SMR: A good scalability MapReduce for Multicore Systems

Abstract

Designing and implementing efficient, simply parallel programming model is challenging. Traditional parallel programming techniques like Pthreads and OpenMPI leave programmers solving many details of design chanllenges. Phoenix, a MapReduce library for multicore, show that MapReduce is a promising model for scalable performance on mulitcore and multiprocess. Applications written with MapReduce framwork have competitive scalability and performance compared to those written with Pthreads.

This work presents ***SMR***, a modified version of phoenix framwork (...) We evaluated ***SMR*** on a 32 CPU cores process. The results show that ***SMR*** achieves up to (NUM1) speedup over 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]. While utilizing multicore sources is still challenging because of the difficulties of parallel programming. Parallel programming is becoming more and more popular because of its potential to improve performance, especially in multicore architectures. In the past, computer software has been written using serial computation concepts which is usually less efficient than multithreaded parallel computation.

MapReduce[7] is a promising programming model for clusters to perform large-scale data processing in a simple and effecient way. In most cases, programmers only need to implement two interfaces: Map, which processes the input data and converts it into a number of key-value pairs; and Reduce, which aggregates values in the key-value pairs according to the key. And the programmer dose no need to control synchronization and schedule tasks manually. While initially MapReduce is implemented on clusters, Ranger et al. have demonstrated the feasibility of running MapReduce applications on shared memory multicore machines with Phoenix[14]. Other libraries such as Metis[13] and Tiled-mapreduce[3], also show that MapReduce is a promising programming model for multicore platforms to take full advantage of processing resources. Phoenix uses the pthread library to assign tasks among CPU cores and relies on shared memory to handle inter-task communications.

Ideally, adding more processes and cores to the library would bring about a linear decrease in execution time. However, the benefits of adding more cores will be reduced due to overhead associated with the additional processes–the contention of lock. A common parallel programming model is shared-memory multithreading, where all threads of an application share a single address space. This shared address space has a cost, can limit the scalability of these applications. All of these operations are synchronized by a single per-process lock. As a result, the performance will be better when the number of cores increases from 1 to 4, while the performance will be worse if using more than 4 cores. Hence, with the continuously increasing number of cores, it can easily cause resource pressures on the runtime, operating systems and the CPU caches, which could sig- nificantly degrade the performance.

To remedy the above problems, firstly, we propose a Scalable thread libray(***Sthread***). Then this paper presents a modified model of ***SMR*** (Scalable MapReduce), that can efficently support MapReduce applications. ***SMR*** reserves the similar Phoenix programing interfaces as well. Specifically, this paper make the following contributions:(need talk more....)

- We research the reason for bad scalability of Phoenix.
- We aim at providing an race-free programming abstraction to support scalable MapReduce.
- We present a scalable MapReduce.

In order to ground our discussion, we present an overview of MapReduce architecture and Phoenix in Section 2. We then develop the design of ***Sthread*** in Section 3, keeping the focus on the implementation mechanism of extension In Section 4 we describe our support for pipelineing map and reduce, and illustrate the potential benefits of that producer-consumer model for MapReduce framework. We present initial performance results in Section 5. Related and future work are covered in Sections 6 and 7.

2. BACKGROUND

In this section, we review the MapReduce programming model and describe the salient features of Phoenix, an implementation of MapReduce for multicore. Then we analyze the limited performance of Phoenix.

2.1 MapReduce Programing Model

The MapReduce programming model is inspired by funnctional languages and propose for data intensive computation in cluster environment. It simple programming interface just require programmer defines two primitives: map and reduce. The map function is applied on the input data and produces a list of intermediate key and value pairs. The reduce function is applied on all intermediate pairs and groups all pairs with the same key to a single key/value pair. The combine function is an aptional operation to aggregates the key and value pairs locally in Map Phase aiming to save networking bandwidth and reduce memory consumption.

The charm of MapReduce is that, for algorithms that can fit that form, the library hides all the concurrency from the programmer. For example, one can count the number of occurrences of each word in a body of text as follows. The map function emits a word, 1 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.

