Evaluating MapReduce for Multi-core and Multiprocessor Systems

Abstract

As multi-core chips become ubiquitous, it is critical to develop parallel programming models and runtime systems that can harness their computational capabilities. In this paper, we evaluate the suitability of the MapReduce model for multi-core and multi-processor systems. MapReduce was developed by Google to program and manage data-centers with thousands of servers. It allows programmers to write functional-style code that is automatically parallelized and scheduled on a distributed system.

We describe Phoenix, an implementation of MapReduce for shared-memory parallel systems that includes a programming API and an efficient runtime system. The Phoenix runtime automatically manages thread-generation, dynamic task scheduling, data partitioning, and fault-tolerance across the processor nodes. We evaluate Phoenix with a diverse set of benchmarks on both multi-core and symmetric multiprocessor systems. We demonstrate that Phoenix leads to excellent speedups with both systems types. The speedups are robust across a wide range of system and dataset characteristics. We also show that Phoenix can automatically recover from transient and permanent errors on map and reduce tasks. Finally, we compare to parallel versions of the benchmarks written in the lower-level Pthreads API and demonstrate that despite the overheads associated with the MapReduce model, Phoenix leads to competitive performance with significantly simpler code. Overall, we establish that MapReduce provides a promising method for programming and managing parallel applications on multi-core and other shared-memory systems.

1 Introduction

As multi-core chips become ubiquitous, we need concurrent executables that can exploit their parallel execution resources. The traditional parallel programming approaches, message-passing and shared-memory multi-threading, are too cumbersome for most developers. These models require that the programmer manages concurrency directly by creating and synchronizing threads through messages or locking primitives. They also require manual management of data locality. Consequently, it is very difficult to write correct and high-performance parallel programs for non-trivial algorithms. Moreover, the programmer must often re-tune the granularity of parallelism, the synchronization approach, or the locality enhancements when the application is ported to a different or larger-scale system.

To simplify parallel application development we need to develop two components: a practical programming model that...
allows users to specify concurrency and locality at a high level; an efficient runtime system that handles low-level mapping, resource management, and fault tolerance issues automatically for a wide range of applications, system characteristics, and system scales. Naturally, the two components are closely linked. Recently, there has been a significant body of research towards these goals using approaches such as streaming [16, 18], shared-memory with transactional semantics [17, 6], speculative multi-threading [27, 8], data-flow based schemes [3], asynchronous parallelism and work stealing [14], and partitioned global address space languages [7, 9, 2].

This paper presents Phoenix, a programming API and runtime system based on Google’s MapReduce [11]. The MapReduce programming model borrows two concepts from functional languages to express data-intensive algorithms. The Map function processes every key/value pair of the input to generate a set of intermediate key/value pairs. The Reduce function properly merges the intermediate pairs with the same key. Given such a program specification, the MapReduce runtime automatically parallelizes the computation by running multiple map and/or reduce tasks in parallel over disjoined portions of the input or intermediate data. The runtime can also handle data partitioning across the parallel system and failures. Google’s MapReduce implementation has been used to process terabytes of datasets on clusters with hundreds to thousands of nodes. The Phoenix implementation is based on the same principles but targets shared-memory systems such as multi-core chips and symmetric multiprocessors.

Phoenix uses threads and buffers in shared memory to spawn parallel Map or Reduce tasks and facilitate their communication. Most communication occurs through pointer manipulation to avoid copying of the actual data. The runtime schedules tasks dynamically across the available processors to achieve load balance and maximize task throughput. Locality can be managed by adjusting the granularity of parallel tasks. Finally, the Phoenix runtime can automatically recover from transient and permanent faults during task execution by repeating or re-assigning tasks and properly merging their output with that from the rest of the computation. Overall, Phoenix can handle in the runtime the complicated concurrency, locality, and fault-tolerance tradeoffs that if left to the user can make parallel programming difficult. Nevertheless, it also allows the programmer to pass application specific knowledge such as custom data partitioning functions (if desired).

The specific contributions of this work are:

- We present the design of Phoenix, an efficient implementation of the MapReduce API and runtime for multi-core chips and symmetric multiprocessors. Phoenix exposes a very simple interface to the user (8 functions) and encapsulates all parallelism management in a runtime system. We describe the key functions, data structures, and control mechanisms that implement the runtime.
- We evaluate Phoenix on commercial chip-multiprocessor (Sun Fire T1200) and symmetric multiprocessor (Sun Ultra-Enterprise 6000) systems. We show that Phoenix leads to excellent parallel efficiency in both environments. We also evaluate various scheduling tradeoffs such as the number of threads per processor, the granularity of tasks size, and the impact of dataset size on the system efficiency. Through fault injection experiments, we show that Phoenix can handle both permanent and transient faults on task execution at a small performance penalty.
- We compare Phoenix programs to Pthreads-based parallel code. Despite the overheads due to the API structure, Phoenix is competitive to Pthreads code. In many cases, Phoenix is actually faster as the programmer gets automatic dynamic
scheduling and concurrency control without having to code it manually using Pthreads locking primitives.

While we do not claim that MapReduce is applicable to all algorithms, this work establishes that it is a valuable tool for simple parallel programming and resource management on multi-core systems.

The rest of the paper is organized as follows. Section 2 provides an overview of MapReduce. In Section 3, we present the Phoenix implementation of MapReduce for shared-memory multiprocessors. Section 4 describes our evaluation methodology and Section 5 presents the evaluation results. Section 6 reviews related work and Section 7 concludes the paper.

2 MapReduce Overview

This section summarizes the basic principles of the MapReduce model. MapReduce is actively used by Google to quickly develop distributed applications that process terabytes of data and run on clusters of hundreds to thousands of machines [11]. There is also an effort to develop an open-source distributed system with similar characteristics [10].

