Evaluating MapReduce for Multi-core and Multiprocessor Systems

Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis*
Computer Systems Laboratory
Stanford University

Abstract

This paper evaluates the suitability of the MapReduce model for multi-core and multi-processor systems. MapReduce was created by Google for application development on data-centers with thousands of servers. It allows programmers to write functional-style code that is automatically parallelized and scheduled in a distributed system.

We describe Phoenix, an implementation of MapReduce for shared-memory systems that includes a programming API and an efficient runtime system. The Phoenix runtime automatically manages thread creation, dynamic task scheduling, data partitioning, and fault tolerance across processor nodes. We study Phoenix with multi-core and symmetric multiprocessor systems and evaluate its performance potential and error recovery features. We also compare MapReduce code to code written in lower-level APIs such as P-threads. Overall, we establish that, given a careful implementation, MapReduce is a promising model for scalable performance on shared-memory systems with simple parallel code.

1 Introduction

As multi-core chips become ubiquitous, we need parallel programs that can exploit more than one processor. Traditional parallel programming techniques, such as message-passing and shared-memory threads, are too cumbersome for most developers. They require that the programmer manages concurrency explicitly by creating threads and synchronizing them through messages or locks. They also require manual management of data locality. Hence, it is very difficult to write correct and scalable parallel code for non-trivial algorithms. Moreover, the programmer must often re-tune the code when the application is ported to a different or larger-scale system.

To simplify parallel coding, we need to develop two components: a *practical programming model* that allows users to specify concurrency and locality at a high level and an *efficient runtime system* that handles low-level mapping, resource management, and fault tolerance issues automatically regardless of the system characteristics or scale. Naturally, the two components are closely linked. Recently, there has been a significant body of research towards these goals using approaches such as streaming [13, 15], memory transactions [14, 5], data-flow based schemes [2], asynchronous parallelism, and partitioned global address space languages [6, 1, 7].

This paper presents Phoenix, a programming API and runtime system based on Google's MapReduce model [8]. MapReduce borrows two concepts from functional languages to express data-intensive algorithms. The *Map* function processes the input data and generates a set of intermediate key/value pairs. The *Reduce* function properly merges the intermediate pairs which have the same key. Given such a functional 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. Google's MapReduce implementation facilitates processing of terabytes on clusters with 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 to spawn parallel Map or Reduce tasks. It also uses shared-memory buffers to facilitate communication without excessive data copying. The runtime schedules tasks dynamically across the available processors in order to achieve load balance and maximize task throughput. Locality is managed by adjusting the granularity and assignment of parallel tasks. The runtime automatically recovers 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, the Phoenix runtime handles the complicated concurrency, locality, and fault-tolerance tradeoffs that make parallel programming difficult. Nevertheless, it also allows the programmer to provide application specific knowledge such as custom data partitioning functions (if desired).

We evaluate Phoenix on commercial multi-core and mul-

*Email addresses: {*cranger, ramananr, penmetsa*}*@stanford.edu, garybradski@gmail.com, and christos@ee.stanford.edu.

1-4244-0805-9/07/$25.00 ©2007 IEEE

tiprocessor systems and demonstrate that it leads to scalable performance in both environments. Through fault injection experiments, we show that Phoenix can handle permanent and transient faults during Map and Reduce tasks at a small performance penalty. Finally, we compare the performance of Phoenix code to tuned parallel code written directly with P-threads. Despite the overheads associated with the MapReduce model, Phoenix provides similar performance for many applications. Nevertheless, the stylized key management and additional data copying in MapReduce lead to significant performance losses for some applications. Overall, even though MapReduce may not be applicable to all algorithms, it can be a valuable tool for simple parallel programming and resource management on shared-memory systems.

The rest of the paper is organized as follows. Section 2 provides an overview of MapReduce, while Section 3 presents our shared-memory implementation. 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.

2.1 Programming Model

The MapReduce programming model is inspired by functional languages and targets data-intensive computations. The input data format is application-specific, and is specified by the user. The output is a set of <key,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 <key,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 the Map function. All other functionality, including the grouping of the intermediate pairs which have the same key and the final sorting, is provided by the runtime.

The following pseudocode shows the basic structure of a MapReduce program that counts the number of occurences of each word in a collection of documents [8]. 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: a document
  // intermediate output: key=word; value=1
Map(void *input) {
  for each word w in input
    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 main benefit of this model is *simplicity*. The programmer provides a simple description of the algorithm that focuses on functionality and not on parallelization. The actual parallelization and the details of concurrency management are left to the runtime system. Hence the program code is generic and easily portable across systems. Nevertheless, the model 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 is how widely applicable is the MapReduce model. Dean and Ghemawat provided several examples of data-intensive problems that were successfully coded with MapReduce, including a production indexing system, distributed grep, web-link graph construction, and statistical machine translation [8]. A recent study by Intel has also concluded that many data-intensive computations can be expressed as sums over data points [9]. Such computations should be a good match for the MapReduce model. Nevertheless, an extensive evaluation of the applicability and ease-of-use of the MapReduce model is beyond the scope of this work. 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 concurrency control. To parallelize the Map function, it *splits* the input pairs into units that are processed concurrently on multiple nodes. Next, the runtime *partitions* the intermediate pairs using a scheme that keeps pairs with the same key in the same unit. The partitions are processed in parallel by Reduce tasks running on multiple nodes. 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. The decisions can be fully automatic or guided by the programmer given application

specific knowledge (e.g., number of pairs produced by each function or the distribution of keys). 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. Finally, the runtime must *merge* and *sort* the output pairs from all Reduce tasks.

