SMR: A good scalability 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, alleviate the burden of programmer. Phoenix, a MapReduce library for multicore and multiprocess, demonstrates that applications written with MapReduce framwork get competitive scalability and performance in comparison to those written with Pthreads. However, evaluation results show that Phoenix scales worse on on a 32-core Intel 4 Xeon E7-4820 system running Ubuntu 12.04 with kernel 3.2.14.

This paper focuses on improving Phoenix in terms of scalability and performance. First, we analyze some important roadblocks that limit scaling of the Phoenix runtime on shared-memory systems. a novel multithreaded model which isolates memory between threads by default and provides a higher level abstraction for scalable MapReduce communication between threads ***Sthread*** (Scalable Thread). Based on ***Sthread***, we design a modified MapReduce model ***SMR***, which employs a new producer-consumer model for pipelining Map and Reduce phase. Finally, we evaluate ***SMR*** on a 32 CPU cores processor and the result demonstrates that applications, like histgram, word_count and pca, can achieve better scalability and performance than Phoenix.

1. Introduction

As 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, and demonstrated the feasibility of running MapReduce applications on shared memory multicore machines with Phoenix[15]. After that, Phoenix is heavily optimized by Yoo et al[18] to obtain better scalability on shared-memory system. Other libraries such as Metis[14] , Tiled-mapreduce[4] and MRPhi[13] attempt to improve performance of Phoenix from various aspects.

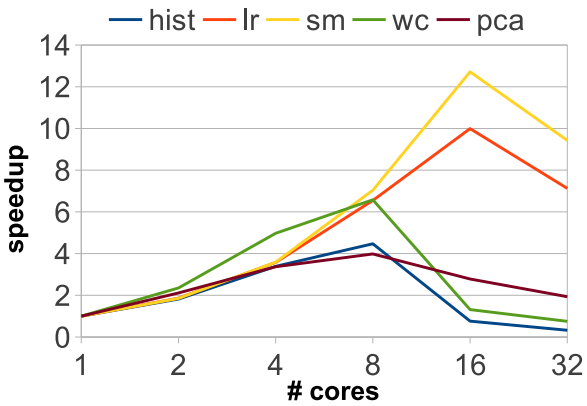


Figure 1. speedup of Phoenix

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] scales worse on an 4-chip, 32-cores system running x86 Linux. Figure 1 shows the speedup of the optimized Phoenix runtime, which is measured on our 32-core system (the detail of this system will be given in Section 5). 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 multithreadig 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 [6]. 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*** (Scalable MapReduce), which can efficiently support MapReduce applications.

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 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 shared-channel for the threads to communicate with 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 3, focusing on implementing mechanism of extension in ***Sthread***. In Section 4, 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 5. Related and future work are covered in Sections 6 and 7.

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. Then the limitation of Phoenix in the aspect of performance is analyzed.

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.

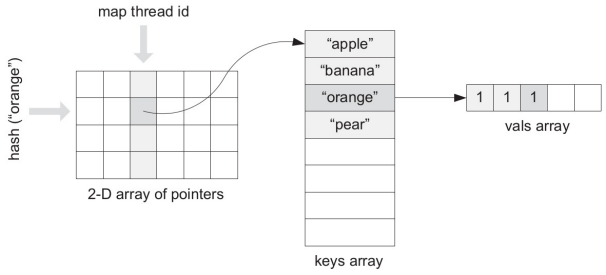


Figure 2. Phoenix intermediate struct

2.3 Optimizing Opportunities of Phoenix

Though Phoenix has demonstrated promising feasibility when running applications on multicore, it has limitations in terms of scalability and performance due to its manners

of design and implementation. The detail analyses are described as follows:

First, 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. By using a share-memory threads model, Pthreads, multiple threads in Phoenix need to share the process's address space[9]. In fact, there is a single lock for per shared address space inside the operating systems virtual memory system. 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].

Second, 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, which limits parallel computing. And 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. it is worth menthion 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 3 and Section 4, respectively.

3. Design of thread program model

In this section, we first investigate the main factor of Phoenix's limited scalability, and then present a new thread model (***Sthread***), which can support ***SMR*** to achieve good scalability. Finally, the detail design and implementation of ***Sthread*** will be given.

3.1 Scalability of Phoenix

Phoenix uses shared-memory multiple threads to implement parallelism, and programs written in Phoenix will start as many threads as the system's cores. Ideally, adding more threads and cores to the runtime would bring about a linear decrease in execution time. However, Phoenix can not scale as well as expected. As indicated in Figure 1, when the system exceeds its scalability limitation, adding more cores might scale negatively. That means the time of completing a workload will increase if there are more cores in the system.