2.2 Phoenix

Phoenix is an implementation of MapReduce for multicore and multiprocessor systems using Pthreads. It shows MapReduce is a promising model and applications written with MapReduce model have competitive scalability and performance with those written with Pthreads[14];

Phoenix stores the intermediate key-value pairs produced by the Map calls in a matrix(Figure 1). Eche map and reduce workers can write or read the global buffer. Concurrent Map workers should avoid touching the same data. To avoid locking and cache contention costs, when map workers and reduce workes operate the buffer concurrently, there are two strategy must be done:
(1) Each row of the buffer is exclusively used by a worker in the Map phase, While each column of the buffer is exclusively used by a reduce worker.
(2) There is a barrier between Map and Reduce phase. Only when all map workers have been completed, reduce workers can begin to work.

As indicated in this figure2, the Phoenix scales rather well when we use no more than four cores. However, when adding more cores in the system, the curve increases exponentially. The scalability of Phoenix is limited, the performance will be better when the number of cores increases from 1 to 4, while the performance will be worse if using more than 4 cores. Perf[].need to talk more about the results...

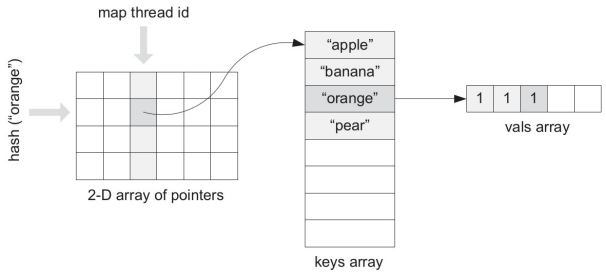

Figure 1. Phoenix intermediate struct

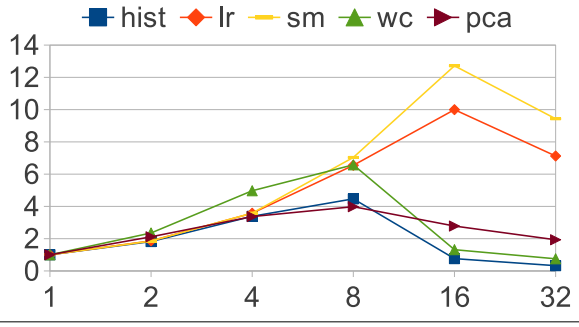

Figure 2. speedup of Phoenix

2.3 Optimizing Opportunities of Phoenix

Though Phoenix has successfully demonstrated the feasibility of running MapReduce applications on multicore, it also comes with some deficiencies when processing jobs with a relatively large amount of threads, which would be common for parallel programming with shared-memory multithreading.

First, there is a strict barrier between the Map and the Reduce phase, which requires workers in Reduce phase can only be startd until all workers in Map phase has been finished. Hence, the execution time of Map phase is determined by the slowest map worker. If one of a map woker is slow, then the runtime of MapReduce will be need more time. Futher, as the user-defined map functions are usually computation-intensive, while the Reduce phase is memory-intensive, the barrier is bad for the overall hardware resource utilization.

Second, in cluster environments, the map tasks and reduce tasks are usually executed in different machines and data exchange among tasks are done through networking, compared to shared memory in multicore environments. Hence, shared data structures , instead of networking communications, are the major performance bottlenecks processing large MapReduce jobs on multicore. Multithreaded programs with hybrid shared mutable data structures do not scale on multicore. They suffer from serious lock contention. Multithreaded applications on many-core processors can be

bottlenecked by contended locks inside the operating systems virtual memory system. Because of complex invariants in virtual memory systems, widely used in Linux kernels, have a single lock per shared address space. [5]

Phoenix do not achieve the desired scalability with increasing core count even with a simple, embarrassingly parallel job. On a serious note, the internal synchronization strategy (e.g., ticket spinlock) of the virtualized instance on a machine with higher core count (e.g., 32-core) dramatically degrades its overall performance. our target of ***SMR***, We will present a new programming model to avoid the contention bewteen threads. we will use a producer-conusmer mode to pipeline map and reduce, therefore making the frameworker more efficient.

