phases of the MapReduce framework, and designed a novel rule-based heuristic to prioritize memory allocations and revocations among execution units (mapper, shuffler, reducer, etc.). In this way, we can maximize the holistic benefits of the Map/Reduce job when scheduling each memory unit. In Spark, the memory can be used for the Resilient Distributed Datasets (RDD) cache and running the framework itself. As for the RDD cache, it depends on the users themselves when and how the data are cached, which increases the uncertainty of the memory usage. For the iterative and the interactive applications, caching the frequently used RDDs will significantly improve the applications performance. However, for many batch processing applications, the RDD cache cannot exhibit its advantages, and therefore those applications can only rely on the memory management in the Spark framework itself. Spark directly requests and revokes the memory from the JVM, and does not have a global memory manager in the application level. Spark uses the hash table to aggregate the data, which is different from the sort way used by Mammoth and Hadoop. When the memory is sufficient, hash will certainly be quicker than sort. However,when the memory is insufficient, it will have to spill the data to disks, and its performance will decrease significantly. Thanks to the holistic manner of memory usage, Mammoth can adapt much better to various memory situations, even when the memory is insufficient. On the contrary, the performance achieved by Spark is excellent when there is the sufficient memory, but not so when the memory is insufficient.

Second, the manners of shuffling in Spark and Mammoth are different. Spark writes the data to the disk on one side and reads them from the disk on the other, while Mammoth stores the Map tasks results in the Send Buffer, and sends them to the Reduce tasks Receive Buffer directly (Mammoth will write the data in the Send Buffer to the disks only for the purpose of fault tolerance).

Finally, Mammoth implements several techniques to optimize the I/O operations, including i) prioritizing different types of I/O operations and scheduling them by a global I/O scheduler, ii) using the interleaved I/O and the sequential I/O to reduce the disk seeking time, iii) overlapping the disk I/O with the CPU computing through multi-cache. Spark simply performs the disk I/Os in an opportunistic and parallel way.

### 7.2 Global resource management in data centers

Some other newly proposed "data-center schedulers" such as Apache YARN, Twitter's Mesos and Google's Omega [27], [13], [24] also support the fine-grained resource sharing in the data center. However, the design objective and the implementation methods of these data-center schedulers are very different from Mammoth in terms of the following aspects. First, the target of these data-center schedulers is to enforce resource allocation strategy based on fairness or priorities when sharing the resources of large-scale data-centers among multi jobs/users, while Mammoth is aimed at improving the performance of each job. Second, these data-center schedulers make the scheduling decisions globally at the cluster level, but the Mammoth execution engine on each node schedules memory locally and independently. Finally, The data-center scheduling frameworks reserve the requested memory capacity on the corresponding node for each task, and use the system-level isolation methods such as virtualization and Linux container to guarantee that the tasks only uses the reserved memory spaces. The Mammoth tasks dynamically request the memory without the pre-determined per-task limit.

## 8 Conclusions and Future Work

In this paper, we propose a MapReduce system called Mammoth that is suitable for a memory-constrained platform, such as

the HPC clusters. In Mammoth, a global memory management scheme is designed for all tasks on a node, which is fundamentally different from the separated memory management in Hadoop. This new design increases the memory utilization for tasks and balances the performance between the CPUs and the I/O system. Mammoth also leverages a multi cache for sequential and interleaved disk accesses, which improves the data access rate. This paper have conducted experiments on the HPC clusters. The experiment results show that Mammoth achieves a speedup of up to 5.19x over Hadoop in terms of job execution time.

It is worthy noting that the Mammoth system does not change the Hadoop processing phases. It is easy to integrate the existing techniques into Mammoth. Furthermore, the MapReduce applications can be transparently run on the Mammoth system without any modifications required.

The following research issues are planned for the future work.

First, the current version of Mammoth processes one job at a time to achieve the optimized performance using the strategies developed in this paper. We plan to optimize the performance for running multiple jobs simultaneously in Mammoth. In order to realize the multi-job mode, i.e., run multiple jobs simultaneously, the same methodology as in this paper (i.e., the global memory and I/O management) can be applied. In order to optimize the performance in the multi-job mode, the key additional consideration is to take into account each job's characteristics. We plan to explore the following two potential approaches to achieving this.

(a) We can integrate Mammoth with Mesos [13] or Yarn [27], which are job-wise resource management systems. Each job can first request the essential resources from Mesos or Yarn, and then the tasks in the job share the allocated resources and follow the methodology described in this paper. This approach aims to make use of the existing job management capability in Mesos or Yarn.

