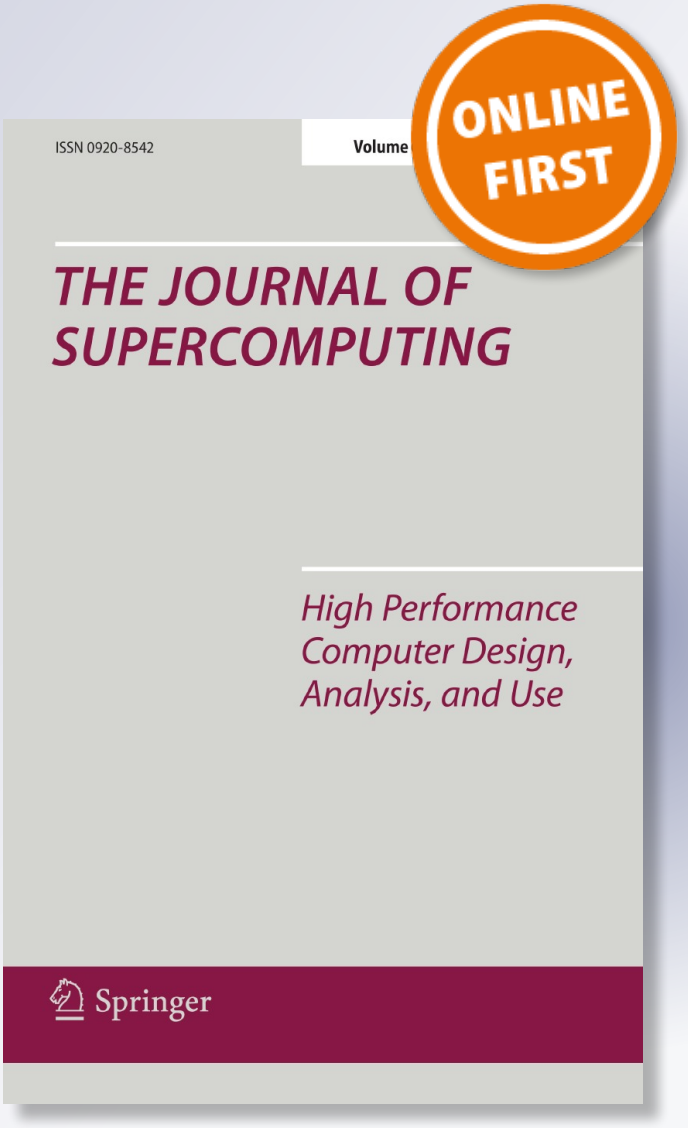
Peacock: a customizable MapReduce for multicore platform

Song Wu, Yaqiong Peng, Hai Jin & Jun Zhang

The Journal of Supercomputing
An International Journal of High-Performance Computer Design, Analysis, and Use

ISSN 0920-8542

J Supercomput
DOI 10.1007/s11227-014-1238-2



Springer

J Supercomput
DOI 10.1007/s11227-014-1238-2

Peacock: a customizable MapReduce for multicore platform

Song Wu · Yaqiong Peng · Hai Jin · Jun Zhang

Abstract MapReduce has been demonstrated to be a promising alternative to simplify parallel programming with high performance on single multicore machine. Compared to the cluster version, MapReduce does not have bottlenecks in disk and network I/O on single multicore machine, and it is more sensitive to characteristics of workloads. A single execution flow may be inefficient for many classes of workloads. For example, the fixed execution flow of the MapReduce program structure can impose significant overheads for workloads that inherently have only one emitted value per key, which are mainly caused by the unnecessary reduce phase. In this paper, we refine the workload characterization from Phoenix++ according to the attributes of key-value pairs, and give a demonstration that the refined workload characterization model covers all classes of MapReduce workloads. Based on the model, we propose a new MapReduce system with workload-customizable execution flow. The system, namely Peacock, is implemented on top of Phoenix++. Experiments with four different classes of benchmarks on a 16-core Intel-based server show that Peacock achieves better performance than Phoenix++ for workloads that inherently have only one emitted value per key (up to a speedup of 3.6×) while identical for other classes of workloads.

Keywords MapReduce · Workload-customizable · Performance · Multicore

Note that Phoenix++ is the best available implementation of MapReduce on shared-memory multicore platform.

S. Wu (✉) · Y. Peng · H. Jin · J. Zhang
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology,
Wuhan 430074, China
e-mail: wusong@hust.edu.cn

Published online: 21 June 2014

1 Introduction

Emerging multicore processors, as another form of following Moore's Law, are commercially replacing traditional single-core processors currently. Nowadays, all major chip vendors have multicore products with quad-cores and eight-cores on a single die, and trends indicate that future chips will consist of even thousands of cores [6].

Parallel applications become as popular as multicore chips in order to take advantage of the abundant computing resources. However, writing parallel programs is difficult for normal programmers. The issues of how to improve application performance without sacrificing productivity is an important topic in the context of single multicore machine. MapReduce [10] is initially implemented on clusters, and it has also been demonstrated to be an effective approach to programming single multicore machine with easy-to-use APIs [7,11,13,18]. Compared to the cluster version, the single-multicore-based MapReduce does not have bottlenecks in disk and network I/O. Thus, it is more sensitive to workload-influenced details [20]. If the single-multicore-based MapReduce adopts a fixed execution flow like cluster-based implementations, it may be inefficient for many classes of workloads.

Although previous works have noted the importance of adapting MapReduce to various workload characteristics [17,19,20], they only focus on data structure optimization while maintaining the same execution flow for all classes of workloads. Considering workloads that inherently have only one emitted value per key, invoking reduce function to aggregate values with the same key is unnecessary for them and can introduce non-trivial overheads. *Our goal* in this paper is to develop a customizable MapReduce system, which can provide appropriate execution flows for different classes of workloads to achieve high performance on multicore.

Overview of this work To achieve this goal, we first construct a workload characterization model, which is a refinement of workload characterization from Phoenix++ and classifies MapReduce workloads into four classes. The model is constructed according to the attributes of key-value pairs. Then we demonstrate that the model covers all classes of MapReduce workloads and discuss its implications on execution flow design. For example, the unnecessary reduce phase can be eliminated for workloads that inherently have only one emitted value per key. Finally, guided by the implications, we design and implement a new and flexible MapReduce system with workload-customizable execution flow on the top of Phoenix++, namely Peacock. The main challenge is how to provide unified buffer management for different execution flow in Peacock. We address the issue by modifying the implementations of containers in Phoenix++, and each type of the container targets at a specified class of workloads. The containers are responsible for managing buffers, which apply aggressive optimization for different classes of workloads.