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 Figure3, the cost of spinlock increases quickly as the cores number cross a specific value (ie. 8). Specially, for hist with 16 and 32 cores, __ticket_spin_lock is the function that has largest execution time percents of 40.15% and 71.25%, respectively. Experiment results demonstrate that Phoenix will suffer from serious lock contention when the cores number exceeds 8. That means most of execution time will be used for waiting but not actual computation. Call-graph information shows that __ticket_spin_lock is caused by pagefault, which is not scale well on large numbers of cores.

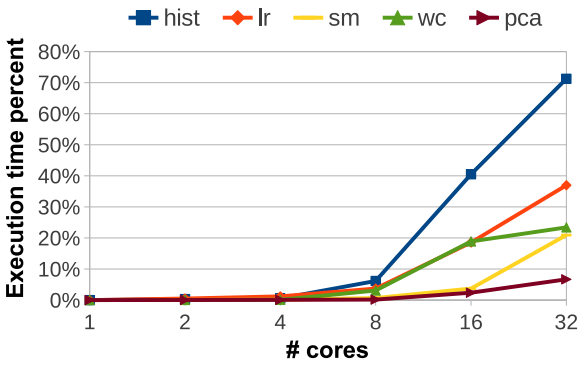


Figure 3. Phoenix ticket_spin_lock percent

The mmap() system call is utilized 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 pagefault in the input data when they invoke map functions. All of these pagefaults need to access the unique mmap_sem semaphore, which is local to a process address space. As a result, multiple threads concurrently pagefaults will casue contention to the single semaphore. One reason why Linux uses this single semaphore on the address space is that it needs a red-black tree to guarantee $O(\log n)$ lookup time when a process has many mmap memory regions [9]. The semaphore is a sleep lock and may run into convoying problems, where waiting threads may get stuck at the end of the wait queue for a long time [1]. The contention is intense when there are large amounts of threads, which will lead to the parallel scalability degradation on the benchmarks.

3.2 Scalable thread library

Our goal is to enable the pagefaults to be scalable for many cores. There are some ways to achieve this target. For example, Corey[3] is a scalable operating system for multi-

core. Clements et.al. proposed a scalable address spaces is 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 pagefaults.

To address the problem and achieve our goal, we propose a novel thread programming model ***Sthread*** (Scalable thread) with better scalability, supporting scalable MapReduce (***SMR***) compatibly, will be presented in this subsection. There are two key points in the design of ***Sthread***. Firstly, to avoid the contention on the single semaphore, we confine the threads in ***Sthread*** to run in separate memory spaces. It means that threads in ***Sthread*** have their local mm_struct and eliminate the contention of the single semaphore. Secondly, when using the separate 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 shared-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 4. Main functions of ***SMR*** thread API.

Figure 4 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 shared-channel between each pair of map and reduce threads by invoking chan_alloc. 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 shared-channel, and the consumer receives the messages from it, which is a typical producer-consumer model (we will detail it in Section 4). Finally, the master starts all threads to work by invoking thread_start.

Although using ***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).

3.3 Design of the Channel

In our design, the shared-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 in 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.

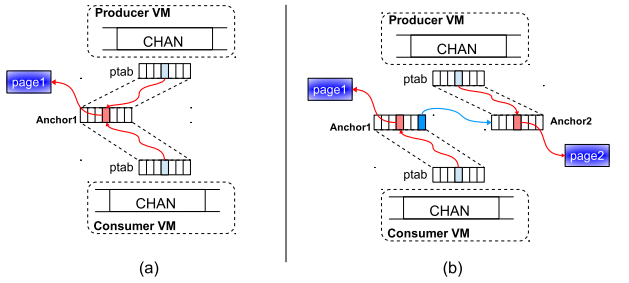


Figure 5. channel extend machanism

Initially, as shown in Figure 5(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, by the anchor page (*Anchor1*), the consumer can locate the page frame *page1* and then read data from *page1*.

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[]. This mechanism allows the producer sending data with no need for waiting when the channel buffer is full. This extend mechanism (Figure 5(b)) will remap the channel buffer (*CHAN* area) to another anchor and allocate some new page frames for the producer. And the consumer will not be disturbed by the extend mechanism 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 shared-channel for threads to communicate. This unbounded shared-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 use this feature to implement the scalable MapReduce (***SMR***) in Section 4.

