SMR: Scalable MapReduce for Multicore Systems

Abstract

Although the multicore chips have been widely used, utilizing multicore sources is still a challenging task due to the difficulties of parallel programming. Traditional parallel programming techniques require the programmer to manually manage many details, such as synchronization, load balance. By automatically managing concurrency, MapReduce, a simple and elegant programming model, alleviates the burden of programmer. Phoenix, a MapReduce library for multicore and multiprocess, demonstrates that applications written with MapReduce framwork can get competitive scalability and performance in comparison to those written with Pthreads. However, evaluation results show that Phoenix scales worse on a 32-core system.

This paper focuses on improving Phoenix in terms of scalability and performance. First, we analyze some critical factors that limit the Phoenix runtime, such as contending for a shared address space per process and existing of the barrier between Map and Reduce phase. Then, we propose a novel multithreaded model, ***Sthread***, which avoids the contention through providing the isolated address spaces between threads. Based on ***Sthread***, we design a scalable mapreduce, ***SMR***, which breaks the barrier and improves the performance by adopting a new producer-consumer model. Finally, the experiments show that ***SMR*** can achieve better scalability and performance than Phoenix for histgram, word_count and pca. Specially, performance improvements range from 9.0X to 26.7X when the number of cores is 32.

1. Introduction

With the prevalence of multicore chips, it is foreseeable that tens to hundreds (even thousands) of cores on a single chip will appear in the near future[2]. However, utilizing multicore sources is still challenging because of the difficulties of parallel programming. Specifically, traditional parallel programming techniques require the programmer to manually manage synchronization, load balancing and locality, and explicitly understand the detail of underlying hardware, which is error-prone and complicated. An alternative approach is dependent on a runtime system for concurrency management.

MapReduce[8] is a promising programming model for clusters to perform large scaled datasets processing in a simple and efficient way. In most cases, programmers only need to implement two functions: map function which processes the input data and produce a series of key-value pairs, and reduce function which is used to aggregate values with the same key. While initially MapReduce is implemented on clusters, Ranger et al. have implemented a MapReduce library for multicore and multiprocess system–Phoenix[15], which demonstrates the feasibility of running MapReduce applications on shared memory multicore machines. After that, Phoenix is heavily optimized by Yoo et al[18] to obtain better scalability on shared-memory system (*i.e.,* Phoenix2). Other libraries such as Metis[14] , Tiled-mapreduce[4] and MRPhi[13] attempt to improve performance of Phoenix from various aspects.

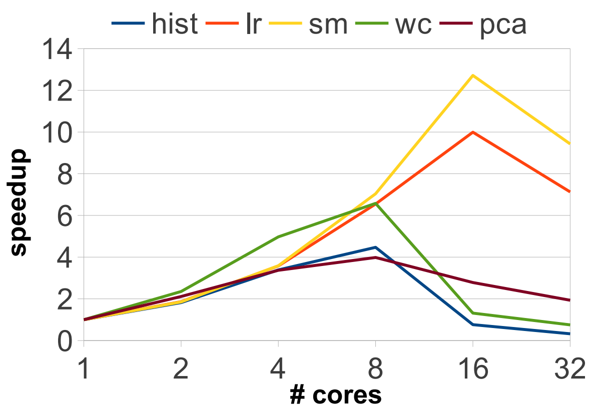


Figure 1. speedup of Phoenix

Phoenix exploits shared-memory threads (i.e., Pthread library) to implement parallelism, and the runtime binds each worker to a thread. Ideally, adding more threads and cores to the system will bring a linear decreasing in execution time. However, we note that the optimized Phoenix [18] (i.e., Phoenix2, which is named Phoenix in this paper) scales worse on a 32-core Intel 4 Xeon E7-4820 system running Ubuntu 12.04 with kernel 3.2.14. Figure 1 shows the speedup of the optimized Phoenix runtime. We observe that despite the Phoenix runtime is able to process these applications in parallel, none of them scales well beyond 16 cores and most of them actually degrade when the number of cores exceeds 8.

Since the Phoenix runtime implements multithreading based on shared-memory Pthreads, all threads of the runtime have to share a single address space, which will lead to contention on the single lock [5]. As a result, the scalability of applications with Phoenix will be limited. To remedy the problems mentioned above, we design a modified MapReduce framework with preserving the programming interfaces of previous MapReduce. Firstly, we propose a scalable thread libray (***Sthread***). Then based on ***Sthread***, this paper presents a scalable MapReduce model for multicore, ***SMR***, which can efficiently scale on multicore system.

The main contributions of this paper are concluded as follows:

- We analyze the important roadblocks that limit scalability of the Phoenix runtime on shared-memory systems. Specifically, we find that the shared address space for multiple threads will be a crucial issue at multicore system.
- We develop a scalable thread library, ***Sthread***, in which threads run in the separated memory space to avoid the contention on the shared space. Furthermore, ***Sthread*** provides the unboundary-channel for the threads to communicate with each others.
- Base on ***Sthread***, we propose a scalable MapReduce model, ***SMR***. And ***SMR*** pipelines the Map and Reduce phase by adapting a new producer-consumer model, in which producer does not have to wait when the buffer is full.
- We implement a prototype of ***Sthread*** and demonstrate the effectiveness of ***SMR*** runtime on a 32-cores system.

In order to ground our discussion, we present an overview of MapReduce framework and Phoenix in Section 2. We then develop the design of ***Sthread*** in Section 4, focusing on implementing mechanism of extension in ***Sthread***. In Section 5, we describe our support for pipelineing map and reduce, and illustrate the potential benefits of the producer-consumer model for MapReduce framework. We give initial performance results in Section 6. Related and future work are covered in Sections 7 and 8.

2. Background

In this section, we will review the MapReduce programming model and detail the salient features of Phoenix, an implementation of MapReduce for multicore.

