Tiled-MapReduce: Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling

Rong Chen, Haibo Chen, and Binyu Zang
Parallel Processing Institute
Fudan University
{chenrong, hbchen, byzang}@fudan.edu.cn

ABSTRACT

The prevalence of chip multiprocessor opens opportunities of running data-parallel applications originally in clusters on a single machine with many cores. MapReduce, a simple and elegant programming model to program large scale clusters, has recently been shown to be a promising alternative to harness the multicore platform.

The differences such as memory hierarchy and communication patterns between clusters and multicore platforms raise new challenges to design and implement an efficient MapReduce system on multicore. This paper argues that it is more efficient for MapReduce to iteratively process small chunks of data in turn than processing a large chunk of data at one time on shared memory multicore platforms. Based on the argument, we extend the general MapReduce programming model with "tiling strategy", called *Tiled-MapReduce* (TMR). TMR partitions a large MapReduce job into a number of small sub-jobs and iteratively processes one sub-job at a time with efficient use of resources; TMR finally merges the results of all sub-jobs for output. Based on *Tiled-MapReduce*, we design and implement several optimizing techniques targeting multicore, including the reuse of input and intermediate data structure among sub-jobs, a NUCA/NUMA-aware scheduler, and pipelining a sub-job's reduce phase with the successive sub-job's map phase, to optimize the memory, cache and CPU resources accordingly.

We have implemented a prototype of *Tiled-MapReduce* based on Phoenix, an already highly optimized MapReduce runtime for shared memory multiprocessors. The prototype, namely Ostrich, runs on an Intel machine with 16 cores. Experiments on four different types of benchmarks show that Ostrich saves up to 85% memory, causes less cache misses and makes more efficient uses of CPU cores, resulting in a speedup ranging from 1.2X to 3.3X.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*

General Terms

Design, Performance

Keywords

MapReduce, Tiled-MapReduce, Tiling, Multicore

1. INTRODUCTION

Multicore or many cores, as another form embodying Moore's Law, is commercially prevalent recently. With the commodity availability of Quad-cores and eight cores on a chip, it is foreseeable that tens to hundreds (even thousands) of cores on a single chip will appear in the near future [1]. With the continuously increasing number of cores, it is vitally important to fully harness the abundant computing resources with still ease-to-use programming models.

MapReduce [2], designed to program large clusters with relatively simple functional primitives, has shown its power in solving non-trivial data-parallel problems such as document clustering, web access stats, inverted index and statistical machine translation. 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. Consequently, MapReduce frees programmers from handling tough tasks such as distributing data, specifying parallelism and fault tolerance.

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 [3], which is heavily optimized by Yoo et al. [4][1]. Phoenix uses the pthread library to assign tasks among CPU cores and relies on shared memory to handle inter-task communications. Compared to the cluster version, MapReduce on multicore is able to take advantage of fast inter-task communications in shared memory, thus avoids the expensive network communications among tasks.

Though Phoenix has demonstrated the applicability of running MapReduce on multicore, it is still limited in exploiting many features of commodity multicore systems due to its way of processing all (thus very large) input data at one time. As a result, the input and intermediate data will persist along the entire life cycle of a processing phase (i.e., *Map* or *Reduce*). Hence, a relatively large data-parallel application can easily cause resource pressures on the runtime, operating systems and the CPU caches, which could significantly degrade the performance. Based on the above observation, we argue that *it is more efficient to iteratively process small chunks of data in turn than processing a large chunk of data at one*

[1]Note that the version of Phoenix we use in this paper is 2.0.0 released in May, 2009, which was a best available implementation on shared memory systems.

time on shared memory multicore platforms, due to the potential of better cache/memory locality and less contentions.

To remedy the above problems, this paper proposes *Tiled-MapReduce*, which applies the *"tiling strategy"* [5] in compiler optimization, to shorten the life cycle and limit the footprint of the input and intermediate data, and to optimize the resource usages of the MapReduce runtime and mitigate contentions, thus increases performance. The basic observation is that the reduce function of many data-parallel applications can be written as commutative and associative, including all 26 MapReduce applications in the test suite of Phoenix [3] and Hadoop [6]. Based on this observation, *Tiled-MapReduce* further partitions a big MapReduce job into a number of small sub-jobs and processes each sub-job in turn. The runtime system will finally merge the results of each sub-job and output the final results. In *Tiled-MapReduce*, the runtime only consumes the resources required for a sub-job as well as the output for each sub-job, which are usually much smaller than those of processing one big task in a time.

Tiled-MapReduce also enables three optimizations otherwise impossible for MapReduce. First, 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. Second, processing a small sub-job provides the opportunity to fully exploit the memory hierarchy of a multicore system, resulting in better memory and cache locality. Finally, according to our measurements, the *Reduce* phase on a multicore machine is usually not balanced, even using dynamic scheduling. This gives us the opportunity to overlap the execution of a sub-job's *Reduce* phase with its successor's *Map* phase, which can fully harness the CPU cores.

The incremental computing nature of *Tiled-MapReduce* would also be beneficial to the online MapReduce model [7], which supports online aggregation and allows users to see the early results of an online job. It would also be useful to handle incremental computing that operates on the newly appended data and combines the new results with the previous results [8]. To support these two computing models , *Tiled-MapReduce* is also built with the support to periodically display the intermediate results after a sub-job is done, as well as the support for continuous computation that saves the partial results of sub-jobs for further computation reuse.

It should be noted that *Tiled-MapReduce* does not require a deep understanding of the underlying MapReduce implementation, thus is orthogonal to a specific MapReduce implementation (*e.g.*, the algorithm and data structures). *Tiled-MapReduce* also mostly retains the existing programming interfaces of MapReduce, with only two optional interfaces for the purpose of input reuse, which also have the counterparts in other MapReduce implementations such as Google's MapReduce [2] and Hadoop [6]. Further, *Tiled-MapReduce* retains the fault tolerance capability in MapReduce, and additionally allows the saving and restoring of data at the granularity of a sub-job, to defend against a whole machine crash, since current multicore systems lack separate hardware failure domains.

We have implemented a prototype of *Tiled-MapReduce* based on Phoenix. The system, called Ostrich, outperforms Phoenix due to the mentioned optimizations. Experiments on a 16-core Intel machine using four different types data-parallel applications (Word Count, Distributed Sort, Log Stats, and Inverted Index) show that: Ostrich can save up to 85% memory, causes less cache misses and makes more efficient uses of CPU cores, resulting in a speedup from 1.2X to 3.3X.

In summary, this paper makes the following contributions:

- The analysis that iteratively processing small chunks of data is more efficient than processing a large chunk of data for MapReduce on multicore platforms.
- The *Tiled-MapReduce* programming model extension that allows the exploits of the multicore environment for data-parallel applications.
- Three optimizations that optimize the memory, cache and CPU usages of the *Tiled-MapReduce* runtime.
- A prototype implementation with experimental evaluation, which demonstrates the effectiveness of *Tiled-MapReduce*.

The rest of the paper is organized as follows. Section 2 presents the background of MapReduce, the Phoenix implementation and the "tiling strategy" in parallel computing. Section 3 discusses the possible performance issues with Phoenix and illustrates the design spaces and optimization opportunities of MapReduce on multicore. Section 4 describes the extension from *Tiled-MapReduce* and its overall execution flow. Section 5 describes the optimization for the resource usages in *Tiled-MapReduce*. Section 6 presents the performance evaluation results. Section 7 relates our work to previous work. Finally, we conclude the paper with a brief discussion on future work in section 8.

2. BACKGROUND

This section presents a short review on the MapReduce programming model, uses Phoenix as an example to illustrate MapReduce for multicore platforms, and briefly describes the tiling strategy in compiler optimization.

2.1 MapReduce Programming Model

The MapReduce [2] programming model mostly only requires programmers to describe the computation using two primitives inspired by functional programming languages, *Map* and *Reduce*. The `map` function usually independently processes a portion of the input data and emits multiple intermediate key/value pairs, while the `reduce` function groups all key/value pairs with the same key to a single output key/value pair. Additionally, users can provide an optional `combine` function that locally aggregates the intermediate key/value pairs to save networking bandwidth and reduce memory consumptions.

Many data-parallel applications could be easily implemented with MapReduce model, such as *Word Count*, *Distributed Grep*, *Inverted Index* and *Distributed Sort* [2]. The pseudo-code in Figure 1 shows the *Word Count* application counting the number of occurrences of each word in a document. 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.