3. Design of thread model

In this section, we will search the main factor of Phoenix's bad scalability. Then we present a new thread model, which has good scalability.

3.1 Scalability of Phoenix

In Phoenix, programs are often written to start as many threads as the system has cores, As indicated by figure2, when the system is larger than their scalability limit adding more threads might actually scale negatively. The time of completing a workload for one core increases when there are more cores in the system. The trend of this curve suggests that the parallel scalability of Phoenix is poor.

In order to understand the scalability behavior, Perf[] is exploited to collect execution time information on the function basis. Experimental results show that Phoenix suffer from serious lock contention when the core count exceeds 8. Figure3 shows the percent of __ticket_spin_lock of each benchmark. Histgram on 16cores and 32 cores show that __ticket_spin_lock is one function which have largest execution time with 71.25% and 40.15% respectively. That means Phoenix waster most of time to wait but not actual work. And it can't fully use mulitcores resource. From Seciton2.2 we know, Phoenix takes two strategies to avoid multiple threads operate the global buffer concurrently. Why the benchmarks suffer from serious lock contention? Actually, it caused by linux kernel.

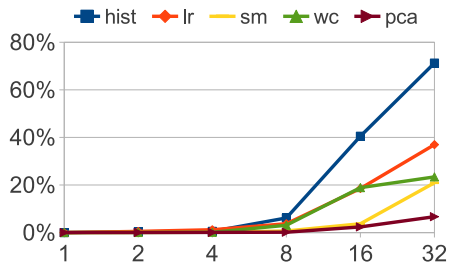

Figure 3. Phoenix spinlock percent

Data structure private locks can be a problem if the data structure is shared by multiple threads. A standard example here are the mm_sem read-write semaphore that protects the list of mappings in a process and the pagetable_lock that protects the pagetable state of a process. These locks are local to a process address space. In Phoenix, there is one master process, and map worker or reduce worker are threads belong to the master process. However when the process is using multiple threads then these threads will be able to access the address space in parallel, which can cause contention on these locks. Call-graph information and source code analysis show that __ticket_spin_lock is caused by pagefault. When a large multi-threaded computing job that causes a lot of parallel page-faults, these page-faults will all run into contention on the mm_sem semaphores. Semaphores are sleeping locks and may run into convoying problems where waiting threads may get stuck at the end of the wait queue for a long time.[1] Thus, spin lock contention degrades the parallel scalability performance of the benchmark.

One reason that widely used operating systems use a lock on the address space is that they use complex index data structures to guarantee O(log n) lookup time when a process has many mapped memory regions. Linux uses a red-black tree for the regions[8]. Because the data structures require rebalancing when a memory region is inserted, they protect the entire data structure with a single lock. The lock is local to a process address space. when the process is using multiple threads then these threads will be able to access the address space in parallel, which can cause contention on the lock. As the increasing of cores number, the scalability will be bad.

3.2 Scalable thread model

Our goal is to make page faults scale to large numbers of cores. There are many way to achive this target such as [4].... However, We don't want to change the implementation of operating system, but just provoid a easy-to-use scalable thread model. This requires addressing a basic problem, how allow page faults to run concurrently to eliminate on the per-process read/write lock. In fact, if applications use processes instead of threads can avoid a single, shared address space, but this complicates sharing.

To achieve our goal, this paper persents a new concurrent address space design that eliminates the above sources of contention by applying a new program model and by introducing channel, a way to share data between threads. We aim at providing an race-free programming abstraction to support scalable MapReduce. With this target, we propose a new thread programming model ***Sthread*** (Scalable thread). ***Sthread*** is C library-based and thread in ***Sthread*** run in separate memory spaces. Threads in Pthreads share the address space of the process that created it, while threads in ***Sthread*** have their own address space, meaning each thread has a mm_struct. Therefore, thread no need contend with others thread for lock. On the other hand, Threads based on share

space can directly communicate with other threads of its process; While ***Sthread*** must use interprocess communication to communicate with the other threads. We provoide a share channel for threads to communicate. The initial state of isolated memory in a thread is inherited from its parent thread when it is started. Currently, there is no shared heap among threads, but private heap for each thread. On the other hand, comparing to thread, it is bad for sharing data between threads. To solute the problem, ***Sthread*** provide interface of channnel to create, send, receive opration.