4. 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.

4.1 Execution flow

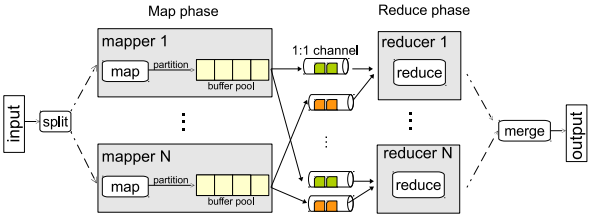


Figure 6. The workflow of ***SMR***

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 6 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 are pushed into a task queue. Then workers in Map phase will get these tasks from the task queue and apply the map function to the tasks. The intermediate key-value pairs produced by the map worker will be inserted into a local buffer. When 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 reduce function to aggregate them with the same key. Finally the results from multiple reduce workers are merged and output.

Compared with existing work, our ***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 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 invoke the reduce function to work with no need for waiting all workers in 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 between map workers and reducer workers in our ***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. According to the key, the library's Merge phase will sort these pairs 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 traverses 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.

4.2 Pipelined execution

Pipelined map and reduce have been adopted in the MapReduce framework for distributed computing[7]. Condie et al. shows since pipelining delivers data downstream operators more promptly, it can increase opportunities for parallelism, improve utilization and reduce response time. And results demonstrate that pipeline can reduce job completion times.

MRPhi[13], a MapReduce framework optimized for the Intel Xeon Phi coprocessor, pipelines the map and reduce phases to better utilize the hardware resource by adopting a typical producer-consumer model. There is a many-to-one queue in this model, in which multiple map workers will insert data into the queue concurrently, and then the single reduce worker removes the data from it. 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 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 wont add data into a full buffer and the reduce worker wont try to remove data from an empty queue. If choosing the latter, expensive malloc and free operations are required to manage the queue, which causing 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 to manage 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 7. Produce-Consume model in SMR

As depicted in Figure 7, 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. With the use of the one-to-one *shared-channel*, map worker sends the data to reducer without contenting with other map workers. When the *buffer* threshold is reached, the map worker will send data in *buffer* to the shared-channel and then set the *buffer* as empty. So the *buffer* will be reused in the hole Map phase, which saves overhead caused by dynamic memory allocation. If the *shared-channel* is full, by triggering the extension mechanism described in Section 3.3, map worker can continuously work without waiting for the reduce worker to remove the data from the shared-channel.

In fact, there is a local buffer pool for each map worker, in which each buffer is used to store key-value pairs which will be sent to a corresponded reducer. When a key-value pair is generated by the map worker, the partition function is invoked to index the corresponded buffer. Once the buffer is determined, the map worker will insert the key-value pair into this buffer. In default, the buffer in ***SMR*** is 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.

4.3 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. Defaultly, buffer in ***SMR*** is a hash table (Figure 8(a)), in which each entry is a pointer, pointing to 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.

Based on ***Sthread***, our producer-consumer model requires that data sent to the channel should be a contiguous block of memory. Since 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 8(b)). This extra copy

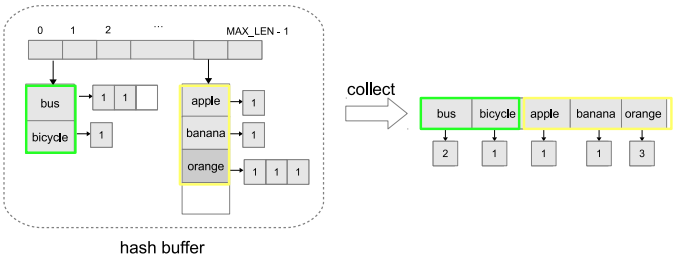


Figure 8. hash buffer and gather

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 affects 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, and it can be reused in the hole of Map phase. The map worker will indicate the buffer as empty at the end of a sending, but will not free the memory until all map jobs have been finished. 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).

5. Evaluation

In this section, we measured the performance and scalability of ***SMR***, and compare it with Phoenix. 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.

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 can provide improved performance and scalability[10]. Therefore, we will evaluate performance and scalability of Phoenix builte with

both jemalloc and ptmalloc, denoted as Phoenix-jemalloc and Phoenix-ptmalloc, respectively.

