A Map-Reduce System with an Alternate API for Multi-Core Environments

Wei Jiang  Vignesh T. Ravi  Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University Columbus OH 43210
{jiangwei,raviv,agrawal}@cse.ohio-state.edu

Abstract—Map-reduce framework has received a significant attention and is being used for programming both large-scale clusters and multi-core systems. While the high productivity aspect of map-reduce has been well accepted, it is not clear if the API results in efficient implementations for different sub-classes of data-intensive applications. In this paper, we present a system MATE (Map-reduce with an AlternatTE API), that provides a high-level, but distinct API. Particularly, our API includes a programmer-managed reduction object, which results in lower memory requirements at runtime for many data-intensive applications. MATE implements this API on top of the Phoenix system, a multi-core map-reduce implementation from Stanford.

We evaluate our system using three data mining applications and compare its performance to that of both Phoenix and Hadoop. Our results show that for all the three applications, MATE outperforms Phoenix and Hadoop. Despite achieving good scalability, MATE also maintains the easy-to-use API of map-reduce. Overall, we argue that, our approach, which is based on the generalized reduction structure, provides an alternate high-level API, leading to more efficient and scalable implementations.

Keywords—data-intensive computing; Multi-Core Architectures; Middleware; Map-Reduce

I. INTRODUCTION

Multi-core and many-core environments are becoming ubiquitous. Programmers can no longer rely on existing sequential implementations to run faster simply because of increased clock frequency of processors. This makes writing parallel programs inevitable. To enable parallel programming, a number of paradigms have been developed, with tradeoffs in terms of expressive power, programmer productivity, and efficiency. For example, models like explicit thread programming and message-passing offer highest level of flexibility, but involve a steep learning curve and potentially low programmer productivity.

An approach that is growing in popularity is based on application-class-specific programming models. Such models can enable both high programmer productivity and provide sufficient expressive power for a particular class of applications, while potentially achieving efficient execution. Map-reduce paradigm, developed by Google, is such an example, targeting data-intensive applications [6]. Over the last few years, multiple projects have focused on improving the API or implementations [7], [11], [16], [17], [19], [20], [21]. At the same time, there are studies focusing on the suitability of map-reduce model for a variety of applications across different platforms [7], [9], [18], [10], [8]. Map-reduce was originally proposed for large scale clusters and data center environments, but its high-level API has been used for developing data-intensive applications on multi-cores as well [5], [18].

While the high productivity aspect of map-reduce API has been very well understood, performance or efficiency has been much less understood. At the same time, the class of data-intensive applications is very broad. It ranges from data-center type of operations like scanning data, which require very little computation, to machine learning or model building on large datasets, which can involve significant amount of computations. It is not yet clear if the current map-reduce API is best suited for all of the different sub-classes of data-intensive applications.

This paper describes support for an alternate API in a map-reduce implementation. This API has been implemented in a system we refer to as Map-reduce with AlternatTE API (MATE). This system has been developed on top of Phoenix, a multi-core implementation of map-reduce [18]. The key distinctive aspect of our API is that it allows the programmers to explicitly declare a reduction object. The map and reduce functions are replaced by a single reduction function that updates such a reduction object, as opposed to emitting \((key, value)\) pairs, and then reducing the values that have the same key. Race conditions can be avoided by replicating the reduction object. A combination function combines different copies of the reduction object to create the final results. The runtime system automatically maps the Reduction/Combination tasks to threads that are spawned on processor cores. Also, the runtime scheduler dynamically partitions the dataset and assigns splits to processor cores.

The distinct API in MATE still offers a high-level interface, leading to much simpler implementations than alternate
approaches like Pthreads or OpenMP. At the same time, it reduces the memory requirements associated with the large number of \{key, value\} pairs that can be generated in the original map-reduce API for several applications.

Our implementation on top of the Phoenix system allows us to carefully evaluate the performance advantages of the new API. We have extensively evaluated our system using three data mining applications and compare it with both Phoenix (with original map-reduce API) and Hadoop on two shared-memory platforms. Our experimental results show that our system outperforms Phoenix for all the three applications and also leads to reasonable speedups in both environments. Hadoop is much slower than both the systems. Overall, we observe that while map-reduce eases the burden of programmers, its inherent API structure may cause performance losses for some applications due to the rigid two-stage computation style. Our approach, which is based on the generalized reduction, offers an alternate API, potentially useful for several sub-classes of data-intensive applications.