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.

Figure4 lists main function of managing threads and channels in ***Sthread***. Initialize in ***SMR***, the master threads invoke thread_alloc to create map workers and reduce workers , and invoke chan_alloc to alloc channels for them. In ***SMR*** map and reduce are implemented using a producer-consumer model. The map threads and reduce threads are the producers and consumers, respectively (Section 4).

Map worker, producing key-value, send message(calling chan_setprod), and reduce worker as a consumer to receive message(calling chan_recv). After creating channels and setting up send-receive relationship, both map and reduce workers start work by invoking thread_start. When the local buffer is full, Map worker invoke chan_send to send the key-value to coresponded channel. Then Reduce worker invoke chan_recv to receive the key-value from the channel. Though, using ***Sthread*** can effectively decrease the overhead of contention, it also take some extra overhead comparing to Phoneix. The extra overhead focus on initialization(section 5).

3.3 Design of the Channel

Once the channel relationships are set up, map workers can invoke chan_send to send messages to channel, and reduce workers can receive from the channel by chan_recv. In order to avoid map waiting when the channel buffer is full, we design an unbounded size of communication buffer. Therefore, a sender can send any number of messages without blocking or waiting. Unboundedness goal is the key to achieve high throughput.

In order to reach a unbounded buffer, We design an extend machanism which allows remap channels buffer to a new page frames. To record and trace generations of page frames among the sharing parties, a special anchor extension page is introduced and shared between producer thread and consumers consumer. Initially a sequence of pages in channel buffer are mapped to a anchor page table (Anchor in Figure 5), in which each page has a corresponding page table entry (PTE). Upon a producer page fault, the fault handler will allocate a real page frame and update the faulting page with a writable producer mapping so that the producer can write the page afterwards. When a thread want to send without waiting, it can call extend primitive to remap channel buffer to new page frames, without changing the old page that consumers may still require. After a consumer receive the old page from the channel, it can call extend to find the new page frames sended by the producer. The older page frames decrease their reference counts and are freed automatically when the counts reach zero.

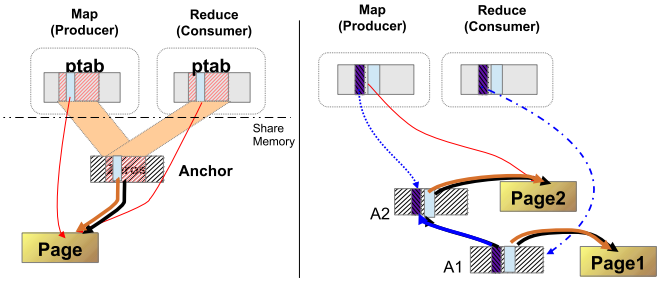

Figure 5. channel extend machanism

4. Implementation and Runtime

In this section we discuss our extensions to the MapReduce programming model and shows the major changes to runtime to support ***SMR*** to support pipelining between Map and Reduce (Section 3.2). We describe how our design supports piplelining (Section 3.3), and discuss the buffer design (Section 3.4). Our focus here is on Producer-Consumer model; We defer performance results to Section 5. e

4.1 Execution flow

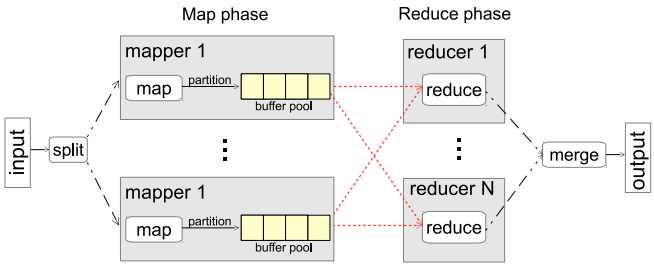

Figure 6. The workflow of DMR

SMR's implementation of MapReduce closely resembles Phoenixs. There is a single master managing a number of slaves worker. Figure6 illustrates the workflow of ***SMR***,

including three main phases: map, reduce, and merge. At the beginning, a split function divides the input data across workers. On multi-core CPUs, a worker is handled by one thread. A worker usually needs to process multiple input elements. Thus the map function is applied to the input elements one by one. Such an operation of applying the map function for an input element is called a map operation. Each map operation produces intermediate key-value pairs. Then a partition function is applied to these key-value pairs. Then in the reduce phase, each reduce operation applies the reduce function to a set of intermediate pairs with the same key. Finally the results from multiple reduce workers are merged and output.