2.1 MapReduce Programing Model

Inspired by funnctional languages, the MapReduce programming model is proposed for data intensive computation in cluster environment. Its simple programming interface requires programmer to define only two primitives: map and reduce. The map function is applied on the input data and produces a set of intermediate key-value pairs. The reduce function is applied on all intermediate pairs and groups them with the same key to a single key-value pair. In order to save networking bandwidth and reduce memory consumption, the combine function, an optional operation, can aggregate the key-value pairs locally in Map phase.

The charm of MapReduce is that, for applications that can adapt, it hides all the concurrency details from programmer. For example, one can count the number of occurrences for each word in a text file. The map function emits a $\langle word, 1\rangle$ pair for each word in document, and the reduce function counts all occurrences of a word as the output. The combine function is similar to the reduce function, but only processes a partial set of key-value pairs in Map phase.

2.2 Phoenix

Phoenix is an implementation of MapReduce for multicore and multiprocessor systems using Pthreads. It shows MapReduce model is a promising model, and the applications written with MapReduce have competitive scalability and performance in comparison to those written with Pthreads[15]; Phoenix stores the intermediate key-value pairs produced by the map workers in a global matrix (Figure 2). Each map and reduce workers can write or read this global matrix. When map and reduce workers operate the matrix concurrently, two strategies must be carried out to avoid lock contention costs.

- Each row of the matrix is exclusively used by a map worker, and each column of the matrix is exclusively used by a reduce worker.
- There is a barrier between the Map and Reduce phase. Only when all map workers have been completed do the reduce workers begin to compute.

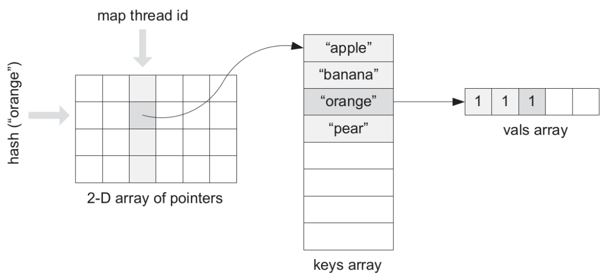


Figure 2. Phoenix intermediate struct

By adopting above strategies, applications programming with Phoenix can effectively reduce lock contention costs. However, these strategies limit the performance of Phoenix, which will be described in next Section.

3. Analysis of Phoenix

In this section, we first evaluate Phoenix built with a scalable memory allocator. Then we will analyze the important roadblocks that limit scalability and performance of the Phoenix runtime on shared-memory systems.

3.1 Scalability

Since there are many heap objects shared among threads in Phoenix, it is sensitive to memory allocator[18]. The memory allocator in glibc (i.e., ptmalloc[11]) does not scale on multicore system, while jemalloc[10] can provide improved performance and scalability. Therefore, we evaluate the scalability of Phoenix built with jemalloc, denoted as Phoenix-jemalloc.

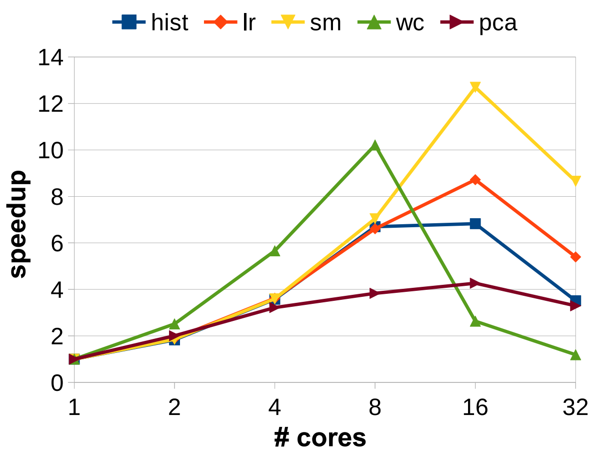


Figure 3. speedup of Phoenix-jemalloc

Figure 3 illustrates the speedup of Phoenix built with jemalloc. We observe that applications scale well when the core count increases from 1 to 8. However, when the core count increases from 8 to 16 to 32, the speedup sharply degrades for wc and from 16 to 32 cores for the other applications. The results indicate that in spite of utilizing the scalable memory allocator, Phoenix can not scale up to 16 cores.

In order to analyze the limited scalability behavior, Linux perf is exploited to collect execution time information of hot function. We note that the map function is the hottest function with less cores, while __ticket_spin_lock will become the hottest function with more cores. __ticket_spin_lock is a type of spinlock which is caused by the contention on the shared structure in Linux kernel. In order to specially explore the impact of spinlock in Phoenix, we collect the execution time percent of __ticket_spin_lock on each benchmark from 1 to 32 cores by Linux perf.

As the result shows in Figure4, the cost of spinlock increases quickly as the cores number cross a specific value (i.e., 8). Specially, for wc 16 and 32 cores, __ticket_spin_lock (*i.e.,* spinlock) is the function that has largest execution time percents of 24.04% and 39.91%, respectively. Experiment results demonstrate that Phoenix suffers from serious lock

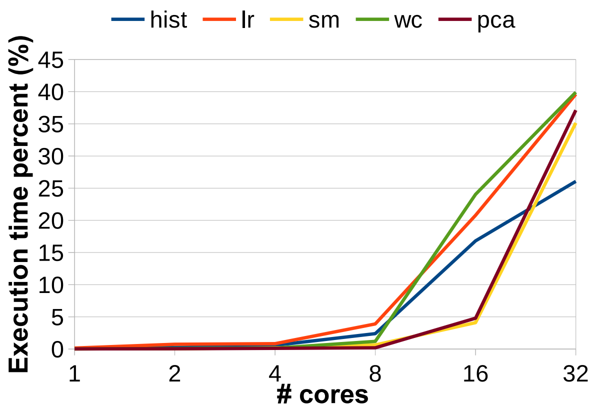


Figure 4. __ticket_spin_lock percent of applications in Phoenix

contention when the cores number exceeds 8. That means most of execution time will be used for waiting but not actual computation.

3.2 Contention of the single address space

Phoenix utilizes shared-memory multiple threads to implement parallelism, where all threads of an application share a single address space. In most widely used operating systems, such as Linux, an address space consists principally of a set of memory mapping regions and a red-back tree is used to store these regions. The red-back enables the operating system to find a particular region quickly when a process has have thousands of memory regions [9].