Summary of evaluations Our experimental results indicate that Peacock outperforms Phoenix++ due to its customizable execution flow for different classes of workloads, resulting in a speedup up to 3.6× for workloads that inherently have only one emitted value per key.

This paper makes the following contributions:

- We analyze that it is more efficient for MapReduce system to provide workload-customizable execution flow on single multicore machine.
- We introduce a MapReduce workload characterization model, which can provide implications on the appropriate execution flow for different classes of workloads.
- We design and implement a new and flexible MapReduce system called Peacock with workload-customizable execution flow.
- We compare Peacock with Phoenix++ through a series of experiments, and the results show that Peacock is more efficient than Phoenix++.

The remainder of the paper is organized as follows: The next section presents the background of MapReduce, existing MapReduce library for multicore, and a motivating example to argue that it is more efficient for the MapReduce runtime to provide workload-customizable execution flow on single multicore machine. Then, we describe our workload characterization model in Sect. 3. Section 4 describes the system design and implementation. Section 5 evaluates the performance of Peacock with four representative benchmarks. Section 6 relates our work to previous work. Finally, we conclude the paper in Sect. 7.

2 Background and motivation

In this section, we briefly overview the MapReduce programming model at first. Then, we look into the challenges of MapReduce library for single multicore machine. As Phoenix++ is the best available implementation of MapReduce on shared-memory multicore platform, we use it as a motivating example to analyze the performance issues of the MapReduce runtime for workloads that inherently have only one emitted value per key.

2.1 MapReduce programming model

MapReduce [10] was created by Google to program large clusters for data-intensive applications with main benefits in simplicity and robustness. Generally, programmers only need to consider the implementations of two interfaces, *map* and *reduce*. The map function independently processes a piece of the input data and generates multiple intermediate key-value pairs. The reduce function is used to group all the key-value pairs with the same key. The MapReduce runtime takes care of low-level details such as parallelization, concurrency control, and fault tolerance. Over the last few years, there are many studies focusing on mapping MapReduce model to different platforms [7,12,13,18].

2.2 MapReduce library for multicore

The popularity of multicore chips has promoted lots of work on novel programming models to write parallel programs for multicore [9,11,16,18]. Programmers need appropriate abstractions which free them from low-level details so that they can

only focus on high-level code. Ranger et al. [18] firstly propose an implementation of MapReduce for shared-memory systems namely Phoenix, which is a promising alternative to take advantage of the additional compute power of multicore platform. Phoenix is heavily optimized [19,20] successively and its success shows that MapReduce based applications can perform competitively with the hand-crafted P-threads version, in the context of single multicore machine. Metis [17] is the first effort that stresses the importance of adapting MapReduce library according to workload characteristics. However, it replaces the hash table representation with a more complex B+ tree, and uses the single B+ tree structure to handle both the small and large key spaces [19]. Phoenix++ characterizes MapReduce workloads according to three key application dimensions and uses object-oriented modularity to provide different containers for different key distributions [19]. Though Phoenix++ is the closest related work to ours in this paper, Peacock adopts workload-customizable execution flow, while Phoenix++ only focuses on providing workload-tailored data structures and processes all classes of MapReduce workload through the same execution flow.

2.3 Motivating example

To the best of our knowledge, Phoenix++ is the best available implementation of MapReduce on shared-memory multicore platform. Now, we use *Distributed Sort* (DS) application [10] to illustrate that the program structure of Phoenix++ can introduce significant overheads for workloads that inherently have only one emitted value per key. DS sorts a set of records based on the Phoenix++ runtime, which is adapted from the Sort benchmark [14]. The map function extracts the key from each record, and emits a $\langle key, record \rangle$ pair, while the reduce function emits all pairs unchanged [10].

Figure 1 illustrates the execution flow of the Phoenix++ library in details. Step ① illustrates that a MapReduce workload starts a job by initializing a MapReduce

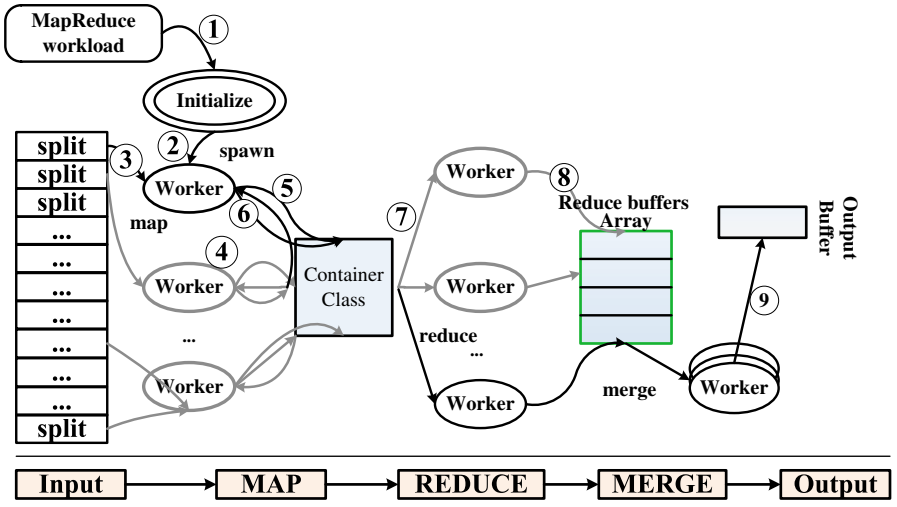


Fig. 1 The execution flow of the Phoenix++ library

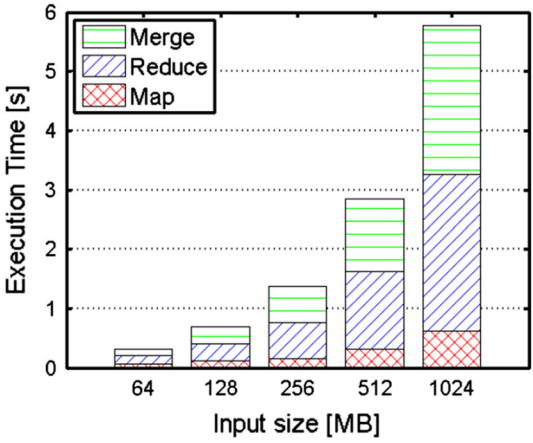


Fig. 2 Execution time breakdown of DS (16-thread) with different sizes of input datasets on a 16-core machine

