Elastic Phoenix: Malleable MapReduce for Shared-Memory Systems

Adam Wolfe Gordon and Paul Lu

Department of Computing Science, University of Alberta,
Edmonton, Alberta, T6G 2E8, Canada
{awolfe,paullu}@cs.ualberta.ca

Abstract. We present the design, implementation, and an evaluation of Elastic Phoenix. Based on the original Phoenix from Stanford, Elastic Phoenix is also a MapReduce implementation for shared-memory systems. The key new feature of Elastic Phoenix is that it supports *malleable jobs*: the ability add *and* remove worker processes during the execution of a job. With the original Phoenix, the number of processors to be used is fixed at start-up time. With Elastic Phoenix, if more resources become available (as they might on an elastic cloud computing system), they can be dynamically added to an existing job. If resources are reclaimed, they can also be removed from an existing job. The concept of malleable jobs is well known in job scheduling research, but an implementation of a malleable programming system like Elastic Phoenix is less common.

We show how dynamically increasing the resources available to an Elastic Phoenix workload as it runs can reduce response time by 29% compared to a statically resourced workload. We detail the changes to the Phoenix application programming interface (API) made to support the new capability, and discuss the implementation changes to the Phoenix code base. We show that any additional run-time overheads introduced by Elastic Phoenix can be offset by the benefits of dynamically adding processor resources.

Keywords: MapReduce, parallel programming, malleable, shared-memory systems.

1 Introduction

In many programming systems, the number of processes, threads, or workers is fixed at start-up time. After all, creating and initializing per-thread data structures is most easily handled at the beginning of the job. Furthermore, the classic batch schedulers (e.g., Load Sharing Facility (LSF), Portable Batch System (PBS), Sun Grid Engine (SGE), LoadLeveler) either do not support a dynamically changing allocation of processors in the middle of running a job, or support it awkwardly. However, with the emergence of elastic cloud computing systems (e.g., Amazon EC2 [1], Eucalyptus [11]) comes a new resource provisioning model where it is feasible to allocate more processors on demand. With additional processors, a MapReduce job might finish with a lower response time. But, what programming model or system can support *malleable jobs* [8] that have the ability to add (and remove) processor resources during job execution?

E. Altman and W. Shi (Eds.): NPC 2011, LNCS 6985, pp. 1–16, 2011.
© IFIP International Federation for Information Processing 2011

Custom applications can, of course, create and destroy threads on demand. But, one of the most appealing aspects of the MapReduce programming model [6] is the separation of the application code from the management of processes and data, for a large class of data-parallel problems. Extending a MapReduce system to support malleable execution seems like a natural thing to do, with minimal additional complexity for the application developer. And, although the benefits of running malleable jobs on elastic cloud systems depends greatly on the specific application, the option of dynamically adding processors to a job is desirable, as long as the overheads are reasonable and the benefits are significant.

We present the design, implementation, and an evaluation of Elastic Phoenix, a MapReduce implementation for shared-memory multiprocessors. With a few minor changes to the original Phoenix [14] application programming interface (API) (Table 1) and some changes to MapReduce applications (Table 2), Elastic Phoenix maintains the original data-parallel programming model and has reasonable overheads. In our empirical evaluation, we show the overheads to be small (6% to 8%) for many applications (Figure 2). And, in a synthetic workload experiment (Figure 4), we show that Elastic Phoenix can improve the response time of the workload by up to 29%, over an equivalent workload where processor allocations can change only at job start-up time.

Phoenix [14] is a MapReduce framework written in C for shared-memory computers. Unlike other MapReduce frameworks, such as Google's MapReduce [6] and Hadoop [2], Phoenix is not distributed: a Phoenix application runs entirely in one address space on a single host, using only the local filesystem and memory. But, we argue that as the core count of individual servers increases over time, and if they have sufficient I/O capacity, the desire to run a MapReduce application on a single, shared-memory server will also grow.

One limitation of the original Phoenix system is that, because all worker threads run in the same process, an application cannot take advantage of additional resources that become available after a job is started. That is, if a Phoenix application is started with four threads, and four more processors become idle while the application runs, there is no way to use these additional processors to accelerate the Phoenix application. In this way, we say that Phoenix is not malleable, or in the terminology of cloud computing, Phoenix is not *elastic*.

Elastic Phoenix is a modification to the original Phoenix framework that allows worker threads to be added or removed while a job is running, which makes our new system elastic and malleable.[1]

2 Background

Phoenix [14] is a MapReduce framework designed for shared-memory systems. A Phoenix application is written in C and linked with the Phoenix library into a single executable. A Phoenix job runs in a single process, within which there are multiple worker threads and a single master thread. As in other MapReduce frameworks, execution proceeds in four stages: split, map, reduce, and merge. Input data is usually stored

[1] Elastic Phoenix is an open-source project and can be obtained from `https://github.com/adamwg/elastic-phoenix`

in a file on the local filesystem. The split stage is performed by the master thread, generating map tasks. The map tasks are executed by the worker threads, generating intermediate key-value pairs in memory. This intermediate data is consumed by the reduce tasks, which are also executed by the worker threads. The final data emitted by the reduce tasks is merged in multiple rounds by the worker threads. The merged data can be written to a file or processed further by the master thread.