Linux provide three address space operations to update and read the memory region: mmap, munmap and page fault. The mmap operation creates memory mapping regions and adds them to the region red-back tree, while the munmap removes the memory regions from the tree. Page faults look up the faulting virtual address in the red-back tree to check whether the virtual address is mapped. To ensure correct behavior when several threads perform mmap and page faults concurrently, Linux use a single read/write semaphore per process (mmap_sem) to serialize mmap and page faults. Therefore, threads need to acquire the semaphore in write mode before executing the mmap operation, while permitting page faults and the other mmap operations. And page faults will acquire the semaphore in read mode and can proceed with each other in parallel.

A process can perform only one mmap or munmap at a time, and these operations will delay page faults. In addition, page faults for different virtual addresses run concurrently, but will block other threads from performing mmap or munmap operations. As a consequence, the performance of multithread applications will be limited by the serialization. The contention is intense when there are large amounts of threads, which will lead to the parallel scalability degradation on the benchmarks.

In Phoenix, benchmarks utilize the mmap() system call to read in input data. Once the user passes the pointer of the mmap() region to runtime as an argument, multiple map threads will concurrently cause page faults in the input data when they invoke map functions. Moreover, there are many mmap operations in memory allocator for dynamic memory management in Phoenix, which will lead to contend for the shared address spaces. As indicated in Figure 3, Phoenix can not scale as well as expected, when the cores count exceed a value, adding more cores might scale negatively due to the increased time spent in spinlock. Experiment call-graph information demonstrates that spinlock is caused by page faults and mmap operations.

3.3 Optimizing opportunities of Phoenix

In cluster environment, network bandwidth is the key factor for performance since the map and reduce workers, executed in different machines usually, communicate by the network. However, when processing MapReduce applications in multicore environment, the data structures shared by multiple threads, instead of the network, are the major performance bottlenecks. There is a single lock for per shared address space inside the operating systems virtual memory system, which we have discussed in the previous section. As a result, multithreaded applications on many-core processors will naturally suffer from serious contended locks. This phenomenon will be common for parallel programming with shared-memory multithreading [6].

In addition, as described in Section 2, there is a strict barrier between the Map and Reduce phase, requiring that the workers in Reduce phase can be started only when all workers in Map phase has been finished. In spite of avoiding lock contention, the barrier becomes an important roadblocks to performance of Phoenix. On the one hand, the barrier limits parallel computing. The execution time of the Map phase is subjected to the slowest map worker, which means that if one of the map workers is slow, then the runtime will need more time. On the other hand, it is worth mention that the user-defined map functions are usually computation-intensive while the Reduce phase is memory-intensive. Thus, the serialization of the Map and Reduce phase is bad for the utilization of system hardware resource.

To aviod the aforementioned issues, a novel MapReduce model, ***SMR***, is proposed in this paper, which exploits the producer-consumer model to break barrier and adopts a new thread program model to reduce lock contending. The implementation details of ***SMR*** are presented in Section 4 and Section 5, respectively.

4. A Novel Thread Model

In this Section, we present a novel thread model, ***Sthread***, which can support ***SMR*** to achieve good scalability. And then, the detail API of ***Sthread*** will be given. we focus on the design and implementation of an unboundary channel in ***Sthread***.

4.1 Scalable thread library

Our goal is to enable page faults to run concurrently with mmap and munmap operations and consequently to eliminate contention on the per-process read/write lock. There are some ways to achieve this target. For example, Corey[3] is a scalable operating system for multicore; Clements et.al. proposed a scalable address spaces by using RCU balanced trees [5]; Andi et.al think process has a better scalability than thread since the process needs not to share the address space with the other processes[1]. However, modifying operation system is impracticable and employing the process rather than the thread will make sharing become complicated. In order to provide a practicable and simple solution, a key problem to be solved is that how to enable threads to eliminate contention on the per-process read/write semaphore when multiple threads concurrently run page faults and mmap operations.

To address the problem and achieve our goal, we propose a novel thread programming model ***Sthread*** with better scalability, supporting scalable MapReduce compatibly. There are two key points in the design of ***Sthread***. Firstly, we confine the threads in ***Sthread*** to run in separate memory spaces to avoid the contention. Therefore, threads in ***Sthread*** have their local mmap_sem and eliminate the contention of the single semaphore with others thread. Secondly, when using the separate memory spaces, communication will be challenging since the threads in ***Sthread*** can not directly communicate with the other threads like thread based on share space. So, we give a *unboundary-channel* for the threads to communicate with the others in ***Sthread***.

int thread_alloc(int gid)
  Allocate a child thread of global ID and return its internal ID.
*int thread_start(int child, void *(*fn)(void*), void *args)*
  Start the given child to run *(*fn)(args)*.
int chan_alloc()
  Allocate a channel and return its ID.
int chan_setprod(int chan, int child, bool ascons)
  Transfer the send-port of the channel to the given child.
int chan_setcons(int chan, int child)
  Assign a receive-port of the channel to the given child.
*size_t chan_send/chan_sendLast(int chan, void *buf, size_t sz)*
  Send a message stored in *buf* of *sz* bytes via the channel.
*size_t chan_recv(int chan, void *buf)*
  Receive a message from the channel and save it in *buf*.

Figure 5. Main functions of ***Sthread*** thread API

Figure 5 lists the main functions of managing threads and channels in ***Sthread***. In the case of ***SMR***, at initial stage, the master thread invokes thread_alloc to allocate map threads and reduce threads, and then creates the *unboundary-channel* between each pair of map and reduce threads by invoking chan_alloc. After that, the master invokes chan_setprod and chan_setcons to set the map and reduce threads as producers and consumers for the shared-channel, respectively.

The producer sends messages to the *unboundary-channel*, and the consumer receives the messages from it, which is a typical producer-consumer model (we will detail how it will be used in Section 4). Finally, the master starts all threads to work by invoking thread_start.

