A Scalable and Composable Map-Reduce System

Mahwish Arif, Hans Vandierendonck, Dimitrios S. Nikolopoulos and Bronis R. de Supinski
Queen's University Belfast
Belfast, United Kingdom
Email: m.arif, h.vandierendonck,d.nikolopoulos@qub.ac.uk, bronis_de_supinski@sbcglobal.net

***Abstract*—This paper presents a novel map-reduce runtime system that is designed for scalability and for composition with other parallel software. We use a modified programming interface that expresses reduction operations over data containers as opposed to key-value pairs. This design choice admits higher efficiency as the programmer can select appropriate data structures. Our runtime targets shared memory systems, which are increasingly capable of performing data analytics on terabyte-sized data sets stored in-memory.**

Our map-reduce runtime is built over the Cilk programming language and outperforms Phoenix++, by 1.5x–4x for 5 out of 7 map-reduce benchmarks on 48 threads.

These results arise from a combination of factors: (i) the reduction of framework overheads, including the elimination of repeated (de-)serialization of key-value pairs; (ii) the use of more appropriate intermediate data structures that reductions over containers support.

***Keywords*-Map-reduce model; performance; composition; programmability**

I. Introduction

Companies and organizations increasingly use data analytics to improve their operations or business model [5]. While data sets in this field are large, current high-end compute boxes may be equipped with a terabyte of memory. Thus, single-node data analytics are often feasible [32] and several authors have investigated shared-memory programming systems for map-reduce style applications [6], [26], [27], [31].

The design of a data analytics programming environment must meet two constraints that are often contradictory. First, performance must support the timely processing of large data sets. Second, programmability must be high since data analysts, who are likely not HPC experts, program the systems. The map-reduce system and its API [9] achieve both objectives for distributed memory systems (clusters): the API hides key performance aspects, such as data access and movement, load balancing and fault-tolerance. while supporting efficient processing.

Map-reduce on shared memory machines is different. Data movement is vertical between levels of the memory hierarchy instead of horizontal between compute nodes. Load balancing can be achieved at a much finer granularity of work items and shared memory map-reduce runtime systems do not usually provide fault tolerance [26], [27], [31], [33], [35].

Our work explores how to structure the map-reduce API and runtime on shared memory systems to maximize performance and ease-of-programming. We derive our solution from an analysis of the main short-comings of existing map-reduce systems.

Framework overheads exist with all systems. Map-reduce systems, however, aggravate them by the need to fit applications to the map-reduce API. The class of programs that can be represented in the map-reduce model is limited theoretically [10] and also from a practical point of view, e.g., in graph analysis [20], [24]. Programmers thus need to overcome hurdles in order to fit their program to the map-reduce model.

Appropriate intermediate data structures are necessary to maximize performance [33]. However, map-reduce systems require the use of lists of key-value pairs that lose the structure of the data, which must be sorted and grouped by key, increasing the time to retrieve individual data elements. In contrast, our map-reduce system admits arbitrary data structures. Programmers can use the most appropriate data structure, including hash maps or arrays.

We build our map-reduce runtime on top of the Cilk programming language [12], [16] and call it *CilkMR*. Cilk offers a simple means to express parallel loops (using the `cilk_for` syntax) and reductions (using generalized reducer hyperobjects [11]). These two concepts have the same conceptual programming complexity as other map-reduce systems [26], [27], [31], [33], [35]. CilkMR, however, retains the structure of the sequential code, which is in stark contrast to previously proposed map-reduce frameworks. Our design choices are not specific to Cilk but could be repeated using other efficient parallel programming languages with similar functionality.

This paper is structured as follows. Section II discusses related work on shared-memory map-reduce programming systems. Section III discusses programming approaches with specialized map-reduce systems and general parallel programming languages. Section IV presents the CilkMR approach to map-reduce programming. Section V explains our benchmarks. Section VI presents our performance evaluation.

978-1-4673-9005-7/16/$31.00 ©2016 IEEE

II. Related Work

Several research projects have investigated efficient map-reduce runtime systems for shared memory systems. The first paper on the Phoenix system [31] compared it against a POSIX threads implementation of the benchmarks. Phoenix matched the performance of the POSIX threads codes that fit the map-reduce model but performed less well on the few that did not. We demonstrate that CilkMR outperforms the latest Phoenix++ runtime system on all but two benchmarks. Because CilkMR can compose map-reduce and other parallel code, it outperforms Phoenix++ by up to 4x on programs that do not match the map-reduce programming model well.

Yoo *et al* [35] improved the scalability of Phoenix for a 256-thread SPARC T2 machine. They found that the internal data structures that store intermediate data are critical for performance. They re-designed the data structures to reduce pointer indirection, fragmentation of data structures and to improve memory allocation.

Talbot *et al* [33] specialized the internal data structures to the applications. This specialization, for instance, replaces generic key-value maps with arrays when appropriate. Further, Phoenix++ replaces Phoenix's C function pointer-based implementation with C++ templates and code inlining to reduce function call overhead.

TiledMR [6] further enhances the memory locality of the map-reduce runtime system. TiledMR splits the input data set and runs small map-reduce jobs in succession while recycling the intermediate data structures efficiently. Many others use this principle, such as in the resilient distributed data sets (RDDs) of SPARK [36].

Lu *et al* [26] optimize map-reduce for the Xeon Phi. As in TiledMR, they pipeline map and reduce to reduce the memory footprint. They also try to vectorize the map task, which only worked for numerical applications such as Black-Scholes and Monte Carlo. Alternatively, they vectorize computation of hash table indices. The programmer must specify which form of vectorization, if any, should be applied, which further burdens the programmer with performance optimization. In our case, vectorization can be enabled by using the array notation of Cilkplus, which is a concise and auto-vectorizable notation for operations that are repeated over all array elements.

