Phoenix++: Modular MapReduce for Shared-Memory Systems

Justin Talbot, Richard M. Yoo, and Christos Kozyrakis
Computer Systems Laboratory
Stanford University
{jtalbot, rmyoo, kozyraki}@stanford.edu

ABSTRACT

This paper describes our rewrite of Phoenix, a MapReduce framework for shared-memory CMPs and SMPs. Despite successfully demonstrating the applicability of a MapReduce-style pipeline to shared-memory machines, Phoenix has a number of limitations; its uniform intermediate storage of key-value pairs, inefficient combiner implementation, and poor task overhead amortization fail to efficiently support a wide range of MapReduce applications, encouraging users to manually circumvent the framework. We describe an alternative implementation, Phoenix++, that provides a modular, flexible pipeline that can be easily adapted by the user to the characteristics of a particular workload. Compared to Phoenix, this new approach achieves a 4.7-fold performance improvement and increased scalability, while allowing users to write simple, strict MapReduce code.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*concurrent programming*; E.2 [**Data**]: Data Storage Representations

General Terms

Algorithms, Design, Measurement, Performance

Keywords

Shared-Memory MapReduce, Modularity, Performance

1. INTRODUCTION

MapReduce [3] is a functional language-inspired parallel programing model for large-scale machines. Its simplicity, coupled with its applicability to practical problems, has led to adoption on a range of computing platforms [11, 4]. The Phoenix project [9, 12], in particular, demonstrated that the MapReduce model could be used on shared-memory machines, with scalability comparable, in some cases, to hand-coded PThreads solutions.

While cluster-based MapReduce performance is limited primarily by disk and network I/O, shared-memory MapReduce performance, without these bottlenecks, is sensitive to a wide range of *workload-influenced* details such as intermediate key-value data layout, memory allocation pressure, and framework overhead [12].

Despite the wide variation in possible MapReduce workload characteristics, Phoenix adopts a *static* MapReduce pipeline similar to cluster-based implementations. For example, intermediate key-value pairs are always stored in a fixed-size hash table and combiner execution always occurs at the end of the map stage. However, these implementation decisions are inefficient for many types of workloads. As a result, we have found that users often have to work around the Phoenix pipeline—managing memory manually, implementing combiner functionality in the map function, etc.—to get high performance. Furthermore, the framework exposes internal task scheduling details, requiring the user to manage library and scheduling overhead. Phoenix 2 [12], although significantly improved in terms of scalability and NUMA-awareness, suffers from similar problems since it is based on the same Phoenix code.

More recent shared-memory MapReduce implementations have attempted to address this performance issue by *modifying the MapReduce paradigm*. The Tiled-MapReduce system [1] executes the map and combine tasks in groups to minimize resource usage and increase locality, resulting in a maximum reported speedup of about 3.5x over Phoenix 2. The MATE system [7] uses a heavily revised MapReduce API that requires the user to write combined map / reduce functions, producing a 1x-3x speedup over Phoenix.

We take a different approach to enabling high performance shared-memory MapReduce. Phoenix++ is a complete revision of the Phoenix framework that achieves speed through *modularity* in performance critical sections. It provides a flexible intermediate key-value storage abstraction that permits workload-tailored implementations and a more effective combiner implementation that can minimize memory usage, all the while hiding the task scheduling details in a modular fashion and maintaining the basic MapReduce model. We demonstrate that on a 32-core Linux machine, the result is an *average* 4.7-fold performance improvement and substantially improved scalability over Phoenix 2, while greatly reducing the amount of code that users must write.

In the next section, we characterize MapReduce workloads along key application dimensions and discuss the implications on MapReduce library design. Section 3 discusses the shortcomings of Phoenix design and demonstrates how users must work around the framework to get good performance. Then, in Section 4, we describe our system, Phoenix++, which overcomes these problems with a flexible, modular pipeline design. In Section 5, we show that this approach achieves better performance and scalability than Phoenix, while significantly reducing the amount of code that a user must write. In Section 6 we position the new implementation among related work, and Section 7 concludes.

2. WORKLOAD CHARACTERIZATION

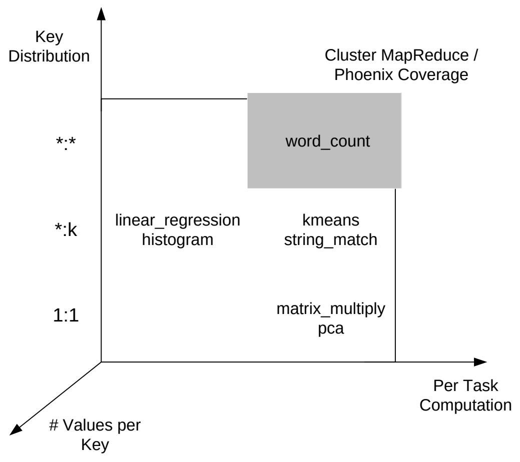


Figure 1: Phoenix sample workloads organized by workload characteristics (see text for description).