Although ***Sthread*** can decrease the overhead of contention, it also takes extra overhead in comparison to Phoneix. The experiments demonstrate that the extra overhead is concentrate in the initial stage (we will analyze this overhead in Section 5.3).

4.2 Unboundary Channel

In our design, the *unboundary-channel* is a virtual memory area, called *CHAN* in the producer and consumer address space. When the producer invokes *chan_send* to send data, the sent data will be copyed to the *CHAN* area. And then the consumer reads the data from the *CHAN* area by invoking *chan_recv*. There is a pagetable (*ptab*) used to store the mapping between *CHAN* memory area and physical address, with each mapping as a corresponding page table entry.

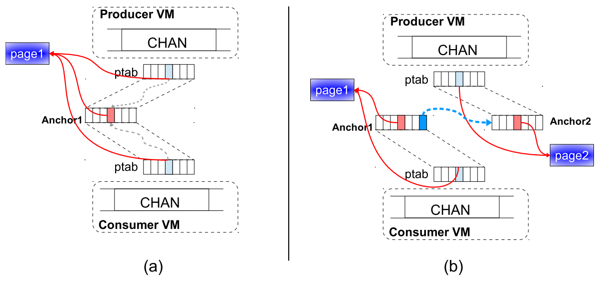


Figure 6. channel extend machanism

Initially, as shown in Figure 6(a), the runtime will not allocate the actual page frames for *CHAN* area but map it to a special page frame— anchor page table (*Anchor1*) which is shared by the producer and consumer. Meanwhile, each entry in the *ptab* of both the producer and consumer will point to the same entry in *Anchor1*.

When the producer sends data, the runtime attempts to write the sent data into the *CHAN* area, which will trigger a pagefault. Then the pagefault handler will allocate a page frame (*page1*) for the producer, and the corresponding *pte* in *Anchor1* will be updated to point to the allocated page (*page1*). After that, the consumer can locate the page frame *page1* and then read data from *page1* by by the anchor page (*Anchor1*).

In general producer-consumer model, if the channel buffer is full, the producer needs to wait until the consumer removes the data from it, which limits the performance and throughput of the system. To avoid the producer waiting, we design a channel buffer with unbounded size by exploiting an extend mechanism[19]. This mechanism allows the producer uninterruptedly sending data with no need for waiting. This extend mechanism (Figure 6(b)) will remap the channel buffer (i.e., *CHAN* area) to another anchor and allocate some new page frames for the producer. At the same time, the consumer will not be disturbed and it can continuously read the data from the original page frames. To record and trace the original and new page frames, an extension pte is introduced (*extension_pte*). When the consumer receives all data from the original page frames, it will locate the new page frames by the *extension_pte*. The original page frames will decrease their reference counts and are automatically freed when the counts reach zero.

In conclusion, we design a scalable thread model (***Sthread***) and provide an unbounded *unboundary-channel* for threads to communicate. This unbounded *unboundary-channel* is a prominent feature of ***Sthread***, which is the main difference between our ***Sthread*** and the previous works. We will detail how to sufficiently exploit this feature to implement the scalable MapReduce (***SMR***) in Section 4.

5. Implementation and Runtime

In this section we discuss our extensions for the MapReduce programming model and show the major changes to runtime of ***SMR***. Then we describe our design how to support pipelining of Map and Reduce phase (Section 4.2.1). The implementation of intermediate buffer between Map and Reduce phase will be given in Section 4.2.2. Finally, we defer performance results to Section 5.

5.1 Execution flow

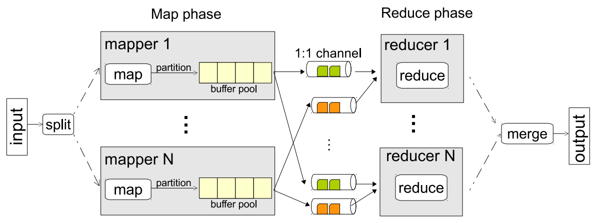


Figure 7. The workflow of ***SMR***

The implementation of MapReduce in ***SMR*** is similar to that of in Phoenix. There is a single master worker managing a number of slave workers. Unlike Phoenix, a worker in ***SMR*** is handled by a ***Sthread*** thread rather than a Pthread thead. Figure 7 illustrates the workflow of ***SMR***, including three main phases: map, reduce and merge.

At the beginning, the input data is divided into some tasks by a split function, and then these tasks will be pushed into a task queue. Workers in Map phase will get these tasks and invoke the map function to produce the intermediate key-value pairs. Then they will be inserted into a local buffer of map worker. If this local buffer is full, the map worker will send the intermediate data of the buffer to a one-to-one channel. After that, the reduce worker can receive this data from the channel and invoke the reduce function to aggregate them with the same key. Finally, the results from multiple reduce workers are merged and output.

Compared with existing work, ***SMR*** takes two main strategies to improve its scalability and performance. On the one hand, a worker in ***SMR*** is implemented as a ***Sthread*** thread instead of a shared-memory thread. Therefore, threads in the isolated address space can avoid contending on a single per-process lock, which has introduced in Section 3.2. On the other hand, the producer-consumer model in Section 3.2 is used to pipeline Map and Reduce phase, i.e., the map worker as a producer will send key-value pairs to the channel and the reduce worker as a consumer will receive the key-value pairs from the channel. Once the reduce workers receive these key-value pairs, it will start working with no need for waiting the Map phase finished.

Combiner. There is a optional *Combiner* operation, which is a local aggregation in Map phase, can maximally reduce memory pressure caused by the intermediate key-value storage. In addition, the *Combiner* operation can reduce the communication traffic for the channel between map and reducer workers in ***SMR***. Furthermore, ***SMR*** is able to support the *Combiner* operation in Reduce phase to cope with the pressure of data skew.

Reduce. Each reduce worker generates a set of output key-value pairs. The library's Merge phase will sort these pairs according to the key, and then produce the final output. If an application no need to do reduce work, i.e., the reduce function is NULL, Phoenix still performs reduction on the intermediate data by using a default reduce function which will traverse key-value pairs. This procedure is inefficient. Unlike Phoenix, for applications without Reduce function, ***SMR*** does not start the Reduce phase but directly starts the Merge after the Map phase.