Mao *et al* [27] exploit huge page support in the kernel. Huge page sizes require fewer entries in the CPU's translation look-aside buffer (TLB), which reduce TLB misses for large data sets. Mao *et al* also advocate the use of NUMA-aware memory allocators, which Yoo *et al* [35] also investigated.

In an alternate approach to map-reduce, Jiang *et al* [18] focus on the reduction stage. Their work interleaves operations from the map and reduce phase instead of rigidly separating the map and reduce phases. They extended their approach to page the reduction data to disk for (too) large data sets [17].

Several authors analyze the characteristics of map-reduce workloads. Talbot *et al* [33] report the task multiplicity (how many keys may be emitted per task), the number of values per key, and the amount of computation in the map task as key characteristics. De Kruijf *et al* [8] follow a numeric approach and measure the amount of computation performed in the partition step, map, reduce or sort. They develop a micro-benchmark that may be dominated by one of these steps. However, their model is incomplete as the appearance of common keys between map tasks may significantly impact performance [33].

While the map-reduce model is conceptually simple, a subtly aspect is the commutativity of reductions. This aspect of the programming model is often undocumented, for instance in the Phoenix systems [31], [33], [35]. However, executing non-commutative reduction operations on a runtime system that assumes commutativity can lead to program bugs [7] even in extensively tested programs [34]. We use Cilk reducers [11] to perform reductions. Unlike many map-reduce models [6], [14], [33], Cilk reducers do not require commutativity. Thus, they are a safe programming construct that will not lead to subtle programming bugs.

Arif *et al* [2] analysed the performance and programmability of OpenMP [29] for map-reduce workloads. OpenMP user-defined reductions allow programmers to express complex and application-specific reduction operations. The mechanism, however, assumes a parallel tree reduction pattern, as reduction operators are defined on two arguments. Arif *et al* point out that a better way to execute reductions on containers in parallel is to assign keys to threads and to make each thread reduce all values for its keys. This way there are no data dependences between threads during the reduction phase. This is however at odds with the way OpenMP and Cilk express reduction operations.

III. Programming Map-Reduce Workloads

The map-reduce programming model typically assumes that key-value pairs represent data. For instance, the links between internet sites may be represented with a source URL as the key and a list of target URLs as the value. This representation exposes high degrees of parallelism through independent operations on different key-value pairs.

Computations on key-value pairs consist of a map function and a reduce function. The map function transforms a single input item (typically a key-value pair) into a list (which may be empty) of key-value pairs. The reduce function combines all values for the same key. Many computations fit this model [9], [23] or can be converted to fit it [20], [24].

A. The Phoenix++ Map-Reduce API and Runtime

The Phoenix++ shared-memory map-reduce system has several steps: splitting input data; map-and-combine; reduce; and sort-and-merge (Figure 1). The split step splits the input

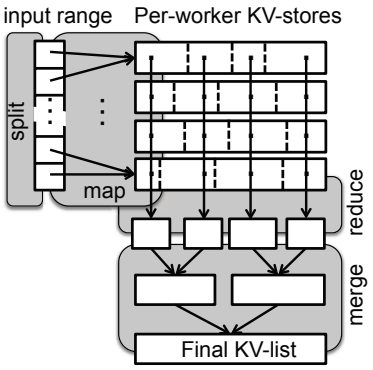

Figure 1: Schematic overview of Phoenix++ runtime system

```
1  int64_t prod(int64_t mi, int64_t ri, int64_t mj, int64_t rj) {
2      return (ri−mi) ∗ (rj−mj);
3  }
4  void cov(int const∗ matrix, int64_t const∗ means,
5           int64_t∗ cov, int N, int length) {
6      cilk_for(int i = 0; i < N; i++) {
7          int64_t row_mean = means[i];
8          int64_t const∗ v1 = matrix + i ∗ N;
9          cilk_for(int j = i; j < N; j++) {
10             int64_t col_mean = means[j];
11             int64_t const∗ v2 = matrix + j ∗ N;
12             int64_t sum=0;
13             for(int64_t k=0; k< length; k++)
14                 sum+= (v1[k] − row_mean) ∗ (v2[k] − col_mean);
15             cov[i∗N+j] = sum / (length−1);
16         }
17     }
18 }
```

Figure 2: The calculation of co-variance (PCA algorithm) expressed in Cilk.

data into independent chunks upon which map tasks can operate. The input data may be a list of key-value pairs read from disk, but can also be other data such as a set of documents. The map-and-combine step breaks each chunk of data apart and transforms it to a list of key-value pairs. The map function may apply a combine function, which performs an initial reduction step. Performing an initial reduction improves performance by reducing the intermediate data set size [33].

Phoenix++ optimizes the storage of intermediate key-value pairs [33]. While a naive implementation would simply use lists, Phoenix++ allows programmers to select intermediate data structures, called *containers*, that are tuned to application properties. Supported containers include hash-maps indexed by key (e.g., for the word count application) and arrays (e.g., when the key is in a densely used integer range). The values in the containers are instances of the *combiner* data structure and hold the (aggregated) associated values, which could be a list or a sum of values.

Every thread produces one instance of the container during the map phase. To facilitate parallel reductions, all instances can be split by key ranges. Each thread reduces the key-value pairs that lie within a key range across all containers. The split is straightforward when the container is a fixed-size array but is more involved for dynamic data structures like a hash map. Finally, the resulting key-value lists are optionally sorted by key and merged into a single key-value list.

Phoenix++ extends the basic map-reduce API. The programmer must select (possibly write) container and combiner data structures that are tuned to the application. This API extension increases the performance of the runtime [33].

B. Performance Limitations

The design of map-reduce systems like Phoenix++ creates many performance limitations. We identify two limitations, which our example application, Principal Components Analysis (PCA), illustrates. The PCA algorithm (Figure 2) calculates a co-variance matrix where position (i, j) lists a correlation metric between items i and j. The Cilk version (Figure 2) exhibits the expected characteristics of the code: two nested loops iterate over the pairs (i, j), calculate a correlation metric and store the value in the output matrix `cov`. In contrast, the map-reduce version (Figure 3) separates out the map and split parts of the computation. The co-variance matrix is represented as key-value pairs by associating the correlation metric (value) to the pair of indices (i, j) (the key).