The `main` function of a Phoenix application is provided by the application, not by the framework. The application configures a MapReduce job by filling in a data structure which is passed to the framework. The application must provide a map function. Most applications provide a reduce function, although an identity reduce function is provided by the framework. The application can optionally provide a splitter function for dividing input data (the default splitter treats the input data as an uninterpreted byte array and splits it into even parts); a combiner that merges intermediate values emitted by a map thread; a partitioner that divides the intermediate data among reduce tasks; and a locator function that helps the framework assign each input split to an appropriate map thread based on the location of the data. The application must also provide a comparator for the application-defined intermediate data keys.

Phoenix creates a worker thread pool, which is used for map, reduce, and merge tasks during execution of a job. Tasks are placed in a set of queues, one for each worker thread, by the master thread. The worker threads in the pool get work by pulling tasks from their queues. If a worker's queue runs out of tasks, it will steal work from other threads to avoid sitting idle.

Map tasks emit intermediate key-value pairs into a set of arrays. Each map thread has an output array for each reduce task (the number of reduce tasks is defined by the application). A reduce task processes the data produced for it by each map thread, emitting final data into another array of key-value pairs. These final data arrays are merged pairwise until a single, sorted output array is produced.

To avoid confusion, we will henceforth refer to Phoenix as "original Phoenix," in contrast to our Elastic Phoenix framework. We use the name Phoenix on its own only for discussion applicable to both frameworks.

3 Related Work

Malleable Programming Environments. Malleable applications are supported by the Message Passing Interface (MPI) through its dynamic process management capability [9]. However, this technique is only applicable to applications developed directly on top of MPI, which offers a low level of abstraction.

At a higher level, support for malleability has been implemented in the Process Checkpointing and Migration (PCM) library, which is built using MPI [7]. PCM is similar to Elastic Phoenix in that it adds malleability to an existing API. However, unlike Elastic Phoenix, the PCM implementation uses explicit splitting and merging of MPI processes; that is, the application developer is required to reason about malleability in order to support it.

An unrelated project, also called Phoenix [12], provides a non-MPI-based message-passing API that supports malleability by allowing nodes in a cluster to join and leave a computation while it runs. Unlike Elastic Phoenix, it is not intended for shared-memory systems.

MapReduce Improvements. Other research groups have also tried to reduce the response time of MapReduce jobs by modifying the framework or semantics of the programming model. The Hadoop Online Prototype (HOP) [5] improves response time of jobs in the Hadoop framework by modifying the flow of data during execution. In HOP, intermediate key-value pairs are sent directly from map workers to reduce workers, before a map task is completed. The framework can also produce partial results by running the reducer on partial map output, though the reducer must be executed over all the map output to produce a final result. By better pipelining and overlapping the work between the map and reduce stages, response time can be improved.

Verma *et al.* [13] present a Hadoop-based MapReduce framework in which reduce tasks not only receive but consume map output as it is produced, eliminating the effective barrier between the map and reduce stages. This technique changes the semantics of the MapReduce model, since the reducer no longer has guarantees about the ordering of its input data. Application developers are thus required to store and sort the data if necessary. Evaluation shows that if careful and efficient techniques are used to handle intermediate data, this modification to MapReduce can improve response time for certain classes of problems.

Chohan *et al.* [4] explore the advantages of using Amazon EC2 Spot Instances (SI), inexpensive but potentially transient virtual machines, to accelerate Hadoop jobs. This is similar to our work in that it provides a type of malleability to a MapReduce framework: a job can be accelerated by adding worker nodes in SIs. However, because of the nature of Hadoop, termination of a spot instance is expensive, since completed work often needs to be re-executed. Elastic Phoenix does not encounter this expense in reducing the number of workers, since the data produced by all completed tasks is in shared memory and will not be lost. We note, though, that our current implementation is not suitable for the semantics of SIs, in which the worker node is terminated without warning, since an Elastic Phoenix worker requires a signal and a little bit of extra time to clean up when it is removed.

The Intermediate Storage System (ISS) [10] for Hadoop improves the response time of MapReduce jobs when faults occur. Hadoop normally stores map output data on local disk until it is required by a reduce worker, requiring the map task that produced the data to be re-executed if a fault occurs on the worker node. The ISS asynchronously replicates the intermediate data, saving the re-execution time if only one replica incurs a fault. However, this technique induces overhead in the absence of faults.

4 Design Goals

Our design goals for Elastic Phoenix were:

- **Elasticity.** The user should be able to add (or remove) worker threads to a running MapReduce job in order to accelerate it. The framework should not make assumptions about the stage of computation during which workers will be added.