2.2 The Phoenix Implementation for Multicore

Phoenix [3] [4] is an implementation of MapReduce on shared memory multiprocessor systems using the pthread library. It showed that applications written with the MapReduce programming model have competitive scalability and performance with those directly written with the pthread library on a multicore platform.

The lower part of Figure 1 uses a flow chart to illustrate the outline of Phoenix from input to output, which goes through three main phases, including *Map*, *Reduce* and *Merge*. The right part illustrates the overall execution flow of processing a MapReduce job on Phoenix runtime. The key data structure of Phoenix runtime is the *Intermediate Buffer*, which is formed as a matrix of buckets

```
// input = a document
map ( input ) {
  for each word in input
    emit_intermediate ( word, 1 );
}

// key = word
// values = a set of values
combine ( key, values ) {
  int sum = 0;
  for each value in values
    sum += value;
  emit_combine ( key, sum );
}
reduce ( key, values ) {
  int sum = 0;
  for each value in values
    sum += value;
  emit ( key, sum );
}
```

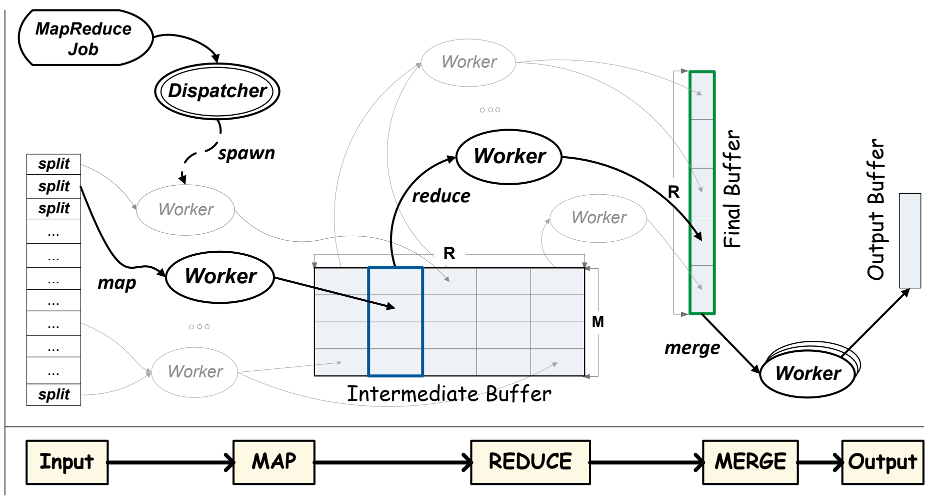

Figure 1: ***Pseudo-code of Word Count on MapReduce and the execution flow of the Phoenix Library: the Map workers generate output to the rows of the Intermediate Buffer and the Reduce workers aggregate the columns of the Buffer to generate output to the Final Buffer.***

and stores the intermediate data produced in the *Map* phase and consumed by the *Reduce* phase. 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 worker in the *Reduce* phase. A MapReduce application starts a job by invoking the dispatcher, which spawns multiple workers and binds them to different CPU cores. In the *Map* phase, each worker repeatedly splits a piece of input data and processes them using the programmer-supplied `map` function. The `map` function parses the input data and emits multiple intermediate key/value pairs to the corresponding row of *Intermediate Buffer*. The runtime also invokes the `combine` function (if provided by users) for each worker to perform local reduction at the end of the *Map* phase. In the *Reduce* phase, each worker repeatedly selects a reduce task, which sends the intermediate data from the corresponding column of *Intermediate Buffer* to the programmer-supplied `reduce` function. It processes all values belonging to the same key and generates the final result for a key. In the *Merge* phase, all results generated by different reduce workers are merged into a single output sorted by key.

2.3 Tiling Strategy in Compiler Optimization

The tiling strategy, also known as *blocking*, is a common technique to efficiently exploit the memory hierarchy. It partitions data to be processed into a number of blocks, computes the partial results of each block, and merges the final results. The tiling strategy is also commonly used in the compiler community to reduce the latency of memory accesses [5] and increase the data locality. For example, loop tiling (also known as loop blocking) is usually used to increase the data locality, by partitioning a large loop into smaller ones. Several variations of tiling are also used to optimize many central algorithms in matrix computations [9], including the fixed-size tiling, the recursive tiling, and a combination of them.

3. OPTIMIZING OPPORTUNITIES OF MapReduce ON MULTICORE

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 data, which would be common for machines with abundant memory and CPU cores.

This problem is not due to the implementation techniques of Phoenix. In fact, Phoenix has been heavily optimized from three layers: *algorithm*, *implementation* and *OS interaction* [4], which results in a significant improvement over its initial version. We attribute the performance issues to mainly the *programming model* of MapReduce on multicore, which process all input data at one time.

First, 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, the contentions on cache and shared data structures, instead of networking communications, are the major performance bottlenecks processing large MapReduce jobs on multicore.

Second, there is a strict barrier between the Map and the Reduce phase, which requires the MapReduce runtime to keep all the input and intermediate data through the *Map* phase. This requires a large amount of resource allocations and comes with a large memory footprint. For relatively large input data, this creates pressure on the memory systems and taxes the operating systems (*e.g.*, memory management), which are with imperfect performance scalability on large scale multicore systems [10]. Further, it also limits the effects of some optimizations (such as the combiner) due to restricted cache and memory locality. For example, the combiner interface, which is the key to reduce networking traffic of MapReduce applications in the cluster environments, was shown to make little performance improvement along with other optimizations (e.g., prefetching) [4].

Third, cache and memory accesses in current multicore systems tend to be non-uniform, which makes exploiting memory hierarchy even more important. Unlike in cluster environments, MapReduce workers in a multicore platform can be easily controlled in a centralized way, making it possible to control memory and cache accesses in a fine-grained way.

Finally, many resources of a commodity multicore platform are universally shared and accessible to MapReduce workers, which makes global failures (*e.g.*, failures resulting in a whole machine crash) more pervasive. Hence, it is not suitable to simply adopt the fault-tolerance model of clusters that mainly focuses on single worker failures.

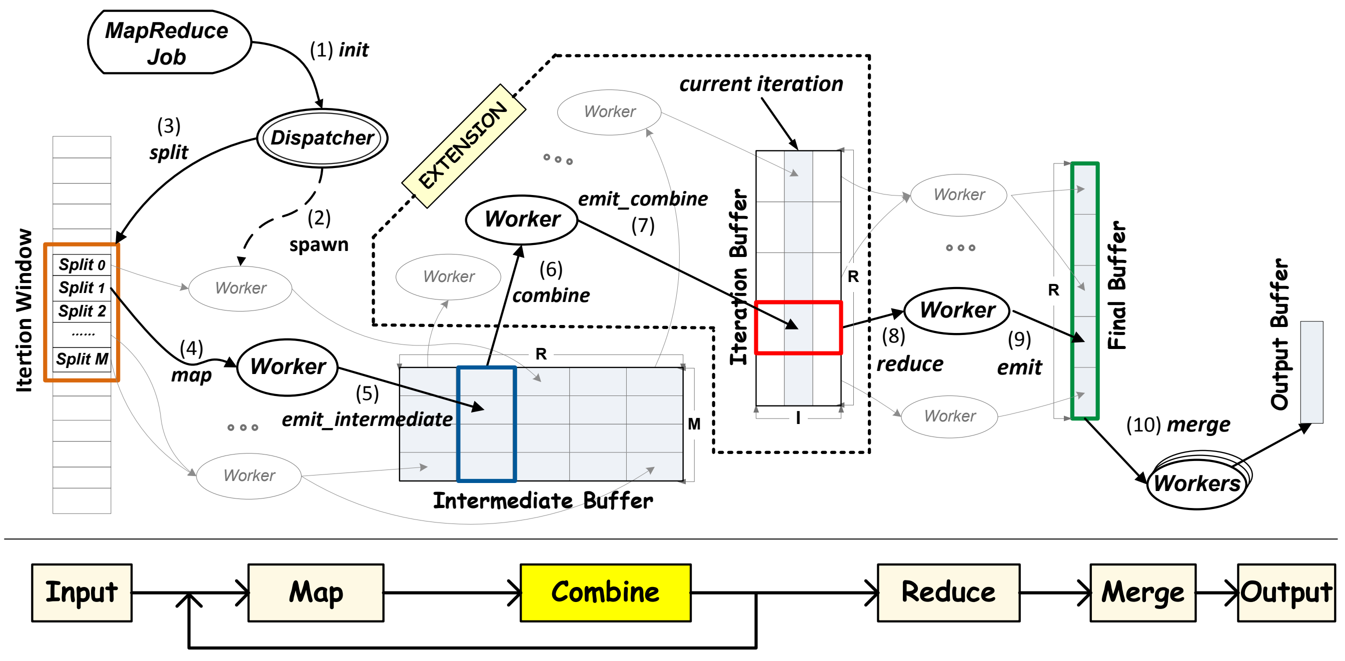

Figure 2: ***Execution Flow of Tiled-MapReduce.***

4. TILED-MAPREDUCE

This section describes the extension to the MapReduce programming model and shows the major changes to runtime to support *Tiled-MapReduce*, as well as the execution flow of *Tiled-MapReduce*.