The map-reduce version is long and tedious. It obfuscates both functionality and performance since the Phoenix++ code (Figure 3) focuses on the mechanics of the computation: how the data is split and processed in parts and how results are reduced.

1) Selection of Data Structures: A list of key-value pairs supports little structuring of the data, which is often rich in structure. Appropriate data structures can improve performance significantly. Phoenix++ provides the option to select a pre-defined data structure, a *container*, to hold the output of the map tasks. An appropriate combiner (reduction operation) must be supplied to combine values with a common key. These issues require a deep understanding of the internals of the map-reduce runtime.

Tuning intermediate data structures breaks the map-reduce abstraction. While key-value pairs often do not support high-performance computation, the programmer must now think about appropriate data structures and how to map key-value pairs to them.

2) Framework Overheads: While many problems match the computational and data organization patterns of map-reduce, others do not. Iterative algorithms with multiple rounds of map and reduce phases, in particular show inefficiencies due to the repeated input, output, shuffling and sorting of key-value pairs. This process has significant computational redundancies, especially if the data has a

```
1  #define MAX_DIM 2000
2  typedef struct {
3      int row_num;
4      int col_num;
5  } pca_cov_data_t;
6  typedef common_array_container<int64_t, int64_t,
7      one_combiner, MAX_DIM*MAX_DIM> cov_container;
8
9  class CovMR : public MapReduceSort<CovMR,
10     pca_cov_data_t, // map input type
11     int64_t,        // key type
12     int64_t,        // value type
13     cov_container   // intermediate key-value container
14 > {
15     int64_t const* matrix;
16     int64_t const* means;
17     int N, length;
18     mutable int row;
19     mutable int col;
20
21 public:
22     CovMR(int64_t const* _matrix, int64_t const* _means,
23           int N, int length) : matrix(_matrix), means(_means),
24         N(N), length(length), row(0), col(0) {}
25     void map(data_type const& data, cov_container& out) const {
26         int const* v1 = matrix + data.row_num*N;
27         int const* v2 = matrix + data.col_num*N;
28         int64_t m1 = means[data.row_num];
29         int64_t m2 = means[data.col_num];
30         int64_t sum = 0;
31         for(int i = 0; i < length; i++)
32             sum += (v1[i] - m1) * (v2[i] - m2);
33         sum /= (length-1);
34         emit_intermediate(out, data.row_num*N + data.col_num, sum);
35     }
36     int split (pca_cov_data_t& out) {
37         if (row >= N) {
38             return 0; // End of data reached
39         } else {
40             out.row_num = row;
41             out.col_num = col;
42             col++;
43             if (col >= N) {
44                 row++;      // Cov is symmetric
45                 col = row;  // only calculate triangle
46             }
47             return 1; // Valid chunk produced
48         }
49     }
50 };
```

Figure 3: Co-variance calculation (PCA) expressed as a map-reduce algorithm in Phoenix++.

richer structure than that captured by its representation as key-value pairs.

K-means clustering is common iterative map-reduce algorithm that demonstrates this issue. This machine learning algorithm summarizes large data sets by assigning points in a multi-dimensional space to one of K clusters. The clusters are tuned to the data by iterating two steps until convergence: (i) assign each point to the closest cluster center; and (ii) update the cluster center based on the assigned points.

In the map-reduce model, each iteration of the two steps is a map-reduce algorithm with its own input and output key-value pairs. Every iteration thus requires data serialization and de-serialization, in particular the updated cluster centers. In contrast, an efficient implementation incurs no cost to communicate cluster centers from one iteration to the next as the data structures are reused across iterations.

```
1  template<class Monoid, class InputIterator, class MapFunctor>
2  void __attribute__(( flatten ))
3  map_reduce( InputIterator ibegin,  InputIterator  iend,
4              MapFunctor mapfn,
5              typename Monoid::value_type & output ) {
6      cilk::reducer<Monoid> imp_;
7      cilk_for(  InputIterator  I=ibegin,  E=iend; I  != E; ++I )
8          mapfn( *I,  imp_.view() );
9      std::swap( output, imp_.view() );
10 }
```

Figure 4: CilkMR map-reduce API call with balanced spawn tree.

C. Summary

The map-reduce model does not support two key performance properties. Instead the programmer must use workarounds or third-party solutions that are hard to compose efficiently. Phoenix++ addresses some of these issues by extending the map-reduce API such that programmers must deeply understand the internals of the runtime. In the following section, we define CilkMR, a map-reduce runtime that presents the reduction operation differently and, thus, is easier to use.

IV. Map-Reduce using Cilk

We use appropriate programming interfaces to support scalable implementations of map-reduce workloads. Our map-reduce runtime builds on the Cilk language [12] because of its support for generalized reductions [11] and Intel's Cilkplus array notation that facilitates auto-vectorization [16].

A. Map-Reduce Code Templates

Cilk is a task-oriented parallel programming model that supports expression of (map) task parallelism (`cilk_for` and `cilk_spawn`) and reduction operations (`cilk::reducer`).