– **Design Flexibility.** The implementation described in this paper uses the POSIX shared-memory mechanism, but our design assumes as little as possible about the mechanism being employed, such that it can be ported to other data-sharing mechanisms in the future.
– **API compatibility.** It should take minimal effort to make an original Phoenix application work with Elastic Phoenix.

These design goals are sometimes conflicting. In particular, maintaining the API of the original Phoenix framework, which was not designed for elasticity, made it more difficult to achieve the other goals.

5 Implementation

The basic design change that enables Elastic Phoenix is separation of the master thread and worker threads into different processes, as illustrated in Figure 1. This presents a number of challenges, since original Phoenix was designed with a shared address space in mind. In addition, our implementation does not assume that all worker processes share a kernel or filesystem, allowing future versions to support distributed operation using another distributed data-sharing mechanism such as distributed shared memory or message passing.

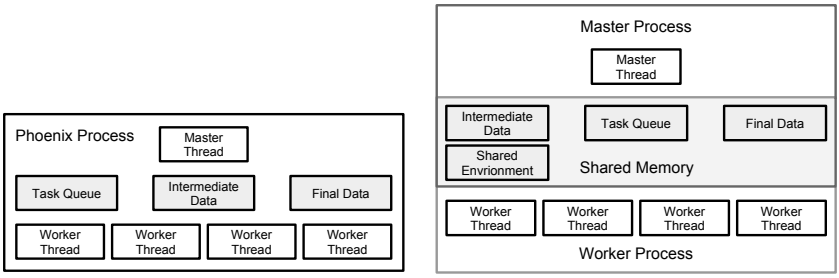


Fig. 1. High-level architecture of (a) original Phoenix and (b) Elastic Phoenix

5.1 Sharing Data

One challenge in designing Elastic Phoenix was how to get input data to the worker threads. The master thread splits the input data, but the worker threads then need to access this data for the map phase. A common idiom in original Phoenix applications is to `mmap` an input file, then have the splitter calculate offsets into the mapped file. Since Elastic Phoenix threads do not share an address space, this clearly will not work. In the interest of flexibility, we also assume that the worker processes cannot access the input file.

Our initial design, which made no changes to the original Phoenix API, required splitter functions to return pointers to the actual input data. The framework then copied

this data into shared memory. This was a simple requirement, but one that did not exist in original Phoenix. As a result, some original applications returned pointers to structures that describe the input data, rather than pointers to the actual data itself. This design allows applications to continue using the `mmap` method described above, with most applications requiring no changes to the splitter code.

However, copying the input data into shared memory after it has been split is a performance bottleneck. Most application-provided splitters must examine at least some portion of the data in order to generate sane splits. Therefore, the copying strategy ends up reading in the data from disk during the split, then copying the data into the shared-memory region, and possibly reading the data off disk again if the buffer cache has been recycled.

Our final design changes the API for splitter functions by adding an argument: a structure containing pointers to memory-allocation functions, which we require the splitter use to allocate space for the input data. The functions we pass to the splitter are custom memory-allocation functions that allocate space in the shared-memory region, keeping all allocation metadata in the region itself. The common design for splitters using this API is to allocate memory for a split, read data directly into this memory, then reduce the size of the split as necessary to reflect application-specific restrictions on the split data.

In addition to the input data, the master and the workers must share the intermediate and final data arrays. The intermediate data is straightforward to move into the shared-memory region, using the previously mentioned memory allocators to dynamically allocate the arrays. The only complication is that there is a set of intermediate data arrays for each worker thread, and in Elastic Phoenix the total number of worker threads is not known at initialization time. We deal with this by setting an upper bound on the number of map threads and pre-allocating space for the necessary pointers.

There is one additional issue with the final data array: it is returned to the application when the MapReduce job finishes. This causes a problem because many applications `free` this array when they are done processing the results it contains. This works in original Phoenix because the framework allocates the final data array with `malloc`. In Elastic Phoenix this array resides in the shared-memory region and is allocated with our custom allocator. We deal with this by adding a function to the API: `map_reduce_cleanup`. The cleanup function mirrors the framework's `map_reduce_ init` function, and is called by the application when it is finished with the results of the MapReduce job.

5.2 Assigning Tasks

In original Phoenix, there is one task queue for each worker thread. The framework does allow threads to steal work from other queues when their own queue runs out, so pre-allocating a task queue for each of the maximum number of worker threads is one potential solution. However, given that Elastic Phoenix will rarely have the maximum number of worker threads, this would lead to the threads exhausting their own queues, then all sharing the other queues until they are exhausted. Given this, we opted for a simpler design with a single task queue. The task queue is pre-allocated in shared memory with a maximum number of tasks, and all worker threads take work from the

queue. Since the task queue in original Phoenix was already thread-safe for dequeue operations, we re-used nearly all of its code.