The runtime can perform several optimizations. It can reduce function-call overheads by increasing the granularity of Map or Reduce tasks. It can also reduce load imbalance by adjusting task granularity or the number of nodes used. The runtime can also optimize locality in several ways. First, each node can prefetch pairs for its current Map or Reduce tasks using hardware or software schemes. A node can also prefetch the input for its next Map or Reduce task while processing the current one, which is similar to the double-buffering schemes used in streaming models [23]. Bandwidth and cache space can be preserved using hardware compression of intermediate pairs which tend to have high redundancy [10].

The runtime can also assist with fault tolerance. When it detects that a node has failed, it can re-assign the Map or Reduce task it was processing at the time to another node. To avoid interference, the replicated task will use separate output buffers. If a portion of the memory is corrupted, the runtime can re-execute just the necessary Map or Reduce tasks that will re-produce the lost data. It is also possible to produce a meaningful partial or approximated output even when some input or intermediate data is permanently lost. Moreover, the runtime can dynamically adjust the number of nodes it uses to deal with failures or power and temperature related issues.

Google's runtime implementation targets large clusters of Linux PCs connected through Ethernet switches [3]. Tasks are forked using remote procedure calls. Buffering and communication occurs by reading and writing files on a distributed file system [12]. The locality optimizations focus mostly on avoiding remote file accesses. While such a system is effective with distributed computing [8], it leads to very high overheads if used with shared-memory systems that facilitate communication through memory and are typically of much smaller scale.

The critical question for the runtime is how significant are the overheads it introduces. The MapReduce model requires that data is associated with keys and that pairs are handled in a specific manner at each execution step. Hence, there can be non-trivial overheads due to key management, data copying, data sorting, or memory allocation between execution steps. While programmers may be willing to sacrifice some of the parallel efficiency in return for a simple programming model, we must show that the overheads are not overwhelming.

3 The Phoenix System

Phoenix implements MapReduce for shared-memory systems. Its goal is to support efficient execution on multiple cores without burdening the programmer with concurrency management. Phoenix consists of a simple API that is visible to application programmers and an efficient runtime that handles parallelization, resource management, and fault recovery.

3.1 The Phoenix API

The current Phoenix implementation provides an application-programmer interface (API) for C and C++. However, similar APIs can be defined for languages like Java or C#. The API includes two sets of functions summarized in Table 1. The first set is provided by Phoenix and is used by the programmer's application code to initialize the system and emit output pairs (1 required and 2 optional functions). The second set includes the functions that the programmer defines (3 required and 2 optional functions). Apart from the Map and Reduce functions, the user provides functions that partition the data before each step and a function that implements key comparison. Note that the API is quite small compared to other models. The API is type agnostic. The function arguments are declared as void pointers wherever possible to provide flexibility in their declaration and fast use without conversion overhead. In constrast, the Google implementation uses strings for arguments as string manipulation is inexpensive compared to remote procedure calls and file accesses.

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 set 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 additional data structure types to facilitate communication between the Splitter, Map, Partition, and Reduce functions. These types use pointers whenever possible to implement communication without actually copying significant amounts of data.

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. 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 com-

Function Description	R/O
Functions Provided by Runtime	
`int phoenix_scheduler (scheduler_args_t * args)` Initializes the runtime system. The `scheduler_args_t` struct provides the needed function & data pointers	R
`void emit_intermediate(void *key, void *val, int key_size)` Used in Map to emit an intermediate output <key, value> pair. Required if the Reduce is defined	O
`void emit(void *key, void *val)` Used in Reduce to emit a final output pair	O
Functions Defined by User	
`int (*splitter_t)(void *, int, map_args_t *)` Splits the input data across Map tasks. The arguments are the input data pointer, the unit size for each task, and the input buffer pointer for each Map task	R
`void (*map_t)(map_args_t*)` The Map function. Each Map task executes this function on its input	R
`int (*partition_t)(int, void *, int)` Partitions intermediate pair for Reduce tasks based on their keys. The arguments are the number of Reduce tasks, a pointer to the keys, and a the size of the key. Phoenix provides a default partitioning function based on key hashing	O
`void (*reduce_t)(void *, void **, int)` The Reduce function. Each reduce task executes this on its input. The arguments are a pointer to a key, a pointer to the associated values, and value count. If not specified, Phoenix uses a default *identity* function	O
`int (*key_cmp_t)(const void *, const void*)` Function that compares two keys	R

Table 1. The functions in the Phoenix API. *R* and *O* identify required and optional fuctions respectively.

piler options and does not require a parallelizing compiler. However, it assumes that its functions can freely use stack-allocated and heap-allocated stuctures for private data. It also assumes that there is no communication through shared-memory structures other than the input/output buffers for these functions. For C/C++, we cannot check these assumptions statically for arbitrary programs. Although there are stringent checks within the system to ensure valid data are communicated between user and runtime code, eventually we 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 was developed on top of P-threads [18], but can be easily ported to other shared-memory thread packages.

3.2.1 Basic Operation and Control Flow

Figure 1 shows the basic data flow for the runtime system. The runtime is controlled by the scheduler, which is initiated by user code. The scheduler creates and manages the threads that run all Map and Reduce tasks. It also manages the buffers used for task communication. 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 that is dynamically assigned some number of Map and Reduce tasks.