2.1 Programming Model

The MapReduce programming model is inspired by functional languages and targets data-intensive computations. The inputs and outputs are sets of \(< \text{key}, \text{value} >\) pairs. The user expresses an algorithm using two functions, Map and Reduce. The Map function is applied on the input data and produces a list of intermediate \(< \text{key}, \text{value} >\) pairs. The Reduce function is applied to all intermediate pairs with the same key. It typically performs some kind of merging operation and produces zero or more output pairs. Finally, the output pairs are sorted by their key value. In the simplest form of MapReduce programs the programmer provides just these two functions. All other functionality, including the grouping of the intermediate pairs with the same key and the final sorting, is provided by the runtime.

The following pseudocode from [11] shows the basic structure of a MapReduce program that counts the number of occurrences of each word in a collection of documents. The map function emits each word in the documents with the temporary count 1. The reduce function sums the counts for each unique word.

```
// input: key=document name; value=document contents
// intermediate output: key=word; value=1
map(String key, String value) {
    for each word w in value
        EmitIntermediate(w, 1);
}

// intermediate output: key=word; value=1
// output: key=word; value=occurences
reduce(String key, Iterator values) {
    int result = 0;
    for each v in values
        result += v;
    Emit(w, result);
```
The benefits of this programming model is simplicity. The programmer provides a simple description of the algorithm that focuses on functionality and not on parallelization. The actual parallelization (if any) and all the details of concurrency management are left to the runtime system. Hence the program description is generic and easily portable across systems and datasets. Nevertheless, the program structure provides sufficient high-level information for parallelization. The Map function can be executed in parallel on non-overlapping portions of the input data and the Reduce function can be executed in parallel on each set of intermediate pairs with the same key. Similarly, since it is explicitly known which pairs each function will operate upon, one can employ prefetching or other scheduling optimizations for locality.

The critical question here is how widely applicable is the MapReduce programming model. In [11], Google provides several examples of data-intensive problems they have successfully coded in this model, including their production indexing system, distributed grep, web-link graph construction, calculations of URL access frequencies, and statistical machine translation. We provide further examples in Section 5. A more extensive study by Intel has also concluded that a large number of data-intensive computations can be expressed as sums over data points [12]. Such computations should be a good match for the MapReduce model. Nevertheless, an extensive study that evaluates the applicability and ease-of-use of the model is beyond the scope of this paper. Our goal is to provide an efficient implementation on shared-memory systems that demonstrates its feasibility and enables programmers to experiment with this programming approach.

2.2 Runtime System

The MapReduce runtime is responsible for parallelization and various other optimizations in a parallel system. To parallelize the Map function, the runtime splits the input pairs into processing units and uses multiple nodes to process the units concurrently. Each node runs a Map task. Next, the runtime splits the intermediate pairs using a scheme that assigns pairs with the same key to the same split. Multiple nodes are used to run a Reduce task on each split. In both steps, the runtime must decide on factors such as the size of the units, the number of nodes involved, how units are assigned to nodes dynamically, and how buffer space is allocated and resized. The decisions can be fully automatic or guided by the programmer given application specific knowledge (e.g., number of pairs produced by each function, distributions of keys, intermediate storage requirements, etc). These decisions allow the runtime to execute a program efficiently across a wide range of machines and dataset scenarios without modifications to the source code. As a final step, the runtime may need to merge and sort the output pairs from all tasks.

The runtime can perform several optimizations. Runtime overheads can be reduced by increasing the granularity of Map or Reduce units. On the other hand, load imbalance across nodes can be decreased by decreasing the granularity of units. Locality can be improved in several ways. First, each node can prefetch pairs from current Map or Reduce units using hardware or software schemes. The output data from each Map or Reduce task can be marked as least-recently-used to optimize cache replacement policies [30]. A node can also prefetch its next Map or Reduce split while processing the current one, which is similar to the double-buffering schemes used in streaming models [28]. Bandwidth and cache space can be preserved using hardware compression of intermediate pairs which tend to have high redundancy [1, 13].

The runtime can also assist with fault-tolerance. When it detects or suspects that a node has failed, it can re-assign the Map
or Reduce task it was processing at the time to another node. In this case, the runtime will use a different output buffer to avoid interference. If a portion of the memory storage is corrupted, it can re-execute the Map or Reduce task that produced the affected data. The runtime can also produce an useful partial or approximated output even if it cannot recover from some faults (e.g. some input or intermediate data are permanently lost). Moreover, the runtime can dynamically adjust the number of nodes it uses to deal with failures, load issues, or power and temperature related issues. Of course, the runtime itself must be made resilient to failures using known techniques such as redundant execution or checkpointing [24, 26].

Google’s runtime implementation targets execution over large clusters of Linux PCs interconnected through Ethernet switches [4]. Tasks are forked using remote procedure calls and buffering and communication occurs by reading and writing files on a distributed file system [15]. The locality optimizations focus mostly on avoiding remote file accesses. While such an implementation is efficient for distributed computing [11], it leads to very high overheads if used with a shared-memory systems. Moreover, a shared-memory system is of a smaller scale than the Google cluster. The Phoenix implementation presented in this paper targets such shared-memory systems by using threads to execute Map and Reduce tasks and shared memory for buffering and communication.

Overall, the critical question for the MapReduce runtime is how significant are the overheads it introduces. The structure of MapReduce computations requires that the inputs and outputs are set up correctly at each stage (distributing original input, combining intermediate outputs with the same key, sorting the final output, etc.). In a shared memory system, we can reduce the data management overhead by manipulating pointers instead of moving the actual data. Nevertheless, the pointer manipulation may still introduce significant overheads that reduce the parallelization benefits. While programmers may be willing to sacrifice some of the parallel efficiency for a simple programming model, we must show that the overheads are not overwhelming.