4.1 Extensions to MapReduce

Being aware of the life-cycle problem with MapReduce on multicore, we extend the general MapReduce programming model with *Tiled-MapReduce*. It uses the "tiling strategy" and divides a large MapReduce job into a number of independent small sub-jobs and iteratively processes one sub-job at a time with efficient uses of resources, given that the reduce interface for many data-parallel applications can be implemented as commutative and associative.

To support iterative processing of a number of sub-jobs derived from a large MapReduce job, *Tiled-MapReduce* replaces the general *Map* phase with a loop of *Map* and *Reduce* phases. In each iteration, *Tiled-MapReduce* processes a sub-job and generates a partial result, which can be either saved for computation reuse [8] or provided for users to know the status of the computation [7]. *Tiled-MapReduce* also extends the general *Reduce* phase to process the partial results of all iterations, rather than the intermediate data. The output generated by the *Reduce* phase is compatible with the output of the general *Reduce* phase. To differentiate with that in the final *Reduce* phase, we rename the *Reduce* phase within one sub-job as the *Combine* phase. The lower part of Figure 2 illustrates the processing phases in *Tiled-MapReduce*.

Note that though it was claimed that the combiner interface made little performance improvements on multicore in the context of the general MapReduce programming model with prefetching of data in the reduce phase. *Tiled-MapReduce* mainly uses such an interface for the purpose of shortening the life cycle of input and intermediate data, which could improves the cache and memory locality and enable further optimizations in section 5.

4.2 Fault Tolerance and Computation Reuse

Being aware of the global failures (e.g., whole machine crash) in a multicore platform and the possible need of computation reuse, Ostrich, *Tiled-MapReduce* extends the fault tolerance model with the support of backing up the results of individual sub-jobs to save the associated computation during a global failure or computation reuse. Generally, each sub-job in Ostrich can backup their partial results to persistent storage (e.g., disk, NFS) independently and in parallel. Ostrich uses self-certified pathname [11] to bookkeep sub-jobs that are backed up. During recovering from a global failure, *Tiled-MapReduce* could load the saved results and only compute those that are not book-kept, which saves a lot of time associated with re-execution. The detailed implementation of fault tolerance and computation reuse in Ostrich is hidden to user programmers, such as I/O parallelism and naming. They are only required to implement two interfaces that are pervasive in MapReduce for clusters (such as Google's MapReduce [2] and Hadoop [6]), `serialize` and `deserialize`, which converts key/value pairs to and from a character stream. In most cases, user programmers can simply use the default `serialize` and `deserialize` functions provided by *Tiled-MapReduce* framework.

4.3 Execution Flow

The top part of Figure 2 illustrates the overall execution flow of a *Tiled-MapReduce* job and the implementation of *Tiled-MapReduce* runtime.

A *Tiled-MapReduce* job is initiated by invoking the `mr_dispatcher` function, which initializes and configures the *Dispatcher* according to the arguments and runtime environments (*e.g.*, available resources). Then, the *Dispatcher* spawns *N Workers* and binds them to CPU cores. The *Dispatcher* also iteratively splits a chunk from input data in the *Iteration Window*, whose size is dynamically adjusted according to the runtime configuration. The chunk of data will be further split into *M* pieces, which forms *M* map tasks.

In the *Map* phase, a *Worker* selects a map task from the *Iteration Window* whenever it is idle, and invokes the programmer-provided `map` function, which parses the input data and generates intermediate key/value pairs. The `emit_intermediate` function provided by the runtime will be invoked to insert a key/value pair to *I*ntermediate Buffer, which is organized as an *M* by *R* matrix of buckets, where *R* is the number of reduce tasks.

In the *Combine* phase, the *Workers* select the reduce tasks in turn and invokes the programmer-provided `combine` function to process a column in the *Intermediate Buffer*. The structure of the *Iteration Buffer* is an *I* by *R* matrix of buckets, where *I* is the total number of iterations.

When all sub-jobs have finished, the *Workers* invoke the programmer-provided `reduce` function to do the final reduce operation on the data in each column of the *Iteration Buffer*. The `reduce` function inserts the final result of a key to *Final Buffer* by invoking the `emit` function. Finally, the results from all reduce tasks are merged and sorted into a single *Output Buffer*.

5. OPTIMIZING RESOURCE USAGES

The support of iterative processing of small sub-jobs allows fine-grained management of resources among sub-jobs. This opens opportunities for new optimizations on resource usages in *Tiled-MapReduce*. In this section, we describe three optimizations that aim at improving the memory efficiency, data locality and task parallelism.

5.1 Memory Reuse

In general MapReduce programming model, the input data is kept in memory in a MapReduce job's whole life cycle. The intermediate data is also kept in memory in a whole MapReduce phase. These create significant pressure to the memory systems. *Tiled-MapReduce* mitigates the memory pressure due to intermediate data by limiting the required memory for the input and intermediate data to a sub-job level. However, it also requires frequent allocations and deallocations of memory along with the sub-job creation and destruction. To reduce such overhead, Ostrich is designed to reuse the memory for *input data* and *intermediate data* among sub-jobs.

5.1.1 Input Data Reuse

The input data in MapReduce is usually required to be kept in memory along with the whole MapReduce process. For example, in Word Count, the *intermediate buffer* will keep a pointer to a string in the input data as a key, instead of copying the string to another buffer. For a relatively large input, the input data will be one of the major consumers of memory and there is poor cache locality in accessing them.

Ostrich is designed to balance the benefit of a large memory footprint and the cost of additional memory copies. For applications that likely have abundant duplicated keys (or values), it would be more worthwhile to copy the keys (or values) instead of keeping a pointer to the input data. Copying the keys (or values) together can also improve the data locality, as the copied keys (or values) can be put together, instead of being scattered randomly across the input data. Hence, Ostrich allows the copy of keys (or values) at the `combine` phase.

By breaking the dependency between the intermediate data and the input data, Ostrich supports the reuse of a fix-sized memory buffer to hold the input data for each sub-job. The memory buffer is remapped to the corresponding portion of input data when a new sub-job starts.

Optional Interfaces: `acquire` and `release`. To support dynamically acquiring and releasing input data, Ostrich provides two optional interfaces, namely `acquire` and `release`, which will be invoked at the beginning and end of each sub-job. An example of the `acquire` and `release` functions for the Word Count application is shown as below. In the example, the `acquire` and `release` functions simply map and unmap a portion of the input file into/from memory.

```
// input = data structure of a document
acquire ( input, offset, len ) {
   input→flen = len;
   input→fdata = mmap ( 0, len, PROT,
            MAP_PRIVATE, input→fd, offset );
}
release ( input ) {
   munmap ( input→fdata, input→flen );
}
```

These two interfaces are not provided from scratch. In fact, there are counterparts of the `acquire` and `release` interfaces in Google's own implementation and Hadoop. For example, Hadoop requires programmers to implement the `RecordReader` interface to process a piece of input split generated from the `InputSplit` interface. The `acquire` interface resembles the `constructor` function of `RecordReader` interface, which is used to get data from a piece of input. The `release` interface also resembles the `close` function of the `RecordReader` interface, which closes a piece of input. Nevertheless, the two interfaces are optional, as Ostrich has provided multiple default implementations for input with general data structures like the Google's MapReduce and Hadoop, in order to reduce the effort of programmers.