1) Balanced Template: Figure 4 shows one variant of the CilkMR map-reduce API, which uses the `cilk_for` keyword to express that iterations of the loop may execute in parallel (line 7). The map task `mapfn` can be applied in parallel to all items in the data set described by the `ibegin` and `iend` iterators. The template is "balanced" as the directed acyclic graph that describes the parallel activities assigns a comparable number of loop iterations to each processor. Work stealing is minimal during execution of this template if each loop iteration has a comparable amount of work.

```
1  template<class Monoid, class SplitFunctor, class MapFunctor>
2  void __attribute__(( flatten ))
3  map_reduce( SplitFunctor splitfn,  MapFunctor mapfn,
4              typename Monoid::value_type & output ) {
5      cilk ::reducer<Monoid> imp_;
6      typename SplitFunctor::value_type value;
7      while( splitfn ( value ) ) {
8          cilk_spawn [&]( typename SplitFunctor::value_type v ) {
9              mapfn( v, imp_.view() );
10         }( value );
11     }
12     cilk_sync;
13     std ::swap( output, imp_.view() );
14 }
```

Figure 5: CilkMR map-reduce API call with unbalanced spawn tree.

```
1  template<class map_type>
2  struct map_monoid : cilk::monoid_base<map_type> {
3      static void reduce(map_type * left, map_type * right) {
4          for(typename map_type::const_iterator
5                  I=right->cbegin(), E=right->cend(); I != E; ++I)
6              (* left )[ I->first] += I->second;
7          right->clear();
8      }
9      static void identity (map_type * p) const {
10         new (p) map_type();
11     }
12 };
```

Figure 6: Example of a hash-map monoid for counting occurrences of words.

2) Unbalanced Template: Figure 5 shows an unbalanced template for map-reduce. The user defines a functor `splitfn` that splits the input into work items. The `cilk_spawn` statement indicates that the `mapfn` functor may be applied in parallel to the work items. The `map_reduce` method blocks at the `cilk_sync` statement until all spawned tasks complete. This code template is "unbalanced" as the underlying directed acyclic graph that describes the parallel activities is highly skewed. Work stealing will be frequent during its execution.

The structure of the code dictates the choice between the balanced and unbalanced templates. The balanced template can be used when the map-reduce template is over a known range. In other cases, such as text parsing problems, the programmer may need to define a split function to divide the input into independent chunks.

3) Notes: While the Phoenix++ runtime strictly separates map, reduce, sort and merge phases, the CilkMR `map_reduce` templates overlap these activities in time. Thus, load imbalance can potentially impact the Phoenix++ runtime much more.

CilkMR returns a container, e.g., a hash map, instead of a list of key-value pairs. An additional step must serialize the hash map when a list of key-value pairs is desired. This separation clearly portrays the cost of this conversion, which may is often unnecessary, to the programmer. Only one of our benchmarks strictly requires it.

B. Generalized Reductions

The application-specific `Monoid` class defines the reduction through three components [11]: a data type; an associative operation; and an identity value. Cilk reductions do not need to be commutative. Thus, Cilk reducers can support reductions like concatenation of lists.

Figure 6 shows the definition of a monoid for a hash-map data type. The template parameter `map_type` defines the underlying non-concurrent hash-map type. Hash-maps are assumed to be reduced by taking the join of all keys and that the values for common keys are further reduced using an `operator +=` (Line 6). The identity value is an empty hash-map as indicated in the initialization function (Line 9).

The runtime system dynamically creates copies of the reduction variable, and reduces those copies as needed. The creation and reduction of these copies, or *views*, aligns with work-stealing activities in the scheduler. Views are created only after a work stealing event. They are reduced when the stolen task completes. Work stealing activities are rare in highly parallel programs because the design of the Cilk scheduler executes most spawn statements as if they are sequential function calls. In these cases, the same view is used across tasks. Views are reduced under conditions of mutual exclusion. Thus, synchronization rarely impacts application-specific reducer code.

C. Performance Characterization

The *balanced* and the *unbalanced* templates have different parallel scalability. Cilk scheduling overhead is bound by the span of the spawn tree [3]. The *balanced* template uses the `cilk_for` loop, which recursively divides the iteration range in half until a fine granularity is reached. Spawning each half of the range results in a balanced spawn tree. No more than $O(\log n)$ work steals are required for n data items. In contrast, the span of the spawn tree of the *unbalanced* template is $O(n)$.

Reduction operations are proportional to steals [11]. Views are reduced off the critical path and are amortized with steals [11]. Thus, they have no overhead if they take constant time [22].

D. Example: Histogram

Figure 7 shows an algorithm that constructs a fixed-size histogram to demonstrate the use of CilkMR. The code has three parts: the definition of the Monoid (Line 1); the definition of the map task (Line 8); and the call of the map-reduce routine (Line 15). The monoid reduces two histograms, adding up all elements pair-wise. The map function adds 3 successive byte values to appropriate elements of the histogram. The call statement uses a variation of the

```
1  struct Monoid : cilk::monoid_base<uint64_t[768]> {
2      typedef uint64_t value_type[768];
3      static void reduce( value_type *left,  value_type *right ) {
4          for(size_t i=0; i< 768; i++)
5              (*left)[i] += (*right)[i];
6      }
7  };
8  struct histogram_map {
9      void operator() ( const char * pix, uint64_t & histogram[768] ) {
10         histogram[(size_t)pix[0]]++;
11         histogram[256+(size_t)pix[1]]++;
12         histogram[512+(size_t)pix[2]]++;
13     }
14 };
15 uint64_t result[768];
16 cilkmr::map_reduce<Monoid>( byte_array, byte_array_length/3,
17                             histogram_map(),
18                             result );
```

Figure 7: The fixed-length histogram algorithm expressed in CilkMR.

map-reduce template with a begin iterator and a count. This template is a convenience short-hand to define the range using a begin and end iterator (Figure 4).

E. Addressing the Performance Limitations

We discuss how CilkMR addresses performance issues of prior map-reduce systems.

1) Selection of Data Structures: While existing map-reduce systems expose an API to reduce two key-value pairs, the CilkMR runtime exposes reductions on data containers. Thus, the programmer controls the type of container that the program uses. In contrast, Phoenix++ pre-defines several containers and associated reduction operators. Adding a new container in Phoenix++ requires an extension to the runtime. The generic CilkMR approach exposes the selection of data structures in its API.

2) Framework Overheads: Prior map-reduce systems serialize the data and return a list of key-value pairs. Successive map-reduce operators must convert data back and forth between the serialized representation and the efficient representation. The CilkMR map-reduce templates return the data set stored in the selected container type. Successive operators are performed without unnecessary data transformations.