3 The Phoenix System

Phoenix implements the MapReduce model on shared-memory systems. Its goal is to allow programmers to benefit from the parallel cores in such systems without explicitly managing synchronization, resource allocation, locality, or scaling on the target system. It consists of two components, a simple API that is visible to application programmers and an efficient runtime that handles parallelization, resource management, and fault-tolerance.

3.1 The Phoenix API

Phoenix provides an application-programmer interface (API) for C and C++. Similar interfaces can be defined for other languages like Java and C#. The API includes two sets of functions summarized by Table 1. The first set is implemented by the Phoenix runtime and is used by the programmer in the application source code (1 required and 2 optional functions). The second set includes the functions that the runtime expects the programmers to provide in their code (3 required and 2 optional functions). Note that the API is quite small compared to other models (4 required and 4 optional functions total). The API is type agnostic. The arguments to the functions are declared as void pointers wherever possible to ensure maximum flexibility in their declaration and fast use without the need for conversion. In contrast, the Google MapReduce implementation uses strings as arguments in all cases.
Table 1. The runtime-defined and user-provided functions in the Phoenix API. R stands for required and O for optional.

<table>
<thead>
<tr>
<th>API Function</th>
<th>R/O</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int phoenix_scheduler (scheduler_args_t * args)</td>
<td>R</td>
<td>Initializes the runtime scheduler; the scheduler_args_t struct includes the data and function pointers required by the runtime.</td>
</tr>
<tr>
<td>void emit_intermediate(void *key, void *val, int key_size)</td>
<td>O</td>
<td>Called by Map to emit an intermediate output &lt;key, value&gt; pair. It is required if the Reduce function is used.</td>
</tr>
<tr>
<td>void emit(void *key, void *val)</td>
<td>O</td>
<td>Called by Reduce to emit a final output pair. It is required if the outputs of Reduce with the same key must be merged.</td>
</tr>
<tr>
<td>int (*splitter_t)(void *, int, map_args_t *)</td>
<td>R</td>
<td>Used to split the Map input; the three arguments are: the input data pointer, the number of pairs assigned to each task; and the pointer to buffer with the input for a specific Map task.</td>
</tr>
<tr>
<td>void (<em>map_t)(map_args_t</em>)</td>
<td>R</td>
<td>Map function</td>
</tr>
<tr>
<td>void (*reduce_t)(void *, void **, int)</td>
<td>O</td>
<td>Reduce function; the three arguments are: pointer to a key, pointer to array of values for that key, and length of the array; if not specified, Phoenix uses a built-in identity reduce.</td>
</tr>
<tr>
<td>int (*partition_t)(int, void *, int)</td>
<td>O</td>
<td>Partitions intermediate pairs based on their keys; the three arguments are: number of reduce tasks, pointer to a key, and size of key; Phoenix provides a default partition function based on hashing keys.</td>
</tr>
<tr>
<td>int (*key_cmp_t)(const void <em>, const void</em>)</td>
<td>R</td>
<td>Compares two keys; used during the final sorting step; since keys are of generic type, an explicit function is required.</td>
</tr>
</tbody>
</table>

The data structure used to communicate basic function information and buffer allocation between the user code and runtime is of type scheduler_args_t. Its fields are summarized in Table 2. The basic fields provide pointers to input/output data buffers and to the user-provided functions. They must be properly filled by the programmer before calling phoenix_scheduler(). The remaining fields are optionally used by the programmer to control scheduling decisions by the runtime. We discuss these decisions further in Section 3.2.4.

There are also data structure types to communicate data between the Splitter, Map, and Reduce functions. The map_args_t type communicates data between the Splitter and the Map tasks. It is updated in the Splitter function before being passed to Map. It contains a pointer to the actual input data partition for this task and its length, measured in user-defined units. The final_data_t type stores the final data to be passed by Phoenix back to the user code. It contains an array of keyval_t structures and the length of the array. The keyval_t structure stores a pointer to a key and a pointer to the final output value for the specific key.

The API guarantees that within a partition of the intermediate output, the pairs will be processed in key order. This makes it easier to produce a sorted final output which is often desired. There is no guarantee in the processing order of the original input during the Map stage. These assumptions did not cause any complications with the programs we examined, but in general it is up to the programmer to verify that the algorithm can be expressed with the Phoenix API given these restrictions.

The Phoenix API does not rely on any specific compiler options and does not assume a parallelizing compiler. The API assumes that Map, Reduce, and Merge tasks can freely use stack-allocated and heap-allocated structures for private data. It also assumes that there is no communication through shared-memory structures other than the input/output buffers for these tasks. For C/C++, we have no way of statically checking if this assumption holds for a random program. Although there are
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input_data</td>
<td>Input data pointer; passed to the Splitter by the runtime.</td>
</tr>
<tr>
<td>Data_size</td>
<td>Input dataset size.</td>
</tr>
<tr>
<td>Map</td>
<td>Pointer to Map function.</td>
</tr>
<tr>
<td>Reduce</td>
<td>Pointer to Reduce function.</td>
</tr>
<tr>
<td>Splitter</td>
<td>Pointer to Splitter function.</td>
</tr>
<tr>
<td>Key_cmp</td>
<td>Pointer to Key Compare function.</td>
</tr>
<tr>
<td>Partition</td>
<td>Pointer to Partition function.</td>
</tr>
<tr>
<td>Output_data</td>
<td>Output data pointer (type final_data_t); buffer space allocated by user.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit_size</td>
<td>Number of input pairs processed by each Map or Reduce task.</td>
</tr>
<tr>
<td>L1_cache_size</td>
<td>L1 data cache size in bytes.</td>
</tr>
<tr>
<td>Num_map_workers</td>
<td>Maximum number of workers (threads) used for Map tasks.</td>
</tr>
<tr>
<td>Num_reduce_workers</td>
<td>Maximum number of workers (threads) used for Reduce tasks.</td>
</tr>
<tr>
<td>Num_merge_workers</td>
<td>Maximum number of workers (threads) used for merge tasks.</td>
</tr>
<tr>
<td>Num_procs</td>
<td>Maximum number of processors (cores) used by Phoenix.</td>
</tr>
<tr>
<td>Key_match_factor</td>
<td>Used to determine the number of reduce tasks.</td>
</tr>
</tbody>
</table>