To start the Map stage, the scheduler uses the `Splitter` to divide input pairs into equally sized units to be processed by the Map tasks. The `Splitter` is called once per Map task and returns a pointer to the data the Map task will process. The Map tasks are allocated dynamically to workers and each one emits intermediate <key, value> pairs. The `Partition` function splits the intermediate pairs into units for the Reduce tasks. 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 this point, the Map stage is over. The scheduler must wait for all Map tasks to complete before initiating the Reduce stage.

Reduce tasks are also assigned to workers dynamically, similar to Map tasks. 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. 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 [8].

Field	Description
Basic Fields	
`Input_data`	Input data pointer; passed to the Splitter by the runtime
`Data_size`	Input dataset size
`Output_data`	Output data pointer; buffer space allocated by user
`Splitter`	Pointer to Splitter function
`Map`	Pointer to Map function
`Reduce`	Pointer to Reduce function
`Partition`	Pointer to Partition function
`Key_cmp`	Pointer to key compare function
Optional Fields for Performance Tuning	
`Unit_size`	Pairs processed per Map/Reduce task
`L1_cache_size`	L1 data cache size in bytes
`Num_Map_workers`	Maximum number of threads (workers) for Map tasks
`Num_Reduce_workers`	Maximum number of threads (workers) for Reduce tasks
`Num_Merge_workers`	Maximum number of threads (workers) for Merge tasks
`Num_procs`	Maximum number of processors cores used

Table 2. The `scheduler_args_t` data structure type.

3.2.2 Buffer Management

Two types of temporary buffers are necessary to store data between the various stages. All buffers are allocated in shared memory but are accessed in a well specified way by a few functions. Whenever we have to re-arrange buffers (e.g., split across tasks), we manipulate pointers instead of the actual pairs, which may be large in size. The 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 as needed. At this stage, there may be multiple pairs with the same key. To accelerate the `Partition` function, the `Emit_intermediate` function stores all values for the same key in the same buffer. 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. At this stage, each key has only one value associated with it. After sorting, the final output is available in the user allocated `Output_data` buffer.

3.2.3 Fault Recovery

The runtime provides support for fault tolerance for transient and permanent faults during Map and Reduce tasks. It focuses 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 a failure is assumed. 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, one should combine the Phoenix runtime with known error detection techniques [20, 21, 24]. Due to the functional nature of the MapReduce model, Phoenix can actually provide information that simplifies error detection. For example, since the address ranges for input and output buffers are known, Phoenix can notify the hardware about which load/store addresses to shared structures should be considered safe for each worker and which should signal a potential fault.

Once a fault is detected or at least suspected, the runtime 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 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 Phoenix code does not provide fault recovery for the scheduler itself. The scheduler runs only for a very small fraction of the 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. We can use known techniques such as redundant execution or checkpointing to address this shortcoming.

Google's MapReduce system uses a different approach

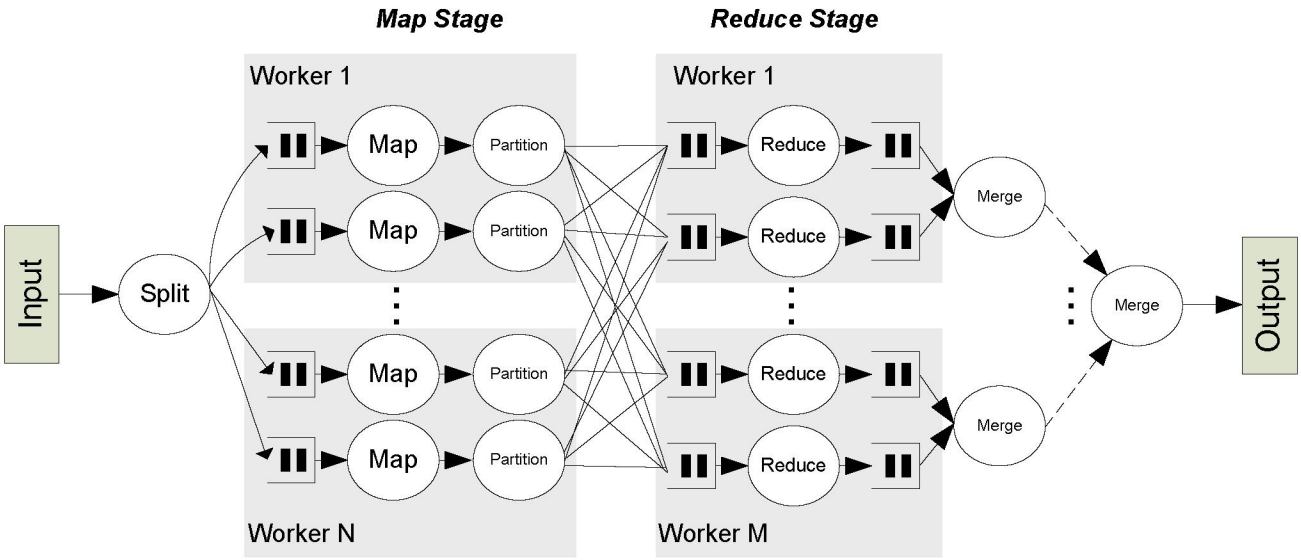

Figure 1. The basic data flow for the Phoenix runtime.

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 makes scheduling decisions that affect the overall parallel efficiency. In general, there are three scheduling approaches one can employ: 1) use a default policy for the specific system which has been developed taking into account its characteristics; 2) dynamically determine the best policy for each decision by monitoring resource availability and runtime behavior; 3) allow the programmer to provide application specific policies. Phoenix employs all three approaches in making the scheduling decisions described below.