Compared with existing work, our ***SMR*** has two main designment to improve its performance and scalability. On the one hand, We propose produce-conumse model to pipeline map and reduce phase. ie. map worker as a producer insert key-value into local buffer, and reduce worker as a consumer will fetch key-value from buffer when the buffer is full. Therefore reduce phase can start without waiting the Map phase finished. On the other hand, ***SMR*** implemente a worker as ***Sthread*** thread instead of as a single multi-thread process. Then workers will be in the isolated address space avoiding contention on a single per-process lock.

Combiner. In order to maximally reduce memory pressure due to intermediate key-value storage. To reduce the amount of communication between mappers and reducers, a mapper can combine its produced key/value pairs with the same key The imbalance of tasks can be solved by dynamic scheduling in the Map phase.

Merge. Each Reduce generates a set of output key/value pairs, and the librarys Merge phase sorts them by key to produce the final output. A MapReduce job may declare reduce function as NULL. For such an application, Phoenix still performs reduction on the intermediate data using a default reduce function, which traverses key-value pairs. This approach is inefficient. Unlike Phoenix, for applications without reduce function, ***SMR*** does not start the reduce phase and directly starts the merge phase after the map phase.

4.2 Pipelined execution

Pipelined map and reduce has been adopted in the MapReduce framework for distributed computing[6**?**]. Condie et al. show that since the intermediate data is delivered to downstream operators more promptly, it is able to improve resource utilization.

While in Phoenix, (To avoid mulit map and reduce contention the same area,) there is a strict barrier between the Map and Reduce phases: the workers in one phase can only be started until all workers in the previous phase has been finished. Hence, the execution time of a job is determined by the slowest worker in each phase. A downstream dataflow element can begin consuming data before a producer element has finished execution, which can increase opportunities for parallelism, improve utilization, and reduce response time. On the other hand, MapReduce workloads are an ideal candidate for pipelining as the user-defined map functions are usually computation-intensive, while the reduce phase to construct the global container is memory intensive[15]. Overlapping the computation-intensive and memory-intensive workloads can effective improve the overall hardware resource utilization.

We design a producer-consumer model to pipeline the map and reduce phases. There are two major data structures, which are local buffers pool for each map worker and a global buffer for reduce worker. Specifically, each map worker has a local buffers pool and each buffer for a corresponding reduce worker. Partition function will be used to push key-value into a corresponding buffer. When the buffer threshold is reached, the corresponding reduce worker can read record in the buffer. One reduce worker will get key-value from each map by Round-Robin and merge the key-value pairs to the global buffer.

Defaultly, the buffer is a hash table as Phoenix. While this technique is more effective for the array buffer container than the hash buffer. We will explain the advantage of array buffer in section 3.2.1. MRPhi[12] is also use producer-consumer model to pipeline the map and reduce phases. There are partitions queues for each reduce worker. While we don't use queues, mapping will be used in DMR(section 3.2.2).

4.2.1 Producer-Consumer model

The pipelined map and reduce are implemented using a producer-consumer model. The map and reduce workers are the producers and consumers, respectively. Specially, the map worker generate key-value and put it into the buffer. At the same time, the reduce worker is consuming the key-value. MRPhi[12]design a producer-consumer model to pipeline the map and reduce phases. Each map worker has a local hash table. When a local hash table is full, key-value pairs stored in this table are partitioned and pushed into corresponding queues. Meanwhile, one reduce worker works on one queue to merge the key-value pairs to the final global hash table.