Table 2. The basic and optional fields of the scheduler_args_t data structure.

stringent checks within the system to ensure valid data is communicated between the user and the runtime, eventually we have to trust the user to provide functionally correct code. For Java and C#, static checks that validate these assumptions are possible.

3.2 The Phoenix Runtime

The Phoenix runtime handles parallelization, resource management, and fault tolerance. It is currently based on Pthreads [21], but it can be easily ported to other shared-memory thread packages.

3.2.1 Basic Operation and Control Flow

Figure 1 shows the basic control flow of the runtime system for the Map and Reduce portions of a computation respectively. The runtime is controlled by the scheduler, which is initiated by the user at the beginning of the program (1). The scheduler creates and manages the threads that run all Map, Reduce, and Merge tasks. It also manages the buffers used to pass data between the steps of the computation. The programmer provides the scheduler with all the required data and function pointers through the scheduler_args_t structure. After initialization, the scheduler determines the number of cores to use for this computation. For each core, it spawns a worker thread (2). Each worker thread will be assigned some number of Map, Reduce, or Merge tasks dynamically.

To start the Map stage, the scheduler uses the Splitter function to split input pairs into equally sized units (3). Each unit is processed by a Map task. The Splitter is called once per Map task and returns a pointer to the data this task will process. The map tasks are allocated dynamically to workers and each one returns a pointer to the intermediate pairs they produced (4). The Phoenix system makes no assumptions about the type of the intermediate data. The user-defined Partition function is used to split the intermediate pairs into units for the Reduce tasks (5). The function ensures all values of the same key go to the same unit. Within each buffer, values are ordered by key to assist with the final sorting. At
Figure 1. The basic control flow of the Phoenix runtime. The Map stage is shown on the left and the Reduce stage on the right.

At this point, the Map stage of the computation is over. The scheduler must wait for all Map tasks to complete before initiating any Reduce tasks.

Reduce tasks are assigned to workers dynamically similarly to Map tasks (Θ). The one difference is that, while with Map tasks we have complete freedom in distributing pairs across tasks, with Reduce we must process all values for the same key in one task. Hence, the Reduce stage may exhibit higher imbalance across workers and dynamic scheduling is more important. The output of each Reduce task is already sorted by key. As the last step, the final output from all tasks is merged into a single buffer, sorted by keys (Ω). The merging takes place in $\log_2(P/2)$ steps, where $P$ is the number of workers used. While one can imagine cases where the output pairs do not have to be ordered, our current implementation always sorts the final output as it is also the case in Google’s implementation [11].

### 3.2.2 Buffer Management

The data structures that facilitate the communication between the user program and the runtime are described in Section 3.1. Additional buffers are necessary to store the data between the Map, Reduce, and Merge tasks. All buffers are allocated in shared memory but are accessed in well specified way by a few functions. Whenever we have to rearrange buffers (e.g., split
across tasks), we actually manipulate pointers not the actual pairs, which may be large in size. These intermediate buffers are not directly visible to user code.

*Map-Reduce buffers* are used to store the intermediate output pairs. Each worker has its own set of buffers. The buffers are initially sized to a default value and then resized dynamically if needed. At this stage, there may be multiple pairs with the same key. To accelerate the *Partition* function, we store all values for the same key in the same buffer. This functionality is implemented by the *Emit_intermediate* function. At the end of the Map task, we sort each buffer by key order. *Reduce-Merge buffers* are used to store the outputs of Reduce tasks before they are sorted in the Merge step. At this stage, each key has only one value associated with it. After sorting, the final output is available in the *Output_data* buffer, which is allocated by the user. All intermediate buffers are released as soon as the data they contained has been successfully processed.

### 3.2.3 Fault Tolerance

The Phoenix runtime provides support for fault-tolerance for transient and permanent faults during Map, Reduce, and Merge tasks. The support is focused mostly on recovery with some limited support for fault detection.

Phoenix detects faults through timeouts. If a worker does not complete a task within a reasonable amount of time, then the runtime assumes there was a failure. The execution time of similar tasks on other workers is used as a yardstick for the timeout interval. Of course, a fault may cause a task to complete with incorrect or incomplete data instead of failing completely. Phoenix has no way of detecting this case on its own and cannot stop an affected task from potentially corrupting the shared memory. To address this shortcoming, we could combine the Phoenix runtime with known error detection techniques, either low-level circuit schemes [23] or high-level architectural approaches [24, 29]. Such a combination can simplify significantly any detection technique that relies on redundant computation. The functional nature of MapReduce programs allows the Phoenix runtime to handle input replication without hardware support. Since the address ranges for input and output buffers are known, Phoenix can notify the hardware which load/store addresses to shared structures should be considered safe for each thread and which should signal a potential fault. We leave the exploration of these issues to future work.

Once a fault is detected or at least suspected, Phoenix attempts to re-execute the failed task. Since the original task may still be running, separate output buffers are allocated for the new task to avoid conflicts and data corruption. When one of the two tasks completes successfully, the runtime considers the task completed and merges its result with the rest of the output data for this stage. The scheduler initially assumes that the fault was a transient one and assigns the new (replicated) task to the same worker. If the task fails a few times or a worker exhibits a high frequency of failed tasks overall, the scheduler assumes a permanent fault and no further tasks are assigned to this worker.