5.2 Pipelined execution

Pipelined map and reduce have been adopted in the MapReduce framework for distributed computing[7]. Condie et al. show since pipelining delivers data downstream operators more promptly, it can increase opportunities for parallelism, improving utilization and reducing response time. And results demonstrate that pipeline can reduce job completion times. Then, MRPhi[13], a MapReduce framework optimized for the Intel Xeon Phi coprocessor, pipelines the map and reduce phases to better utilize the hardware resource.

MRPhi adopts a typical producer-consumer model for pipelining, in which there is a many-to-one queue for communication between the map and reduce worker. Multiple map workers will insert data into the queue concurrently, and then the single reduce worker removes the data from it. On the one hand, as the number of concurrent map workers increases, the synchronization overheads, such as contention and waiting times, rise sharply and severely impair application performance. On the other hand, the queue is classically implemented as a fixed-size buffer or as a variable-size buffer with the help of dynamic memory allocation. If the former is adopted, operations on the queue need be synchronized to make sure that the map worker won't add data into a full buffer and the reduce worker won't try to remove data from an empty queue. If choosing the latter, expensive malloc and free operations are required to manage the queue, which will cause overhead.

We think a good producer-consumer model need to achieve two targets: (1) map worker can continue to working when the buffer is full. (2) there is no need for too much overhead on managing dynamic memory allocation. Based on ***Sthread***, an efficient producer-consumer model for ***SMR*** is designed in this subsection. It not only pipelines Map and Reduce phase, but also break through the limitation of issues mentioned above.

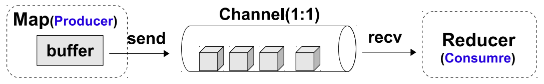


Figure 8. Produce-Consume model in SMR

As depicted in Figure 8, in our producer-consumer model, there are two major data structures: a local *buffer* for each map worker is designed to store the intermediate key-value pairs; a one-to-one *shared-channel* in ***Sthread*** is used to communicate between map and reduce workers. The one-to-one *shared-channel* means that the map worker sends the data to the reduce worker without contenting with other map workers. Furthermore, as depicted in Section 3.3, the *shared-channel* is a no-boundary channel. So, when the *shared-channel* buffer is full, the map worker can continuously work with no need for waiting the reduce worker to remove the data from the *shared-channel*. Moreover, since the *shared-channel* is implementation by mapping, it can save overhead caused by dynamic memory allocation.

In fact, each map worker has a local buffer pool, in which each buffer stores key-value pairs sent to the corresponded reduce worker. When the map worker generates a key-value pair, the partition function is invoked to index the corresponded buffer. After the buffer is determined, the map worker will insert the key-value pair into this buffer. In default, the buffer is implemented by a hash table which is similar to Phoenix's buffer. We also provide an array buffer implementation for ***SMR*** and we will detail the advantage of array buffer in next subsection.

5.3 Buffer Design and Optimize

Mites [14] shows that the organization of intermediate data is critical to the performance of many MapReduce applications. In cluster, it is the network bandwidth dominates the performance, but on multicore system the performance is dominated by the operations on the data structure that holds intermediate data. By default, the buffer in ***SMR*** is a hash table (Figure 9(a)), in which each entry is a pointer, pointing to a key array sorted by key. If the hash table has enough entries, collisions will be rare and the key arrays will be short, so that lookup and inserting will cost O(1), which is an attractive quality for workloads.

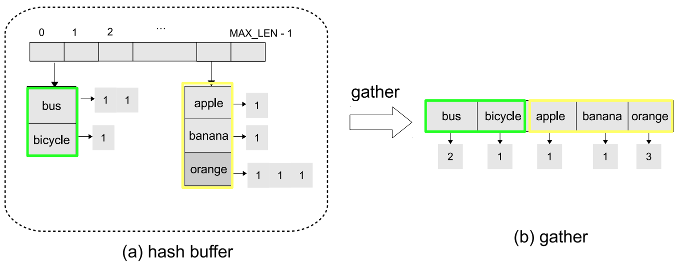


Figure 9. hash buffer and gather

Since the channel is implemented by mapping (in Section 3.3), it requires that data sent to the channel should be a contiguous block of memory. However, the key arrays in the hash buffer is scattered, which implies that all of the key arrays should be gathered together before sending them to the channel. This issue can be solved by copying these scattered key arrays out from the hash buffer and then inserting them into a new contiguous memory region (Figure 9(b)). This extra copy is unfortunate and time-consuming. Furthermore, the hash buffer also requires frequent allocations and deallocations of memory for key and value, and simultaneously couples with the data structure creation and destruction, which will negatively affect performance.

To avoid the time-consuming gather in hash buffer, we implement an easy-to-use array buffer, and the map worker could store its output by appending key-value pairs to array buffer. The array buffer is initially sized to a default value. After sending buffer's data to the channel, the map worker will indicate the buffer as empty, but will not free the memory until all map jobs have been finished. Therefore, memory of array buffer can be reused in the whole of Map phase. This manner avoids the expensive costs of memory allocation and deallocation as well as the data structures construction and destruction. Above all, unlike hash buffer, the array buffer is a contiguous memory block, it not have to gather key-value pairs before sending. The experiment results show that the array buffer is more effective than the hash buffer for some applications that likely have abundant key-value pairs, such as word_count (Section 5).

6. Evaluation

We evaluate ***SMR*** and Phoenix on a 32-core Intel 4 Xeon E7-4820 system equipped with 128GB of RAM. The operating system is Ubuntu 12.04 with kernel 3.2.0 and glibc-2.15. Benchmarks were built as 64-bit executables with gcc -O3. We logically disable CPU cores using Linuxs CPU hotplug mechanism, which allows to disable or enable individual CPU cores by writing 0 (or 1) to a special pseudo file (/sys/devices/system/cpu/cpuN/online), and the total number of threads was matched to the number of CPU cores enabled. Each workload is executed 10 times. To reduce the effect of outliers, the lowest and the highest runtimes for each workload are discarded, and thus each result is the average of the remaining 8 runs.