The pre-allocation design causes an issue for the splitter: there is a maximum number of map tasks, so the splitter can only create so many splits. Elastic Phoenix uses the application-provided unit size estimate to request an appropriate number of data units for each split such that the task queue is filled but not exceeded. However, an application's estimated unit size will, in some cases, be inaccurate because the data is not uniform and not known beforehand. In this case, the framework may run out of space for map tasks before all the data has been split. We deal with this situation by requiring that splitters be "resettable". When the splitter is passed a special argument, it should thereafter operate as if it has not yet been called. When we run out of space for map tasks, we reset the splitter and re-split the data, requesting more units for each task. This is a rare case, as many applications provide good unit size estimates. We have observed that when re-splitting is required it often takes only one try, which tends to be fast due to filesystem caching.

5.3 Coordinating Workers

An Elastic Phoenix job is started by first running the master process, then running one or more worker processes. Coordinating the worker threads is somewhat different from original Phoenix, since the master does not create the worker threads.

We use two basic mechanisms for coordination in Elastic Phoenix: a worker counter and a barrier, both stored in shared memory. When the master starts, it executes the split phase, then waits for workers to join the computation. The first thing a worker process does when it starts is acquire a worker ID using the counter in shared memory. This is done using atomic operations to avoid the need for a lock.

Because we do not assume that all workers run under the same kernel, we cannot use a blocking barrier for synchronization between computation phases. Instead, we have implemented a barrier using spinlocks and polling. The barrier is used after each stage to synchronize the master and worker threads. The master thread reaches the barrier first, since it does no work in the map, reduce, and merge phases. To make the barrier code simpler, we give each worker process a constant number of threads; in our current implementation the default is four. In addition, we use a shared spinlock to prevent new workers from joining the computation at inconvenient times, such as after the first worker thread reaches the barrier (since this would change the number of threads expected, complicating the barrier code) or before the master has finished setting up the necessary shared environment.

Elastic Phoenix keeps track of the current phase of execution using a flag in shared memory. This allows workers to join during any stage of execution: split, map, reduce, or merge. A worker that joins during a later phase skips the previous phases entirely.

An additional coordination problem in Elastic Phoenix is that because we have attempted to preserve the original Phoenix API as much as possible, the application code is not intended to be executed in multiple processes. Many original Phoenix applications contain code that should be executed only once as part of initialization or cleanup. We add two optional application-provided functions to the API to deal with this: `prep` and `cleanup`. If an application may provides these functions, Elastic Phoenix guarantees

that they will be executed only in the master process. The `prep` function is called before any other work is done, and can be used to, for example, open the input file and set up data for the splitter. The `cleanup` function is called by the `map_reduce_cleanup` function described earlier. These functions reduce the need for applications to explicitly manage concurrency, and allow our implementation to handle initialization and cleanup differently in future versions.

6 Porting Phoenix Applications

One of our design goals, as described in Section 4, was to allow original Phoenix applications to be easily ported to Elastic Phoenix. In this section, we describe our experiences porting applications, and try to quantify the amount of change required. Table 1 lists the complete set of API changes.

The biggest change that is required in every application is to the splitter function. Since Elastic Phoenix requires that the splitter use a framework-provided memory allocator for allocating input splits, it has an additional argument. In some splitters, this is simply a matter of replacing calls to `malloc` with calls to the provided allocator, but most need to be restructured to read in a file incrementally instead of using `mmap`, as described in Section 5.1.

Table 1. API changes from original Phoenix to Elastic Phoenix

Description	Original Phoenix	Elastic Phoenix
Splitter function signature	`int splitter(void *, int, map_args_t *)`	`int splitter(void *, int, map_args_t *, splitter_mem_ops_t *)`
Splitter uses provided allocator	`free(out->data);` `out->data = malloc(out->length);`	`mem->free(out->data);` `out->data = mem->alloc(out->length);`

Splitter can be reset	N/A	`if(req_units < 0) {`
		`  lseek(data->fd, 0, SEEK_SET);`
		`  data->fpos = 0;`
		`}`

New `cleanup` API function	`printf("Num. values: %d\n", mr_args.result->length);`
	`return(0);`
		`printf("Number of values: %d\n", mr_args.result->length);`
		`map_reduce_cleanup(&mr_args);`
		`return(0);`

New application `prep` function	`int main(int argc, char **argv) {`
	`...`
	`  struct stat s;`
	`  stat("in", &s);`
	`  data.fd = open("in");`
	`  mr_args.data_size = s.st_size;`
	`...`
		`int prep(void *data, map_reduce_args_t *args) {`
		`  struct stat s;`
		`  stat("in", &s);`
		`  ((app_data_t*)data)->fd = open("in");`
		`  args->data_size = s.st_size;`
		`}`