To better understand how different workloads impact shared-memory MapReduce performance, we tried to systematically characterize the 7 example workloads included in the Phoenix 2 distribution [12]. After some iteration, we settled on three dimensions (shown in Figure 1):

1. **the map task to intermediate key distribution**, with three possible values,

   `*:*` any map task can emit any key, where the number of keys is not known before execution,

   `*:k` any map task can emit any of a fixed number of keys, k, and

   `1:1` each task outputs a single, unique key.

2. **the number of values emitted per key**, which typically scales with the data set size, and

3. **the amount of per task computation** compared to the total framework overhead.

While not completely orthogonal, we believe that these dimensions capture the important workload characteristics that impact performance on shared-memory MapReduce. Table 1 summarizes the impact of each workload dimension on key framework design decisions.

The *key distribution* is the relationship between the set of map tasks and intermediate keys. For highest performance, the runtime must leverage key distribution knowledge to provide efficient key-value storage between the map and reduce phases. While `*:*` workloads such as `word_count` must use a hash table storage structure, other key distributions can use more efficient data structures that consume less memory and have lower fixed costs.

The *number of emitted values per key* typically grows with the size of the data set used in the workload. Larger data sets stress the runtime's capability to efficiently manage in-memory buffer space. Hence it is preferable to invoke combiner functions frequently, but individual combiner call overheads can add up.

Finally, *per task computation* is the ratio between the complexity of the user-provided map or reduce functions and the overhead of the MapReduce framework, such as time necessary to dequeue the task or to emit the intermediate key-value pair. When the per task computation is low, as in the `linear_regression` and `histogram` workloads, such overhead can dominate.

3. LIMITATIONS OF PHOENIX

Like the cluster-based MapReduce implementations that preceded it, Phoenix is statically optimized for a particular class of workloads (shown in gray in Figure 1), featuring high per task computation and a large, unknown number of keys. Phoenix's design decisions are summarized in Table 1. In the rest of the section, we describe Phoenix's design in more detail and demonstrate how users are forced to work around the Phoenix framework to improve performance.

3.1 Inefficient Key-Value Storage

In a shared-memory MapReduce implementation, maintaining the intermediate key-value storage is a complicated task, since containers must provide fast lookup and retrieval over potentially large dataset, all the while coordinating accesses across multiple threads.

The approach proposed in Phoenix is to utilize map thread-specific fixed-width hash tables (i.e., the number of hash buckets is constant) [12]. The fixed size is necessary to permit reduce threads to access the hash tables in a cross-cutting fashion, taking matching key-value pairs from the same bucket in each hash table, without requiring locking.

However, the fixed-width hash table is not a good fit for any of the workload key distribution categories we identified; its fixed-width limits the performance over `*:*` workloads with large number of keys, and it is inefficient for other distributions. Listing 1, for example, shows the map function of the `histogram` application, which counts up the number of times a particular red, green, or blue component value occurs in a large image. As can be seen in lines 5∼7, in order to avoid the costly overhead of using the fixed-width hash table of Phoenix, the programmer uses *fixed-size arrays* instead, to manually store the component counts before emitting them. This optimization is possible since the user knows *a priori* that the number of possible keys is limited to the 768 pixel values (i.e., a `*:k` workload).

3.2 Ineffective Combiner

In standard MapReduce, a user can optionally specify a *combiner function* [3], typically associative and commutative, which is run locally on key-value pairs emitted by the map function. On clusters of machines, the combiner function reduces the number of key-value pairs that must be exchanged between machines. In Phoenix 2, combiner functionality was introduced to lower memory traffic between

Dimension	Affects	Phoenix	Phoenix++
key distribution	intermediate key-value storage	hash table	workload-tailored storage
# values per key	combiner implementation	combiner run after the map stage	combiner run for every emitted key-value pair
per task computation	scheduling / library overhead	amortized in user code via task chunking	eliminated by the compiler through code inlining