Five MapReduce applications is used in the evaluation, including histogram (hist), linear_regression (lr), string_match (sm), wordcount (wc), and pca (pca). we report results using the large data sets available, which are summarized in Table **??**.

6.1 Performance and Scalability

Since there are many heap objects shared among threads in Phoenix, it is sensitive to memory allocator[18]. The memory allocator in glibc (i.e. ptmalloc[11]) does not scale on multicore system, while jemalloc[10] can provide improved performance and scalability. Therefore, in this section, we first evaluate the scalability of Phoenix built with jemalloc, denoted as Phoenix-jemalloc. Then we measure the performance and scalability of ***SMR*** compared with Phoenix-jemalloc.

6.1.1 Performance of SMR

Based on ***Sthread***, we implemented the optimization of Phoenix in Section 5 and measured ***SMR*** performance by executing each benchmark. We compare the execution time of ***SMR*** with Phoenix-ptmalloc and Phoenix-jemalloc, respectively.

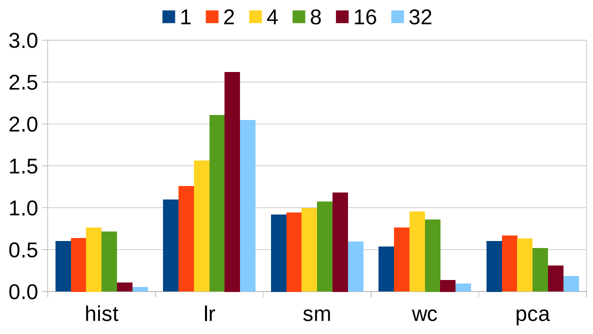


Figure 10. SMR versus Phoenix with ptmalloc

Figure 10 shows the performance comparison between ***SMR*** and Phoenix2. For hist, wc and pca, the optimized runtime of ***SMR*** leads to improvement across all core counts. The average improvement was 2.0x for less then 8 cores. For more then 8 cores, specially with 16 and 32 cores, the improvement were obvious, reaching 10.x in maximum and 5.0x on average. For sm, it has performed as well as Phoenix when the cores number less then 16, while it surpasses Phoenix when there is more cores. The only workload that ***SMR*** runs worse than Phoenix is lr. Figure 11 compares the performance of ***SMR*** against Phoenix built with jemalloc.

It can be saw that Phoenix-jemalloc has a better performance, while the similar tendency like Figure 10.

The results indicate, for workloads, such as hist, wc and pca, ***SMR*** performs significantly better than Phoenix, reduc-

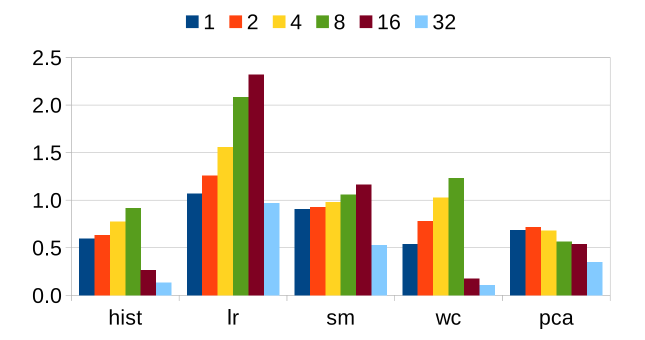


Figure 11. SMR versus Phoenix with jemalloc

ing the average running time to 30%. It owns to reducing the overhead caused by contending on the shared address space (Section 4) and pipelining the Map and Reduce phase (Section 5). However, ***SMR*** can not overcome Phoenix when running lr and sm. The reason of worse performance on lr is that most of execution time is waste in ***SMR***'s initialization. And for sm, since it is a computation intensive workload, which does not cause many pagefault, Phoenix can scale well for these benchmarks in less cores. We will evaluation overhead of initialization time in Section 5.4 and the bottlenecks of ***SMR*** will be discussed in detail in Section 5.4.

6.1.2 Scalability of SMR

System Contention is evaluated using the execution time percent of __ticket_spin_lock. Figure 12 reports of __ticket_spin_lock overhead in ***SMR***. Evaluation indicates that the lock overhead can be significantly reduced to less than 1% of total runtime from 1 to 32 cores. This demonstrates that using isolated-memory ***Sthread*** thread can significantly reduce the contention in Linux kernel.

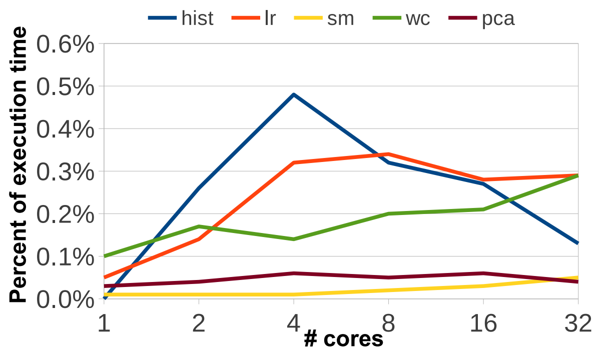


Figure 12. Execution time percent of __tickect_spin_lock in ***SMR***

Figure 13 summarizes the scalability results for ***SMR***. Specially, workloads sm, wc and pca scale up to 32 cores. And the performance of these applications is scale linearly with the number of cores. However, linear_regression and histgram still did not scale particularly because of wasting considerable time in initialization. We will discuss this bottleneck in detail in subsection 5.4.