However, there are two main defects of producer-consumer model in MRPhi. Firstly, The queue used in MRPhi is a many-to-one queue. Multiple map worker will append data to the tail of queue contendly. when one thread is appending, any thread waiting to append is stalled. That means more time spent in waiting lock. Secondly, They dont describe, however, how to manage the queue, which using a allocted space or by dynamic memory allocation. if using a allocated space, when the queue is full, the producer will blocking waiting for a slot in the queue. it is bad for parallelion excution of Map and Reduce. if dynamic memory allocation, map no need waiting, while there will have a large number of malloc and free, which causing overhead.

our tagert of designing the produce-consume model

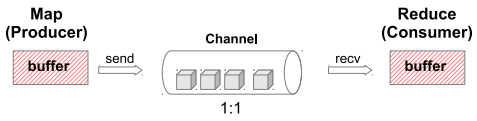

Figure 7. Produce-Consume model in SMR

Figure.7 shows producer-conusmer model in our ***SMR***. There is a one to one channel between map worker and reduce worker, avoiding the contention of multiple map workers. When the buffer is full, map worker send the buffer to channel, at the same time, reduce worker receive data from channel, and copy it to local buffer. on the other hand, we use the a special mapping to allow producer and consumer parallelization effectively, without waiting when the queue is full, or malloc and free operations. When the local buffer is full, it send the data to channel and then use the buffer again. That means map no need wait and there is no many malloc and free operations. Figure7 Thus, this parallelization effectively decouples the behavior of proucer and conusmer, and allows them to be overlapped without many malloc and free.

In Section**??** we will present the design of Channel.

4.2.2 Buffer Design and Optimize

Research shows that the organization of MapReduce intermediate data is a challenge of design a MapReduce library. The organization of the Map output is critical to the performance of many MapReduce applications, since the entier body of intermediate data must be reorganized between the Map and Reduce phase: Map produces data in the same order as the input, while Reduce must consume data grouped by key.[13] In a data center this operation is dominated by the performance of the network, but when running on single multicore processor the performance is dominated by the operations on the data structure that holds intermediate data.

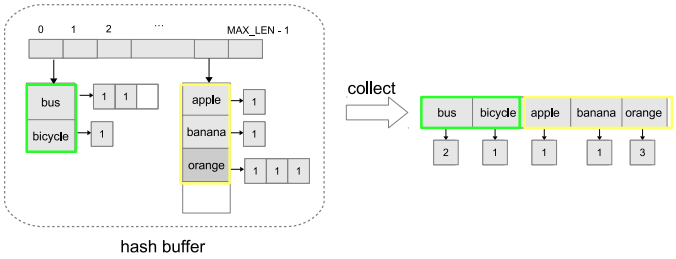

Figure 8. Produce-Consume model in SMR

Defaultly, buffer in ***SMR*** is a hash table(Figure8), and each element is a array of pointers to key arrays, where the fixed is by a default value(256); During the map phase, each map worker uses patition function to index the buffer. Once the buffer is determined, the element is indexed by the hash value of the key. Inside each entry of a hash table, ***SMR*** stores the key-value pairs in an array sorted by key. If the hash table have enough entries, collisions will be rare and the key arrays will be short, so that lookup and insert will have cost O(1). The hash table's O(1) lookups make it particularly attractive for workloads with many repeated keys. (the reduce operator is immediately applied to that pair based on the local container. This process is performed using a combiner.)

Hoverver, our producer-consumer model requires reduce task is a contiguous block of memory, which implies that key-value of hash table should be gathered before sending to reduce worker. We address this issue by copying values out of the hash tables at the end of the map phase and inserting them into a new contiguous array. This extra copy is unfortunate and time-consuming. Furthemore, it also requires frequent allocations and deallocations of memory along with the data structure creation and destrcution.

To avoid the time-consuming collection in hash buffer, we implement a easy buffer, namely array buffer. The buffers are initially sized to a default value and then reuse the buffer among sub-jobs. It will indicate that the buffer is empty at the end of a sending, but will not free the memory until all map jobs have finished. Each map thread could store its output by appending each key/value pair to array buffer, and then as each sub-job is processed in turn, the data structures and memory spaces for the input and intermediate data can be reused across the sub-job boundaries. This avoids the costs of expensive memory allocation and deallocation, as well as the data structures construction. This could save the expensive operations such as concurrent memory allocations and deallocations, as well as the building of data structures. Furthemore, unlike hash buffer, array buffer is a contiguous block of memory, it not need collection before send the buffer to channel. For applications that likely have abundant duplicated keys (or values), such as word_count, it would be more worthwhile to use array buffer.