(b) We can run the tasks of different jobs in a single task execution engine JVM and share the global resources (memory, disk, network and CPU) on each node. This way, some runtime information of each individual job may need to be obtained separately. For example, different jobs may have different ratios between the size of the intermediate data and the size of the input data. Meanwhile, we may need to consider more job information to set the priority order for resource usage. The additional job information could include each job's priority, each job's execution progress and each job's resource demand, etc..

In summary, in order to support the multi-job mode, the core methodologies developed in this paper can still be applied, but the additional work may need to be conducted to either integrate Mammoth with the job-wise resource management systems if the first approach is taken, or if taking the second approach, set the above parameter values so that the performance can be optimized when the developed framework is run in the multi-job mode. We plan to tackle these issues in our future work.

Second, Mammoth is essentially a memory-intensive framework, which manages as much memory as possible in the application level, following the similar philosophy as Spark [31] and PACMan [8]. A new research angle along this research direction is that we still manage the memory in the application level, but try to manage the memory opportunistically instead of carefully crafting the memory usage as we did in Mammoth. In the further, we plan to design and implement the opportunistic approach to utilize the memory in Mammoth.

Third, Mammoth only focuses on the disk I/O currently. We plan to integrate the support of the network I/O into Mammoth.

Finally, we plan to investigate hybrid scheduling algorithms, e.g., scheduling both CPU-intensive and data-intensive applications, to balance the CPU and I/O processing on the HPC clusters.

### *Acknowledgments*

This work is supported by the NSFC (grant No. 61370104 and No. 61133008), National Science and Technology Pillar Program (grant No. 2012BAH14F02), MOE-Intel Special Research Fund of Information Technology (grant No. MOE-INTEL-2012-01), Chinese Universities Scientific Fund (grant No. 2014TS008).

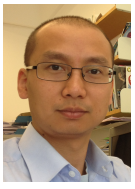
## REFERENCES

[1] Apache hadoop. http://hadoop.apache.org/.
[2] Apache storm. http://storm.incubator.apache.org//.
[3] Building a terabyte-scale data cycle at linkedin with hadoop and project voldemort. http://data.linkedin.com/blog/2009/06/building-a-terabyte-scale-data-cycle-at-linkedin-with-Hadoop-and-project-voldemort.
[4] New idc worldwide hpc end-user study identifies latest trends in high performance computing usage and spending. http://www.idc.com/getdoc.jsp?containerId=prUS24409313.
[5] Scaling hadoop to 4000 nodes at yahoo! http://developer.yahoo.net/blogs/Hadoop/2008/09/scaling_Hadoop_to_4000_nodes_a.html.
[6] Tuning spark. http://spark.apache.org/docs/0.9.0/tuning.html.
[7] Visit to the national university for defense technology changsha, china. http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf.
[8] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: Coordinated memory caching for parallel jobs. In *Symposium on Network System Design and Implementation (NSDI)*, 2012.
[9] D. Borthakur, K. Muthukkaruppan, K. Ranganathan, S. Rash, J. S. Sarma, N. Spiegelberg, D. Molkov, R. Schmidt, J. Gray, H. Kuang, A. Menon, and A. Aiyer. Apache hadoop goes realtime at facebook. In *ACM Conference on Management of Data (SIGMOD)*, 2011.
[10] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Eficient iterative data processing on large clusters. In *International Conference on Very Large Data Bases (VLDB)*, 2010.
[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementations (OSDI)*, 2004.
[12] Z. Fadika, E. Dede, M. Govindaraju, L. Ramakrishnan, and S. Ghemawat. Adapting mapreduce for HPC environments. In *High-Performance Distributed Computing (HPDC)*, 2011.
[13] B. Hindman, A. Konwinski, MateiZaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Symposium on Network System Design and Implementation (NSDI)*, 2011.
[14] T. Hoefler, J. Dongarra, and A. Lumsdaine. Towards efficient mapreduce using mpi. In *volume 5759 of Lecture Notes in Computer ScienceSpringer (2009), pp. 240C249*.
[15] B. Howe. lakewash_combined_v2.genes.nucleotide. http://www.cs.washington.edu/research/projects/db7/escience_datasets/seq_alignment/.
[16] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system implementation and observations. In *IEEE International Conference on Data Mining (ICDM)*, 2009.
[17] S. Krishnan, C. Baru, and C. Crosby. Evaluation of mapreduce for gridding lidar data. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 33–40, Nov 2010.
[18] K. Kuiper, R. Arora, and R. Dimpsey. Java server performance: A case study of building efficient, scalable JVMs. In *IBM System Journal, 39(1)*, 2000.
[19] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *ACM Conference on Management of Data (SIGMOD)*, 2011.
[20] M. V. Neves, T. Ferreto, and C. D. Rose. Scheduling mapreduce jobs in hpc clusters. In *European Conference on Parallel and Distributed Computing (Euro-Par)*, 2012.
[21] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A balanced large-scale sorting system. In *Symposium on Network System Design and Implementation (NSDI)*, 2011.