Number of Cores and Workers/Core: Since MapReduce programs are data-intensive, we currently spawn workers to all available cores. In a multi-programming environment, the scheduler can periodically check the system load and scale its usage based on system-wide priorities. The mechanism for dynamically scaling the number of workers is already in place to support fault recovery. In systems with multithreaded cores (e.g., UltraSparc T1 [16]), we spawn one worker per hardware thread. This typically maximizes the system throughput even if an individual task takes longer.

Task Assignment: To achieve load balance, we always assign Map and Reduce task to workers dynamically. Since all Map tasks must execute before Reduce tasks, it is difficult to exploit any producer-consumer locality between Map and Reduce tasks.

Task Size: Each Map task processes a *unit* of the input data. Given the size of an element of input data, Phoenix adjusts the unit size so that the input and output data for a Map task fit in the L1 data cache. 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 determines the distribution of intermediate data. The default partition function partitions keys evenly across tasks. This may be suboptimal since keys may have a different number of values associated with them. The user can provide a function that has application-specific knowledge of the values' distribution and reduces imbalance.

There are additional locality optimizations one can use with Phoenix. The runtime can trigger a prefetch engine that brings the data for the next task to the L2 cache in parallel with processing the current task. The runtime can also provide cache replacement hints for input and output pairs accessed in Map and Reduce tasks [25]. Finally, hardware compression/decompression of intermediate outputs as they are emitted in the Map stage or consumed in the Reduce stage can reduce bandwdith and storage requirements [10].

4 Methodology

This section describes the experimental methodology we used to evaluate Phoenix.

4.1 Shared Memory Systems

We ran Phoenix on the two shared-memory systems described in Table 3. Both systems are based on the Sparc architecture. Nevertheless, Phoenix should work without modifications on any architecture that supports the P-threads library. The CMP system is based on the UltraSparc T1 multi-core chip with 8 multithreaded cores sharing the L2 cache [16]. The SMP system is a symmetric multiprocessor with 24 chips. The use of two drastically different systems allows us to evaluate if the Phoenix runtime can 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.

	CMP	SMP
Model	Sun Fire T1200	Sun Ultra-Enterprise 6000
CPU Type	UltraSparc T1 single-issue in-order	UltraSparc II 4-way issue in-order
CPU Count	8	24
Threads/CPU	4	1
L1 Cache	8KB 4-way SA	16KB DM
L2 Size	3MB 12-way SA shared	512KB per CPU (off chip)
Clock Freq.	1.2 GHz	250 MHz

Table 3. The characteristics of the CMP and SMP systems used to evaluate Phoenix.

4.2 Applications

We used the 8 benchmarks described in Table 4. They represent key computations from application domains such as enterprise computing (Word Count, Reverse Index, String Match), scientific computing (Matrix Multiply), artificial intelligence (Kmeans, PCA, Linear Regression), and image processing (Histogram). We used three datasets for each bemchmarks (S, M, L) to test locality and scalability issues. We started with sequential code for all benchmarks that serves as the baseline for speedups. From that, we developed a MapReduce version using Phoenix and a conventional parallel version using P-threads. The P-threads code is statically scheduled.

Table 4 also lists the code size ratio of each parallel version to that of the sequential code (lower is better). Code size is measured in number of source code lines. In general, parallel code is significantly longer than sequential code. Certain applications, such as WordCount and ReverseIndex, fit well with the MapReduce model and lead to very compact and simple Phoenix code. In contrast, the MapReduce style and structure introduce significant amounts of additional code for applications like PCA and MatrixMultiply because key-based data management is not the most natural way to express their data accesses. The P-threads code would be signifciantly longer if dynamic scheduling was implemented. Phoenix provides dynamic scheduling in the runtime. Of course, the number of lines of code is not a direct metric of programming complexity. It is difficult to compare the complexity of code that manages keys or type-agnostic Phoenix function interfaces against the complexity of code that manually manages threads. 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.

Word Count: It counts the frequency of occurence for each word in a set of files. The Map tasks process different sections of the input files and return intermediate data that consist 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 (key).

Reverse Index: It traverses a set of HTML files, extracts all links, and compiles an index from links to files. 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 the identity function.

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 the identity function.

KMeans: It implements the popular kmeans algorithm that groups a set of input data points into clusters. Since it is iterative, the Phoenix scheduler is called multiple times until it 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 point and each mean and assigns the point to the closest cluster. For each 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 centriod (mean vector). It emits

	Description	Data Sets	Code Size Ratio	
			Pthreads	Phoenix
Word Count	Determine frequency of words in a file	S:10MB, M:50MB, L:100MB	1.8	0.9
Matrix Multiply	Dense integer matrix multiplication	S:100x100, M:500x500, L:1000x1000	1.8	2.2
Reverse Index	Build reverse index for links in HTML files	S:100MB, M:500MB, L:1GB	1.5	0.9
Kmeans	Iterative clustering algorithm to classify 3D data points into groups	S:10K, M:50K, L:100K points	1.2	1.7
String Match	Search file with keys for an encrypted word	S:50MB, M:100MB, L:500MB	1.8	1.5
PCA	Principal components analysis on a matrix	S:500x500, M:1000x1000, L:1500x1500	1.7	2.5
Histogram	Determine frequency of each RGB component in a set of images	S:100MB, M:400MB, L:1.4GB	2.4	2.2
Linear Regression	Compute the best fit line for a set of points	S:50M, M:100M, L:500M	1.7	1.6

Table 4. The applications used in this study. Relative code size with respect to sequential code.