Table 1: Workload characteristics and their impact on design decisions. Phoenix and Phoenix++ denote each implementation's choices.

```
1  void hist_map(map_args_t *args) {
2      unsigned char *data = (unsigned char *)
           args->data;
3
4      /* Manually buffer intermediate results
           */
5      intptr_t red[256] = {0};
6      intptr_t green[256] = {0};
7      intptr_t blue[256] = {0};
8
9      /* Count occurrences, amounts to manual
           combine */
10     for (int i = 0; i < args->length * 3; i
           +=3)
11     {
12         red[data[i]]++;
13         green[data[i+1]]++;
14         blue[data[i+2]]++;
15     }
16
17     /* Selectively emit key-value pairs */
18     for (int i = 0; i < 256; i++)
19     {
20         if(red[i] > 0)
21             emit(i, red[i]);
22         if(green[i] > 0)
23             emit(i+256, green[i]);
24         if(blue[i] > 0)
25             emit(i+512, blue[i]);
26     }
27 }
```

Listing 1: The Phoenix map function for the `histogram` workload. In an effort to improve performance, the body of the map function has been made overly complex.

```
class Histogram : public MapReduceSort<
    Histogram,
    pixel, intptr_t, uint64_t,
    array_container<intptr_t, uint64_t,
        sum_combiner, 768> > {
public:
    void map(pixel const& p, container& out)
         const {
        emit(out, p.r, 1);
        emit(out, p.g+256, 1);
        emit(out, p.b+512, 1);
    }
};
```

Listing 2: The `histogram` map function written for Phoenix++. Templates are used to adapt the MapReduce pipeline to the workload while maintaining a simple programmatic interface.

the map and reduce phases [12]; the user specifies the combiner function through a separate function pointer, and for each thread, it is run at the end of the map phase on top of the key-value hash table.

However, unlike cluster-based implementations, on SMP machines memory allocation costs tend to dominate, even more than the memory traffic. Running the combiner at the end of the map phase may reduce the total overhead to invoke the combiner, but it fails to reduce the memory allocation pressure, since generated key-value pairs must still be stored. Further, by the time the combiners are run, those pairs may no longer be in the cache causing expensive memory access penalties.

A better approach is to run the combiner more frequently. In lines 10∼15 of Listing 1, we can see that the Phoenix-provided combiner is ignored in favor of manually implementing the combiner inside the map function. This bypass substantially reduces memory allocation pressure.

3.3 Exposed Task Chunking

Typical MapReduce implementations internally group tasks into chunks to reduce scheduling costs and amortize per-task overhead. Phoenix exposes this chunking to the programmer, passing chunks, rather than individual tasks, to the map function (e.g., line 10 in Listing 1).

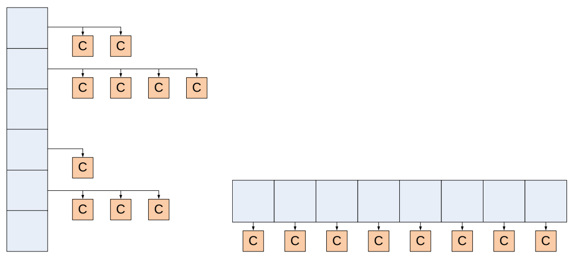
This design enables the user-implemented optimizations described in the previous two sections. However, it also has two drawbacks—first, extra code to deal with chunks is pushed into user code, complicating the map function. Second, and more important, if the user leverages the exposed chunk to improve performance, the framework can no longer freely adjust the chunk size since doing so will affect the efficiency of the map function. Phoenix addresses this dependency by requiring the user to specify the chunk size as a parameter; however, the user's fixed chunk size may be far from optimal and could work against load balancing.

3.4 Performance Impact

We tried rewriting the Phoenix `histogram` application to use an "ideally" simple map function that used the built-in hash table storage for key-value pairs, relied on the Phoenix-provided combiner, and operated on a single pixel at a time. Our tests showed that this ran 10x slower than the version in Listing 1. In a more extreme case, the `linear_regression` workload showed a 24x slowdown. Thus, there is considerable incentive for users to work around the Phoenix framework to improve performance.

4. PHOENIX++ DESIGN AND IMPLEMENTATION

Motivated by the shortcomings of Phoenix discussed in the previous section, we designed a *modular* MapReduce



(a) Hash Container (b) Array Container

Figure 2: Container class implementations. Each "C" represents a combiner object, which stores emitted values associated with a single key.

framework, Phoenix++, where users can adapt performance critical elements of the pipeline to match the known characteristics of their workload. In particular, we expose the runtime's intermediate key-value grouping and storage responsibility through two abstractions: *containers*, which allow us to tune the framework to each workload's key distribution, and *combiner objects*, which permit efficiently handling workloads with large numbers of values per key. We further increase the modularity of the pipeline by allowing the user to easily replace the memory allocator and by making the final key-value sort optional. Lastly, we hide the task chunking granularity to provide a cleaner interface.