New application `cleanup` function	`int main(int argc, char **argv) {`
	`...`
	`  close(data.fd);`
	`...`
		`int cleanup(void *data) {`
		`  app_data_t *ad = (app_data_t *)data;`
		`  close(ad->fd);`
		`}`

API `init` modifies `argc` and `argv`	`int nargs = argc;`
	`char *fname = argv[1];`
	`map_reduce_init();`
		`map_reduce_init(&argc, &argv);`
		`int nargs = argc;`
		`char *fname = argv[1];`

In Phoenix applications, most of the code in `main` serves one of four purposes: setting up the splitter; setting up the MapReduce job; processing the results; and cleaning up. The first can usually be moved verbatim into the `prep` function. The second is usually idempotent, operating only on local data structures, and can remain intact without harm. The third and fourth can usually be moved to the `cleanup` function to ensure they execute only in the master process.

Table 2. Porting MapReduce applications: original to Elastic Phoenix (lines of code)

Application	Original	Elastic	Lines altered
Histogram	245	285	110
Word Count	234	202	114
Linear Regression	230	243	97
Matrix Multiply	168	185	48

The original Phoenix framework includes seven sample applications: histogram, k-means, linear regression, matrix multiply, PCA, string match, and word count. Of these, we have ported histogram, word count, and linear regression. PCA and k-means require multiple MapReduce jobs, a feature not currently supported in Elastic Phoenix (we discuss this limitation in Section 8). The matrix multiply included in original Phoenix is not written in the MapReduce style: the map tasks calculate the resultant matrix directly into a known output location, rather than emitting the calculated values as intermediate data. We wrote a new matrix multiply for original Phoenix that uses the MapReduce style, emitting a key-value pair for each entry in the resultant matrix, then ported it to Elastic Phoenix. String match also does not emit any intermediate or final data; in fact, it produces no results at all.

To quantify the amount of work required to convert an application to Elastic Phoenix, we analyzed the applications we converted. First, we calculated the lines of code for each application before and after conversion using SLOCCount [3]. Second, we estimated how many lines were altered by taking a `diff` of the two source files and counting the number of altered lines in the Elastic Phoenix version. This does not entirely account for the fact that many of the changes simply involved moving code from one function to another. The results of this analysis are displayed in Table 2. Of course, lines of code are an incomplete and controversial metric for programming effort, but we include it as a data point for consideration.

Word count was the first sample application we ported, and we modified it incrementally during the development of Elastic Phoenix. Therefore, it underwent more change than would an application being ported directly to the final version of Elastic Phoenix. We ported histogram and linear regression to one intermediate version of Elastic Phoenix, then to the final version, so they display less change than word count. We developed our new version of matrix multiply for original Phoenix first, then we ported it directly to the final version of Elastic Phoenix. Therefore, matrix multiply provides the best indication of the effort needed to directly port an application. Anecdotally, a single developer wrote the new matrix multiply application for original Phoenix in several hours, and took only a few minutes to port it to Elastic Phoenix.

7 Evaluation

We evaluated Elastic Phoenix in two ways. First, we evaluated the overhead of using Elastic Phoenix by running our ported applications with both frameworks. Then, we explored the advantages of Elastic Phoenix by developing a workload of repeated MapReduce jobs on a system where the availability of processors varies over time. All experiments were performed on an Intel Xeon X5550 (2.67 GHz) server with two sockets, eight cores, 16 hyperthreads, and 48 GB of RAM. We compiled original Phoenix with its default tuning parameters, and used the default four worker threads per process for Elastic Phoenix.

Table 3. Applications and inputs for evaluation

Application	Description	Input Size
Histogram	Counts the frequency of each color value over pixels in a bitmap image	1.4 GB
Word Count	Counts the frequency of each word in a text file	10 MB
Linear Regression	Generates a linear approximation of a set of points	500 MB
Matrix Multiply	Parallel matrix multiplication using row and column blocks	1000x1000

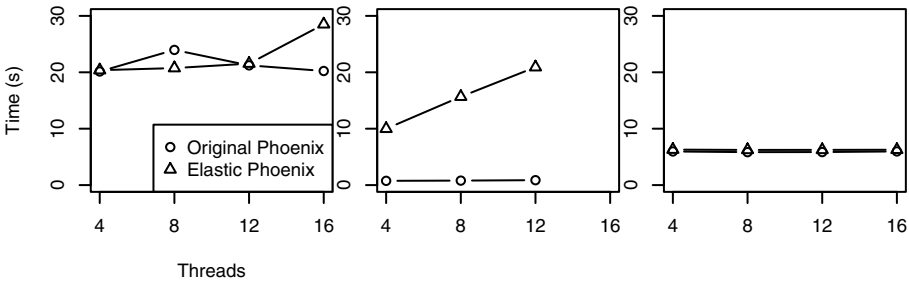
7.1 Overhead

The performance advantage of malleability is that MapReduce jobs can be accelerated as they run by adding threads to the computation. If the overhead of Elastic Phoenix is so large that, for example, using four threads in original Phoenix is faster than using eight threads in Elastic Phoenix, then this advantage disappears. In this section we evaluate the overhead of using Elastic Phoenix, showing that for some applications it is possible to increase performance by adding threads.