5.1.2 Intermediate Data Reuse

The use of *Tiled-MapReduce* provides opportunities to reuse the *Intermediate Buffer* among sub-jobs. This could save the expensive operations such as concurrent memory allocation and deallocation, as well as the building of data structures. Instead of constructing an *Intermediate Buffer* for each sub-job, Ostrich uses a global buffer to hold the intermediate data generated in the *Map* phase. Ostrich will indicate that the buffer is empty at the end of a sub-job, but will not free the memory until all sub-jobs have finished.

It is possible that the *Iteration Buffer* used to store partial results from all iterations is too large for the final *Reduce* phase. Hence, Ostrich adds a new internal phase, namely *Compress*, to further combine the partial results among sub-jobs. When the size of partial results exceeds a threshold (*e.g.*, 32), the *Compress* phase will invoke the `combine` function to combine the partial results among sub-jobs. The results generated in the *Compress* phase are also stored in the *Iteration Buffer*. The previous partial results will be simply discarded and the memory space is reused for the subsequent sub-jobs. It should be noted that the *Compress* phase is an internal phase of Ostrich and thus is transparent to programmers.

5.2 Exploiting Locality

5.2.1 Data Locality

The memory access pattern of MapReduce inherently has poor temporal and spatial locality. With regards to temporal locality, a MapReduce application usually sequentially touches the whole input data only once in the *Map* phase to generate the intermediate data. It also randomly touches discrete parts of intermediate data multiple times in the *Map* and *Reduce* phases to group key/value pairs and generates the final results. For spatial locality, though the input data is sequentially accessed in the *Map* phase, a large number of *key-compare* operations results in poor spatial locality. Even worse, each reduce task accesses the key/value pairs generated by different map workers in different time, causing poor spatial locality in the *Reduce* phase and even severe thrashing when the physical memory is exhausted.

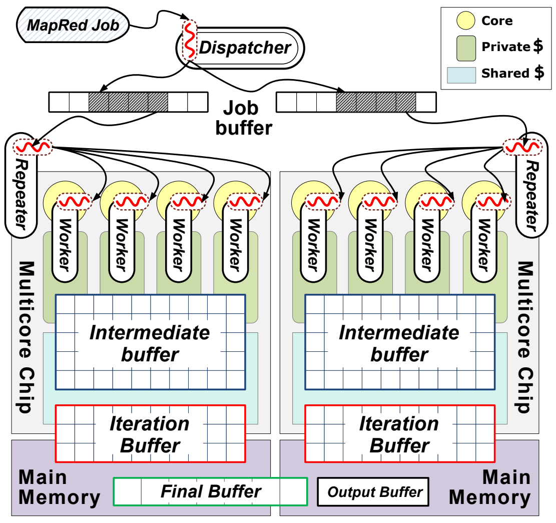

Figure 3: ***The logical view of the multicore oriented scheduling model used by Tiled-MapReduce.***

Tiled-MapReduce provides opportunities to improve data locality for data-parallel applications. As each time only a small sub-job is handled, the working set of the sub-job could be relatively small. A small working set is beneficial to exploit the cache hierarchy in a multicore platform. Specifically, Ostrich estimates the working set of a sub-job by first running a sample sub-job to collect its memory requirements. Based on the collected data, Ostrich automatically estimates the size of each sub-job according to the cache hierarchy parameters. Currently, Ostrich tries to make the working set of each sub-job fit into the last-level cache.

5.2.2 *NUCA/NUMA-Aware Scheduler*

Currently commodity multicore hardware usually organizes caches in a non-uniform cache access (NUCA) way. The memory also tends to be organized in a non-uniform way (i.e., NUMA) as the number of cores increases. Thus, the latency of accessing caches or memory on remote chips is much slower than that on local cache [10]. Further, the cost of synchronization among threads on different cores also increases with the growing of cores. In a MapReduce runtime, even if the working set of each sub-job fits into the local cache, the cross-chip cache accesses still cannot be avoided in the *Combine* phase. Thus, current MapReduce schedulers (e.g., Phoenix) have problems in scaling well on multicore systems, mainly due to the fact that they use all cores belonging to different chips to serve a single MapReduce job.

According to our measurement on Phoenix, MapReduce *scales better with the increase of the input size than on the number of cores*. For example, the total execution time (66s) of the Word Count application using four cores to process 1 Gbyte input four times and merging the final results is notably less than the time (80s) of using 16 cores to process 4 Gbyte input once. Hence, we propose a NUCA/NUMA-aware scheduler that runs each iteration on a single chip and allocates memory from local memory pool, which could significantly reduce the remote cache and memory access in the *Combine* phase. NUCA/NUMA-aware scheduler also increases the utilization of caches by eliminating the duplication of input data and intermediate data among the many core caches.

Figure 3 provides the overview of the NUCA/NUMA-aware scheduler. The main thread as the *Dispatcher* spawns a *Repeater* thread for each chip (a scheduling group), which further spawns *Worker* threads on each cores within the chip. Each *Repeater* has a private *Job Buffer* to receive sub-jobs from *Dispatcher*, a private *Intermediate Buffer* used by workers to store the output of the *Map* phase and a private *Iteration Buffer* to store the partial results generated by workers in the *Combine* phase.

Using a NUCA/NUMA-aware scheduler might cause some imbalance among *Repeaters* in different scheduling groups, due to unbalanced workloads assignment to these groups. To balance workload among them, Ostrich implements the work-stealing mechanism proposed by Cilk [12], to allow a *Repeater* to actively steal jobs from other scheduling group when it has done all the local jobs.

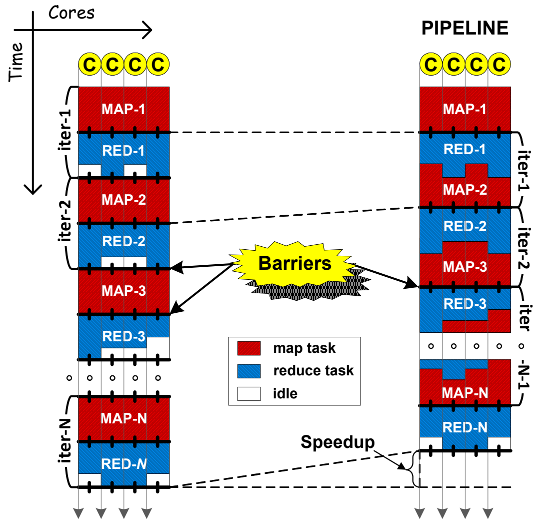

Figure 4: ***The task pipeline optimization of Tiled-MapReduce: the left figure shows the original execution flow, while the right figure depicts the execution flow after the task pipeline optimization.***

5.3 Task Parallelism and Thread Reuse

5.3.1 *Software Pipeline*

In general MapReduce programming model, 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. The imbalance of tasks can be solved by dynamic scheduling in the *Map* phase. However, in the *Reduce* phase, as all values for the same key must be in one reduce task, it is not always feasible to generate a large number reduce tasks for dynamic scheduling. For example, applications with only a small number of keys can only generate a small number of reduce tasks. Moreover, the workload of each key could be imbalanced as the number of key/value pairs with the same key can vary significantly.

Tiled-MapReduce uses software pipeline to create parallelism among adjacent sub-jobs, since there is no data dependency between one sub-job's *Reduce* phase and its successor's *Map* phase. Figure 4 illustrates the Pipeline optimization in *Tiled-MapReduce*. The y-axis is the time and the x-axis is a list of cores. The left of the figure is the execution flow of the normal *Tiled-MapReduce* runtime, which has strict barriers between each phase and can have many idle states due to the imbalance of tasks. The right of figure is the execution flow with Pipeline optimization, which overlaps the *Reduce* phase of the current sub-job and the *Map* phase of its successor.

As the reuse of *Intermediate Buffer* among sub-jobs also creates resource dependency between adjacent sub-jobs, Ostrich uses a dual-buffer to eliminate the dependency. This will increase the memory consumption, but can shorten the total execution time. Users can avoid the use of a dual-buffer by disabling the software pipeline optimization.

5.3.2 Thread Pool

The partition of a large job into a number of sub-jobs increases the number of thread creations and destructions. To avoid such overhead, Ostrich uses a worker thread pool to serve tasks in all phases within a sub-job. The worker thread in the pool is also reused among sub-job boundaries. The pool is initialized at the beginning of a MapReduce job and serves all tasks in all phases, including the final *Reduce* and *Merge* phase.