V. Benchmarks

We implement all 7 Phoenix++ benchmarks in CilkMR. Table I describes their Phoenix++ properties. The first column shows the key multiplicity as *m:e*, where m indicates how many map tasks can generate a unique key and e indicates how many keys a map task can emit. Previous reports [33] on these properties are inconsistent with the distributed code. For **matmul** and **strmatch**, the map tasks emit no key-value pairs so the multiplicity is **:0*. Instead, they use shared memory operations to produce output results, which is inconsistent with the spirit of the map-reduce

Table I: Phoenix++ codes: map task multiplicity, combiner data type, sorting, merge and reduction operation.

	map	combiner	sort	reduction
histogram	*:768	array	Y	array add
lreg	*:5	array	N	array add
wc	*:*	hash	Y	hash map join
kmeans	*:K	array	N	array add
matmul	*:0	n/a	N	n/a
pca	1:1	array	Y	array add
strmatch	*:0	n/a	N	n/a

model. The second column shows the intermediate key-value data structure. In most cases, a generic key-value list is optimized to an array indexed by an integer key. For word count, intermediate key-value pairs are stored in a hash map indexed by a character string key. The CilkMR implementations are similar except:

- For **wc**, we use the same hash table as Phoenix++ but we reduce hash table instances following Cilk's schedule of reduction operations, which is markedly different from the separation of map and reduce phases under Phoenix++;
- For **histogram**, we avoid sorting a list of key-value pairs by storing it as an array while Phoenix++ generates a list of key (index)-value pairs that it then sorts;
- For **matmul**, we use a tried-and-tested matrix multiply implementation with good parallel scalability and locality;
- For **pca**, we partition the co-variance matrix among threads and use a scalar Cilk reducer to aggregate the total co-variance;
- For **kmeans**, we use a Cilk reducer object to merge partial results in the computation of cluster averages, which is more efficient than a key-value pair representation;
- For **strmatch**, we make no significant changes to the Phoenix++ distribution, which is simply a parallel for-loop over the input data.

The CilkMR codes (Table II) use similar reduction data structures as the Phoenix++ versions. In some cases, we further specialize to the benchmarks, e.g., a struct of scalars vs. an array for **lreg**. Some benchmarks use the balanced (B) vs. the unbalanced (U) template, nested parallelism, vectorization or sorting. The label (Y) indicates that vectorization is possible, but did not improve performance as expected.

Cilk codes that do not require a reduction are not implemented using the map-reduce API but are implemented using parallel for loops. In the case of **matmul** we used the MIT Cilk-5 implementation, which is known to perform well. The Phoenix++ **strmatch** implementation does not perform a reduction as it produces no output. Thus, we parallelized the loop without using the map-reduce API. These choices are possible as our runtime composes with other parallel code written in the same language. Thus, the programmer

Table II: CilkMR codes: balanced parallelism (bal), using map-reduce API (mr), nested parallelism (nest), vectorization (vec) and sorting (sort).

	reduction	bal	mr	nest	vec	sort
histogram	fixed-size array add	B	Y	N	(Y)	N
lreg	5-scalar struct add	B	Y	N	Y	N
wc	hash table union	U	Y	N	N	Y
kmeans	cluster center add	B	Y	N	(Y)	N
matmul	n/a	B	N	N	N	N
pca	scalar integer add	B	Y	Y	Y	N
strmatch	none	U	N	N	N	N

Table III: Input dataset sizes

	medium	large	huge
histogram	400MB	1.4GB	11.2GB
lreg	100MB	500MB	4GB
wc	50MB	100MB	800MB
kmeans	50,000	75,000	100,000
matmul	512x512	768x768	1024x1024
pca: items	1000	1500	1500
vector length	1000	1500	4500
strmatch	100MB	500MB	4GB

may choose not to use the map-reduce API when appropriate without increasing programming complexity.

VI. Evaluation

We evaluate the programming systems on a quad-socket 2.6GHz Intel Xeon E7-4860 v2, totaling 48 threads. The operating system is CentOS 6.5 with the Intel C compiler version 14.0.1. We compare against Phoenix++ version 1.0 using three input dataset sizes for the Phoenix++ benchmarks (Table III). Three input sizes for the Phoenix++ benchmarks correspond to those used by Talbot *et al* [33]. We create another input through further scaling. For **kmeans** we search for 100 clusters in a 100-dimensional space. The 'wc' dataset contains few large files with sizes varying from 10MB to 800MB. Reported results are averaged over 15 executions.

We experimented with various multi-threaded memory allocators including Hoard [4], SSMalloc [25] and TCMalloc [13]. The resulting performance is similar to that with the default system allocator. They achieve slightly higher performance for some benchmarks but sometimes introduce performance anomalies. For example, **lreg** did not scale well with TCMalloc. The memory allocator does not affect our conclusions as it cannot make up for algorithmic inefficiencies. Thus, we report results for the default allocator.

A. Performance Evaluation: Speedup

Figures 8–10 present the speedup using CilkMR, Cilk and Phoenix++ over the sequential version of the benchmarks. Figure 8 shows the benchmarks dominated by computation in the map phase: **matmul**, **pca** and **kmeans**. Phoenix++ repeatedly serializes and de-serializes the centers to key-value lists for **kmeans**, which reduces scalability. **matmul** and **pca** do not strictly require a reduction operation as each map task deposits its results in distinct locations of an array. Phoenix++ uses this observation for **matmul** (Table I). As previously discussed, we use *plain* Cilk for **matmul** and **pca**.