instance, and then the runtime spawns multiple workers bound to different cores (or hyper-threads) in step ②. To start the Map phase, the input is partitioned into equally sized units, each of which is processed by a map task to be allocated dynamically to map workers as step ③ shows. In Phoenix++, containers are responsible for grouping emitted values with the same key and then store them in combiners. There are three built-in container implementations (hash container, array container, and common array container) tuned to three classes of workloads, respectively. Each map worker asks the container class for a container "instance" and returns its instance when finishing emitting key-value pairs into the instance shown in steps ④, ⑤, ⑥.

In order to maximally reduce the pressure of intermediate key-value storage on memory, Phoenix++ invokes the combiner once each key-value pair is emitted by the map function [19]. Furthermore, combiners will be run on all cross-thread emitted values before the combiner object is passed to reduce phase [19] (step ⑦). The additional function calls are eliminated by leveraging a C++ template technique known as the *Curiously Recurring Template Pattern* (CRTP) [8].

In the Reduce phase, each worker repeatedly selects a reduce task and generates the final result in reduce buffers array (step ⑧) followed by merged into a single output (step ⑨) in the merge phase.

Although Phoenix++ can efficiently handle workloads with large number of values per key, there are three key problems for workloads that inherently have one emitted value per key such as DS. The first problem lies in that invoking reduce function to aggregate values with the same key is unnecessary for them. Figure 2 shows the execution time breakdown of Distributed Sort with varying input. Our testbed will be described in Sect. 5. As shown in the figure, the unnecessary reduce phase takes up a significant portion of the whole execution time.

Phoenix++ runtime allows each map worker to control the width of its own hash table, and copy values out of the hash table at the end of the map phase followed by inserting them into a new fixed-width hash table. The overheads of resizing hash table and extra copy at the end of map phase are unnecessary for one emitted value per key workloads such as DS, because it has no need to aggregate values with the same key. We call these overheads as invalid overheads, which cause the second problem.

The third problem is that using combiners will introduce additional overheads of converting combiner objects to final values in the reduce phase for workloads that inherently have only one emitted value per key.

We attribute the performance issues of Phoenix++ in the context of processing workloads that inherently have only one emitted value per key to that the runtime provides the same execution flow for all classes of workloads. Can we provide a customizable execution flow for different classes of workloads? The first thing is to classify workloads, and then design an appropriate execution flow for each class of workloads.

3 Workload characterization model

In order to implement a workload-customizable MapReduce system, we first model the important workload characteristics with the goal to classify MapReduce workloads into different classes.

Phoenix++ characterizes MapReduce workloads along three important application dimensions: (1) *key distribution* namely "the relationship between the set of map tasks and intermediate keys" [19], (2) *number of emitted values per key*, and (3) *per task computation* namely "the ratio between the complexity of the user-provided map or reduce functions and the overhead of the MapReduce framework" [19].

Phoenix++ only uses the *key distribution* information to classify MapReduce workloads, and designs efficient key-value storage structure between the map and reduce phases for each class of MapReduce workloads. However, we think that it is necessary to take into account the information of both the key distribution and the number of emitted values per key, when classifying MapReduce workloads. In this way, we can adopt more aggressive optimizations for workloads that inherently have only one emitted value per key.

Table 1 shows our workload characterization model, which refines the workload characterization from Phoenix++ [19] by combining the first two dimensions (men-

Table 1 Workload characterization model

Classes of workloads	Characteristics
:(1:1)	"any map task can emit any key, where the number of keys is not known before execution" [19], and only one value is emitted per key (e.g., Distributed Sort, Tera-Sort and Self-Join)
:(1:n)	"any map task can emit any key, where the number of keys is not known before execution" [19], and the number of values emitted per key may be more than one (e.g., Word Count, Term-Vector and Sequence-Count)
$*:(1:n)^k$	"any map task can emit any fixed number of keys" [19], and the number of values emitted per key may be more than one (e.g., Histogram)
1:(1:1)	"each task outputs a single, unique key" [19], and the key must be integer or char type, and only one value is emitted per key (e.g. PCA and Matrix Multiply)

tioned above) into a single dimension. As shown in Table 1, our model classifies all MapReduce workloads into four different classes. In the following, we will answer two questions about the model: (1) does the model cover all classes of MapReduce workloads? (2) what are the implications on execution flow design?

3.1 Does the model cover all classes of MapReduce workloads

MapReduce workloads fall into three classes based on the map task to intermediate key distribution: (1) "**:** any map task can emit any key, where the number of keys is not known before execution" [19], (2) "**:k* any map task can emit any of a fixed number of keys, *k*" [19], and (3) "*1:1* each task outputs a single, unique key" [19]. In each class of workloads, the number of values emitted per key has two values: (1) exactly one, or (2) not known before execution, maybe more than one. For example, distributed sort does not have any duplicate keys and thus it has only one emitted value per key. For workloads such as word count, the number of values emitted by each key may grow with the size of the input data sets, and the exact values can not be known before execution.

Therefore, according to the definition of our model shown in Table 1, **:** key distribution workloads fall into **:(1:1)** or **:(1:n)** class. If a **:k* key distribution workload inherently has only one emitted value per key, it can be represented as a *1:(1:1)* workload. Otherwise, it is a $*:(1:n)^k$ workload. This is because keys in the **:k* key distribution workloads are fixed, and each key is emitted by an unique task when each key has only one value. For *1:1* key distribution workloads, each key-value pair only appears once in all map tasks, thereby they must have only one emitted value per key. We can conclude that *1:1* key distribution workloads fall into *1:(1:1)* class.

In summary, our workload characterization model covers all classes of MapReduce workloads.

3.2 What are the implications on execution flow design

According to Table 1, both **:(1:1)** and *1:(1:1)* workloads have only one emitted value with the same key, and thus aggregating emitted values with the same key is unnecessary for them. Therefore, we need to design a new execution flow for these workloads with the goal to minimize the unnecessary overheads.