6. EVALUATION

This section presents the experimental results for Ostrich on an Intel multicore machine. We mainly compare the performance of Ostrich with Phoenix, currently the best available MapReduce implementation on shared memory systems. We also compare Ostrich with Hadoop, as Hadoop can also be executed in a single machine with multicore.

All experiments were conducted on an Intel 16-Core machine with 4 Xeon 1.6GHz Quad-cores chips. Each chip has two 8-way 2 Mbyte L2 caches and each L2 cache is shared by two cores in the same die. The size of the physical memory is 32 Gbyte. We use Debian Linux with kernel version 2.6.24.3, which is installed on a 137G SCSI hard disk with an ext3 file system. All input data are stored in a separated partition in a 300G SCSI disk. The memory allocator for both Ostrich and Phoenix is the Streamflow allocator [13], which was shown to have good scalability on multicore platforms. We run the experiment five times and report the average result.

6.1 Tested Applications

As the performance benefit of *Tiled-MapReduce* is sensitive to characteristics of key/value pairs of MapReduce applications, we chose four different MapReduce applications representing four major types of MapReduce applications regarding the attributes of key/value pairs. Word Count has a large number of keys and each key has many duplicated key/value pairs, for which *Tiled-MapReduce* is expected to have significant performance speedup. Distributed Sort also has a large number of keys, but there is no duplicate among key/value pairs. In such a case, *Tiled-MapReduce* is expected to show still notable performance improvements. Log Statistics has a large number of key/value pairs, but with only a few numbers of keys. Inverted Index has only a few key/value pairs with no duplicated pairs. For these two cases, *Tiled-MapReduce* is expected to have relatively less performance improvement because there is very few numbers of keys or key/value pairs.

The following are brief descriptions of four benchmarks. If not mentioned, the `combine` function is the same to `reduce` function.

Word Count (WC) It counts the number of occurrences of each word in a document. The `key` is the word and the `value` is the number of occurrences. The `map` function emits a $\langle word,\ 1\rangle$ pair for each word in document and the `reduce` function sums the values for each word and emits a $\langle word,\ total\ count\rangle$ pair.

Distributed Sort (DS) It models the TeraSort benchmark [14] and uses the internal sorting logic of a MapReduce runtime to sort a set of records. The `key` is the key of the record and the `value` is the record itself. The `map` function emits a $\langle key,\ record\rangle$ pair for each record and the `reduce` function leaves all pairs unchanged. The sorting of records happens in both the reduce and the merge phase, which perform the local and global sorting according to keys, respectively.

Log Statistics (LS) It calculates the cumulative online time for each user from a log file. The `key` is the user ID and the `value` is the time of login or logout. The `map` function parses the log file and emits $\langle user\ ID,\ \pm time\rangle$ pairs. The `reduce` function adds up the time for each user and emits a $\langle user\ ID,\ total\ time\rangle$ pair.

Inverted Index (II) It generates a position list for the word a user specified in a document. The `key` is the word and the `value` is its position. The `map` function scans the document and emits $\langle word,\ position\rangle$ pairs. The `combine` function emits all pairs unchanged. The `reduce` function sorts all positions of the word and emits a $\langle word,\ list(position)\rangle$ pair.

Modification	WC	DS	LS	II
Input Data Reuse	11 + 3	Default	Default	11 + 3
Fault Tolerance	Default	Default	Default	Default

Table 1: ***The required code modification for Tiled-MapReduce, in addition to the code in Phoenix. "Default" indicates that programmers simply use the default implementation provided by Ostrich.***

Table 1 lists the code modifications to port a MapReduce application in Phoenix to Ostrich. To support input data reuse, programmers need to provide 11 and 3 lines of code for the acquire and release functions for WC, which maps/unmaps a portion of the input file into/from memory accordingly. Such code can be written by simply reusing the code that processes input data in Phoenix. For II, the acquire and release function is identical to WC, as the input type is the same with WC. For all the four applications, programmers can simply use the default `serialize` and `deserialize` functions provided by the runtime to support fault tolerance.

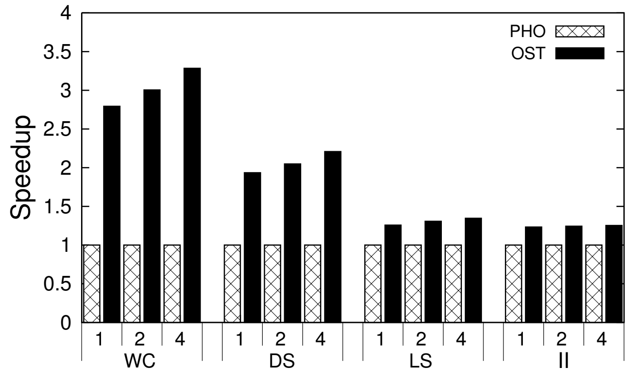

Figure 5: ***Overall speed over Phoenix using 16 cores with 1, 2 and 4 Gbyte input***

6.2 Overall Performance

Figure 5 shows the speedup of Ostrich compared to Phoenix. The input size for each benchmark is 1, 2 and 4 Gbyte. As shown in the figure, the largest speedup comes from Word Count, in which Ostrich outperforms Phoenix by 2.79X, 3X and 3.28X for the input size of 1, 2 and 4 Gbyte accordingly. While for Distributed Sort,

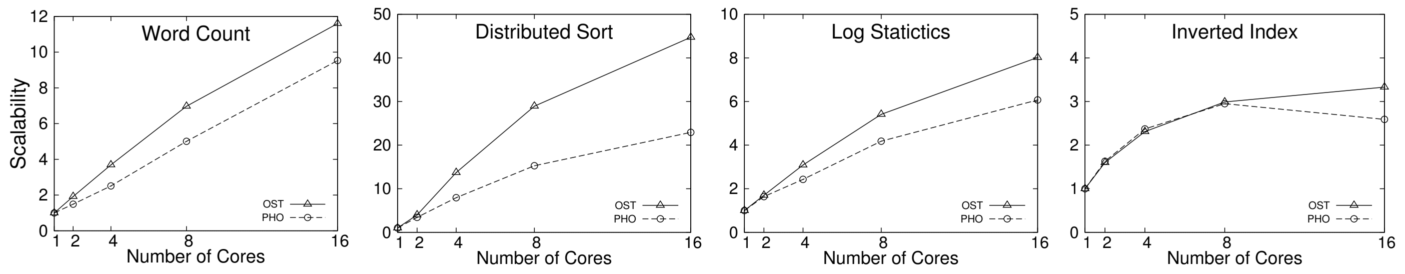

Figure 6: ***A comparison of the scalability of Ostrich with that of Phoenix.***

the speedup is 1.93X, 2.05X and 2.21X accordingly. The speedup is relatively small for Log Statistics (1.26X, 1.31X and 1.35X) and Inverted Index (1.23X, 1.24X and 1.25X). This is because the first two applications have relatively larger number of intermediate data and keys, which expose more optimization space compared to the later two.

Figure 6 compares the scalability of Ostrich with that of Phoenix. As shown in the figure, Ostrich has much better scalability than Phoenix for the four tested applications. With an increasing number of cores, Ostrich enjoys much larger speedup compared to the performance data on a single core. This confirms the effectiveness of *Tiled-MapReduce* and the associated optimizations. It should be noted that the scalability ratio of DS exceeds the number of cores. This is due to the fact that DS spends most of the execution time in the *Merge* phase, whose performance is much sensitive to the cache locality. As the number of cores increases, the size of total caches increases and the working set for each core decreases, which lead to less cache misses and thus a super-linear speedup. Further, the performance of II actually degrades when the number of cores increases from 8 to 16, due to the poor scalability of the *Reduce* phase and the increased time spent on such a phase.

In the following sections, we will categorize the source of the speedups and improved scalability.