For **matmul** we use two matrix multiply implementations distributed with MIT Cilk [12]. The **matmul** version uses recursive decomposition where on each level of recursion the problem is split along its largest dimension. The **rectmul** version splits the target matrix along both dimensions on each level of recursion and has a much higher degree of parallelism. Also, its leaf task, a 16x16 block multiply, is highly optimized. Figure 8 normalizes performance to the sequential version of **matmul**. We also present the performance of `dgemm` from the Intel MKL library. These results show that specifically optimized codes clearly outperform a generic map-reduce framework like Phoenix++, which shows that blindly applying the map-reduce concept to every problem is not sensible. Further, the map-reduce runtime is used inappropriately for **matmul** as the map task accesses shared memory and does not emit key-value pairs.

The memory-bound benchmarks **histogram** and **lreg** show good scalability with both map-reduce systems (Figure 9). Both benchmarks perform few operations per input byte – 4 integer operations for **histogram** and 7 for **lreg**. The CilkMR version accelerates faster with increasing thread counts but eventually saturates. Saturation occurs at lower thread counts with smaller inputs, which suggests that the Cilk scheduler carries a higher burden than the Phoenix++ scheduler. Interestingly, the specialized map-reduce system performs better on smaller inputs.

The benchmarks **wc** and **strmatch** use the unbalanced CilkMR template (Figure 10). The performance of these benchmarks is nearly identical to that of the Phoenix++ version up to around 8–24 threads, depending on the problem size, after which the inefficiency of the unbalanced template limits scalability.

B. Addressing Performance Limitations

1) Internal Data Structures: Our initial CilkMR implementation of **wc** performed poorly because we used the default C++ STL `unordered_map`. This hash map data structure performs badly for map-reduce applications because it balances performance against space. In map-reduce applications, however, insert operations dominate execution time, so the performance trade-off does not arise.

We optimized the STL unordered map by defining a resizing policy that restricts the hash table size to a power of 2 and a hash function that selects the lowest bits of the integer key. Despite improvements, the Phoenix++ hash table still outperforms it since the STL code dynamically allocates memory for each element of the hash table.

We conclude that map-reduce applications are sensitive to the performance of the data structures due to the gener-

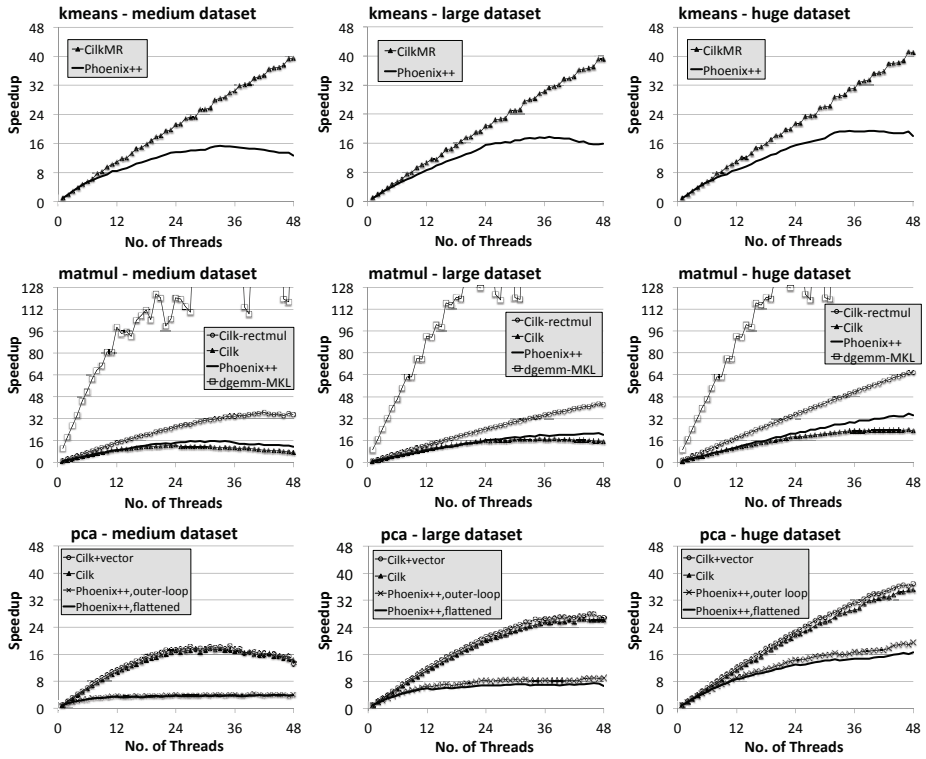

Figure 8: Results (a): Applications dominated by map time (compute-bound).

Table IV: Peak heap memory usage (MB) in excess of data set size when using the large data set.

		Memory usage (MB) for thread count.			
		1	**16**	**32**	**48**
histogram	CilkMR	0.06	0.95	1.67	2.50
	Phoenix++	0.04	0.43	0.86	1.23
lreg	CilkMR	0.06	0.69	1.39	2.08
	Phoenix++	<0.01	0.06	0.11	0.17
wc	CilkMR	11.70	28.10	34.30	34.00
	Phoenix++	15.10	60.60	98.30	117.00
pca	Cilk	25.82	26.44	27.13	27.82
	Phoenix++	159.90	161.50	160.00	160.10
kmeans	CilkMR	39.81	41.66	42.98	44.55
	Phoenix++	68.62	502.00	963.2	1423.4
strmatch	CilkMR	0.06	0.69	1.38	2.07
	Phoenix++	0.56	0.58	0.61	0.73
matmul	Cilk	4.06	4.69	5.39	5.35
	Phoenix++	4.06	4.16	4.27	4.39

ally low amount of computation per data structure access. Thus, appropriate data structure selection is essential. This observation holds across programming models.

C. Memory Consumption

Low memory consumption is important as map-reduce workloads tend to be applied to large data sets and main memory is limited. Table IV shows the peak heap memory when using the large input. We use the valgrind tool "massif" [28] to collect this data. We subtract the memory required to store the input data set for clarity. For many benchmarks, the runtime requires little additional space over the data set. Nonetheless, the internal data structure size grows moderately as the thread count increases by about 18 KB per thread for CilkMR and about 1 KB per thread for Phoenix++. This difference arises because CilkMR requires a varying number of stacks depending on how work stealing progresses [21].