The rest of the paper is organized as follows. In Section II we give an overview of the map-reduce and generalized reduction, with a case study showing the use of APIs from both MATE and Phoenix. The design and the implementation of our system are discussed in Section III. We report the results from our experimental study in Section IV. We give a brief overview of the related efforts in this area in Section V and conclude in Section VI.

II. MAP-REDUCE AND GENERALIZED REDUCTION

This section describes the map-reduce model and the alternate API implemented in our MATE system, which we will refer to as the generalized reduction API. We will further explain the API usage of the two models using apriori association mining, a popular data mining algorithm [3].

A. Map-Reduce Model

Map-reduce [6] was created by Google for application development on data-centers with thousands of computing nodes. It is a programming model that enables easy development of applications that process vast amounts of data on large clusters. Through a simple interface of two functions, map and reduce, this model facilitates parallel implementations of many real-world tasks, ranging from data processing for search engine support to machine learning [5], [9].

The map-reduce programming model can be summarized as follows [6]. The computation takes a set of input points, and produces a set of output \{key, value\} pairs. The user of the map-reduce library expresses the computation as two functions: map and reduce. Map, written by the user, takes an input point and produces a set of intermediate \{key, value\} pairs. The map-reduce library groups together all intermediate values associated with the same key, and passes them to the reduce function. The reduce function, also written by the user, accepts a key and a set of values associated with that key. It merges together these values to form a possibly smaller set of values. Typically, just zero or one output value is produced per reduce invocation.

The main benefits of this model are in its simplicity and robustness. It allows programmers to write functional-style code that is easily parallelized and scheduled in a distributed computing environment. For a highly parallel system, one can view map-reduce offering two important components [18]: a practical programming model that allows users to develop applications at a high level and an efficient runtime system that deals with the low-level details. Parallelization, concurrency control, resource management, fault tolerance and many other issues are handled by the map-reduce runtime.

B. Generalized Reduction Structure

The API offered by our MATE system is quite similar to the map-reduce API, with some subtle, but important differences. First, the MATE API allows developers to explicitly declare a reduction object, and perform updates to its elements directly. On the other hand, map-reduce maintains an implicit reduction object unexposed to the user. Another important distinction is that, in map-reduce, all data elements are processed in the Map step and the intermediate results are then combined in the Reduce step. In MATE, both map and reduce steps are combined into a single step called Reduction, where each data element is processed and reduced before next data element is processed. This choice of design avoids the overheads due to high memory requirements of intermediate results, sorting, grouping, and shuffling, which can be significant costs in the map-reduce implementation.

The MATE system’s generalized reduction based API is motivated by our earlier work on a system called FREERIDE (FRamework for Rapid Implementation of Datamining Engines) [14], [15]. The FREERIDE system was motivated by the difficulties in implementing and performance tuning parallel versions of data mining algorithms. FREERIDE is based upon the observation that parallel versions of several well-known data mining techniques share a relatively similar structure, which is that of a generalized reduction. Though the Map-Reduce paradigm is built on a similar observation [6], it should be noted that the first work on FREERIDE was published in 2001 [14], prior to the map-reduce paper by Dean and Ghemawat in 2004.
Fig. 1. Processing Structure: MATE (left) and Map-Reduce (right)

The API supported by MATE (and FREERIDE) exploits the similarity among parallel versions of several data mining and scientific data processing algorithms. The following are some important functions, which need to be written by the application developer.

**Reduction**: A reduction function specifies how, after processing one data instance, a reduction object (initially declared by the programmer), is updated. The result of this processing must be independent of the order in which data instances are processed on each processor. The order in which data instances are processed is determined by the runtime system.

**Combination**: In this function, final results from multiple copies of a reduction object are combined into a single reduction object. A user can choose from one of the several common combination functions already implemented in the system, or could provide one of their own.

This API can be used to parallelize applications on multi-cores and clusters of multi-cores. Throughout the execution of the application, the reduction object is maintained in the main memory, with a private copy for each core. After every iteration where all data instances are processed, the results from multiple threads in a node are combined locally to obtain the final results. The local combination can be handled internally by the middleware or the programmers can write one of their own.