5.1 Phoenix Scalability Analysis

Since Phoenix is sensitive to memory allocator[18], in this subsection, we first evaluate its scalability with jemalloc, a good scalable memory allocator, and than we measure the scalability of ***SMR*** to demonstrate the effectivity of ***SMR***.

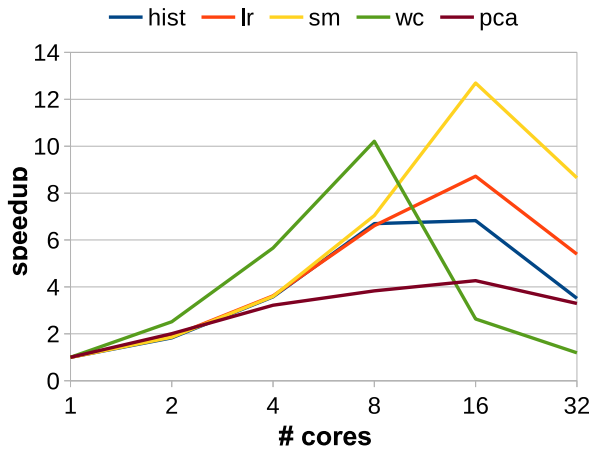


Figure 9. speedup of Phoenix-jemalloc

Figure 9 shows 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.

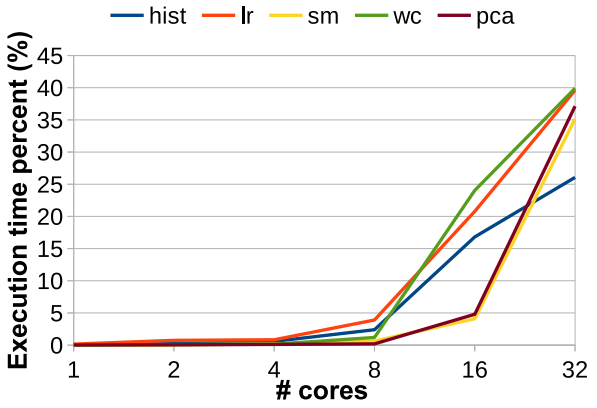


Figure 10. speedup of Phoenix-jemalloc

At the same time, we collect __ticket_spin_lock execution time percent by Linux perf to peek the contention in Linux kernal. The result as shown in Figure 10, denote serious lock contention takes place when increase the number of cores over 8 and as a result execution time will be drastically increased.

In conclusion, at low cores, increasing cores number result in speedup for Phoenix. However, as more cores were added, the benefit was reversed due to the increased time spent in ticket_spin_lock. That means Phoenix can not scale up to 16 cores in spite of using the scalable memory allocator.

5.2 Performance and Scalability

5.2.1 Performance

Based on ***Sthread***, we implemented the optimization of Phoenix in Section 4 and measured ***SMR*** performance by executing each benchmark, including histogram (hist), linear_regression (lr), string_match (sm), wordcount (wc), and pca (pca).

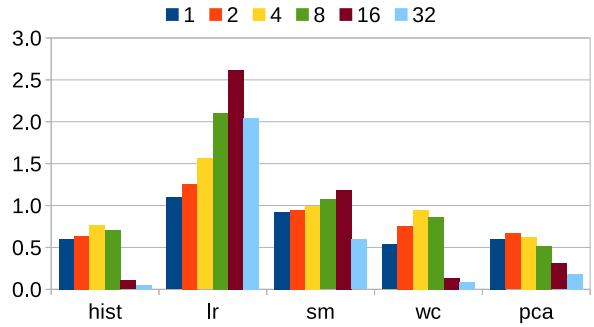


Figure 11. SMR versus Phoenix with ptmalloc

Firstly, we compare the execution time of ***SMR*** with Phoenix-ptmalloc. As shown in the Figure 11, 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. However, ***SMR*** runs worse than Phoenix on lr (about 0.8x).

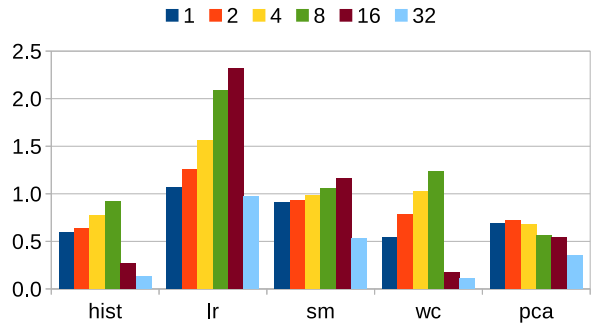


Figure 12. SMR versus Phoenix with jemalloc

Then we evaluate the performance of Phoenix with jemalloc. The result show that Phoenix with jemalloc has a better performance. when compare it with ***SMR***, as showen in Figure 12, we observe the similar tendency like Phoenix-ptmalloc.