[22] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. ThemisMR: An i/o-efficient mapreduce. In *ACM Symposium on Cloud Computing (SOCC)*, 2012.
[23] M. C. Schatz. CloudBurst: Highly sensitive read mapping with mapreduce. In *Bioinformatics, 25(11):1363-9*, 2011.
[24] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *European Conference on Computer Systems(EuroSys)*, 2013.
[25] A. Shinnar, D. Cunningham, and B. Herta. M3R: Increased performance for in-memory hadoop jobs. In *International Conference on Very Large Data Bases (VLDB)*, 2012.
[26] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *International Symposium on Memory Management (ISMM)*, 2000.
[27] V. K. Vavilapalli, A. C. Murthy, and e. a. Chris Douglas. Apache hadoop yarn: Yet another resource negotiator. In *Symposium on Cloud Computing (SOCC)*, 2013.
[28] J. Venner. *Pro Hadoop*. apress, 2009.
[29] T. White. *Hadoop: The Deinitive Guide*. Yahoo Press, 2010.
[30] R. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *ACM Conference on Management of Data (SIGMOD)*, 2013.
[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Symposium on Network System Design and Implementation (NSDI)*, 2012.

**Xuanhua Shi** is a professor in Service Computing Technology and System Lab and Cluster and Grid Computing Lab, Huazhong University of Science and Technology (China). He received his Ph.D. degree in Computer Engineering from Huazhong University of Science and Technology (China) in 2005. From 2006, he worked as an INRIA Post-Doc in PARIS team at Rennes for one year.). His current research interests focus on the scalability, resilience and autonomy of large-scale distributed systems, such as peta-scale systems, and data centers.

**Ming Chen** is a master student in Service Computing Technology and System Lab and Cluster and Grid Computing Lab at Huazhong University of Science and Technology (China). He is now doing some research on cluster and big data technology.

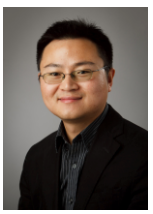


**Ligang He** received the Bachelors and Masters degrees from the Huazhong University of Science and Technology, Wuhan, China, and received the PhD degree in Computer Science from the University of Warwick, UK. He was also a Post-doctoral researcher at the University ofCambridge, UK. In 2006, he joined the Department of Computer Science at the University of Warwick as an Assistant Professor, and then became an Associate Professor. His areas of interest are parallel and distributed computing,high performance Computing and Cloud computing.

**Xu Xie** is a masters student in Service Computing Technology and System Lab and Cluster and Grid Computing Lab at Huazhong University of Science and Technology (China). His research interests focus on large scale distributed data processing.

**Lu Lu** received his Bachelor degree in Computer Science and Technology from the Naval University of Engineering (China) in 2008. He is now a Ph.D. candidate student in Service Computing Technology and System Lab and Cluster and Grid Lab at Huazhong University of Science and Technology (China). His research interests focus on large scale distributed data processing.

**Hai Jin** a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He is now Dean of the School of Computer Science and Technology at HUST. Jin received his PhD in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. Jin worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is a senior member of the IEEE and a member of the ACM.



**Yong Chen** is an Assistant Professor and Director of the Data-Intensive Scalable Computing Laboratory in the Computer Science Department of Texas Tech University. His research interests include data-intensive computing, parallel and distributed computing, high-performance computing, and Cloud computing.

**Song Wu** a professor of computer science and engineering at Huazhong University of Science and Technology (HUST) . He is now the director of Parallel and Distributed Computing Institute at HUST. He has also served as the vice director of Service Computing Technology and System Lab and Cluster and Grid Computing Lab of HUST. His current research interests include cloud computing , system virtualization and resource management in datacenters.