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 iterations. To find the mean, each Map task in the first iteration 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 iteration, the Map task is assigned to compute a few elements in the required 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 in both iterations.

Histogram: It analyzes a given bitmap image to compute the frequency of occurence of a value in the 0-255 range for the RGB components of the pixels. The algorithm assigns different portions of the image to different Map tasks, which parse the image and insert the frequency of component occurences into arrays. The reduce tasks sum up these numbers across all the portions.

Linear Regression: It computes the line that best fits a given set of coordinates in an input file. The algorithm assigns different portions of the file to different map tasks, which compute certain summary statistics like the sum of squares. The reduce tasks compute these statistics across the entire data set in order to finally determine the best fit line.

5 Evaluation

This section presents the evaluation results for Phoenix using the CMP and SMP shared-memory systems. All speedups are calculated with respect to the sequential code of the application. Unless otherwise specified, we use the large datasets for each application.

5.1 Basic Performance Evaluation

Figure 2 presents the speedup with Phoenix as we scale the number of processor cores used in the two systems. Higher speedup is better. With the CMP, we use 4 workers per core taking advantage of the hardware support for multithreading. This choice leads to good throughput across all applications. Hence, the CMP speedup with 8 cores can be significantly higher than 8 as we use 32 workers. Figure 3 presents the execution time breakdown between Map, Reduce, and Merge tasks for the CMP system. The breakdown is similar for the SMP.

Phoenix provides significant speedups with both systems for all processor counts and across all benchmarks. In some cases, such as MatrixMultiply, we observe superlinear speedups due to caching effects (beneficial sharing in the CMP, increased cache capacity in the SMP with more cores). At high core counts, the SMP system often suffers from saturation of the bus that interconnects the processors (e.g., PCA and Histogram). With a large number of cores, we also noticed that some applications suffered from load imbalance in the Reduce stage (e.g., WordCount). ReverseIndex achieves the highest speedups due to a number of reasons. Its code uses array based heaps to track indices. As work is distributed across more cores, the heaps accessed by each core are smaller and operations on them become significantly faster. Another contributor to the superlinear speedup is that it spends a signficant portion of its execution time on the final merging/sorting of the output data. The additional cores and their caches reduce the merging overhead.

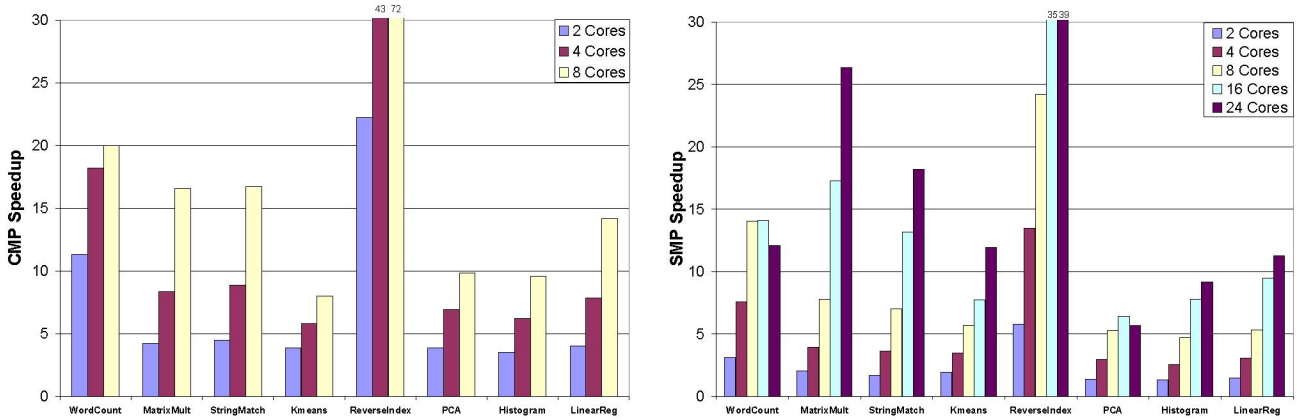

Figure 2. Speedup with Phoenix for the large datasets as we scale the number of processors cores in the two systems.

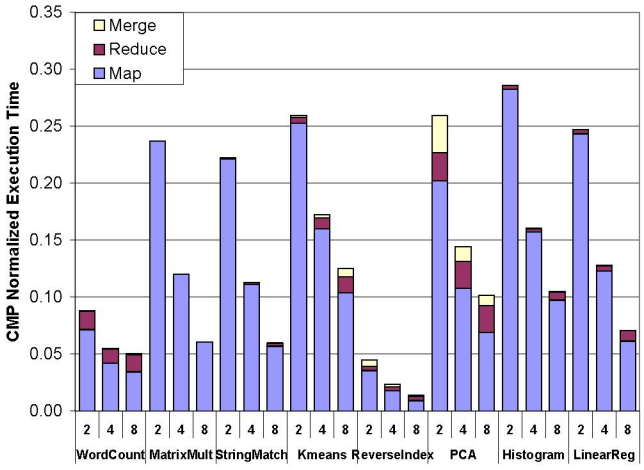

Figure 3. Execution time breakdown for the CMP system.