Compared to Phoenix, we significantly increase the performance. For workloads, such as hist, wc and pca, the desired performance of ***SMR*** owns to reducing the overhead caused by contending shared address space (Section 3) and pipelining the Map and Reduce phase (Section 4). 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. We discuss their bottlenecks in detail in Section 5.4, but in the remainder of this section we first focus on the optimizations that turned out to be successful.

5.2.2 Scalability

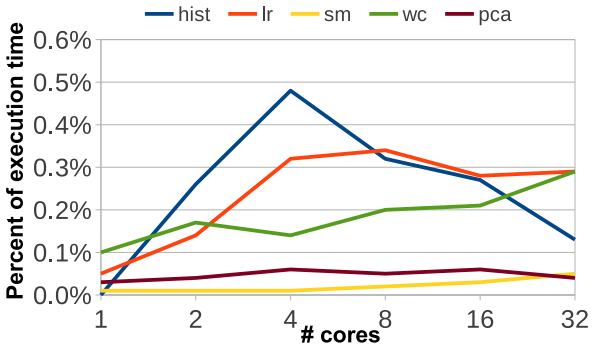


Figure 13. Execution time percent of __tickect_spin_lock in ***SMR***

In our ***SMR***, using ***Sthread*** thread was sufficient to reduce the contention in Linux kernel. System Contention is evaluated using the execution time percent of __ticket_spin_lock. Evaluation indicates (Figure13) that the locking overhead can be significantly reduced to less than 1% of total runtime on 32 cores

Figure 14 summarizes the scalability results for ***SMR***. Specially, workloads sm, wc and pca scaled 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.

To summarize, ***SMR*** demonstrated its performance and scalability for applications such as word_count, histgram and pca. For applications such as linear_regression, ***SMR*** does not show its superiority, which we will analyze this result in detail in Section 5.4.

5.3 Impact of buffer Optimizations

In Section 4.3, we discussed the organization of intermediate data is critical to the performance of many MapReduce

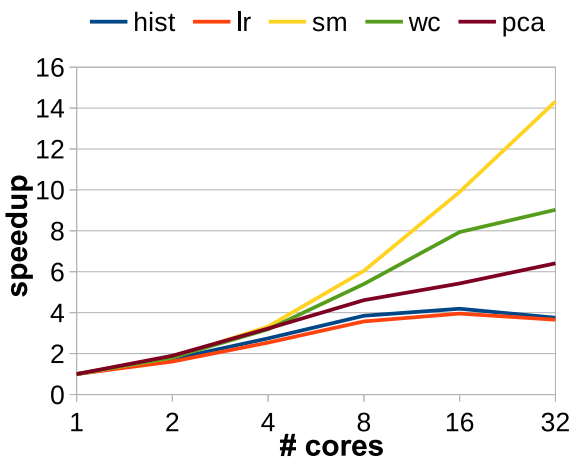


Figure 14. speedup of ***SMR***

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. However, not all the workloads benefited from the optimization.

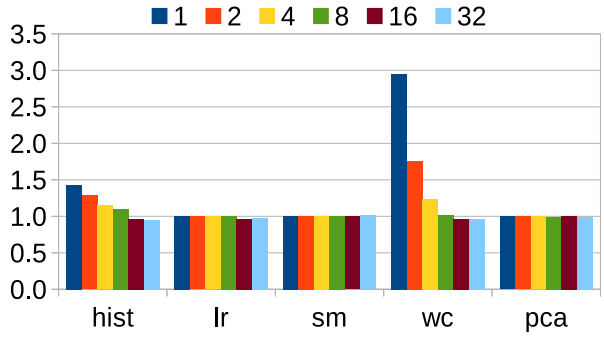


Figure 15. Execution time of hash buffer over array buffer

Figure15 shows the execution time of hash buffer in comparison of the 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. So, for workloads that did generate a large number of key-value pairs, gather operation in hash buffer waster more time, as a result, array buffer will be better.

From the experimental results, it is observed that benchmarks yield better performance using ***SMR*** than Phoenix (in most cases). ***SMR*** has the advantage over Phoenix for pipelining map and reduce phases and separating address space by using ***Sthread*** thread instead of shared memory Pthread. The performance of the histgram implementation on Phoenix is the best, 30%-95%x better than the Phoenix implementation on the our 32-cores system.