We do not consider optimizing **:(1:n)** and $*:(1:n)^k$ workloads in this paper. The execution flow for these workloads can be the same with Phoenix++. However, it is necessary to guarantee that the performance of **:(1:1)** and *1:(1:1)* workloads can be improved without sacrificing the performance of **:(1:n)** and $*:(1:n)^k$ workloads.

4 Peacock design and implementation

In this section, we describe the design and implementation of Peacock system, including the new key-value storage structure and the flexible execution flow used in Peacock. Our design is motivated by the observations outlined in the previous sections. Our goals

Table 2 Containers for different classes of workloads

Classes of workloads	Containers
:(1:1)	Append container, and do not use combiners new in Peacock
:(1:n)	Hash container, and uses combiners to maintain a single aggregate value, identical to Phoenix++
$*:(1:n)^k$	Array container, and uses combiners to maintain a single aggregate value, identical to Phoenix++
1:(1:1)	Common array container, and do not use combiners, modified in Peacock

are to minimize the unnecessary overheads for workloads that inherently have only one emitted value per key.

4.1 Containers

In order to provide workload-customizable execution, we need to modify the containers in Phoenix++. Table 2 summarizes the containers provided for each class of workloads. The containers have significant impact on the corresponding execution flow design. As shown in Table 2, we design a new container for **:(1:1)** workloads called append container in Peacock. Because these workloads have only one emitted value with the same key, aggregating emitted values with the same key is unnecessary for them. Based on the observation, the append container does not use combiners to store emitted values, and simply stores the intermediate key-value pairs generated from map phase. The append container is an array of entries with length equal to the number of map workers. Each entry is an append-only buffer for a certain map worker. Each map worker has its independent output append-only buffer in order to avoid costs of locking and cache line contention. Appending each key-value pair to an append-only buffer can maintain *O(1)* insertion performance [17]. When all the map workers are finished, all the buffers are passed to the merge phase directly. In this way, the unnecessary procedure of converting combiner objects to final values can be eliminated, and the overhead of reduce phase can be also minimized.

We redesign the common array container in Phoenix++ by eliminating the using of combiner object to store emitted values. The common array container is a global structure for all map workers without any synchronization, due to the fact that each emitted key is unique in *1:(1:1)* workloads. At the end of map phase, the global array is divided equally to each merge worker to conduct merge sort, if sorting is enabled.

In addition, Peacock keeps the same design of hash container and array container with Phoenix++, which is designed for **:(1:n)** and $*:(1:n)^k$ workloads to store their intermediate key-value pairs, respectively.

Interface Peacock has been implemented on top of Phoenix++, and the system APIs directly exposed to the programmers are completely compatible with Phoenix++. Table 3 depicts the container interface functions in Peacock. Similar to developing applications with Phoenix++, the programmer specifies the container type in appli-

Table 3 Container interface functions

Function description
initialize(num_map_threads, num_reduce_tasks)
Initialize the container
container_type::reduce_buffer_type* get_rbs(int& length)
Return the pointer to reduce buffers array
container_type::input_type get_ins()
Create a thread-local container "instance"
container_type::add(container)
Return control of the instance to the container
combiner_iterator out(reduce task_id)
Output combiner objects for processing by a reduce task

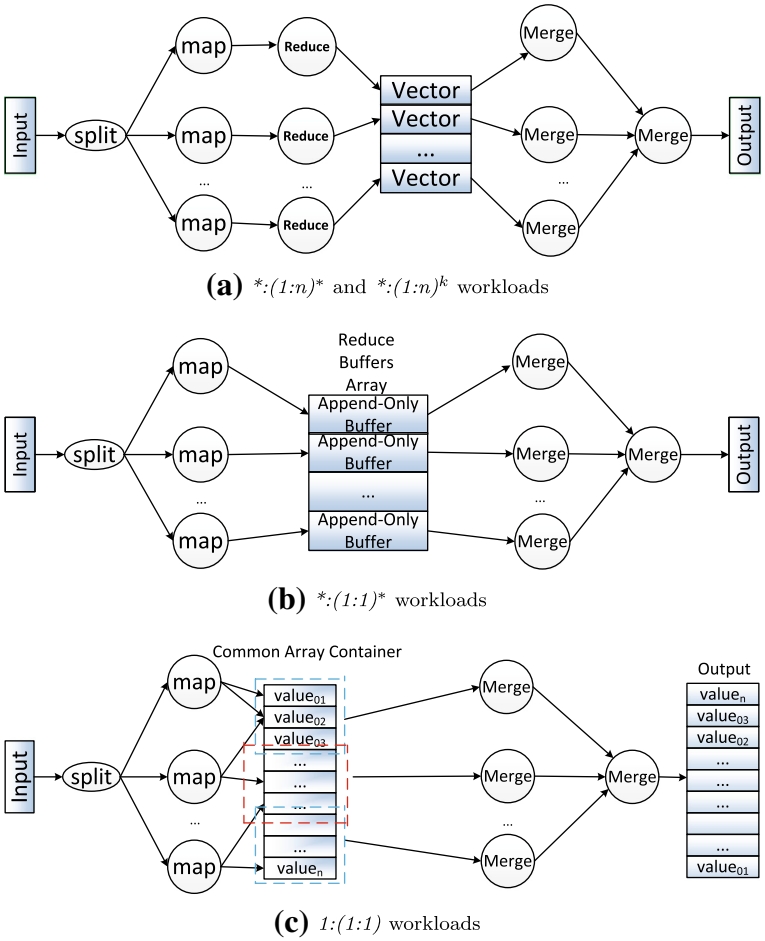
cation code and the container object is initialized by the runtime. The runtime asks the container object for the array of reduce buffers that will be passed to the merge phase through *get_rbs()* method. The *get_ins()* method is used to create a thread-local container instance for the invoked map worker, which is an append-only buffer, hash table, thread-local array and non-blocking array for **:(1:1)**, **:(1:n)**, **:(1:n)^k*, and *1:(1:1)* workloads, respectively. At the end of map phase, every map worker invokes the add function to return its instance to the container. The container will merge similar structures from all map workers for **:(1:n)** and **:(1:n)^k* workloads.