Unfortunately, increasing the modularity of the framework can reduce performance due to additional function calls, less opportunities for compiler optimizations, etc. This is particularly a concern in Phoenix++ because the intermediate key-value storage is on the critical performance path. This issue motivated us to implement Phoenix++ in C++ where templates provide a mechanism to statically inline code, allowing us to build an adaptive framework without paying a performance penalty.

The result is a consistent and simple MapReduce-style programmatic interface with very high performance and scalability across all workload dimensions. Listing 2 shows the `histogram` workload written in Phoenix++. Compared to Phoenix, the amount of user-written code is substantially decreased. Even so, this runs faster than the corresponding Phoenix 2 code. The Phoenix++ design choices are summarized in Table 1 and are discussed in the following sections.

4.1 Containers

In Phoenix++, *containers* provide the standard group-by functionality between the map and reduce phases, grouping emitted key-value pairs by key and then storing them in *combiners*. Figure 2 depicts two example container configurations and their relationship to combiner objects. By mixing and matching *container* and *combiner* implementations we can get high performance key-value storage for a wide range of workloads.

To provide the highest possible performance for each key distribution, we define 3 default container implementations tuned to each of the `*:*`, `*:k`, and `1:1` workload classes.

- **hash container** (`*:*`): a variable-width hash table implementation where each map thread can resize its own hash table (in contrast to Phoenix where all tables are the same fixed width),
- **array container** (`*:k`): a fixed-size, thread-local array implementation, which requires that the keys be integers within an *a priori* known range [0,K-1], and
- **common array container** (`1:1`): a non-blocking array structure shared across all threads.

initialize(num_map_threads, num_reduce_tasks) - Initialize the container
container get() - Create a thread-local container "instance"
put(container) - Return control of the instance to the container
combiner_iterator out(reduce_task_id) - Output combiner objects for processing by a reduce task

Table 2: Container interface functions.

Figure 2(a) depicts the hash container implementation. In contrast to the default fixed-width hash table in Phoenix, we allow each map thread to control the width of its own hash table. This ensures that we can maintain O(1) insertion performance even in the presence of an unexpectedly large number of keys. However, since each hash table has different width, the correspondence between keys and bucket indices has been lost. This makes it more difficult to group values with the same key across threads. 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 fixed-width hash table with the same number of buckets as reduce tasks. This extra copy is unfortunate, but as we show in Section 5, the overall performance is higher than the fixed-width hash table.

Figure 2(b) shows the arrangement of the array container. For tasks with a known, small key cardinality, insertion into a fixed size array avoids the cost of hashing keys and repeatedly checking if the hash table should be resized. This structure can provide a substantial performance increase, especially in cases where the amount of computation per task is small. In Listing 2, line 4, we can see that the Phoenix++ `histogram` workload uses the `array_container`.

Lastly, the common array container leverages the fact that each emitted key is unique in `1:1` workloads; all threads can write into the same array without any synchronization.

Interface. The user is also free to specify her own *container* through the common interface shown in Table 2. Through the `get()` method, each map thread can ask the container class for a container "instance". When done emitting key-value pairs into the instance, each map thread returns its instance through the `put()` method. Using this instance interface permits us to support a wide range of possible container implementations. For example, when the `get()` function is called, the container is free to return either a thread-local data structure, which it will later merge with similar structures from other map threads, or a global data structure, which is controlled by locks or other synchronization mechanisms. When the workload properties permit, the container can even expose a global data structure without any synchronization mechanisms at all, as is the case for the common array container.

4.2 Combiner Objects

The *combiner object* abstraction in Phoenix++ is used to store all emitted values with the same key. Thus, in Phoenix++, combiners are not just functions, but are stateful objects. In order to maximally reduce memory pressure

initialize() - Initialize the combiner object
add(value) - Insert a new value into the combiner object
add(combiner) - Combine the state of two combiner objects
bool discard() - Return true if the combiner object should not be passed to the reduce stage
value* next() - Iterate over values in the combiner object - Return NULL if all values have been returned

Table 3: Combiner class member functions.

due to intermediate key-value storage, Phoenix++ invokes the combiner immediately after each key-value pair is emitted by the map function.

The default combiner, `buffer_combiner`, implements the standard MapReduce behavior, buffering up all emitted values until the reduce function is called. The implementation follows Phoenix 2 in eliminating the need to copy values when concatenating buffers from different threads.

By contrast, the `associative_combiner` does not buffer values. Instead, it leverages the fact that the combiner will be called for every emitted value, and maintains a single aggregate value which is incrementally combined with each emitted value. This completely eliminates the overhead associated with maintaining a buffer. In the sample workloads, we make wide use of the `sum_combiner` (Listing 2, line 4), an instantiation of `associative_combiner` which simply stores the sum of all the values inserted into it.