Three benchmarks consume significant additional memory: **wc**, **pca** and **kmeans**. They store large volumes of data in the intermediate data structures that support the reduction. This space increases rapidly with thread count for Phoenix++, while the memory utilization remains fairly constant for CilkMR. The Cilk reducers repeatedly merge small data sets throughout the computation while Phoenix++ collects all key-value pairs during the map phase prior to initiating the reduction phase. Thus, CilkMR avoids creating large numbers of copies. The excessive memory consumption of **kmeans** arises from a deliberate memory leak, created for performance reasons.

VII. Conclusion

We have presented a scalable and composable map-reduce runtime system. Our runtime system, called CilkMR, builds on the Cilk parallel programming language in order to reap opportunities for performance optimization that are out of scope of state-of-the-art specialized map-reduce systems.

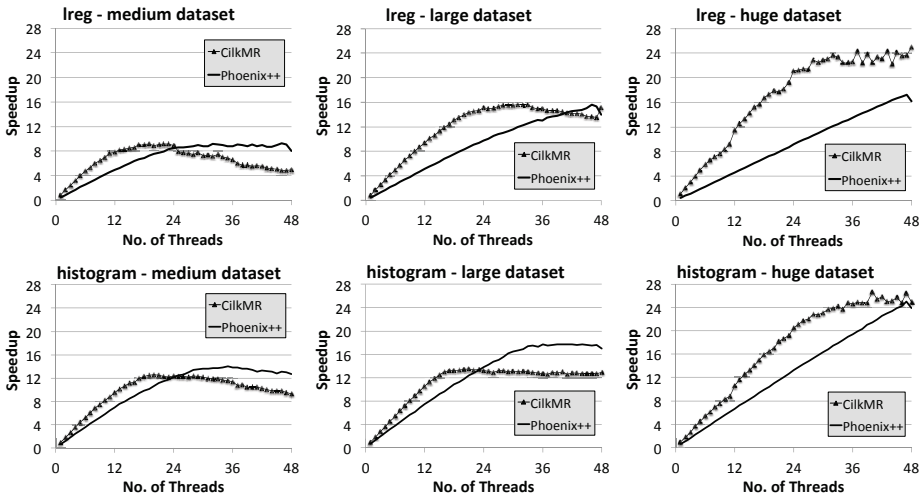

Figure 9: Results (b): Memory-bound applications.

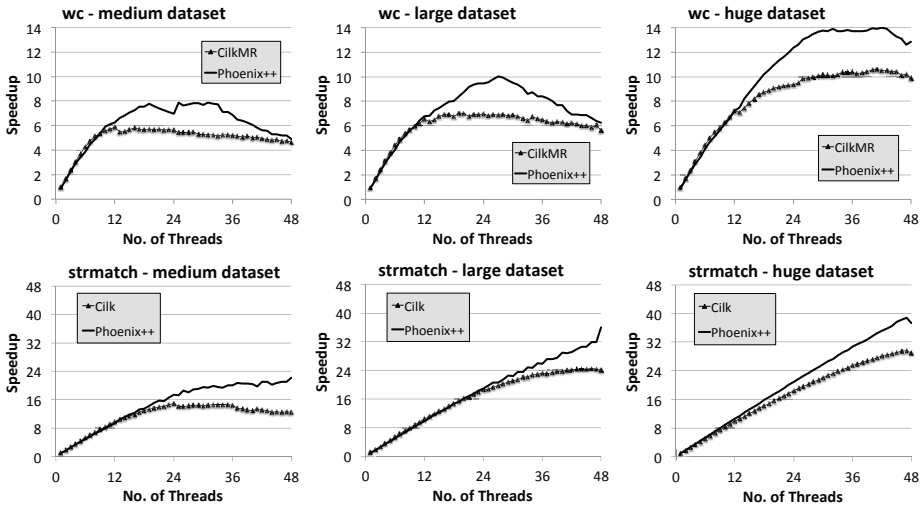

Figure 10: Results (c): Applications with unbalanced spawn trees.

Ease of programming with CilkMR is similar to other map-reduce systems. CilkMR supports composition of multiple, potentially nested, map-reduce kernels. In contrast, state-of-the-art map-reduce programs require redesign of the parallel structure from first principles when composing codes.

We evaluated several map-reduce benchmarks implemented in the Cilk parallel programming language and in Phoenix++, a state-of-the-art shared-memory map-reduce runtime. Our evaluation shows that on a 48-core workstation, the Cilk codes perform 1.5x–4x better, although performance is reduced by up to 30% for two applications where the Cilk versions use an inefficient parallel code structure.

Our performance evaluation demonstrates that CilkMR offers a better parallel implementation of the map-reduce pattern than Phoenix++. It differs from the Phoenix++ approach by not representing data as key-value pairs when appropriate. Viewing our map-reduce templates as a library extension to a generic parallel programming language, they also avoid inappropriate map-reduce-inspired algorithm design.

Some algorithms have been extensively studied and high-performance implementations have been constructed for them. From a viewpoint of programmability, it is advisable to re-use these implementations, e.g., through standardized libraries. As shown in our evaluation of matrix multiply, K-means and PCA, forcing these applications to match the map-reduce API hinders performance. CilkMR supports composing map-reduce code with existing code.

This work first applies to large-scale shared memory servers, which can scale to terabyte-sized main memory. An important avenue for future work is to investigate how the representation of map-reduce programs affects the design of

distributed map-reduce systems, in particular whether these benefit equally from representing the reduction operation over containers as opposed to individual key-value pairs.

REFERENCES

[1] Z. Anderson, *Efficiently Combining Parallel Software Using Fine-grained, Language-level, Hierarchical Resource Management Policies*. In OOPSLA '12. 717–736.