The current implementation of Phoenix does not provide fault-tolerance for the scheduler itself. The scheduler runs only for a very small fraction of the execution time and has a small memory footprint, hence it is less likely to be affected by a transient error. On the other hand, a fault in the scheduler has more serious implications for the program correctness. In the future, we will use known techniques such as redundant execution or checkpointing to make the scheduler robust to transient faults [24, 26].

The runtime system for Google’s MapReduce uses a different approach for worker fault tolerance. Towards the end of
the Map or Reduce stage, they always spawn redundant executions of the remaining tasks, as they proactively assume that some workers have performance or failure issues. This approach works well in large clusters where hundreds of machines are available for redundant execution and failures are more frequent. On multi-core and symmetric multiprocessor systems, the number of processors and frequency of failures are much smaller hence this approach is less profitable.

### 3.2.4 Concurrency and Locality Management

The runtime must make several scheduling decisions that can affect the parallel efficiency of the overall application. In general, there are three ways to determine a policy: 1) use a default policy for the specific system which has been developed taking into account the system characteristics (e.g., cache sizes); 2) dynamically determine the best policy for each scheduling decision by monitoring the availability of resources and their runtime behavior; 3) allow the programmer to provide application specific policies as part of the source code or the runtime arguments\(^1\). Phoenix employs all three approaches in managing the parallel execution of a program. The remaining of this section reviews the main scheduling decisions and the tradeoffs they involve. We evaluate some of these tradeoffs in Section 5.

**Number of Cores and Workers/Core:** Since MapReduce programs are data-intensive, we currently spawn workers to all available cores in the shared-memory system in order to improve parallel speedup. In a multiprogramming environment, the scheduler can periodically check the system load and scale its core usage according to some system-wide priorities. The mechanism for dynamically scaling the number of workers is already in place to support fault tolerance. In systems with hardware support for multiple hardware threads per core (e.g., the UltraSparc T1 \cite{19}), we spawn one worker per hardware thread. This typically maximizes the system throughput even if an individual task takes longer. We also allow the programmer to directly set the number of processors and workers for Map, Reduce, or Merge tasks if desired.

**Task Assignment:** The assignment of Map and Reduce tasks to workers is performed dynamically to achieve load balance. It is difficult to target improved locality between Map and Reduce tasks with the assignment policy as all Map tasks must execute before any Reduce tasks.

**Unit Size:** Phoenix splits the input data into units (a portion of the input data) and processes one unit per Map task. Given the size of an element of input data, Phoenix adjusts the unit size so that the input and output data for a task can fit in the L1 caches in the system. Note that for some computations there is little temporal locality within Map or Reduce stages. Nevertheless, partitioning the input at L1 cache granularity provides a good tradeoff between lower overheads (few larger units) and load balance (more smaller units). The programmer can vary this parameter given specific knowledge of the locality within a task, the amount of output data produced per task, or the processing overheads.

**Partition Function:** The partition function splits the intermediate output pairs across Reduce tasks. If the user does not provide a partition function, Phoenix uses a default function that evenly partitions keys across Reduce tasks. This may not lead to the best performance since keys may have a different number of values associated with them at this stage. The user can provide a function that has application-specific knowledge of the values distribution and leads to lower imbalance during the reduce stage.

There are some locality optimizations that we are not exploring in our current implementation. The first one is hardware

---

\(^1\)The user preferences are communicated to the scheduler through the optional fields of the `scheduler_args.` structure in Table 2.
Table 3. The hardware characteristics of the shared-memory system used with Phoenix.

<table>
<thead>
<tr>
<th>Model</th>
<th>Multi-core System (CMP)</th>
<th>Symmetric Multiprocessor (SMP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Count</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>Processor Model</td>
<td>UltraSparc T1</td>
<td>UltraSparc II</td>
</tr>
<tr>
<td></td>
<td>single-issue, in-order, shared FPU</td>
<td>4-way issue, in-order, separate FPU</td>
</tr>
<tr>
<td>Threads/Processor</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>8KB, 4-way associativity</td>
<td>16KB, direct-mapped</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>3MB, 12-way set associative shared between processors</td>
<td>512KB, off-chip cache per processors</td>
</tr>
<tr>
<td>Main Memory</td>
<td>16 GB</td>
<td>6 GB</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>1.2 GHz</td>
<td>250 MHz</td>
</tr>
</tbody>
</table>

compression/decompression of intermediate outputs as they are emitted in the Map stage or consumed in the Reduce stage. Such an optimization requires special hardware support [1, 13] not available in the systems we used. The second optimization is aggressive prefetching of the data for the next task in parallel with processing the current one (double-buffering). Finally, we could provide cache replacement hints for input and output pairs accessed in Map and Reduce tasks [30].

4 Methodology

This section describes the experimental methodology we used to evaluate the functionality and performance of the Phoenix system.

4.1 Shared Memory Systems

We used Phoenix on the two shared-memory machines described in Table 3. The first system (CMP) is based on the UltraSparc T1 multi-core chip with 8 in-order multithreaded cores sharing an L2 cache [19]. The second system (SMP) is a conventional symmetric multiprocessor with 24 processors each on a separate chip. Using one CMP and one SMP allowed us to evaluate Phoenix under drastically different scenarios with respect to number of cores, threads per core, cache sizes, clock frequencies, and cost of communication. For MapReduce to deliver on its promise, the same program should run as efficiently as possible on any type of shared-memory system without any involvement by the user.

Both systems are based on the Sparc architecture. Nevertheless, our implementation of Phoenix relies only on Pthreads, hence it can run on virtually any other shared-memory system. In the future, we plan to test Phoenix on other multi-core systems.