4.2 Integrate various execution flows into the unified MapReduce model

Peacock tries to provide flexible execution flow abstraction which permits workload-customizable implementations. As eliminating the reduce phase will change the abstracted MapReduce model, the main challenge of Peacock is how to integrate various execution flows into the unified MapReduce model. Peacock overcomes the challenge by modifying the internal data structures of Phoenix++ to provide unified buffer management for different execution flows, while maintaining the standard MapReduce API for programmers. In fact, eliminating the reduce phase is to make the reduce tasks do nothing.

According to [19], the Phoenix++ runtime system handles three fixed types of temporary buffers. The first is thread-local container instance for each map worker to do its own computation over dispatched splits. The second is the output combiner objects for processing by a reduce task. The third is the array of reduce buffers, each buffer for a certain reduce task to generate its output which will be sent to merge tasks. In contrast to Phoenix++, Peacock handles variable types of temporary buffers for different classes of workloads in order to achieve high performance. Each class of workloads has its specific data structure of the reduce buffers array encapsulated in the container class. The container object is responsible for initializing the data structure of the reduce buffers array. Figure 3 illustrates the execution flows supported by the Peacock runtime for different classes of MapReduce workloads when the final sort is enabled by users.

As shown in Figure 3a, the execution flow for **:(1:n)** and **:(1:n)^k* workloads is the same as Phoenix++. The corresponding data structure is an array of vectors with length equal to the number of reduce tasks. Each vector is a separate buffer for a certain



(a) **:(1:n)** and **:(1:n)^k* workloads

(b) **:(1:1)** workloads

(c) *1:(1:1)* workloads

Fig. 3 Flexible execution flow when the final sort is enable

reduce task to generate its output, and all the vectors will be sent to merge tasks after the reduce phase to merge a single output.

Figure 3b shows the execution flow for **:(1:1)** workloads. Peacock leverages the fact that there is only one emitted value with every distinct key in **:(1:1)** workloads. Every map worker appends its emitted key-value pairs into an append-only buffer to maintain *O(1)* insertion performance, and the buffers array also acts as the role of reduce buffers array passed to the merge phase directly, thereby minimizing the overheads introduced by the unnecessary reduce phase. In this way, **:(1:1)** workloads will also consume less memory than the implementations based on Phoenix++.

Figure 3c shows the execution flow for *1:(1:1)* workloads. Similar to append-only buffers array, the common array container is the reduce buffers array at the same time. Furthermore, it is also the final output buffer. At the end of map phase, the global array is divided equally to each merge worker to conduct merge sort, if sorting is enable.

When users disable the final sort, the runtime simply merges all the reduce buffers into a single final output buffer or returns to the application code as the case for *1:(1:1)* workloads.

4.3 Discussion

Normal programmers are not interested in becoming parallel programming experts. Thus, a good parallel programming model needs to provide appropriate abstractions for programmers to consider high-level codes while not worrying about low-level details. This can avoid concurrency bugs mainly made by inexperienced programmers and improve productivity at the same time. The design of parallel programming model should be driven by the goal of guaranteeing good application performance and programmer productivity. The next section will show that Peacock can achieve better performance than Phoenix++ for MapReduce workloads that inherently have only one emitted value per key while identical for other classes of workloads. Here, we first discuss Peacock in terms of productivity.

Similar to Phoenix++, Peacock needs programmers to specify the container type in their codes. Programmers should be aware of the knowledge in *the map task to intermediate key distribution*, and *the number of values emitted per key*, so that they can specify the most efficient implementations of containers for their programs. Compared to traditional MapReduce model, the programming style of Peacock and Phoenix++ does not put much more burden on programmers. This is because both the *key distribution* and *the number of values emitted per key* belong to the high-level knowledge that programmers must consider to achieve high application performance even when using sequential programming. After specifying the container type, programmers only need to write strict, simple MapReduce code to develop parallel applications with Peacock.

In summary, it is easy for normal programmers to develop parallel applications with Peacock. We plan to explore how to apply the workload-customizable optimizations automatically in our future work.

5 Evaluation

With the prototype of Peacock implementation, we test its performance on our testbed. In this section, we first introduce the testbed and the tested benchmarks used in our experiments, then present the experimental results for Peacock with different classes of workloads. We conduct several experiments to compare Peacock with Phoenix++, which is currently the best available implementation of MapReduce on single multi-core machine.

5.1 Testbed

Table 4 describes the hardware and software settings for our experiments. We run all experiments on the machine at least five times when it is idle, and report the average result. The memory allocator for both Peacock and Phoenix++ is Intel's TBB scalable_allocator [2].

Table 4 Experimental settings

CPU	2 Intel Xeon E5-2670 chips with 8 2.60 GHz cores per chip Total 16 cores
Cache	Per core data / instruction L1, 32 KB Per core L2, 256 KB Shared L3, 20 MB
Memory	64 GB
Interconnect	Point-to-point QuickPath interconnect
Operating system	Linux kernel 2.6.32
Compiler	GCC 4.4 with -O3 optimization

Table 5 Benchmarks

Applications	Description	Default input size
Distributed sort (DS)	Sort a set of records	1 GB
Word count (WC)	Compute frequency of each distinct word in a file	1 GB
Histogram (HIS)	Compute frequency of each RGB component in a set of images	1.4 GB
PCA	Principal components analysis on a matrix	1,000 × 1,000

5.2 Benchmarks

As the performance benefit of Peacock depends on workload characteristics, we choose four applications representing **:(1:1)**, **:(1:n)**, *1:(1:n)^k* and *1:(1:1)* workloads, respectively: DS, WC, HIS, and PCA. Table 5 lists the brief descriptions of four benchmarks we use and the default input size for each benchmark. Among these benchmarks, WC, HIS, and PCA are available in the Phoenix++ release [19]. We port them to work with Peacock by only modifying the container type in their codes. We use the (relatively large) data sets available on the Phoenix++ website [3]. DS is included in the Ostrich [7], and we port DS to work with Peacock and Phoenix++. To make the comparison fair, we specify the most appropriate data structure of Phoenix++ in the code of DS application. The input dataset for DS is generated by the *gensort* tool, which is available in [14].