[2] M. Arif and H. Vandierendonck. 2015. *A Case Study of OpenMP Applied to Map/Reduce-Style Computations*. In IWOMP'15. 162–174.

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. 1998. *Thread Scheduling for Multiprogrammed Multiprocessors*. In SPAA '98. 119–129.

[4] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. 2000. *Hoard: A Scalable Memory Allocator for Multithreaded Applications*. In ASPLOS IX. 117–128.

[5] R.E. Bryant, R.H. Katz, and E.D. Lazowska. 2008. *Big-Data Computing: Creating revolutionary breakthroughs in commerce, science and society*. http://www.cra.org/ccc/files/docs/init/Big_Data.pdf. (Dec. 2008).

[6] R. Chen and H. Chen. 2013. *Tiled-MapReduce: Efficient and Flexible MapReduce Processing on Multicore with Tiling*. ACM Trans. Archit. Code Optim. 10, 1, Article 3 (April 2013), 30 pages.

[7] C. Csallner, L. Fegaras, and C. Li. 2011. *New Ideas Track: Testing Mapreduce-style Programs*. In ESEC/FSE '11. 504–507.

[8] M. de Kruijf and K. Sankaralingam. 2009. *MapReduce for the Cell Broadband Engine Architecture*. IBM Journal of Research and Development 53, 5 (Sept 2009), 10:1–10:12.

[9] J. Dean and S. Ghemawat. 2004. *MapReduce: Simplified Data Processing on Large Clusters*. In OSDI'04. 10–10.

[10] B. Fish, J. Kun, Á D. Lelkes and L. Reyzin, and György Turán. 2014. *On the Computational Complexity of MapReduce*. DISC 2015.

[11] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. 2009. *Reducers and other Cilk++ hyperobjects*. In SPAA '09. 79–90.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall. 1998. *The implementation of the Cilk-5 multi-threaded language*. In PLDI '98 . 212–223.

[13] S. Ghemawat and P. Menage. *Tcmalloc: Thread-caching malloc*. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[14] Hadoop 2015. Apache Hadoop. http://hadoop.apache.org. (2015).

[15] T. Harris, M. Maas, and V J. Marathe. 2014. *Callisto: Co-scheduling Parallel Runtime Systems*. In EuroSys '14. Article 24, 14 pages.

[16] Intel 2013. Intel Cilk Plus Language Extension Specification (version 1.2. 324396-003us ed.).

[17] W. Jiang and G. Agrawal. 2011. *Ex-MATE: Data Intensive Computing with Large Reduction Objects and Its Application to Graph Mining*. In CCGrid"11. 475–484.

[18] W. Jiang, V. T. Ravi, and G. Agrawal. 2010. *A Map-Reduce System with an Alternate API for Multi-core Environments*. In CCGRID '10. 84–93.

[19] K. Kennedy and J. R. Allen. 2002. *Optimizing Compilers for Modern Architectures: a Dependence-based Approach*. Morgan Kaufmann Publishers Inc., USA.

[20] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. 2011. *Filtering: A Method for Solving Graph Problems in MapReduce*. In SPAA '11. ACM, 85–94.

[21] I-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson. 2010. *Using Memory Mapping to Support Cactus Stacks in Work-stealing Runtime Systems*. In PACT '10. 411–420.

[22] C E. Leiserson and T B. Schardl. 2010. *A Work-efficient Parallel Breadth-first Search Algorithm (or How to Cope with the Nondeterminism of Reducers)*. In SPAA '10. 303–314.

[23] J. Leskovec, A. Rajaraman, and J. D. Ullman. 2014. *Mining of Massive Datasets* (2 ed.). Cambridge University Press.

[24] J. Lin and M. Schatz. 2010. *Design Patterns for Efficient Graph Algorithms in MapReduce*. In MLG '10. 78–85.

[25] R. Liu and H. Chen. 2012. *SSMalloc: A Low-latency, Locality-conscious Memory Allocator with Stable Performance Scalability*. In APSys '12. 15–15.

[26] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R.S.M. Goh, and R. Huynh. 2013. *Optimizing the MapReduce framework on Intel Xeon Phi coprocessor*. In IEEE Big Data,2013. 125–130.

[27] Y. Mao, R. Morris, and F. Kaashoek. 2010. *Optimizing MapReduce for Multicore architectures*. Technical Report MIT-CSAIL-TR-2010-020. MIT Computer Science and Artificial Intelligence Laboratory.

[28] N. Nethercote and J. Seward. 2007. *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation*. In PLDI '07. 89–100.

[29] OpenMP Application Programming Interface, version 4.0. http://www.openmp.org/. (July 2013).

[30] H. Pan, B. Hindman, and K. Asanović. 2010. *Composing Parallel Software Efficiently with Lithe*. In PLDI '10. 376–387.

[31] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. 2007. *Evaluating MapReduce for Multi-core and Multiprocessor Systems*. In HPCA '07. 13–24.

[32] J. Shun and G. E. Blelloch. 2013. *Ligra: A Lightweight Graph Processing Framework for Shared Memory*. In PPoPP '13. 135–146.

[33] J. Talbot, R. M. Yoo, and C. Kozyrakis. 2011. *Phoenix++: Modular MapReduce for Shared-memory Systems*. In MapReduce '11. 9–16.

[34] T. Xiao, J. Zhang, H. Zhou, Z. Guo, S. McDirmid, W. Lin, W. Chen, and L. Zhou. 2014. *Nondeterminism in MapReduce Considered Harmful? An Empirical Study on Non-commutative Aggregators in MapReduce Programs*. In ICSE Companion 2014. 44–53.

[35] R. M. Yoo, A. Romano, and C. Kozyrakis. 2009. *Phoenix Rebirth: Scalable MapReduce on a Large-scale Shared-memory System*. In IISWC '09. 198–207.

[36] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M J. Franklin, S. Shenker, and I. Stoica. 2012. *Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing*. In NSDI'12. 2–2.