4.2 Applications

We evaluated Phoenix using the six benchmarks described in Table 4. The benchmarks represent key computations from a diverse set of application domains including enterprise computing (word count, reverse index, string match), scientific computing (matrix multiply), and artificial intelligence (Kmeans and PCA) [12]. We used each benchmark with three datasets (small, medium, and large) in order to test locality and scalability issues. We started with a sequential version for each application as the baseline for speedup and developed two additional versions. The first one is a MapReduce version using the Phoenix API. To evaluate the Phoenix overheads due to the structure of the programs and the actions in the runtime,
we also developed an explicitly parallel version that uses Pthreads directly. The Pthreads versions are statically scheduled (see discussion in Section 5.6). Table 4 lists the number of lines of code (LOC) for each version, which can serve as one indication of coding complexity. In most cases, the Phoenix code is smaller than the Pthreads code. The sequential code is always significantly smaller than both Phoenix and Pthreads. Of course, lines of code as a metric does not capture the intellectual difficulty associated with each programming model. With the MapReduce model, a significant percentage of the code is due to the style and structure of the API (e.g., processing of pairs, void datatypes, etc). However, the programmer does not have to consider dynamic scheduling, synchronization through mutexes, or barriers and other programming constructs that can lead to significant correctness issues if not coded correctly. For reference, the Phoenix runtime is approximately 1,500 lines of code (including headers).

The following are brief descriptions of the main mechanisms used to code each benchmark with Phoenix. We found it straightforward to develop algorithms that fit the MapReduce model for these benchmarks. A more extensive study by Intel has also concluded that a large number of data-intensive computations can be expressed as sums over data points [12]. Such computations should be a good match for the MapReduce model.

**Word Count:** Given a file, it counts the frequency of occurrence for each word. The Map tasks process different sections of the input file and return intermediate data that consists of a word (key) and a value of 1 to indicate that the word was found. The Reduce tasks add up the values for each word.

**Reverse Index:** It traverses a set of HTML files, extracts all links, and compiles a reverse index from links to files. This can be used by search engines, for example, to determine all pointers to a particular location on the web. Each Map task parses a collection of HTML files. For each link it finds, it outputs an intermediate pair with the link as the key and the file info as the value. The Reduce task combines all files referencing the same link into a single linked-list.

**Matrix Multiply:** Each Map task computes the results for a set of rows of the output matrix and returns the (x,y) location of each element as the key and the result of the computation as the value. The Reduce task is just an identity function that performs sorting.

<table>
<thead>
<tr>
<th>Application Name</th>
<th>Description</th>
<th>Data Sets</th>
<th>LOC Seq.</th>
<th>LOC Pthreads</th>
<th>LOC Phoenix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Count</td>
<td>Determine frequency of words in a file</td>
<td>10MB (S), 50MB (M), 100MB (L) files</td>
<td>186</td>
<td>269</td>
<td>175</td>
</tr>
<tr>
<td>Matrix Multiply</td>
<td>Integer matrix multiplication (dense)</td>
<td>100x100 (S), 500x500 (M), 1000x1000 (L) matrices</td>
<td>89</td>
<td>171</td>
<td>143</td>
</tr>
<tr>
<td>Reverse Index</td>
<td>Build reverse index for links in HTML files</td>
<td>100MB (S), 500MB (L), 1GB (L) of HTML files</td>
<td>268</td>
<td>290</td>
<td>274</td>
</tr>
<tr>
<td>Kmeans</td>
<td>Iterative clustering algorithm to classify data points into groups</td>
<td>Matrix with 10K (S), 50K (M), 100K (L) points in 3-D space</td>
<td>212</td>
<td>275</td>
<td>310</td>
</tr>
<tr>
<td>String Match</td>
<td>Search file with keys for an encrypted word</td>
<td>50MB (S), 100MB (M), 500MB (L) text file with possible keys</td>
<td>134</td>
<td>197</td>
<td>193</td>
</tr>
<tr>
<td>PCA</td>
<td>Principal components analysis (statistical clustering technique)</td>
<td>500x500 (S), 1000x1000 (M), 1500x1500 matrix</td>
<td>172</td>
<td>373</td>
<td>359</td>
</tr>
</tbody>
</table>

Table 4. The applications used in this study. LOC stands for lines of code.
String Match: It processes two files: the “encrypt” file contains a set of encrypted words and a “keys” file contains a list of non-encrypted words. The goal is to encrypt the words in the “keys” file to determine which words were originally encrypted to generate the “encrypt file”. Each Map task parses a portion of the “keys” file and returns a word in the “keys” file as the key and a flag to indicate whether it was a match as the value. The reduce task is just an identity function for sorting.

KMeans: It implements the popular kmeans clustering algorithm used to group a set of input data into clusters. Since it is iterative, the Phoenix scheduler is called multiple times until the algorithm converges. In each iteration, the Map task takes in the existing mean vectors and a subset of the data points. It finds the distance between each data point and each mean, and assigns the point to the closest cluster. For each data point, it emits the cluster id as the key and the data vector as the value. The Reduce task gathers all points with the same cluster-id, and finds their centroid (mean vector). It emits the cluster id as the key and the mean vector as the value.

PCA: It performs a portion of the Principal Component Analysis algorithm in order to find the mean vector and the covariance matrix of a set of data-points. The data is presented in a matrix as a collection of column vectors. The algorithm uses two MapReduce stages. To find the mean, each Map task in the first stage computes the mean for a set of rows and emits the row numbers as the keys, and the means as the values. In the second stage, the Map task is assigned a few elements in the covariance matrix, and is provided with the data required to calculate the value of those elements. It emits the element row and column numbers as the key, and the covariance as the value. The Reduce task is the identity one in both cases.