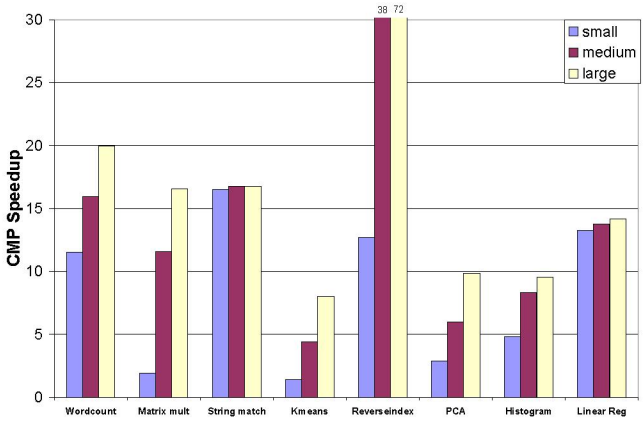

Figure 4. CMP speedup with 8 cores as we vary the dataset size.

In general, the applications in Figure 2 can be classified into two types. The key-based structure that MapReduce uses fits well the algorithm of WordCount, MatrixMultiply, StringMatch, and LinearRegression. Hence, these applications achieve significant speedups across all system sizes. On the other hand, the key-based approach is not the natural choice for Kmeans, PCA, and Histogram. Hence, fitting these algorithms into the MapReduce models leads to significant overheads compared to sequential code and reduces the overall speedup. We discuss this issue further when we compare the performance of Phoenix to that of P-threads in Section 5.4.

5.2 Dependency to Dataset Size

Figure 4 shows the speedup Phoenix achieves on the CMP with 8 cores when we vary the input dataset size. We observed similar behavior for the SMP system. It is clear that increasing the dataset leads to higher speedups over the sequential version for most applications. This is due to two reasons. First, a larger dataset allows the Phoenix runtime to better amortize its overheads for task management, buffer allocation, data spliting and sorting. Such overheads are not dominant if the application is truly data intensive. Second, caching effects are more significant when processing large datasets and load imbalance is more rare. StringMatch and LinearRegression perform similarly across all dataset sizes. This is because even their small datasets contain a large number of elements. Moreover, they perform a significant amount of computation per element in their dataset. Hence, even the small datasets are sufficient to fully utilize the available parallel resources and hide the runtime overheads.

5.3 Dependency to Unit Size

Each Map task processes a unit of the input data. Hence, the unit size determines the number of number of Map tasks, their memory footprint, and how well their overhead is amortized. Figure 5 shows the speedup for CMP system as we vary the unit size from 4KB to 128KB. Many applications perform similarly with all unit sizes as there is little temporal locality in the data access. Larger units can lead

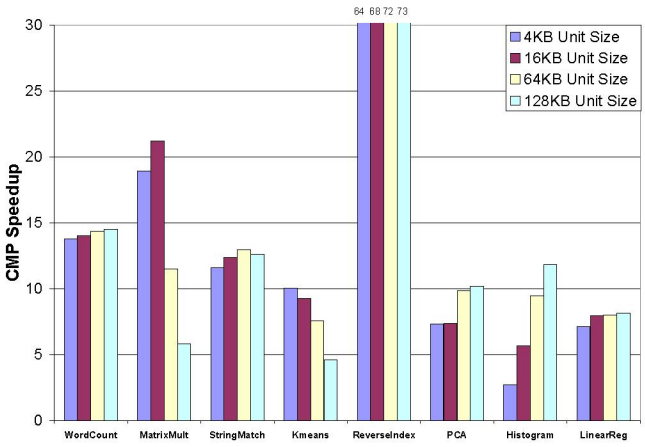

Figure 5. Speedup for the CMP (8 cores) as we vary the unit size for Map tasks.

to better performance for some applications as they reduce significantly the portion of time spent on spawning tasks and merging their outputs (fewer tasks). Histogram benefits from larger units because it reduces the number of intermediate values to merge across tasks. On the other hand, applications with short term temporal locality in their access patterns (e.g. Kmeans and MatrixMultiply) perform better with smaller units as they allow tasks to operate on data within their L1 cache or the data for all the active tasks to fit in the shared L2.

The current implementation of Phoenix uses the user-supplied unit size or determines the unit size based on the input dataset size and the cache size. A better approach is to use a dynamic framework that discovers the best unit size for each program. At the beginning of a data intensive program, the runtime can vary the unit size and monitor the trends in the completion time or other performance indicators (processor utilization, number of misses, etc.) in order to select the best possible value.

The choice of Partition function was not particularly important for the applications we studied as they spend most time on Map tasks. Nevertheless, an imbalanced partitioning of the intermediate outputs can lead to significant imbalance.

5.4 Comparison to Pthreads

Figure 6 compares the speedup achieved with the Phoenix and P-threads code for the two systems. All speedups are with respect to the same sequential code. We use 8 cores with the CMP and 24 cores with the SMP (largest possible configurations). The P-threads code uses the lower level API directly and has been manually optimized to be as fast as possible. The parallel code manages threads directly and does not have to comply with the MapReduce model (computation models, data formats, buffer management approach, final output sorting etc.). Nevertheless, in many cases we re-optimized the P-threads code once we observed how the Phoenix code operates and why it leads to good performance. The only shortcoming of the P-threads code we developed is the use of static scheduling for simplicity. The Phoenix system handles dynamic scheduling in the runtime in a manner transparent to the programmer.

Figure 6 shows that for five of the applications, Phoenix leads to similar or slightly better speedups. These are the applications that fit naturally into the MapReduce model. Either the data is always associated with keys (e.g., WordCount) or introducing a key per large block of data does not lead to significant overheads due to key maniputation and sorting (e.g., MatrixMultiply). The fact that Phoenix operates mostly on pointers and avoids actual data copies as much as possible helps reduce its overhead. The exact comparison between Phoenix and P-threads for these applications depends on the usefulness for the specific configuration of the dynamic scheduling that Phoenix implements in the runtime. Note that we could change the P-threads code to implement similar dynamic scheduling at the cost of significant programming complexity.