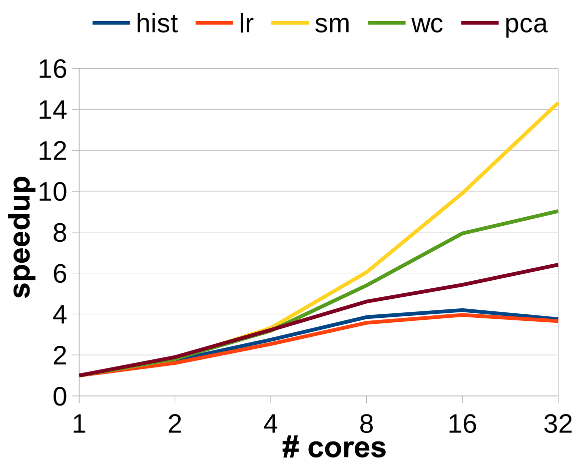


Figure 13. speedup of ***SMR***

Overall, for applications such as word_count, histgram and pca, the results show clearly that ***SMR*** has better performance and scalability when compared to Phoenix. ***SMR*** has the advantage over Phoenix for pipelining map and reduce phases and separating address space by using ***Sthread*** thread. However, for applications such as linear_regression, ***SMR*** does not show its superiority, which will be analyzed in detail in Section 5.4.

6.2 Impact of buffer Optimizations

In Section 4.3, we discussed the organization of intermediate data is critical to the performance of many MapReduce applications. The default hash buffer need to gather the scatter key arrays together before sending them to the shared-channel, which is time-wasting. We addressed this issue by providing a array implementation of buffer, i.e., array buffer, which is a continuous memory space avoiding key value pairs coping and buffer reallocation.

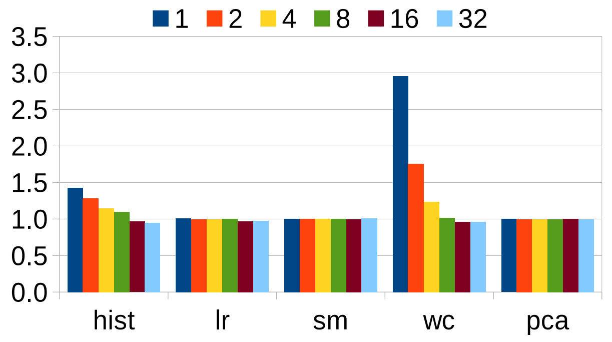


Figure 14. Execution time of hash buffer over array buffer

To measure the impact of buffer's implementation for performance, we run all applications with hash buffer and array buffer, respectively. Figure14 compares the execution time of hash buffer and array buffer. From the Figure, it can be seen that although cases such as word_count and histgram used array buffer has better performance, other benchmarks, e.g., linear_regression, string_match and pca, did not. A common feature of word_count and histgram is that they will generate a large number of key-value pairs in MapReduce computation. Therefore, gather operation waste more time in hash buffer, resulting in better performance by using array buffer.

6.3 Challenges and Limitations

Although we were able to significantly improve the performance and scalability of ***SMR***, workloads histogram and linear_regression still did not scale up to 32cores (Figure 13). In addition, ***SMR*** spent more time then Phoenix when run linear_regression (Figrure 10). To find out where the execution time was being spent, we collect each phase time information by using stub. The results denote that workloads on ***SMR*** waste more execution time in initialization phase when compared to Phoenix.

Figure 15(a) and 15(b) exhibit the initialization time of ***SMR*** and Phoenix, respectively. The results show that the initialized time of ***SMR*** is range 0.25s to 0.35s, while it is just about 0.001s in Phoenix, which is far less than in ***SMR***. Moreover, except for pca, we observe that initialized time is small variation for different benchmarks and in different cores count from Figure 15(b). Since there are two times mapreduce computation in pca, which leads to two times initialization, it take more time than others benchmarks.

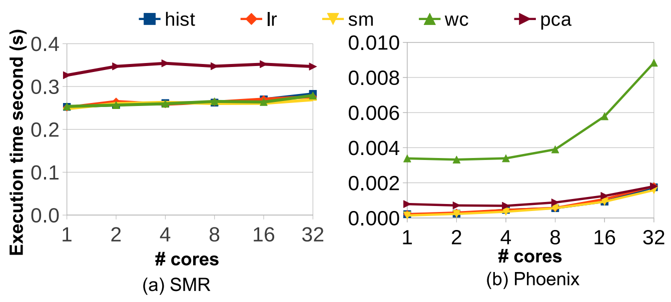
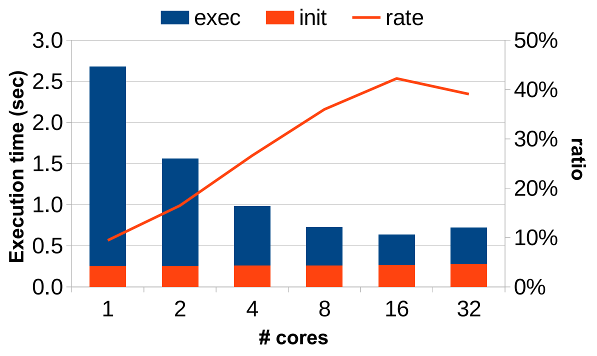


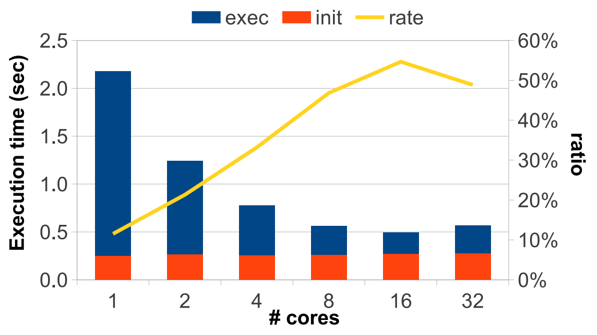
Figure 15. initialization time of ***SMR*** and Phoenix

For multicore MapReduce workflow, most of initialized time is occupied in creating map and reduce threads. Phoenix, in which multiple threads share the process's resource by using Pthread, can save time for creating threads. On the contrary, there are many resource need to be allocated for each ***Sthread*** thread, including separated address space, shared channel and setting up producer and consumer relationship. As a result, ***SMR*** will take more time for in initialization.

Figure 16(a) shows the results with exec defined as mapreduce actual execution time and init as the initialization time. It was clear that the 2 non-scaling workloads (i.e., histgram and linear_regression) share two common trends.