5. 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.

5.1 Performance of benchmarks

Performance We compare ***SMR*** with Phoenix, built with different memory allocator: ptmalloc and jemalloc. Since

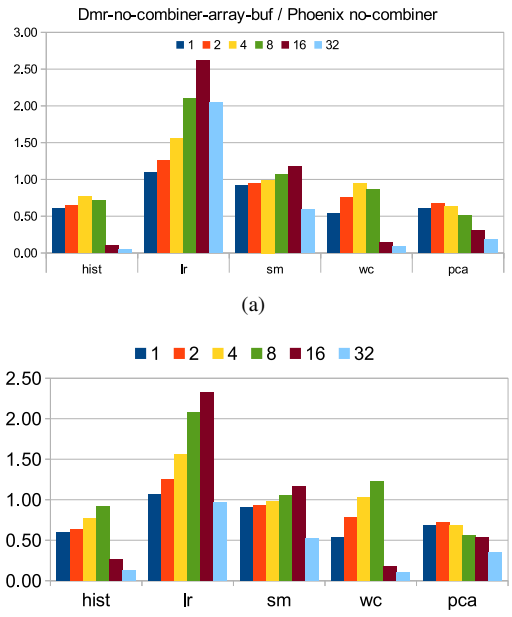

Figure 9. Execution time of SMR versus Phoenix

there are many heap objects shared among threads in Phoenix, and ptmalloc[10], the memory allocator in glibc, does not scale on multicore system. Then we evaluation Phoenix built with jemalloc[9], a scalable memory allocator for multicore system.

Figure11(a) and 12(a) present the Execution time of ***SMR*** versus Phoenix. ***SMR*** matches or outperforms Phoenix on 4 out of 5 workloads, but runs worse than Phoenix only on linear_regression. For hist, pca and word_count, ***SMR*** outperforms Phoenix betwen xxx and xxx faster. The reason of worse performance on linear_regression is that most of time is waste in ***SMR***'s initailization. We will evaluation overhead of initialization time in section**??**.

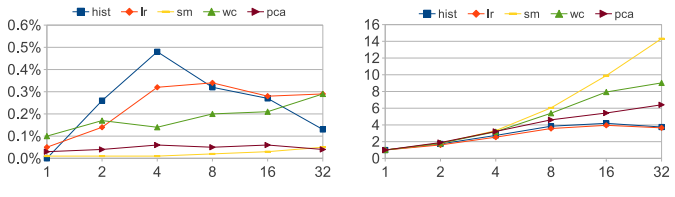

Figure 10. Scalability of SMR

Scalability

We say a waiting thread is the thread which is waiting for shared data that produced by another thread. If using semaphore for synchronization, a waiting thread will be blocked and enlisted on the waiting queue by the OS scheduler. If using spinlock for synchronization, a waiting thread will do busy wait, i.e. spinning in a while loop. Embedded software applications that using semaphore to handle the synchronization could result in performa+nce overkill, since it involves system calls translated into thousands of CPU instructions [1]; while using spinlock have problem of long busy-waiting time.

When a running thread tries to read shared data, it must do busy wait until it has exhausted its time-slice or until another thread has withdrawn the occupation on the shared data.

System performance is evaluated using instructions per cycle (IPC). Higher IPC means better performance. Figure **??** shows the IPC of Phoenix first increases and then decreases as more threads are run on multi-core system.

SMR Evaluation indicates that the locking overhead can be significantly reduced to less than 1

The performance of hist, wc, sm are scale linearly with the number of cores.