5.3 Overall performance

Figure 4 compares Peacock with Phoenix++ in terms of execution time breakdown of DS with different sizes of input datasets. As shown in the figure, Peacock significantly reduces the execution time of DS by not only minimizing the overhead of the unnecessary reduce phase, but also using append-only buffer to improve the performance of the map phase. Table 6 specifies how much performance benefit comes from the

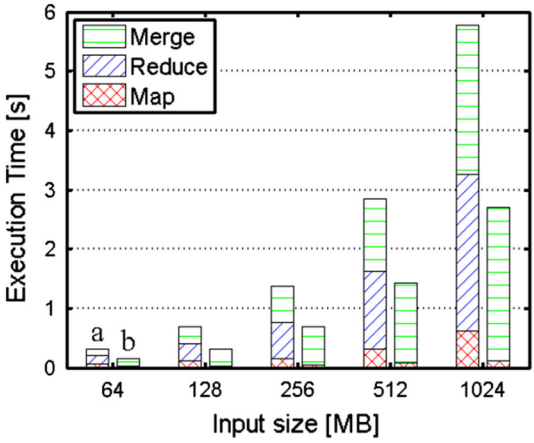


Fig. 4 Execution time breakdown of DS (16-thread) with different sizes of input datasets on a 16-core machine (**a** Phoenix++, **b** Peacock)

Table 6 The benefit breakdown of Peacock for DS (16-thread) with different sizes of input datasets on a 16-core machine

Input size (MB)	64	128	256	512	1,024
The benefit from the append-only buffer (%)	20	23	11	6	14
The benefit from eliminating the reduce phase (%)	80	77	89	94	86

append-only buffer and how much comes from eliminating the reduce phase. As a result, Peacock outperforms Phoenix++ by 2.19×, 2.25×, 1.99×, 1.98×, and 2.13× for the input size of 64, 128, 256, 512, 1,024 MB, respectively. We analyze the key reasons as follows.

- The Phoenix++ runtime allows each map worker to control the width of its own hash table, and copy values out of the hash table at the end of the map phase followed by inserting them into a new fixed-width hash table for workloads such as DS. However, the overheads of resizing hash table and extra copy at the end of map phase are unnecessary for DS. Peacock eliminates these overheads by making each map worker simply insert key-value pairs into its independent append-only buffer. Therefore, compared to Phoenix++, Peacock achieves better performance of the map phase for DS.
- When using Peacock to develop DS, programmers specifies the container type of append container in application code. The Peacock runtime will pass the append-only buffers to the merge phase directly, and thus eliminate the unnecessary reduce phase. As the unnecessary reduce phase takes up a significant portion of the whole execution time of Phoenix++ based DS, Peacock can successfully find enough performance optimization room from Phoenix++.

Figure 5 shows the effect of Peacock on four benchmarks with the default size of input datasets (relatively large). Among the four benchmarks, PCA uses two MapReduce iterations, and thus the figure shows that the first and second iteration as PCA/1 and PCA/2, respectively. Similar to DS, PCA/1 and PCA/2 have only one emitted value

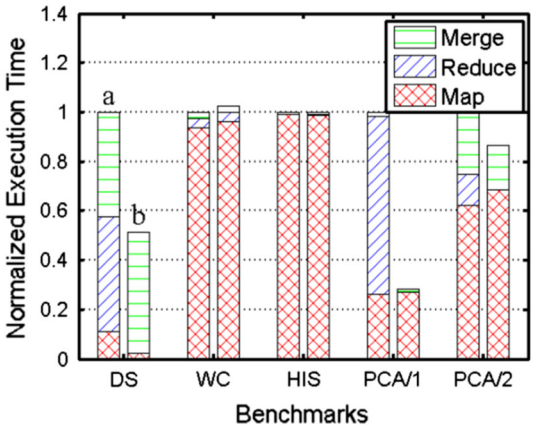


Fig. 5 Normalized execution time breakdown of four representative benchmarks on a 16-core machine (**a** Phoenix++, **b** Peacock)

per key. They do not have to spend time in the unnecessary reduce phase by using customizable execution flow provided by Peacock. As a result, compared to Phoenix++, DS, PCA/1 and PCA/2 enjoy 94 %, 253 %, and 16 % relative performance improvement, respectively. For WC and Histogram, Peacock almost keeps the performance of Phoenix++ due to the fact that the key-value storage structures and overall execution flows of these two classes of workloads are the same with Phoenix++.

5.4 Testing scalability

Figure 6 shows performance comparison between Peacock and Phoenix++ for DS with different number of threads (the left Y-axis indicates the total runtime, and the right Y-axis indicates the speedup normalized to Phoenix++). As shown in the figure, Peacock has better scalability than Phoenix++ for DS and gets up to 3.6× speedup over Phoenix++.

Figure 7 shows performance comparison between Peacock and Phoenix++ for PCA with different number of threads. We can observe that both Peacock and Phoenix++ do not scale well for the first MapReduce iteration of PCA. In spite of this, Peacock still gets up to 3.5× speedup over Phoenix++. For the second MapReduce iteration of PCA, Phoenix++ spends little time in the reduce phase when its number of threads is low. Therefore, Peacock does not find sufficient room for performance optimization. As the number of threads increases to 16, the improvement becomes apparent. The second MapReduce iteration of PCA takes up the most portion of the overall execution time, and Peacock gets up to 1.16× speedup over Phoenix++ for the overall PCA application.

Figure 8 shows performance comparison between Peacock and Phoenix++ for WC and HIS with different number of threads. WC and HIS are **:(1:n)** and *1:(1:n)*^{k} workloads, and thus they may have duplicate keys. As Peacock mainly addresses the performance issues of **:(1:1)** and *1:(1:1)* MapReduce workloads, it does not modify the key-value storage structures and execution flows of Phoenix++ for **:(1:n)** and *1:(1:n)*^{k} workloads. The figure indicates that Peacock does not add extra overhead for WC and HIS.