Additionally, unlike the standard MapReduce formulation which specifies that combiner functions are run per machine, and then reduce functions are run *across* machines [3], in Phoenix++ we guarantee that combiners will be run on all emitted values across all threads, before the combiner object is passed to the reduce phase. This means that the user only has to implement the combiner function once; code does not have to be duplicated in the reduce function.

Interface. The set of functions supported by the combiner class is given in Table 3. We use an `initialize()` method rather than a constructor since, to minimize memory allocations, the runtime may reuse combiner objects. Within each thread, the values will be added to the combiner object through `add()`, as they are emitted from the map function. The second `add()` method combines combiner objects created for the same key in different threads. The `discard()` method permits the user to mask certain combiner objects—typically empty combiner objects that were proactively created by the runtime—from being passed on to the reduce function. Finally, the `next()` method is used by the reduce function to iterate through the values in the combiner object. Table 4 describes how the `buffer_combiner` and the `sum_combiner` implement this common interface.

4.3 Other Modularity in Phoenix++

Generic, optional sort. In the original Google implementation, MapReduce sorts the final output [3]. Considering that I/O is the dominant factor in cluster-based MapReduce, additional sorting does not cause much overhead. However, for shared-memory MapReduce implementations, sorting at the merge phase can impose significant overhead. Thus, we permit the user to completely disable the final sort.

(a) `buffer_combiner` Implementation

State - Linked list of vectors of values
initialize() - Clear linked list
add(value) - Append value to first vector in linked list
add(combiner) - Concatenate combiner objects' linked lists (avoids copying)
bool discard() - Return true if no values were inserted
value* next() - Iterate over all elements in all vectors in linked list

(b) `sum_combiner` Implementation

State - Single value (typically numeric)
initialize() - Set state to 0
add(value) - state += value
add(combiner) - state += combiner.state
bool discard() - Return true if state == 0
value* next() - Return state and then NULL on the next call

Table 4: Combiner class implementations provided by Phoenix++.

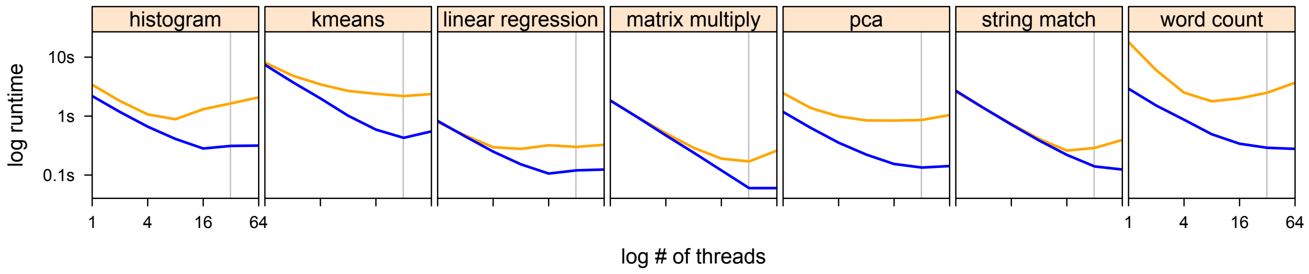
Additionally, if sorting is enabled, we allow the user to provide a custom sorting function that is defined over key-value pairs, rather than the keys alone. In cases such as `word_count`, which require a more complicated result sorting, the custom sorting function avoids the need for a second MapReduce pass.

Custom memory allocators. In Phoenix 2, the authors noted the importance of the memory allocator on the overall library scalability [12]. In Phoenix++, we use STL's custom allocator interface to support more scalable allocators (e.g. Intel's TBB `scalable_allocator` [6]) for the containers and combiners.

4.4 Efficient Modularity

As discussed in the previous sections, in Phoenix++, we permit the user to swap in and out core functionalities of the framework. Moreover, we have hidden the task chunking granularity, allowing the user to write a map function that works on a single map task. While these changes simplify programming, they do introduce a large number of function calls in the inner task loops which can quickly dominate the runtime for applications with low per task computation.

To eliminate the function calls we leverage a C++ template technique known as the Curiously Recurring Template Pattern (CRTP) [2]. As shown in Listing 2, the first argument to the template (line 2) is the inheriting class (`Histogram`) itself. This means that the user-provided map and reduce functions are statically determined, and as a result the compiler can inline them into the task loop specified in the framework's code. Additionally, since the container and combiner objects are also specified as template parameters, the compiler can inline them as well. The result is that all the function calls in the inner loops can be eliminated.