5.4 Challenges and Limitations

Although we were able to significantly improve the scalability of ***SMR***, workloads histogram, linear_regression still did not scale up to 32cores. Figure 11 presents their execution time at various core counts. We collect each phase time information to find out where the execution time was being spent by using stub. The results denote that workloads on ***SMR*** waste more execution time in initialization phase, compared to Phoenix. Figure 16 shows the initialization time of ***SMR*** and Phoenix. Exactly, the initialized time of ***SMR*** is range 0.25s to 0.35s, while it is just about 0.001s in Phoenix.

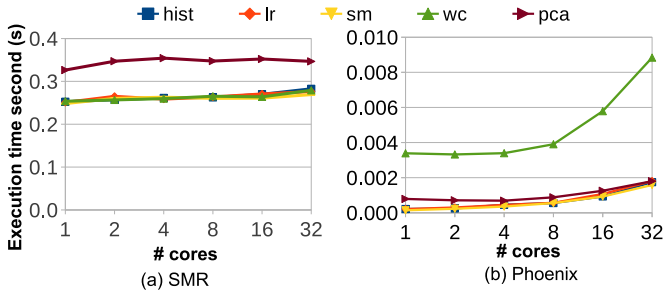


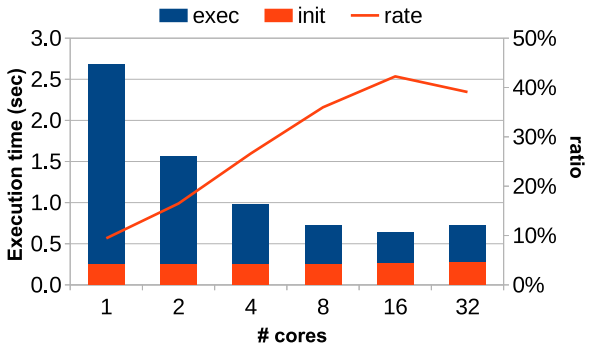
Figure 16. initialization time of ***SMR*** and Phoenix

For multicore MapReduce library, most of initialized time is occupied in creating map and reduce threads. Compared to thread in Phoenix, which based on Pthread, creating thread will spend more time in ***SMR*** because of more resource need to be allocated. In addition, it will take more time for creating shared-channel and setting up it. From the experimental results (Figure16(b)), we observed that initialized time is small variation for different benchmarks and in different cores count, except for pca. Since there are two times mapreduce computation in pca, which lead to two times initialization, it take more time than others benchmarks.

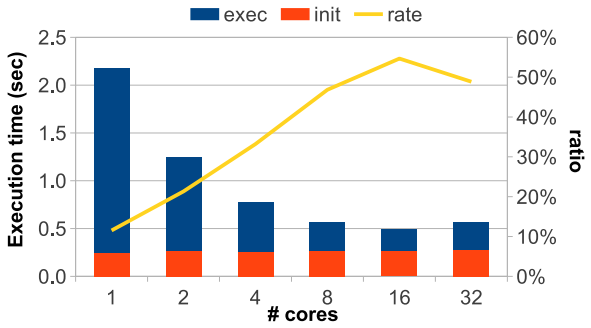
Figure 17(a) shows the result with exec time defined as mapreduce actual execution time and init time as the initialization time before workers starting work. It was clear that the 2 non-scaling workloads shared two common trends. First, the total execution time of both histgram and linear_regression less than 2.7s for all cores, which if far less than other benchmarks. Second, As the increasing number of cores, the portion of actual computation time (execution time) significantly decreased. However, the initialization time is almost constant, which cause the init time ratio increase and dominate the total execution time at high cores count. As a result, the total execution time of linear_regression on Phoenix will less than on ***SMR***.

6. 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



(a) Execution time and initialization time on histgram



(b) Execution time and initialization time on linear_regression

Figure 17. Initialize time with ***SMR***

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. It 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 problem of scalability.

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 ....

7. 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. we provide a novel thread programming model for support 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 the scalability and reliablity of ***SMR*** for iterative processing. As to poor performance on multicast across processors, our future work will provide scalable iterative processing.

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. *Acm Sigarch Computer Architecture News*, 47(4):199–210, 2012.

[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*, pages 211–224. ACM, 2013.

[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. Mapreduce-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`.