5 Evaluation

This section presents the evaluation results for Phoenix using the CMP and SMP shared memory systems. Apart from the basic speedup results, we also evaluate several aspects of the runtime. All speedups presented in this section are calculated with respect to the sequential version of the application running on a single processor. Unless otherwise specified, we use the large datasets for each application.

5.1 Basic Performance Evaluation

Figure 2 presents the speedup of the Phoenix applications as we scale the number of processors on the two systems (higher is better). The CMP numbers use 4 workers per core taking advantage of the hardware support for multithreading. Hence, even though the speedups seem superlinear, that is not really the case. Figure 3 presents the same data in terms of normalized execution time so that we can see the breakdown of time between the Map, Reduce, and Merge stages for each application.

With the exception of PCA, Phoenix provides significant speedups with both systems for all processor counts and across all benchmarks. In some cases, applications with temporal locality in their access get superlinear speedups as the tasks operate on small blocks that fit in the private caches (e.g. Matrix Multiply on SMP). Scaling is practically linear for all applications. For ReverseIndex and Wordcount the SMP performance does not scale to 24 processors as the system experiences saturation of the bus that interconnects the processors.

From the execution time breakdown in Figure 3, we see that most benchmarks spend the majority of time on the Map stage. Only ReverseIndex and PCA spend significant time on Reduce. For PCA, this is mostly because of the large number
5.2 Dependency to Number of Threads per Processor

The CMP system supports up to 4 hardware threads (contexts) per processor core. Figure 4 shows the behavior of Phoenix as we scale the number of threads used from 1 to 4, including the speedup (left) and the normalized number of L2 misses (right) 2. Each thread is assigned a worker for Map, Reduce, and Merge tasks. In general, the use of multiple workers per core leads to higher speedups in most cases as it improves the task throughput even though an individual task may take longer. With multiple workers on each core, the core resources are fully utilized even when one of the workers stalls due to cache misses or dependencies.

Due to a configuration mismatch on dataset sizes and unit sizing, the speedups in Figures 2 and 4 are not directly comparable. We will correct this error in future versions of the paper.

---

2 Due to a configuration mismatch on dataset sizes and unit sizing, the speedups in Figures 2 and 4 are not directly comparable. We will correct this error in future versions of the paper.
Figure 4. The speedup and normalized number of L2 misses with the large datasets and 8 processors on the CMP as we scale the number of threads per core.

Figure 5. Speedup as we vary the dataset size with 8 processors for the CMP and 24 for the SMP.

At 4 workers per core we start observing diminishing returns for many of the applications (e.g. String Match and Kmeans). This is because 3 workers are sufficient to fully utilize each core and saturate the shared FPU on the UltraSparc T1. Wordcount does not benefit from 3 or 4 threads, as the additional workers cause negative interference problems in the L2 cache as shown by the right graph in Figure 4. The increase in L2 misses reduces the concurrency between threads. With PCA, on the other hand, workers interfere in a positive way in the L2 cache. The results suggest that the use of multiple workers per core should controlled by the runtime dynamically as its efficiency can vary across applications.

5.3 Dependency to Dataset Size

Figure 5 shows the speedup Phoenix achieves on the two systems as we vary the dataset size. For the CMP system we use 8 processors and for the SMP 24 processors. It is clear that increasing the dataset sizes leads to higher speedups over sequential. This is due to two reasons. First, a larger datasets allows the Phoenix runtime to better amortize the overheads for splitting, merging, or sorting data between the execution stages and any overhead due to a sequential portion of an application. Second, caching and locality effects are more significant when processing datasets leading to even larger superlinear effects.
for some applications. String Match is the one application that performs virtually identically regardless of the dataset size. This program is highly parallel to begin with and has very low runtime overheads as it does not include a reduce stage. Hence, even the small datasets are sufficient to fully utilize the available parallel resources.

5.4 Dependency to Unit Size

The unit size determines the amount of data that each Map task processes. Figure 6 shows the speedup for the two systems as we vary the unit size from 4KB to 128KB. We use 8 processors for the CMP and 24 for the SMP. Many applications perform equally well with all unit sizes as there is little temporal locality in the data access (e.g., Word Count). Larger units can actually lead to better performance for such applications as they reduce the amount of time spent on spawning tasks and merging their outputs (fewer tasks).

On the other hand, applications with temporal locality in their access patterns (e.g., Kmeans) perform better with smaller units as they allow tasks to operate on data within their L1 cache. Matrix Multiply has temporal locality but behaves differently between the two systems. The CMP has an L2 cache shared between cores that captures temporal locality sufficiently and multithreading can hide the L2 access latency. On the SMP, L2 caches are private and there is no multithreading.

5.5 Fault Tolerance

Figure 7 presents the results for a fault injection experiment. For the case of a permanent error, a processor in the system stops responding at an arbitrary point within the program execution. For the case of a transient error, an arbitrary task stops responding, but the processor remains functional. In both cases, the failure affects the execution and buffers for the affected tasks, but does not corrupt the data structures of the scheduler (see discussion in Section 3.2.3). Figure 7 shows execution time for these experiments normalized to the case with no faults. Hence, a lower bar is better.

The first important result from Figure 7 is that the Phoenix runtime detects both types of faults and recovers to complete the execution correctly. In the case of a permanent error, the runtime stops issuing any further tasks to the faulty processor, hence the execution time increases by 5% to 15%, depending on the number of processors in the system to begin with and how well they were utilized by the application. In the case of a transient fault, the runtime simply re-executes the faulty task.
and integrates its output with the rest of the data. Since the overall number of tasks is large, one or two failed tasks does not affect execution time by more than 0.5% in either system.