5.2 Overhead of SMR

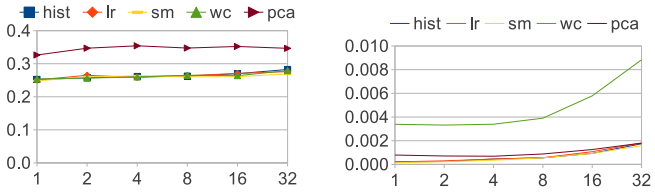

Figure 11. Initialize time of SMR and Phoenix

Figure11 shows the initialization time of ***SMR*** and Phoenix. The time of ***SMR*** used is range 0.25s to 0.35s, while it is just about 0.001s in Phoenix. why the initialization time is so large in ***SMR***

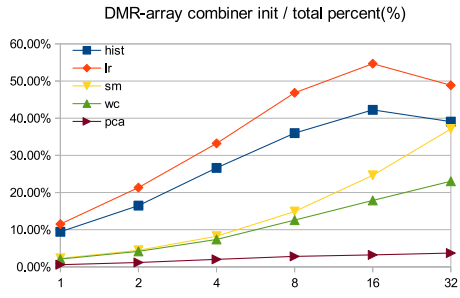

Figure 12. Initialize time of SMR and Phoenix

5.3 Diff buffer performance

As the number of threads increases (from 1 to 8), the execution time decreases for all cases. For 9 to 32 threads, both Phoenix and ***SMR*** execution times keep decreasing with the increase of the number of threads When the number of dependent processes increases above the number of cores, serious contending lock takes place, and as a resultexecution

time will be drastically increased. As the number of threads increases from 32 to 33, there is a significant increase in execution time owing to Opterons NUMA architecture with point-to-point communication links.

On the workstation, the ***SMR*** implementation is 13%-95% better than Phoenix implementation (peaking at 8 threads).

As the number of threads increases (from 1 to 8), the execution times for both Pthread/C and MPI programs decrease for both workstation and supercomputer. For 9 threads to 32 threads, Pthread/C and MPI execution times changes slightly on workstation. However for 9 threads to 32 threads, both Pthread/C and MPI execution times keep decreasing (in most cases) with the increase in number of threads on the supercomputer node. This is because we use 8 cores in the workstation while we use 32 cores in the supercomputer node. Execution time for large number of threads (9 to 33 and beyond) remains almost the same for the workstation because of no communication overhead. Results (see Figure 4) also show that MPI execution time on supercomputer increases significantly when the number of threads is increased from 32 to 33 and beyond; but Pthread/C execution time on supercomputer remains almost unchanged. This is due to the communication overhead among the cores (32 cores in a supercomputer node) when using MPI message passing.

From the experimental results, it is observed that both computer systems yield better performance using Pthread than MPI (in most cases). Pthread has the advantage over MPI for sharing memory and distributing tasks among lightweight threads instead of processes. The performance of the Pthread implementation on the supercomputer is the best, 30better than the Open MPI implementation on the supercomputer node.

6. Related Work

MapReduce is a popular distributed framework for massive-scale parallel data analysis. There are many existing implementation of MapReduce which adopt on the basic architecture and programming model of originally Google's MapReduce, such as Hadoop[], Dryad[11].

The Phoenix MapReduce libray[14] is the most relevant work to ***SMR***. Phoenix demostrate that MapReduce is a promising parallel programming models in muliticore and multiprocess systems. It creates a thread pool by Pthreads and can schedule tasks dynamically to support itrative applications. ***SMR*** differs from Phoenix mainly on that Phoenix need barrier between iterative MapReduce, while ***SMR*** brokes barrier to speed up computing.

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.

7. Conclusions And Furture Work

We have presented ***SMR***, a scalable iterative MapReduce model, to support efficient iterative processing.

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 NUMA-aware scalable iterative processing. And supporting online-MapReduce.

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] 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`.

[4] 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.

[5] 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.

[6] 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.

[7] 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`.

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

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

[10] 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.

[11] 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`.

[12] 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. .

[13] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing mapreduce for multicore architectures. *Computer Science and Artifi-*

cial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep, 2010.

[14] 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.

[15] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *2nd MapReduce*, pages 9–16, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0700-0. . URL http://doi.acm.org/10.1145/1996092.1996095.