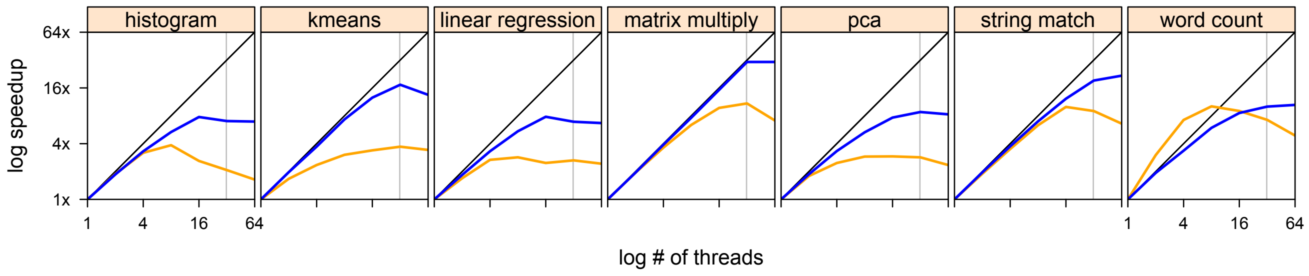


Figure 3: Performance (top) and scalability (bottom) comparisons of Phoenix++ (blue) and Phoenix 2 (orange). Vertical lines mark 32 threads, the number of physical cores on our test machine. For the scalability comparison, black lines represent ideal scaling.

Hardware Settings	
CPU	4 Nehalem-EX chips 8 cores per chip 2-way Hyper-Threading per core Total 64 hardware contexts on system
Cache	Per core data / instruction L1, 32 KB Per core L2, 256 KB Shared L3, 24 MB
Memory	32 GB
Interconnect	Point-to-point QuickPath interconnect
Software Settings	
Operating System	Linux kernel 2.6.32
Compiler	GCC 4.4.3 with -O3 optimization

Table 5: Experiment settings.

5. EXPERIMENTAL RESULTS

Table 5 describes the hardware and software settings for our experiment. We ported the test cases that ship with the Phoenix 2 release [12] to work with Phoenix++. During the porting process, we updated a few of the workloads to more efficient implementations, in order to evaluate against more realistic usage scenarios. To make the comparison fair, we back-ported those optimizations to the Phoenix 2 versions. Likewise, to make replication possible, we report results using the (relatively small) data sets available on the Phoenix 2 website [10]. All the results are the averages of 5 runs when the machine was idle; the execution of the various configurations were interleaved to reduce bias.

5.1 Performance Summary

Figure 3 compares the performance of Phoenix++ against Phoenix 2. Phoenix++ is substantially faster and more scalable across all workloads; notice that the axes are log scales. Some workloads exhibit peak performance at 32 threads, as the performance degrades slightly when we start to utilize Hyper-Threads. At 32 threads, Phoenix++ achieves a 4.7x speedup on average over Phoenix 2.

Profiler analysis reveals that Phoenix++ achieves this performance improvement through combinations of all the major enhancements described in Section 4. The newly introduced *containers* improved most of the workloads, even `word_count`, for which Phoenix 2 had been optimized. The more effective *combiner objects* contributed to better data locality and lower memory allocation pressure, resulting in substantial scalability improvement on most applications. For `matrix_multiply` and `string_match`, which do not emit any intermediate key-value pairs, the combiner's `discard()` interface function turned out to be useful, as Phoenix 2 suffered high overheads executing empty reduce tasks. Finally, `histogram` and `linear_regression`, which exhibit the smallest per task computation, benefited significantly from the reduced function call overhead due to function inlining enabled with CRTP.

In the following sections, we try to quantify the benefit of each of these optimizations.

5.2 Container Performance

As described in Section 4.1, containers were introduced to provide a modular way to cover different key distributions. In Figure 4, we compare the performance of the three default containers provided by Phoenix++ against a fixed-width hash container which mimics the container used in Phoenix. Notice that not every container can be used with every workload, and that `matrix_multiply` and `string_match` are not included in the plot, since they do not emit any key-values during the map phase and are, thus, insensitive to container choices.