For three applications in Figure 6 (Kmeans, PCA, and Histogram), P-threads outperforms Phoenix significantly. For these applications, the MapReduce program structure is not an efficient fit. Kmeans invokes the Phoenix scheduler iteratively, which introduces significant overhead. At the end of each iteration, there is also an expensive operation to translate the output pair format to the input pair format. In addition, the Reduce function frequently performs memory allocation. For PCA, the MapReduce code does not use the original array structure and must track the coordinates for each data point separately. Hence, for each integer in the input set, it must manipulate two other integers. In contrast, the P-threads code uses direct array accesses and does not experience any additional overhead. For Histogram, the P-threads code does not use keys as the output format is predictable. It also avoids the final sorting of the output data.

The conclusion from Figure 6 is that, given an efficient implementation, MapReduce is an attractive model for some classes of computation. It leads to good parallel efficiency with simple code that is dynamically managed without any programmer effort. Nevertheless, its model is not general enough to cover all application domains. While it always leads to significant speedups, it does not always lead to the best possible performance. A good sign is that MapReduce performs suboptimally for applications that are difficult to express with its model anyway.

5.5 Fault Recovery

Figure 7 presents the results for a fault injection experiment on the CMP system. We observed similar results with fault injection experiments on the SMP system. The graphs

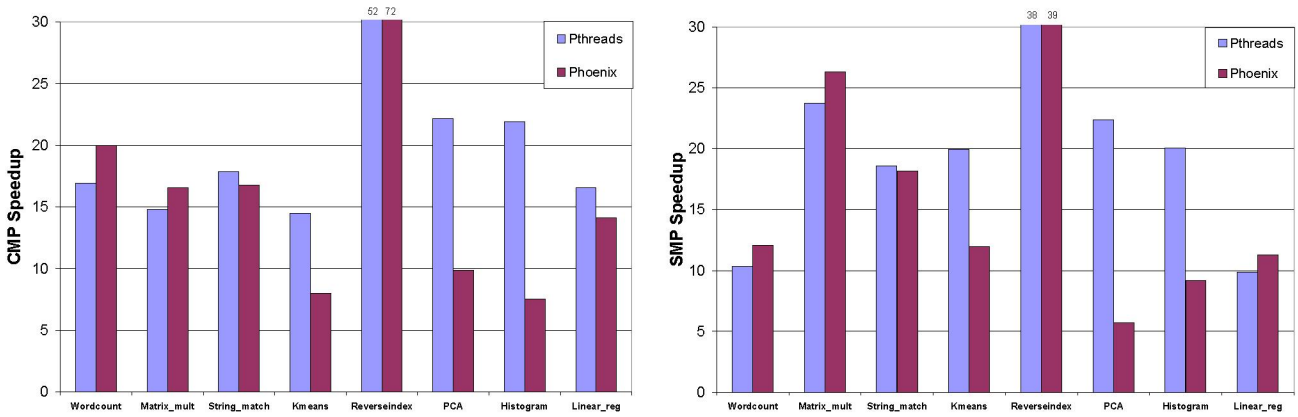

Figure 6. Phoenix Vs. Pthreads speedup for the CMP (8 cores) and SMP (24 cores) systems.

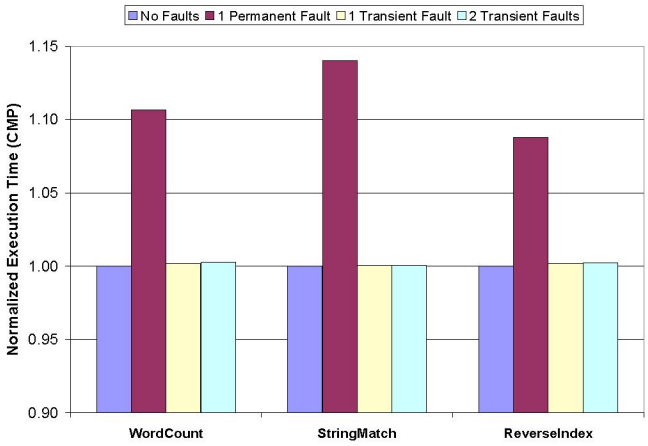

Figure 7. Normalized execution time in the presence of transient and permanent faults for the CMP with 8 cores.

represent normalized execution time, hence lower is better. For the case of a permanent error, a core in the system stops responding at an arbitrary point within the program execution. For the case of a transient error, an arbitrary Map or Reduce task fails to finish, but the core it was assigned to remains functional. In both cases, the failure affects the execution and buffers for the tasks, but does not corrupt the runtime or its data structures (see discussion in Section 3.2.3).

The first important result from Figure 7 is that the Phoenix runtime detects both types of faults through timeouts and recovers to complete the execution correctly. Failure recovery is completely transparent to the application developer. In the case of the permanent error, the runtime does not assign further tasks to the faulty core. Hence, execution time increases by 9% to 14%, depending at which point the fault occured within the program execution and how well was the core utilized by the application. With a lower processor count, the impact of a core failure is higher. 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 other words, once Map and Reduce tasks are sized for concurrency and locality, they are also efficient units for failure recovery.

6 Related Work

MapReduce is similar to models that employ scan primitives or parallel prefix schemes to express parallel computations [17, 4]. Dubey has recently suggested the use of similar primitives in order to easily parallelize and schedule recognition, mining, and synthesis computations [9]. Concepts similar to MapReduce have also been employed in application-specific systems [19].