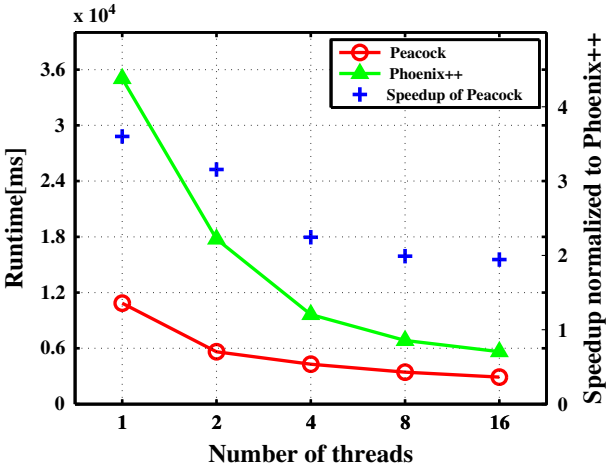


Fig. 6 Performance comparison between Peacock and Phoenix++ for DS with different number of threads

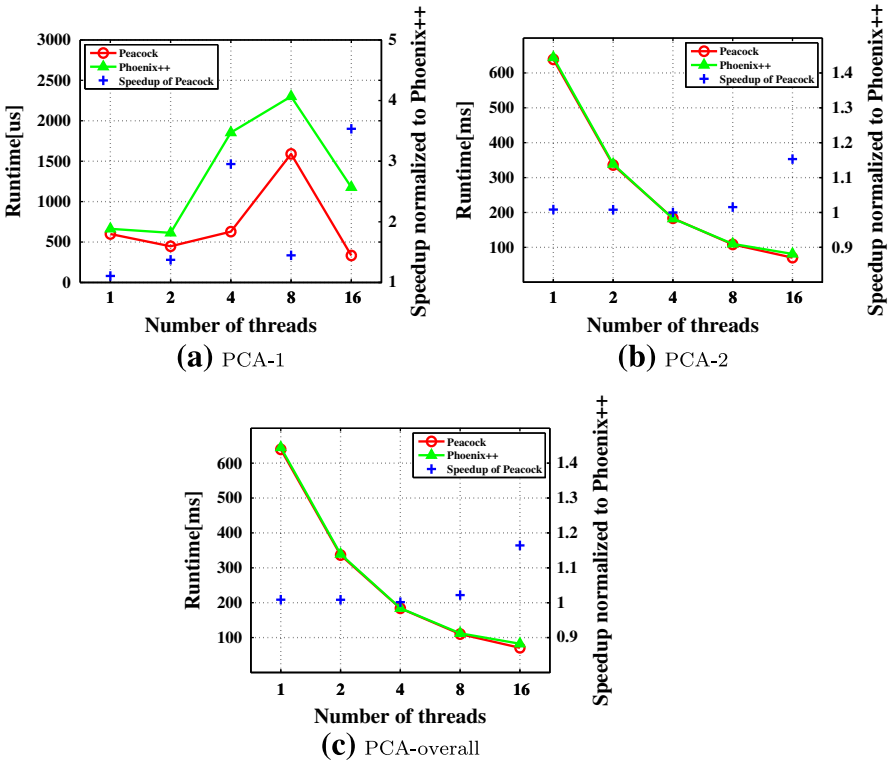


Fig. 7 Performance comparison between Peacock and Phoenix++ for PCA (including the first and second MapReduce iterations of PCA, and the overall application) with different number of threads

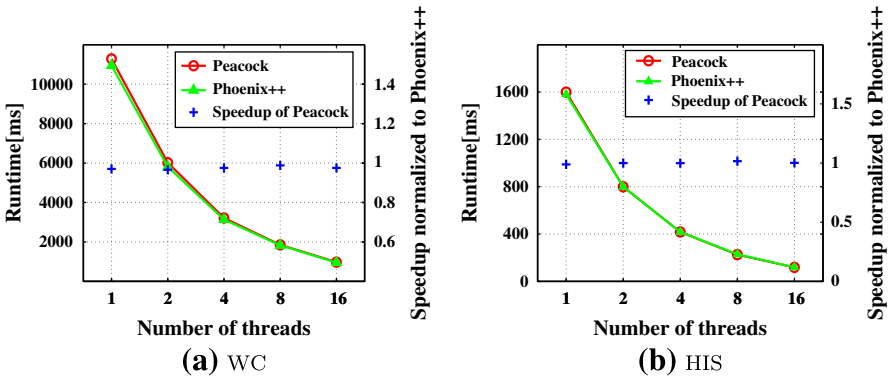


Fig. 8 Performance comparison between Peacock and Phoenix++ for WC and HIS with different number of threads

In summary, results on four representative benchmarks indeed validate that the advantage of Peacock is achieving better performance than Phoenix++ for MapReduce workloads that inherently have only one emitted value per key by applying aggressive optimization for them, while not sacrificing the performance for other classes of workloads. The overhead of supporting new execution flows is small.

6 Related work

Multicore machines are prevalent. Therefore, programmers must write parallel programs to take advantage of the abundant computing resources. However, writing parallel programs is not straightforward because programmers are more familiar with sequential programming. Moreover, parallel applications have non-deterministic behaviour. Concurrency bugs are easy to happen in a hand-crafted parallel application. It is difficult for normal programmers to deal with all this complexity. Currently, making parallel applications work well on single multicore machine is done in following ways:

(1) *Testing and debugging tools for parallel applications* Because parallel applications have non-deterministic behaviour, fixing concurrency bugs is time-consuming and error-prone. In order to make testing and debugging of parallel applications easier, several studies propose approaches to executing parallel applications deterministically [4,5,16]. However, these systems often guarantee the deterministic parallelism at the cost of decreasing the application performance. There are also some studies that detect concurrency bugs automatically [15,21,22]. The work in development of testing and debugging tools for parallel applications are orthogonal to Peacock, and they are still open problems today. However, when these testing and debugging techniques become mature enough, they can further enhance the productivity of Peacock for normal programmers.

(2) *High-level programming models for parallel applications* Inexperienced programmers are easy to write parallel programs with concurrency bugs. To help the normal programmers to write robust and efficient parallel programs, many researchers try to develop high-level programming models for single multicore

machine [9,11,12,17,18]. Many applications exhibit data-parallel pattern, and they can be easily implemented with MapReduce model [10]. MapReduce is initially implemented on clusters, and it has a popular open-source implementation called HADOOP [1]. The current version of HADOOP mainly focuses on clusters rather than single multicore machines.