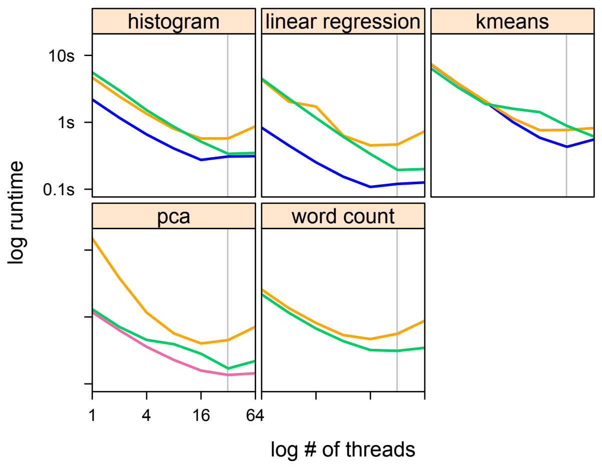


Figure 4: Workload sensitivity to container choice: variable-width hash table (green), array (blue), common array (pink), and, for comparison, a Phoenix-like fixed-width hash table (orange). The default Phoenix++ containers provide high performance and scalability across a range of workloads.

The first observation to make is that no single container prevails across all the workloads, justifying our approach of providing various container implementations; the `*:k` workloads, `histogram`, `linear_regression`, and `kmeans`, perform the best with the array container, and the `1:1` workload, `pca`, benefits the most from the common array container. Second, the default Phoenix++ containers provide better performance and scalability than the Phoenix-like fixed-width hash container—even the `*:*` workload, `word_count`, benefits from the new hash container.

5.3 Combiner Performance

Figure 5 shows the performance when using an `associative_combiner` to incrementally aggregate emitted values, compared to using a `buffer_combiner` to simply buffer up key-value pairs for the reduce task.

Combiners provide a dramatic absolute performance improvement, especially for those workloads with low per task computation: e.g., `histogram` and `linear_regression`. With the exception of `word_count`, they also provide a substantial scalability improvement across all thread counts.

Note that this contradicts the Phoenix 2 conclusion [12], where the authors observed minimal performance improvement through combiners. The difference in conclusions can be attributed to two factors; first, Phoenix 2 only invoked the combiner at the end of the map phase. In contrast, Phoenix++ invokes the combiner after every emitted value, thus improving locality and reducing memory allocation pressure. Second, as shown in Listing 1, users tend to implement their own combiners in Phoenix 2, which hides the utility of a library-provided combiner.

5.4 Function Call Overhead

As described earlier, we use templates with CRTP to eliminate function calls in the map and reduce inner loops in-

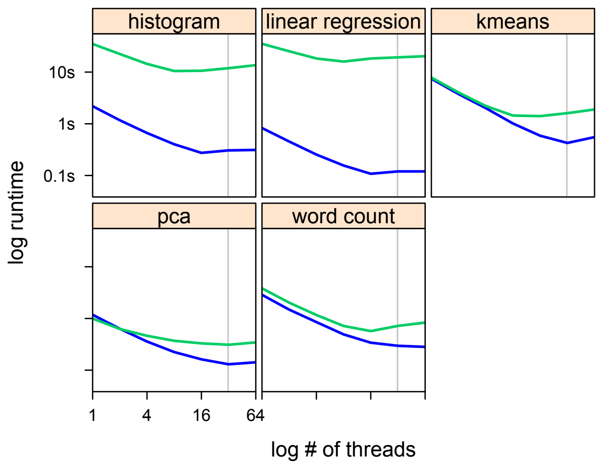


Figure 5: Comparison of workloads with (blue) and without (green) using a combiner. In contrast to the Phoenix 2 results, we see a substantial improvement from the combiner.

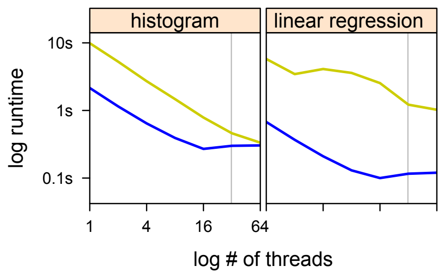


Figure 6: Improvements due to function inlining via CRTP (blue) compared to without (yellow) on workloads with low per task computation.

troduced by our modularization of the pipeline. To demonstrate that these introduced function calls can be problematic, we force the compiler, using GCC's `noinline` function attribute, to not inline the `emit_intermediate` calls into the map function nor inline the map function into the framework's task loop.

The `histogram` and `linear_regression` workloads, which exhibit the smallest per task computation, are particularly sensitive to the function call overhead (Figure 6). The other workloads, with higher per task computation, were not substantially affected by the change.

5.5 Code Size Comparison

In addition to the significant performance improvements, the other important objective of Phoenix++ was to provide a cleaner interface, allowing users to write strict, simple MapReduce code. To quantify the improvement, we compare the lines of source code for the map, reduce, and combiner functions for each workload implemented in Phoenix 2 and Phoenix++. Table 6 shows the results.