The recent turn towards multi-core chips has sparked significant work on novel programming models and runtime systems. StreamIt uses a synchronous data-flow model that allows a compiler to automatically map a streaming program to a multi-core system [13]. The Click language for network routes is also based on data-flow concepts and is amenable to optimizations and static scheduling by the compiler [15]. The Data-Demultiplexing approach combines data-flow execution with speculative parallelization of sequential programs [2]. Demultiplexed functions are speculatively executed as soon as their inputs are ready. Languages based on transactional memory introduce database semantics for concurrency control to multithreaded programming [14, 5]. Cilk is a faithful extension of C for multithreading that uses asynchronous parallelism and an efficient work-stealing schedule [11]. There are also proposals for langauges based on partitioned global address space that provide the programmer with explicit or implicit control over locality in large parallel systems [6, 1, 7]. Finally, there are also mature commercial models for parallel programming on shared memory systems such as OpenMP that uses high-level directives to specify fork-join parallelism from loops or independent tasks [22].

It is too early to discuss the applicability and practical success of each approach. It is likely that multiple models will succeed, each in a separate application domain. Apart from ease-of-use and scalability, two factors that may affect their acceptance is how well they run on existing hardware and if they can tolerate errors. Phoenix runs on stock hardware and automatically provides fault recovery for map and reduce tasks.

7 Conclusions

This paper evaluated the suitability of MapReduce as a programming environment for shared-memory systems. We described Phoenix, an implementation of MapReduce that uses shared memory in order to minimize the overheads of task spawning and data communication. With Phoenix, the programmer provides a simple, functional expression of the algorithm and leaves parallelization and scheduling to the runtime system. We showed that Phoenix leads to scalable performance for both multi-core chips and conventional symmetric multiprocessors. Phoenix automatically handles key scheduling decisions during parallel execution. It can also recover from transient and permanent errors in Map and Reduce tasks. We compared the performance of Phoenix to that of parallel code written directly in P-threads. Despite runtime overheads, Phoenix leads to similar performance for most applications. Nevertheless, there are also applications that do not fit naturally in the MapReduce model for which P-threads code performs significantly better.

Overall, this work establishes that MapReduce provides a useful programming and concurrency management approach for shared-memory systems.

References

[1] E. Allen et al. *The Fortress Language Specification.* Sun Microsystems, 2005.

[2] S. Balakrishnan and G. S. Sohi. Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs. In *the Proceedings of the 33rd Intl. Symp. on Computer Architecture*, June 2006.

[3] L. Barroso et al. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2), Mar. 2003.

[4] G. E. Blelloch. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, 38(11), Nov. 1989.

[5] B. D. Carlstrom et al. The Atomos Transactional Programming Language. In *the Proceedings of the Conf. on Programming Language Design and Implementation*, June 2006.

[6] P. Charles et al. X10: an Object-oriented Approach to Non-uniform Cluster Computing. In *the Proceedings of the 20th Conf. on Object Oriented Programming Systems Languages and Applications*, Oct. 2005.

[7] Cray. *Chapel Specification.* Feb. 2005.

[8] J. Dean and J. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *the Proceedings of the 6th Symp. on Operating Systems Design and Implementation*, Dec. 2004.

[9] P. Dubey. Recognition, Mining, and Synthesis Moves Computers to the Era of Tera. *Technology@Intel Magazine*, Feb. 2005.

[10] M. Ekman and P. Stenstrom. A Robust Main-Memory Compression Scheme. In *the Proceedings of the 32nd Intl. Symp. on Computer Architecture*, June 2005.

[11] M. Frigo et al. The Implementation of the Cilk-5 Multithreaded Language. In *the Proceedings of the Conf. on Programming Language Design and Implementation*, June 1998.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *the Proceedings of the 9th Symp. on Operating Systems Principles*, Oct. 2003.

[13] M. I. Gordon et al. A Stream Compiler for Communication-exposed Architectures. In *the Proceedings of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[14] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *the Proceedings of the 18th Conf. on Object-oriented Programing, Systems, Languages, and Applications*, Oct. 2003.

[15] E. Kohler et al. Programming Language Optimizations for Modular Router Configurations. In *Proceedings of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[16] P. Kongetira et al. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE MICRO*, 25(2), March 2005.

[17] R. E. Ladner and M. J. Fischer. Parallel Prefix Computation. *Journal of the ACM*, 27(4), Oct. 1980.

[18] B. Lewis and D. J. Berg. *Multithreaded Programming with Pthreads.* Prentice Hall, 1998.

[19] M. Linderman and T. Meng. A Low Power Merge Cell Processor for Real-Time Spike Sorting in Implantable Neural Prostheses. In *the Proceedings of the Intl. Symp. on Circuits and Systems*, May 2006.

[20] S. Mitra et al. Robust System Design with Built-In Soft-Error Resilience. *IEEE Computer*, 38(2), Feb. 2005.

[21] S. S. Mukherjee et al. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *the Proceedings of the 29th Intl. Symp. on Computer architecture*, May 2002.

[22] OpenMP Architecture Review Board. OpenMP Application Program Interface, v. 2.5, May 2005.

[23] S. Rixner. *Stream Processor Architecture.* Kluwer, 2002.

[24] J. C. Smolens et al. Fingerprinting: Bounding Soft-error Detection Latency and Bandwidth. In *Proceedings of the 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[25] Z. Wang et al. Using the Compiler to Improve Cache Replacement Decisions. In *the Proceedings of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2002.