To evaluate the overhead of our design, we ran four applications under both original and Elastic Phoenix. For each application, we used the largest input data we could given 4 GB of shared memory (we discuss these limits in Section 8). The applications we used and their input sizes are shown in Table 3. All times are averages over five runs, with cold filesystem caches.

We divide these applications into two classes, which we will discuss separately: I/O-bound and CPU-bound. Histogram, word count, and linear regression are I/O-bound. This is due to the fact that the computation performed in the map and reduce stages is quite trivial, even with large input. The bottleneck, therefore, is reading and splitting the input data, which must be done serially in both original and Elastic Phoenix. Our matrix multiply application is CPU-bound because its input is generated in the splitter function, avoiding any file I/O.

Benchmark results for the I/O-bound applications are presented in Figure 2. Being I/O-bound, we do not expect any speedup as we add threads. We observe that Elastic Phoenix has little overhead compared to original Phoenix for the histogram and linear regression applications (8% and 6%, respectively, on average), showing some slowdown as we add more threads due to the additional coordination overhead. These applications have large input data, and generate intermediate data with a fixed number of keys.



(a) Histogram (b) Word Count (c) Linear Regression

Fig. 2. Overhead experiment results for I/O-bound applications

In contrast, we observe significant overhead (1792% average) in the word count application. This overhead likely stems from the fact that the input size is small, and that the application produces a large amount of intermediate data with diverse keys, since many more words appear infrequently in the input than appear frequently. This causes a large amount of extra shared memory to be allocated but never used, as the framework allocates space for multiple values (10, by default) with the same key when the first value for a given key is emitted. In fact, word count, with only 10 MB of input data, can be run with at most 12 worker threads in Elastic Phoenix; with more threads, the small amount of additional per-thread allocation in shared memory causes it to exceed the 4 GB limit. Our shared-memory allocator is simple, and does not match the efficiency of the system's `malloc`, which is used in original Phoenix.

Benchmark results for our single CPU-bound application (i.e., matrix multiply) are shown in Figure 3. Here we observe significant speedup from four to eight threads, then moderate speedup as we increase to 12 and 16 threads. We theorize that the extreme speedup from four to eight threads is caused by the fact that our CPUs are quad-core, and with eight threads Phoenix uses both sockets. Original Phoenix pins threads to cores, spreading work out as much as possible, but we have configured it to use the same number of processors as threads, so the four threads are pinned onto one CPU. Using both sockets doubles the total usable cache size and the total memory bandwidth available. The decline in speedup beyond eight threads is likely due to the fact that we have only eight real CPU cores, and the hyperthreaded cores offer little advantage in an application such as matrix multiply with regular memory access patterns, few branches, and no disk I/O.

With four threads (i.e., a single worker process), Elastic Phoenix adds no overhead compared to original Phoenix. Elastic Phoenix adds a moderate amount of overhead when more than one process is used, due to the added overhead of coordinating threads in multiple processes, such as increased contention on spinlocks. Overall, the overhead averages 38%. Nonetheless, Elastic Phoenix with eight threads is significantly faster than original Phoenix with four threads, suggesting that the ability to add additional workers as a job runs can improve performance in some cases.

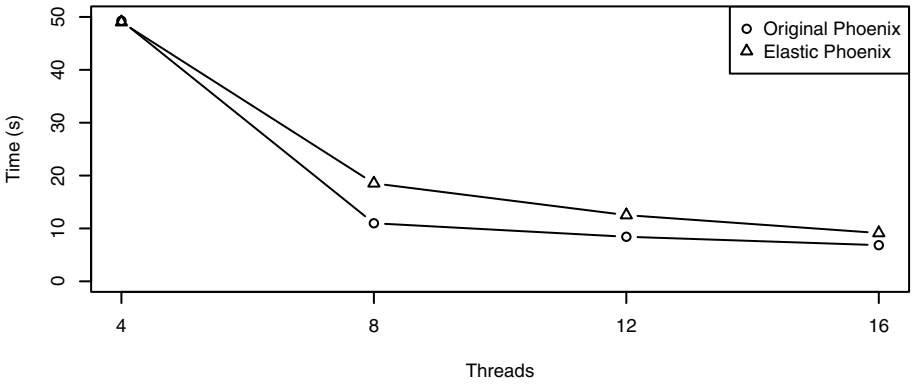


Fig. 3. Overhead experiment results for the CPU-bound matrix multiply application

7.2 Elasticity Advantages

Elastic Phoenix can improve throughput and job latency by allowing idle processors to be used immediately by a running MapReduce job. A performance improvement is possible when the overhead of using Elastic Phoenix does not exceed the speedup provided by the additional threads. As we showed earlier, this is the case for some applications. In this section, we evaluate how much performance can be gained in such a situation. For this benchmark, job latency can be reduced by 29% by adding threads as processors become available.