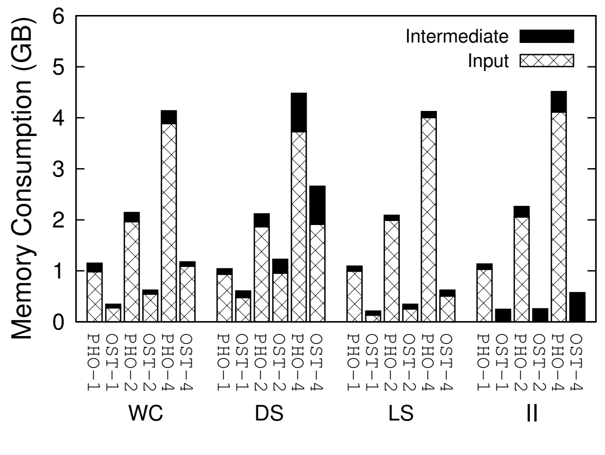

Figure 7: ***The benefits of input and intermediate data reuse: a comparison of the memory consumption for Ostrich to Phoenix with 1, 2 and 4 Gbyte input. OST-N and PHO-N represent the performance data for Ostrich and Phoenix with N Gbyte input.***

6.3 The Benefit of Memory Reuse

We first investigate the benefit of the memory reuse optimization. In Figure 7, we compare the memory consumption of the four benchmarks on Ostrich versus Phoenix with the input size of 1, 2 and 4 Gbyte. The main memory consumption in Phoenix is from input data which are kept in memory in a MapReduce job's whole life cycle. Due to the input reuse optimization, Ostrich significantly reduces the memory consumption for the input data. For the intermediate data, as Phoenix has already applied the combine optimization to reduce the memory space for intermediate data, Ostrich only has a small performance improvement for WC. For DS and II, as the key/value pairs cannot be locally combined, Ostrich shows no saving in intermediate data but additionally adds a small space overhead due to the copied keys and the spaces for Iteration Buffer. For LS, as the intermediate data is relatively small, Ostrich shows no space saving for intermediate data. Anyway, copying keys and grouping data into a small centralized intermediate buffer do increase the cache locality.

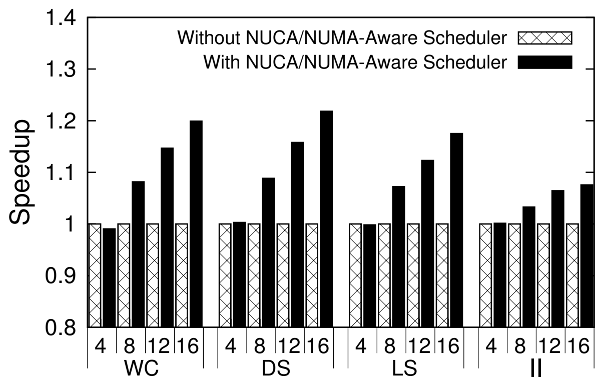

Figure 8: ***The effect of NUCA/NUMA-Aware Scheduler on four benchmarks for 4 to 16 cores with 1 Gbyte input.***

6.4 The Benefits of Exploiting Cache Locality

6.4.1 Improvement from NUCA/NUMA-Aware Scheduler

We compared the performance of Ostrich in two configurations. The configuration without NUCA/NUMA-aware scheduler partitions all threads into a single group and makes them sequentially dispose sub-jobs together, which is also used by Phoenix. By contrast, the NUCA/NUMA-aware scheduler partitions cores on the same chip into the same group and dispatch sub-jobs in parallel.

Figure 8 shows the effect of NUCA/NUMA-aware scheduler on four benchmarks with 4, 8, 12 and 16 cores. The improvement of NUCA/NUMA-aware scheduler is from the elimination of cross-chip cache access in *Combine* phase. WC, GS and LS benchmarks spend relatively more time in *Combine* phase to reduce the intermediate data, resulting in more performance improvement. Hence, the first three applications (i.e., WC, DS, and LS) enjoy around

20% performance speedup (19%, 22% and 19% accordingly) for 16 cores. Inverted Index has relatively smaller speedup (8%) because there are relative few operations in *Combine* phase, so the benefit from scheduler is limited. As Ostrich treats all cores within a chip as a group, the behavior of the new scheduler is the same to original scheduler when the number of cores is less than or equal to 4 (the number of cores in a chip). Hence, there is little performance improvement.

For all the four applications, the performance improvement increases with the number of cores. Thus, it is reasonable to foresee that Ostrich could have even more improvement for future multicore with abundant cores, especially for the tiled architecture [15].

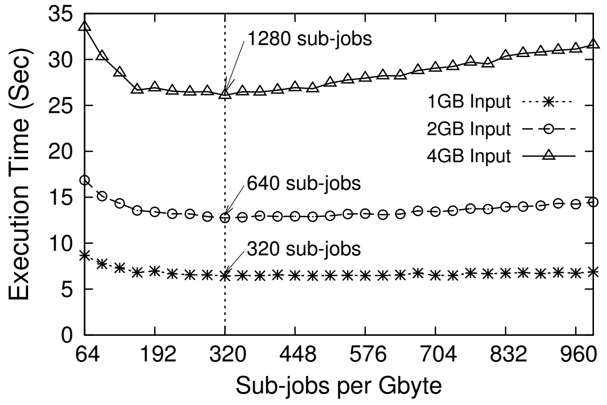

Figure 9: ***The impact of iteration size on execution time using 16 cores with 1, 2 and 4 Gbyte input.***

6.4.2 Relevance of Iteration Size

Tiled-MapReduce provides good opportunities to exploit the memory hierarchy, by limiting the footprint of a sub-job within a certain range. Figure 9 shows the execution time of the Word Count application with the number of sub-jobs to process each 1 Gbyte input data using 16 cores. The results shows that WC benchmark has the best performance when the each iteration size is close to 3.2 Mbyte (320 sub-jobs per Gbyte), for 1, 2 and 4 Gbyte input data. This is because the size of the overall last level caches (i.e., L2) in a chip is 4 Mbyte. The rest space of the cache is used to store other data such as intermediate buffer. Thus, *Tiled-MapReduce* tends to enjoy the best performance when the footprint of a sub-job just fit in the last level cache, as creating more sub-jobs would suffer more from the associated overhead with merging these sub-jobs. One can use only a small sample input to estimate the size of each sub-job to get an optimal job partition.

6.4.3 Improved Cache Locality

Figure 10 presents the L2 cache miss rate of four benchmarks on Ostrich versus Phoenix using 16 cores with 1, 2 and 4 Gbyte input. The L2 cache miss rate is collected using OProfile [16], which indicates the total number of L2 cache misses divided by the total number of retired instructions. The miss rate of WC, DS and LS benchmark running on Ostrich is from 3.1X to 7.1X fewer compared to that on Phoenix. As the working set of each sub-job matches the total L2 cache size of a chip (i.e., 4 Mbyte), thus it ensures that the data fetched in *Map* phase could be reused in the *Combine* phase within private cache of a chip and unlikely be accessed after being flushed to memory. The cache miss rate of II benchmark for Ostrich is close to that of Phoenix, as there are few optimizing spaces in II due to the fact there is very few intermediate

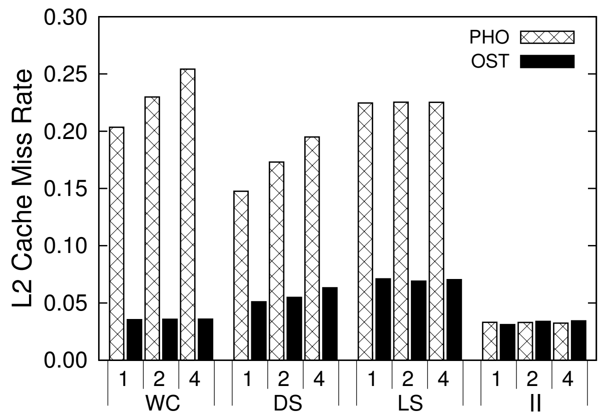

Figure 10: ***A comparison of the L2 miss rate on Ostrich to Phoenix using 16 cores for four benchmarks with 1, 2 and 4 Gbyte input.***

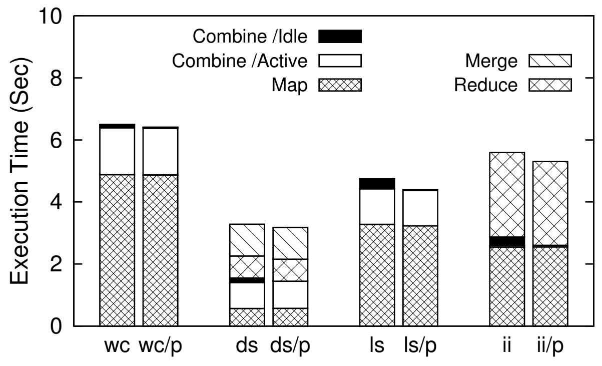

Figure 11: ***The benefit of Pipeline with a breakdown of time in all phases using 16 cores with 1 Gbyte input. The label '/p' means with Pipeline.***

data and the *Combine* phase could not decrease the workload in *Reduce* phase.