Compared to the cluster version, MapReduce does not have bottlenecks in disk and network I/O on a single multicore. Ranger et al. [18] firstly propose an implementation of MapReduce for shared-memory systems namely Phoenix. They demonstrate that MapReduce is a promising alternative to take advantage of the additional compute power of multicore platforms. Yoo et al. [20] release an optimized version of Phoenix called Phoenix 2 in 2009, which allows the users to control the width of the hash table. This can alleviate buffer reallocation issues in the initial version to some extent, but it is difficult to determine an appropriate initial width of the hash table for normal programmers [19]. Mao et al. [17] conducts the first research work that stresses the importance of adapting MapReduce library according to workload characteristics. However, they replace the hash table representation with a more complex B+ tree, and use the single B+ tree structure to handle both the small and large key spaces [19]. Talbot et al. propose more efficient data structures tailed to different key distributions, and implement them in a modular MapReduce system called Phoenix++ [19]. As shown in this paper, there is still some optimization room left for workloads that inherently have only one emitted value per key in Phoenix++.

7 Conclusion

Parallel applications become as popular as multicore chips in order to take advantage of the abundant computing resources. MapReduce has been demonstrated to be a promising alternative to simplify parallel programming with high performance on single multicore machine.

This paper argues that the MapReduce program structure can introduce significant overheads for workloads with only one emitted value per key. Based on the observation, this paper proposes a refinement of the workload characterization of Phoenix++, and presents Peacock, which is a new MapReduce system with workload-customizable execution flow. The benefit of workload-customizable execution flow is the ability of applying aggressive optimization for different classes of workloads. As a result, our system can achieve better performance than Phoenix++ on MapReduce applications that inherently have only one emitted value per key. The overhead of supporting new execution flows is also negligible for other classes of workloads.

In our future work, we will explore how to apply the workload-customizable optimizations automatically. We will also study more characteristics based optimizations and apply them to our system.

Acknowledgments The research is supported by National Science Foundation of China under Grant No. 61232008, National 863 Hi-Tech Research and Development Program under Grant No. 2013AA01A213, Guangzhou Science and Technology Program under Grant 2012Y2-00040, Chinese Universities Scientific Fund under Grant No. 2013TS094, and Research Fund for the Doctoral Program of MOE under Grant No. 20110142130005.

References

1. The apache software foundation. Hadoop. http://hadoop.apache.org
2. Intel Corporation. Threading building blocks. http://www.threadingbuildingblocks.org
3. Stanford University. The Phoenix system for mapreduce programming. http://mapreduce.stanford.edu
4. Aviram A, Weng SC, Hu S, Ford B (2010) Efficient system-enforced deterministic parallelism. In: Proceedings of the 9th USENIX conference on operating systems design and implementation, OSDI'10USENIX Association, Berkeley, CA, USA, pp 1–16
5. Bergan T, Anderson O, Devietti J, Ceze L, Grossman D (2010) Coredet: a compiler and runtime system for deterministic multithreaded execution. In: Proceedings of the fifteenth edition of ASPLOS on architectural support for programming languages and operating systems, ASPLOS XVACM, New York, NY, USA, pp 53–64
6. Borkar S (2007) Thousand core chips: a technology perspective. In: Proceedings of the 44th annual design automation conference, DAC '07ACM, New York, NY, USA, pp 746–749
7. Chen R, Chen H, Zang B (2010) Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling. In: Proceedings of the 19th international conference on parallel architectures and compilation techniques, PACT '10ACM, New York, NY, USA, pp 523–534
8. Coplien JO (1995) Curiously recurring template patterns. C++ Rep 7(2):24–27
9. Dagum L, Menon R (1998) Openmp: an industry-standard api for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55
10. Dean J, Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
11. Feng M, Gupta R, Hu Y (2011) Spicec: scalable parallelism via implicit copying and explicit commit. SIGPLAN Not 46(8):69–80
12. He B, Fang W, Luo Q, Govindaraju NK, Wang T (2008) Mars: a mapreduce framework on graphics processors. In: Proceedings of the 17th international conference on parallel architectures and compilation techniques, PACT '08ACM, New York, NY, USA, pp 260–269
13. Jiang W, Ravi VT, Agrawal G (2010) A map-reduce system with an alternate api for multi-core environments. In: Proceedings of the 2010 10th IEEE/ACM international conference on cluster, cloud and grid computing, CCGRID '10IEEE Computer Society, Washington, DC, USA, pp 84–93
14. Jim G. Sort benchmark home page. http://sortbenchmark.org
15. Jin G, Zhang W, Deng D, Liblit B, Lu S (2012) Automated concurrency-bug fixing. In: Proceedings of the 10th USENIX conference on operating systems design and implementation, OSDI'12USENIX Association, Berkeley, CA, USA, pp 221–236
16. Liu T, Curtsinger C, Berger ED (2011) Dthreads: efficient deterministic multithreading. In: Proceedings of the twenty-third ACM symposium on operating systems principles, SOSP '11ACM, New York, NY, USA, pp 327–336
17. Mao Y, Morris R, Kaashoek MF (2010) Optimizing mapreduce for multicore architectures. Technical report, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology
18. Ranger C, Raghuraman R, Penmetsa A, Bradski G, Kozyrakis C (2007) Evaluating mapreduce for multi-core and multiprocessor systems. In: Proceedings of the 2007 IEEE 13th international symposium on high performance computer architecture, HPCA '07IEEE Computer Society, Washington, DC, USA, pp 13–24
19. Talbot J, Yoo RM, Kozyrakis C (2011) Phoenix++: modular mapreduce for shared-memory systems. In: Proceedings of the second international workshop on MapReduce and Its Applications, MapReduce '11ACM, New York, NY, USA, pp 9–16
20. Yoo RM, Romano A, Kozyrakis C (2009) Phoenix rebirth: scalable mapreduce on a large-scale shared-memory system. In: Proceedings of the 2009 IEEE international symposium on workload characterization (IISWC), IISWC '09IEEE Computer Society, Washington, DC, USA, pp 198–207
21. Yuan D, Zheng J, Park S, Zhou Y, Savage S (2012) Improving software diagnosability via log enhancement. ACM Trans Comput Syst 30(1):4:1–4:28
22. Zhang W, Lim J, Olichandran R, Scherpelz J, Jin G, Lu S, Reps T (2011) Conseq: detecting concurrency bugs through sequential errors. In: Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems, ASPLOS XVIACM, New York, NY, USA, pp 251–264