(a) Execution time and initialization time on histgram



(b) Execution time and initialization time on linear_regression

Figure 16. Initialize time with ***SMR***

First, the total execution time of both histgram and linear_regression less than 2.7s for all cores, which is far less than other benchmarks. Second, as the increasing number of cores, the portion of actual computation time (i.e., exec) significantly decreased. However, the initialization time (i.e., init) is almost constant, which cause the init time ratio increase and dominate the total execution time at high cores count. As a result, histgram and linear_regression can not scale up to 32 cores. And the total execution time of linear_regression on Phoenix will less than on ***SMR***.

7. Related Work

MapReduce is a popular distributed framework for massive-scale parallel data analysis developed by Google. There are many existing implementation of MapReduce which adopt on the basic architecture and programming model of originally Google's MapReduce. Hadoop[16], an open source implementation of MapReduce, has been adopted enterprises including Yahoo! and Facebook. Map-Reduce-Merge[17] is proposed for supporting joins of heterogeneous datasets directly by adding a Merge phase. Dryad[12] generalizes MapReduce into an acyclic dataflow graph.

The Phoenix, a MapReduce libray implementation on multicore platform, is the most relevant work to ***SMR***. Phoenix demonstrates tha applications that fit the MapReduce model can perform competitively with parallel code using Pthreads. ***SMR*** differs from Phoenix mainly on two point: Phoenix need barrier between Map and Reduce phase, while ***SMR*** brokes barrier to speed up computing; And ***SMR*** exploits ***Sthread*** thread to implement map and reduce workers, which results in fewer contention in Linux kernel caused by a shared address space.

Tilt-MapReduce[4] uses the tiling strategy to partition a large MapReduce job into a number of small sub-jobs and handles the sub-jobs iteratively. MRPhi[13] is a MapReduce framework optimized for the Intel Xeon Phi coprocessor. Its pipeline the map and reduce phases to better utilize the hardware resource by producer-consumer model, which has some limitations. Mao et al.[14] consider that the organization of the intermediate values produced by Map invocations and consumed by Reduce invocations is central to achieving good performance on multicore processors. They present a optimized MapReduce library (i.e., Metis) using an efficient data structure consisting of a hash table per map thread with a b+tree in each hash entry. Although, we don't compare ***SMR*** with mites, Tilt-MapReduce, our research shows that if the MapReduce library implemented by Pthreads, there will be limitation of scalability for these framework.

Serveral paper have looked at scaling systems on mulitcore system. Boyd-Wickizer et al. aimed at designing a scalable operating system for multicore, named Corey[3], and presents three new abstractions (address ranges, shares and kernel cores), to scale a MapReduce application running on Corey. RadixVM[6], a new virtual memory design that allows VM-intensive multithreaded applications to scale with the number of cores. others ....

8. Conclusions And Furture Work

Phoenix shows MapReduce model is a promising model and approach to use multicore resource for multicore and multiprocessor systmes. However, it has limitations in terms of scalability and performance due to its manners of design and implementation.

In this paper, we analyzed scalability and performance limitations of Phoenix and provide practicable solution. Then we provide a novel thread programming model (***Sthread***) supporting scalable mapreduce, i.e., ***SMR***. We have presented ***SMR***, a scalable MapReduce model for multicore system. We significantly improved the scalability over Phoenix, achieving an average speedup improvement of 2.5x, and a peek speedup improvement of 10x. Our evaluation further shows much time cost in initialization which limit the performance and scalability of ***SMR***. Therefore, our future work will reduce overhead of initialization as well as guarantee the performance and scalability of ***SMR***.

References

[1] G. Andi Kleen, Intel Corporation. Linux multi-core scalability. *In Proceedings of Linux Kongress*, October 2009.

[2] S. Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1. . URL http://doi.acm.org/10.1145/1278480.1278667.

[3] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai, et al. Corey: An operating system for many cores. In *OSDI*, volume 8, pages 43–57, 2008.

[4] R. Chen, H. Chen, and B. Zang. Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *19th PACT*, pages 523–534, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. . URL http://doi.acm.org/10.1145/1854273.1854337.

[5] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. . URL http://doi.acm.org/10.1145/2150976.2150998.

[6] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Radixvm: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 211–224, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. . URL http://doi.acm.org/10.1145/2465351.2465373.

[7] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Usenix Conference on Networked Systems Design and Implementation*, pages 647–667, 2010.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *6th OSDI*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251264.

[9] L. T. et al. Linux source code. *http://www.kernel.org/*.

[10] J. Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *BSDCan Conference*, Ottawa, Canada, 2006.

[11] W. Gloger. Dynamic memory allocator implementations in linux system libraries, May 2006. URL http://www.malloc.de/en/index.html. http://www.malloc.de/en/index.html.

[12] M. Isard, M. Budiu, Y. Yu, et al. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, Mar. 2007. ISSN 0163-5980. . URL http://doi.acm.org/10.1145/1272998.1273005.

[13] M. Lu, L. Zhang, H. P. Huynh, et al. Optimizing the MapReduce framework on Intel Xeon Phi coprocessor. In *BigData Congress '2013*, pages 125–130, Oct. 2013. .

[14] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing mapreduce for multicore architectures. *Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep*, 2010.

[15] C. Ranger, R. Raghuraman, A. Penmetsa, et al. Evaluating MapReduce for multi-core and multiprocessor systems. In

13th HPCA, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 1-4244-0804-0. . URL `http://dx.doi.org/10.1109/HPCA.2007.346181`.

[16] T. White. *Hadoop: The Definitive Guide*. Yahoo! Press, 2010.

[17] H. C. Yang, A. Dasdan, R. L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. *In SIGMOD*, pages 1029–1040, 2007.

[18] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *IISWC '09*, pages 198–207, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-5156-2. . URL `http://dx.doi.org/10.1109/IISWC.2009.5306783`.

[19] Y. Zhang and B. Ford. Lazy Tree Mapping: Generalizing and scaling deterministic parallelism. In *4th APSys*, July 2013.