6.5 The Benefit of Pipeline

As the major benefit of Pipeline optimization is from the elimination of wait time in the *Combine* phase, we further split the time spent on the *Combine* phase into *active* time and *idle* time. The *active* time is the average of total execution time of all workers during *Combine* phase, and the *idle* time is the difference between the *active* time and the time spent on *Combine* phase.

Figure 11 shows the time breakdown of benchmarks with and without Pipeline optimization using 16 cores with 1 Gbyte input. The *idle* time dominates the *Combine* phase of II (83%), because it only indexes one word. Thus only one worker is active in the *Combine* phase, which results in a notable imbalance. For LS, the number of login records of each User ID is remarkably different, so the workload of each worker in *Combine* phase is imbalanced. Hence, Pipeline brings a relative large improvement, reducing 21% of time spent on the *Combine* phase. For DS, the default partition function cannot thoroughly balance the workload, thus Pipeline still have some benefit, close to 15%. For WC, which has a large number of keys and duplicated pairs for dynamic load balancing, the improvement from Pipeline is limited (5%).

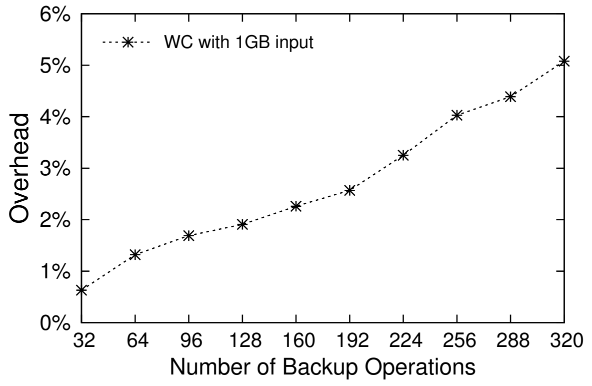

Figure 12: ***The performance overhead of backup operations for WC benchmark using 16 cores with 1 Gbyte input. Each backup operation writes the unsaved partial results of all sub-jobs.***

6.6 Fault Tolerance

Figure 12 shows the relation between performance overhead and the number of backup operations, for WC benchmark using 16 cores with 1 Gbyte input. The performance overhead of the fault tolerance mechanism depends on the frequency of backup operations. In the tested configuration using 1 Gbyte input, there are 320 sub-jobs in the optimal configuration. Performing backup operations on every 10 sub-jobs incurs less than 1% performance overhead, while doing backup operations after each sub-job incurs about 5.1% overhead. Thus, there is a tradeoff between the cost of backup and the benefit from the efficiency of recovery from fault.

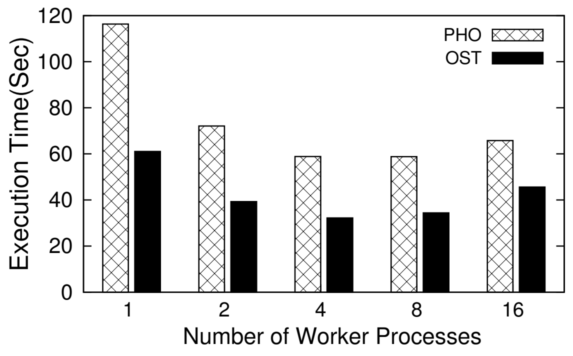

Figure 13: ***A comparison on the throughput of MapReduce servers based on Ostrich and Phoenix with the number of worker processes from 1 to 16. The workload has 100 mixed MapReduce jobs, which are randomly generated from our four benchmarks with about 100 Mbytes input.***

6.7 Ostrich as MapReduce Server

To demonstrate the effectiveness of using Ostrich when processing multiple MapReduce jobs in batch mode, we built a MapReduce server using both Ostrich and Phoenix. The server forks multiple worker processes to process MapReduce requests in parallel. Each worker process uses a part number of cores to serve requests. Figure 13 shows the performance comparison of MapReduce servers based on Ostrich and Phoenix, with the number of worker processes from 1 to 16. For the server with 16 worker processes, each process exclusively uses one core and continuously serves the MapReduce requests. As shown in the figure, the MapReduce server based on Ostrich outperforms that based on Phoenix under each configuration, and the speedup increases from 1.44X to 1.90X with the decrease of worker processes, due to the better cache locality and performance scalability in Ostrich. The results also show that running multiple MapReduce jobs in parallel is a much better way to maximize the overall throughput, since the scalability of MapReduce is not linear in many cases, especially when the number of cores exceeds 8 in our evaluation. However, contention on shared caches might result in performance degradation, which is shown by our results when the number of worker processes exceeds 8.

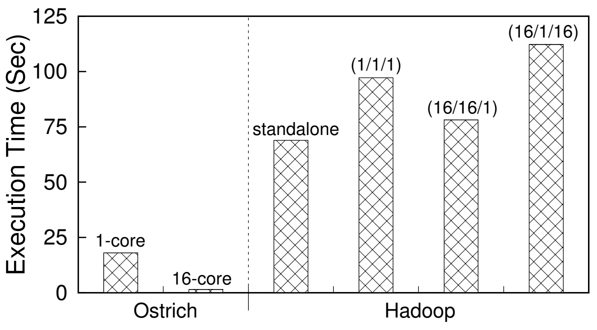

Figure 14: ***The performance comparison between Ostrich and Hadoop for WC benchmark with 100 Mbytes input. The labels upon histogram are the configuration, and the symbol of (C/J/T) corresponds with #cores, #jvm and #threads/jvm.***

6.8 Performance Comparison with Hadoop

As Hadoop can also run on a single machine, and is built with support for task parallelism in scheduler, readers might be interested in how the performance of Hadoop compares to that of Ostrich. We thus compared the performance of WC benchmark with 100 Mbytes input on Ostrich and Hadoop (version 0.19.1). We did not choose a larger input size for comparison because Hadoop scales poorly with the input size and consumes around half an hour in our machine to process 1 Gbyte input.

As shown from the Figure 14, Ostrich is orders of magnitude faster (more than 50X) than Hadoop on Intel 16-Core machine. Note that the stand-alone mode of Hadoop (used for debugging) avoids the overhead of HDFS and communication with the remote JobTracker, but the current version can only run on 1 core. The reason for the deficiency is that the Hadoop needs to create multiple objects (e.g., String, Text(Line, Token)) of a single piece of input, which causes redundant uses of memory and processing of the objects (e.g., ReadLine, Tokenize, Serialize and Deserialize). Besides, the exchange of intermediate data between Map and Reduce phase uses disk files. Finally, using a JVM makes it hard for the runtime to exploit memory hierarchy of multicore. Nevertheless, some techniques like group-based scheduling in Ostrich could improve the performance of Hadoop in multi-threaded mode (*i.e.*, (16/1/16)), which uses only a single buffer and requires a big lock to protect the buffer.

6.9 Discussions

Our evaluation results confirm that Ostrich has good performance due to the described optimizations. However, to get an optimal result, there are still some design spaces to exploit, such as the size of the `Iteration Window` and whether copying keys/values or not. Hence, an auto-tuning scheme would be useful. For example, Ostrich could first process a small portion of input and monitor the cache miss rate or operating system activities (swapping status) to make an optimal decision, which is a part of our future work.

7. RELATED WORK

Our work is related to the research in programming models for data-parallel applications, nested data parallelism and multicore related research. We briefly discuss the most related work in turn.

7.1 Programming Model and Runtime Related to MapReduce

MapReduce [2] is a popular programming model developed in Google. An open-source implementation, namely Hadoop [6] is provided by Apache, which is implemented in Java and uses HDFS as the underlying file system. Zaharia et al. [17] uses a heterogeneity-aware scheduler to improve Hadoop's performance in heterogeneous environments, such as virtual clusters. The currently implementation of Hadoop focuses on cluster environments rather than on a single machine. It does not exploit the data locality at the granularity of a single machine, neither does it provide a multicore-oriented scheduler.