Consider a scenario in which a sequence of MapReduce jobs is run repeatedly, starting again each time it finishes. For example, the sequence may consist of a job that creates a link graph from a set of hypertext pages, then a job that computes the most popular page. Assume the jobs run on a system where CPU resources become available unpredictably, as is typical in a multi-user, batch-scheduled environment. Original Phoenix always uses the number of CPUs that are available when it starts, while in Elastic Phoenix CPUs are added to the computation as they become available.

Our benchmark uses alternating linear regression and matrix multiply jobs with random-sized inputs of 250 MB to 500 MB for linear regression and 500 x 500 to 1000 x 1000 for matrix multiply. This sequence represents a typical workload of an I/O-bound job that generates input for a compute-bound job, although we do not actually use the linear regression output as input for the matrix multiply. With one worker thread, the jobs take between 1 second and 5 minutes each. For this experiment, we compiled Elastic Phoenix with one thread per worker process, so that it can take advantage of a single processor becoming available.

We generated a set of random CPU availability changes, where every 5 seconds the number of CPUs available can change, with at least 2 and at most 8 CPUs always being available, since our host has 8 CPU cores (we do not employ the 8 hyperthreaded cores). The workload we generated contains 500 jobs in total: 250 linear regression jobs interleaved with 250 matrix multiply jobs.

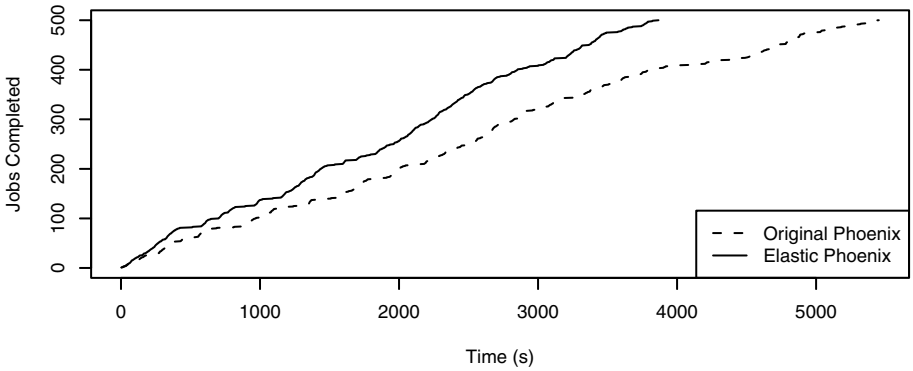


Fig. 4. Job completion over time in the elasticity benchmark

The results of the experiment are displayed in Figure 4. Original Phoenix completed the jobs in 5,451 seconds and Elastic Phoenix completed the jobs in 3,866 seconds, a 29% improvement. We see from these results that adding processors as they become available can drastically speed up a workload of this type.

Note that we have successfully tested the ability to dynamically remove workers from a running Elastic Phoenix job, for proper malleability, but we do not present a performance evaluation in this paper.

8 Limitations and Future Work

The design of Elastic Phoenix introduces a number of limitations not present in original Phoenix. Some of these are inherent to the design we have chosen; others are due to our current implementation and could be eliminated in the future without a significant design change. There are also some ways in which Elastic Phoenix could be improved, orthogonal to these limitations.

Job Size. Currently, the size of the shared-memory region used with Elastic Phoenix limits the amount of input data that can be used. This is because the input data must all fit in shared memory, and will share the space with some of the intermediate data (input data is freed immediately after being used by a map task, but intermediate values are stored to shared memory as they are produced). Similarly, the intermediate data must share the space with some final data, so this data is also limited in size. The input size limit is not the same for all applications and cannot be calculated in advance. As mentioned in Section 7.1, the number of intermediate key-value pairs, as well as the diversity of the keys, affects the maximum input size.

Combiners. The original Phoenix framework allows applications to provide a combiner function, which performs a partial reduce on the map output of a single worker thread. This can speed up the reduce stage, and reduce the amount of space used for intermediate data. In particular, it helps applications that have a large number of repeated intermediate keys.

Elastic Phoenix currently does not support combiners. The application may specify a combiner, to keep the API consistent with original Phoenix, but it will not be used by the framework. This is a limitation of the current implementation and could be changed in the future. It is possible that this modification would help with the job size limitation by compacting intermediate data.

Multiple MapReduce Jobs. A common idiom in MapReduce applications, including Phoenix applications, is to run multiple MapReduce jobs in sequence. For example, one might run a job to do a large computation, then another job to sort the results by value. In original Phoenix, multiple jobs can be performed without writing data to disk: the output data from one job becomes the input data to another job. That is, multiple calls are made to the Phoenix MapReduce scheduler within a single executable application.