5.6 Comparison to Pthreads

Figure 8 presents a comparison between the Phoenix and the Pthreads implementations of the benchmarks for the two systems. We use 8 processors with the CMP and 24 processors with the SMP. The figure presents speedup so higher is better. In theory, the Pthreads version should be as fast or faster than the Phoenix one in all cases. Pthreads is a more flexible, lower level interface that allows the programmer to code without having to comply with the MapReduce model. Moreover, Phoenix introduces some overhead when merging and sorting data between stages.

Nevertheless, Figure 8 shows that Phoenix outperforms the Pthreads version in half of the cases. Of course, this is not fundamental. Using Pthreads, we could implement the Phoenix programming style and runtime control for every application from the scratch, getting at least equal performance. Our Pthreads versions have two shortcoming. First, some of them have
to use mutexes to synchronize concurrent threads and spend a significant time blocking or synchronizing (e.g., Word Count and Reverse Index). The Phoenix programs, on the other hand, communicate in a message-passing style, using separate buffers that are merged later on. Second, the Pthreads programs use static scheduling. In contrast, the Phoenix programs get dynamic scheduling and efficient partitioning at no overhead to the programmer. Of course, there are programs such as Kmeans and PCA for which the Pthreads is faster as it allows for more flexible implementations with lower overheads. String Match exhibits different behavior in the two systems because the overhead of locking is much higher in a SMP system than in an SMP system.

Again, all Pthreads programs could be optimized carefully to match or exceed the Phoenix performance. In this case, the size and complexity of the Pthreads programs would be much higher. This is exactly the main advantage of MapReduce: scalable performance and efficient scheduling with very simple code.

6 Related Work

Phoenix follows the Google MapReduce model [11]. The Apache Nutch/Lucene project is an open-source project that uses MapReduce as the computation model for web-search services [10]. Concepts similar to MapReduce are also employed in application-specific systems [22]. In the past, several researchers explored the use of scan primitives or parallel prefix computations to construct restricted programming models that allow for automatic parallelization [20, 5]. More recently, researchers at Intel have been exploring the use of similar primitives to easily parallelize and schedule recognition, mining, and synthesis computations [12]. Phoenix attempts to directly use the MapReduce model with shared-memory systems.

The recent turn towards multi-core chips has sparked significant work on novel parallel programming and runtime models. StreamIt uses a synchronous data-flow model to express streaming computations [16]. The StreamIt compiler can automatically apply fusion and fission transformation and statically partition a streaming program on multi-core architectures. The Click language for network routes is also based on data-flow concepts and is amenable to optimizations and static scheduling by the compiler [18]. Speculative multithreading allows programmers to speculatively parallelize loop iterations or function calls in sequential programs [27, 8]. Hardware and/or a runtime system track dependencies during runtime and uses re-execution to enforce them. The Data-Demultiplexing approach combines the speculative parallelization and data-flow schemes [3]. Demultiplex functions can be speculatively executed as soon as its inputs are available. Languages based on transactional memory allow programmers to synchronize threads using non-blocking transactions that are guaranteed to be atomic and isolated through hardware or software means [17, 6]. Cilk is a faithful extension of C for multithreading that uses asynchronous parallelism and an efficient work-stealing schedule [14]. The three languages developed within the Darpa HPCS project (X10, Fortress, and Chapel) follow the partitioned global address space approach and target larger systems with hundreds of multi-core chips [7, 9, 2]. They provide the illusion of a single address space even across a distributed machine (e.g., a cluster) but provide the programmer with explicit or implicit control over locality.

There are also mature commercial models for parallel programming on shared memory systems. Pthreads is the lowest-level and most flexible model a programmer can use to have full-control over parallel execution [21]. OpenMP provides a set of higher-level compiler directives that target fork-join parallelism from loops or independent tasks [25]. With both models, the programmer must manage certain aspects of parallelism explicitly.
It is currently premature to discuss the applicability and practical success of the above approaches. It is likely that several models will co-exist with each one being dominant for different application domains. Apart from ease-of-use and scalability, two factors that may affect their acceptance is how well they run with existing hardware (no model-specific enhancements) and if they help address the upcoming problems of reliability. Phoenix runs on stock hardware and automatically provides fault-recovery for map and reduce tasks.

7 Conclusion and Future Work

We presented Phoenix, a programming API and runtime system for shared memory systems based on Google’s MapReduce model. With Phoenix, the programmer provides a simple, functional expression of the algorithm and parallelization and scheduling are handled automatically by the runtime system. Compared to MapReduce implementations for distributed systems, Phoenix uses shared memory to minimize the overheads of task spawning and data communication. We evaluated Phoenix on a modern multi-core chip and a conventional symmetric multiprocessor and showed that it leads to scalable performance on both systems without the need for source code tuning by the programmer. We also showed that Phoenix can automatically recover from transient and permanent errors that affect Map and Reduce tasks. Finally, we compared Phoenix programs to parallel programs written with a lower-level parallel API like Pthreads. We demonstrated that despite the rigid program structure and runtime overheads, Phoenix leads to similar and sometimes better performance due to its automatic dynamic scheduling capabilities.

For future work, we intend to make several improvements to the current Phoenix implementation. First, we will improve robustness by providing fault tolerance for the runtime scheduler. Second, we will improve dynamic optimization and tuning by feeding to the scheduler performance profiling data from the operating system (e.g., processor utilization) or the performance counters (e.g., cache misses). Such data can be used to fine tune any scheduling decisions. Moreover, we want to provide the user with an interface for specifying priorities and maximum resource utilization that the scheduler should take into account as it runs applications in multiprogrammed environments. Finally, we will continue with further application studies to test both the usability of the API and the efficiency of the runtime. We also plan to publically release the Phoenix implementation as open-source software.

References