Much of the code reduction in map functions is due to the fact that Phoenix++ allows users to write map functions

	map		reduce		combiner	
	P++	P2	P++	P2	P++	P2
histogram	5	39	0	13	0	11
kmeans	30	47	5	33	11	0
linear_regression	9	34	0	14	0	14
matrix_multiply	12	26	0	0	0	0
pca	24	56	0	0	0	0
string_match	31	36	0	0	0	0
word_count	26	53	0	13	0	11

Table 6: Code size comparison in lines of source code for workloads in Phoenix++ and Phoenix 2.

that work on a single map task, rather than an entire chunk. Thus, chunk looping code can be eliminated from most workloads, except for `string_match` and `word_count` which already were single task functions in Phoenix 2.

Comparing the reduce and combiner function code size reveals that (1) our combiner objects obviate the need to specify the aggregation function in multiple locations, and that (2) the two combiner abstractions, `buffer_combiner` and `associative_combiner`, provided by the runtime are sufficient to cover most of the workloads; thus, users rarely have to specify their own. One exception is the `kmeans` workload which requires the combiner to allocate memory to hold the aggregate value. This is not supported by the simple `sum_combiner`, so we had to provide a custom `associative_combiner` implementation. Phoenix 2 does not use a combiner for this workload.

Finally, with the generic sorting interface (Section 4.3), we could eliminate a second MapReduce pass to sort the results of the `word_count` workload, further reducing the necessary code.

6. RELATED WORK

Previous efforts have noted the importance of adapting to variations in workload characteristics. Phoenix 2 [12] allows the users to specify the width of the hash table, but the initial value could be difficult to determine. Metis [8] substitutes the hash table representation with a more complex B+tree, trying to handle both the small and large key spaces with a single structure. Phoenix++, on the contrary, leverages object-oriented modularity to provide different containers that are tailored to different key distributions.

Other shared-memory MapReduce implementations, such as Tiled-MapReduce [1] and MATE [7], have tried to achieve higher performance by modifying the MapReduce paradigm. In contrast, Phoenix++ uses modularity and function inlining to achieve low resource usage and high locality, without substantially changing the standard MapReduce interface.

There exist other MapReduce implementations written in C++. Hadoop [11], for example, provides C++ bindings, and the Boost.MapReduce project [5] plans to bring MapReduce functionality to the Boost C++ libraries. However, neither of these strategically exploits C++ features to achieve high performance and concise code.

7. CONCLUSION

We have demonstrated that shared-memory MapReduce frameworks can achieve much higher performance by adapting to the characteristics of their workloads. To make this possible, we specified interfaces for *containers* and *combiners*, and showed that modularity overhead could be effectively reduced using templates. As a result, Phoenix++ allows the user to write very high performance code, without having to manually circumvent the library or the MapReduce paradigm. The source code is publicly available at `http://mapreduce.stanford.edu`.

Acknowledgments

This work was supported by the Stanford Pervasive Parallelism Laboratory. We appreciate Intel Corporation for donating the machine used in this study. Justin Talbot is supported by FODAVA grant 0937123. Richard Yoo is generously supported by the David and Janet Chyan Stanford Graduate Fellowship.

8. REFERENCES

[1] R. Chen, H. Chen, and B. Zang. Tiled-MapReduce: Optimizing resource usages of data-parallel applications on multicore with tiling. In *Proc. of the 19th Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 523–534, 2010.

[2] J. O. Coplien. Curiously recurring template patterns. *C++ Rep.*, 7:24–27, February 1995.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. of the 6th Symposium on Operating Systems Design & Implementation*, 2004.

[4] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proc. of the 17th Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 260–269, 2008.

[5] C. Henderson. Boost.MapReduce. `http://www.craighenderson.co.uk/mapreduce`.

[6] Intel Corporation. Threading Building Blocks. `http://www.threadingbuildingblocks.org`.

[7] W. Jiang, V. T. Ravi, and G. Agrawal. A Map-Reduce system with an alternate API for multi-core environments. In *Proc. of the 10th IEEE/ACM Int'l Symposium on Cluster, Cloud, and Grid Computing*, pages 84–93, 2010.

[8] Y. Mao, R. Morris, and F. Kaashoek. Optimizing MapReduce for multicore architectures. *Technical Report MIT-CSAIL-TR-2010-020*, 2010.

[9] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proc. of the 13th Int'l Symposium on High Performance Computer Architecture*, pages 13–24, 2007.

[10] Stanford University. The Phoenix system for MapReduce programming. `http://mapreduce.stanford.edu`.

[11] The Apache Software Foundation. Hadoop. `http://hadoop.apache.org`.

[12] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In *Proc. of the 2009 IEEE Int'l Symposium on Workload Characterization*, pages 198–207, 2009.