The current Elastic Phoenix implementation does not support running multiple MapReduce jobs in one execution, due to the way in which Elastic Phoenix initializes data that will be shared among the master and worker processes. This limitation could be eliminated through some further design.

Number of Workers. Elastic Phoenix pre-allocates a number of data structures that in original Phoenix are allocated based on the number of threads being used. In particular, there is one set of intermediate data queues and one final data queue for each worker thread, and the pointers for these are pre-allocated. They could be allocated dynamically as workers join the computation, but this would require using a linked list or similar structure, making lookups slower when inserting data.

In order to pre-allocate this data, we set a fixed limit on the number of worker tasks that can join a MapReduce job. The default limit in our current implementation is 32 threads, which allows for eight worker processes with four threads each. On any given system, the number of worker threads is naturally limited by the number of processors, so this is a minor limitation.

Other Future Work. Elastic Phoenix makes extensive use of spinlocks instead of blocking locks. This is because we designed Elastic Phoenix with flexibility in mind, and for a potential hybrid shared-memory and distributed-memory version of Elastic Phoenix where the worker processes may not share a kernel, making the use of standard blocking locks, such as pthreads mutexes and semaphores, impossible. One potential performance improvement would be to use blocking locks when they are available, or to exploit optimistic concurrency by making more extensive use of atomic operations. This would likely involve abstracting locking and atomic operations so that they could be implemented in an appropriate way for any given data sharing system.

The original Phoenix software implemented many optimizations to improve cache usage and memory locality for worker threads. In the process of building Elastic Phoenix, we have eschewed many of these optimizations, either because they do not apply in the case where worker threads are in different processes or because getting the initial version of Elastic Phoenix was easier without them. In the future, we may re-introduce some of these optimizations.

9 Concluding Remarks

We have presented the design, implementation, and an evaluation of Elastic Phoenix, a MapReduce implementation for shared-memory multiprocessors. Based on the Stanford Phoenix system, Elastic Phoenix adds the ability to support malleable jobs that can dynamically add processors to the computation, with the goal of reducing the turnaround time of the job or workload.

Using a synthetic workload (Figure 4), we have shown that Elastic Phoenix can improve the response time of the workload by up to 29%, over an equivalent workload where processor allocations can only change at job start-up time. Although the benefits of malleability or an elastic programming system will vary from application to application, Elastic Phoenix provides those benefits while maintaining the MapReduce programming model and much of the Phoenix API.

Acknowledgements. This research was supported by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Amazon Elastic Compute Cloud (Amazon EC2), `http://aws.amazon.com/ec2`
2. Apache Hadoop, `http://hadoop.apache.org/`
3. SLOCCount, `http://sourceforge.net/projects/sloccount/`
4. Chohan, N., Castillo, C., Spreitzer, M., Steinder, M., Tantawi, A., Krintz, C.: See spot run: using spot instances for mapreduce workflows. In: 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud 2010 (June 2010)
5. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears, R.: MapReduce online. In: 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2010), USENIX Association (2010)
6. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: 6th Symposium on Operating Systems Design and Implementation (OSDI 2004), USENIX Association (2004)
7. El Maghraoui, K., Desell, T.J., Szymanski, B.K., Varela, C.A.: Dynamic Malleability in Iterative MPI Applications. In: IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), pp. 591–598. IEEE, Los Alamitos (2007)
8. Feitelson, D., Rudolph, L., Schwiegelshohn, U., Sevcik, K., Wong, P.: Theory and practice in parallel job scheduling. In: Feitelson, D.G., Rudolph, L. (eds.) IPPS-WS 1997 and JSSPP 1997. LNCS, vol. 1291, pp. 1–34. Springer, Heidelberg (1997)
9. Gropp, W., Lusk, E.: Dynamic process management in an MPI setting. In: 7th IEEE Symposium on Parallel and Distributed Processing. IEEE Computer Society, Los Alamitos (1995)
10. Ko, S.Y., Hoque, I., Cho, B., Gupta, I.: Making cloud intermediate data fault-tolerant. In: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, pp. 181–192. ACM, New York (2010)
11. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The Eucalyptus Open-Source Cloud-Computing System. In: 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 124–131. IEEE, Los Alamitos (2009)

12. Taura, K., Kaneda, K., Endo, T., Yonezawa, A.: Phoenix: a parallel programming model for accommodating dynamically joining/leaving resources. ACM SIGPLAN Notices 38(10) (October 2003)
13. Verma, A., Zea, N., Cho, B., Gupta, I., Campbell, R.: Breaking the MapReduce Stage Barrier. In: IEEE International Conference on Cluster Computing, Heraklion, Greece, pp. 235–244 (2010)
14. Yoo, R.M., Romano, A., Kozyrakis, C.: Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system. In: 2009 IEEE International Symposium on Workload Characterization (IISWC), pp. 198–207. IEEE, Los Alamitos (2009)