There is also some work aiming at extending the programming model of MapReduce for other purposes. For example, the database community extends the MapReduce programming model by adding one additional phase, namely Merge, to join two tables [18]. Online MapReduce [7] supports the online aggregation and allows users to get the partial results of an online job. Such a MapReduce model has a good match with the incremental computing nature of Ostrich.

Ranger et al. [3] provide a MapReduce implementation on multicore platform. Their implementation, namely Phoenix, successfully demonstrated that applications written using MapReduce are comparable in performance to their pthread counterparts. Compared to MapReduce, *Tiled-MapReduce* partitions a big MapReduce job into a number of independent sub-jobs, which improves resource efficiency and data locality, thus significantly improves the performance. Yoo et al. [4] heavily optimize Phoenix from three layers: algorithm, implementation and OS interaction. Recently, Mao et al. [19] also builds a MapReduce runtime for multicore that many algorithm and data-structure level optimizations over Phoenix [3]. In contrast, Ostrich optimizes MapReduce mainly at the programming model level by limiting the data to be processed in each MapReduce job, which significantly reduces the footprint and enables other locality-aware optimizations. Hence, Ostrich is orthogonal to these optimizations and can further improve the performance of these systems.

The popularity of MapReduce is also embodied in running MapReduce on other heterogeneous environments, such as on GPUs [20] and Cell [21]. To ease the programming of MapReduce applications on heterogeneous platforms such as GPUs and CPUs, Hong et al. [22] recently developed a system called MapCG, which provides source-code level compatibility between these two platforms. Coupled with a lightweight memory allocator and hashtable on CPUs, they showed that MapCG has considerable performance advantage over Phoenix and Mars.

Merge [23] is a programming model that targets heterogeneous multicore platform. It uses library-based model which is similar to MapReduce to hide the underlying machine heterogeneity. It also allows automatic mapping of the computation tasks into available resources.

Dryad [24] is a program model for data-parallel applications from Microsoft. Dryad abstracts tasks as nodes in the resource graph and relies on the runtime to map the nodes to the graph. DryadLINQ [25] integrates Dryad with high-level language (i.e., .NET) and allows users to program using SQL like programming language. DryadInc [8] supports the incremental computing model and the reuse of existing computation results. *Tiled-MapReduce* supports the save of the partial results of a sub-job, which also enables the continuous computation and computation reuse.

Romp [26] is a toolkit that translates GNU R [27] programming to OpenMP program. It was also claimed to support the translation of the MapReduce program to the OpenMP counterpart.

7.2 Nested Data Parallelism

Many data-parallel languages and their implementations, *e.g.*, NESL [28], are designed to support nest data parallelism, which is critical for the performance of nested parallel algorithm. Meanwhile, the nested data processing has been explored by the database community for several decades, such as Volcano [29] dataflow query processing systems. We observed that the MapReduce programming model also could be paralleled in two dimensions and it could be efficiently implemented on multicore architecture. Our work is on the programming model level and introduces several additional optimizations to reduce the pressure on the system resources, including main memory, caches and processors.

7.3 Multicore Research

A multicore operating system, named Corey [10], proposes three new abstractions (*address ranges*, *shares* and *kernel cores*), to scale a MapReduce application (i.e., Word Revert Index) running on Corey. The work in Corey is orthogonal to Ostrich. The abstractions in Corey, if available in commodity OSes, could further improve the efficiency of Ostrich due to the reduced time spent in the OS kernel.

Thread clustering [30] schedules threads with similar cache affinity to close cores, thus improves the cache locality. *Tiled-MapReduce* also aims to improve cache locality, but tries to limit the working set and fits data in a sub-job in cache.

8. CONCLUSION AND FUTURE WORK

Multicore is prevalent and it is important to harness the power of the likely abundant CPU cores. MapReduce is a promising programming model for multicore platforms, to fully utilize the power of such processing resources.

This paper argued that the environmental differences between clusters and multicore open new design spaces and optimization opportunities to improve performance of MapReduce on multicore. Based on the observation, this paper proposed *Tiled-MapReduce*, that uses the "tiling strategy" to partition a large MapReduce job into a number of small sub-jobs and handles the sub-jobs iteratively. This paper also explored several optimizations otherwise impossible for MapReduce, to improve the memory, cache and CPU efficiency. Experimental results showed that our implementation, namely Ostrich, outperforms Phoenix by up to 3.3X and saves up to 85% memory. Our profiling results confirmed that the improvement comes from the reduced memory footprint, better data locality and task parallelism.

In our future work, we plan to explore the integration of Ostrich with the cluster version of MapReduce (e.g., Hadoop) on multicore based clusters, to exploit multiple levels (e.g., both multicore and cluster level) of data locality and parallelism. We will also investigate the potential use of the tiling strategy in the cluster version of MapReduce.

9. ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful comments. This work was funded by China national 973 program under grant numbered 2005CB321602, a research grant from Intel numbered

MOE-INTEL-09-04 and Shanghai Leading Academic Discipline Project (Project Number: B114).

10. REFERENCES

[1] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference (DAC)*, 2007, pp. 746–749.

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the 13th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2007, pp. 13–24.

[4] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System," in *Proceedings of 2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 198–207.

[5] S. Coleman and K. McKinley, "Tile size selection using cache organization and data layout," in *Proceedings of the 1995 ACM SIGPLAN Conference on Program Language Design and Implementation (PLDI)*, 1995, pp. 279–290.

[6] A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley, "Hadoop: a framework for running applications on large clusters built of commodity hardware," *Wiki at http://lucene. apache. org/hadoop*, 2005.

[7] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[8] L. Popa, M. Budiu, Y. Yu, and M. Isard, "DryadInc: Reusing work in large-scale computations," in *Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[9] G. Golub and C. Van Loan, *Matrix computations*. Johns Hopkins University Press, 1996.

[10] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2008, pp. 43–57.

[11] D. M. Eres, M. Kaminsky, M. F. Kaashoek, and E. Witchel, "Separating key management from file system security," in *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP)*, 1999, pp. 124–139.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the 1998 ACM SIGPLAN Conference on Program Language Design and Implementation (PLDI)*, 1998, pp. 212–223.

[13] S. Schneider, C. Antonopoulos, and D. Nikolopoulos, "Scalable locality-conscious multithreaded memory allocation," in *Proceedings of the 5th International Symposium on Memory Management (ISMM)*, 2006, pp. 84–94.

[14] J. Gray, http://www.hpl.hp.com/hosted/sortbenchmark/.

[15] E. Waingold, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua *et al.*, "Baring It All to Software: The Raw Machine," *IEEE Computer*, vol. 30, no. 9, pp. 86–93, 1997.

[16] J. Levon, *OProfile Manual*, http://oprofile.sourceforge.net/doc/, Victoria University of Manchester.

[17] M. Zaharia, A. Konwinski, A. Joseph, U. Berkeley, R. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments," in *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2008, pp. 86–93.

[18] H. Yang, A. Dasdan, R. Hsiao, and D. Parker, "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2007, pp. 1029–1040.

[19] Y. Mao, R. Morris, and F. Kaashoek, "Optimizing MapReduce for Multicore Architectures," Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep. MIT-CSAIL-TR-2010-020, 2010.

[20] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 260–269.

[21] M. de Kruijf and K. Sankaralingam, "MapReduce for the Cell BE Architecture," Department of Computer Sciences, The University of Wisconsin-Madison, Tech. Rep. TR1625, 2007.

[22] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "MapCG: Writing Parallel Program Portable between CPU and GPU," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[23] M. Linderman, J. Collins, H. Wang, and T. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008, pp. 287–296.

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2007, pp. 59–72.

[25] Y. Yu, M. Isard, D. Fetterly, M. Budiu, ÃŽlfar Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2008, pp. 1–14.

[26] F. Jamitzky, "Romp: OpenMP binding for GNU R," http://code.google.com/p/romp/, 2009.

[27] D. B. et al., "The R project for statistical computing," http://www.r-project.org/, 2010.

[28] G. E. Blelloch, "Programming parallel algorithms," *Communications of the ACM*, vol. 39, no. 3, pp. 85–97, 1996.

[29] G. Graefe, "Volcano, an extensible and parallel query evaluation system," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 1, pp. 120–135, 1994.

[30] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2007, pp. 47–58.