Fig. 1 further illustrates the distinction in the processing structure enabled by MATE and map-reduce. The function Reduce is an associative and commutative function. Thus, the iterations of the for-each loop can be performed in any order. The data-structure RObj is referred to as the reduction object.

**C. A Case Study**

We will now use the apriori association mining algorithm, to show the similarities and differences between map-reduce and the generalized reduction APIs. Apriori is a well known algorithm for association mining, which also forms the basis for many newer algorithms [2]. Association rule mining is a process of analyzing a set of transactions to extract association rules and is a commonly-used and well-studied data mining problem. Given a set of transactions, each of them being a set of items, the problem involves finding subsets of items that appear frequently in these transactions. Formally, let \( L_i \) represent the set consisting of frequent itemsets of length \( i \) and \( C_i \) denote the set of candidate itemsets of length \( i \). In iteration \( i \), \( C_i \) is generated at the beginning of this stage and then used to compute \( L_i \). After that, \( L_i \) is used as \( C_{i+1} \) for iteration \( (i + 1) \). This process will iterate until the candidate itemsets become empty.

Fig. 2 shows the pseudo-code of apriori using the generalized reduction API. Using this API, in iteration \( i \), first,
the programmer is responsible for creating and initializing the reduction object. The reduction object is allocated to store the object ids for each frequent-i itemset candidate with associated counts of support. Then, the reduction operation takes a block of input transactions, and for each transaction, it will scan through the frequent-i itemset candidates, which are the frequent-(i-1) itemsets generated from last iteration. During the scan, if some frequent-i itemset candidate is found in the transaction, the object id for this itemset candidate is retrieved and its corresponding count of support is incremented by one. After the reduction operation is applied on all the transactions, the update_frequent_candidates operation is invoked to remove the itemset candidates whose count of support is below the support level that is defined by the programmer.

Fig. 3 gives the pseudo-code of apriori using the map-reduce API. The map function is quite similar to the reduction given in Fig. 2 and the only difference is that the emit_intermediate function is used instead of the accumulate operation for updating the reduction object. In iteration i, the map function will produce the itemset candidate as the key and the count one as the value, if this itemset can be found in the transaction. After the map phase is done, for each distinct itemset candidate, the reduce function will sum up all the one’s associated with this itemset. If its total count of occurrences is not less than the support level, this itemset and the total count will be emitted as the reduce output pair. The reduce output will be used in the update_frequent_candidates to compute L_i for the current iteration and use it as the C_{i+1} for next iteration, the same as in Generalized Reduction.

By comparing the implementations, we can make the following observations. In Generalized Reduction, the reduction object is explicitly declared to store the number of transactions that own each itemset candidate. During each iteration, the reduction object has to be created and initialized first as the candidate itemsets are dynamically growing. For each itemset candidate, the reduction object is updated in the reduction operation, if it exists in a transaction. When all transactions are processed, the update_frequent_candidates operation will compute the frequent itemsets and use it as the new candidate itemsets for next iteration.

In map-reduce, the map function checks the availability of each itemset candidate in a transaction and emits it with the count one if applicable. The reduce function will gather the counts of occurrences with the same itemset candidate and compare the total count with the support level. To summarize, for each distinct itemset candidate, the generalized reduction sums up the number of transactions it belongs to and checks whether the total number is not less than the support level. In comparison, map-reduce examines whether the itemset candidate is in the transaction in the Map phase, without performing the sum. It then completes the accumulation and compares the total count with the support level in the Reduce phase. Therefore, the reduction can be seen as a combination of map and reduce connected by the reduction object. Besides, the intermediate pairs produced by the Map tasks, will be stored in the file system or main memory and accessed by the Reduce tasks. The number of such intermediate pairs can be very large. In apriori, processing each transaction can produce several such pairs. In comparison, the reduction object only has a single count for each distinct itemset. Therefore, it requires much less memory. In addition, map-reduce requires extra sorting, grouping, and shuffling of the intermediate pairs, since each reduce task accumulates pairs with the same key value. In comparison, the generalized reduction API does not have any of these costs.

III. System Design

In this section, we discuss the design and implementation of the MATE system. This system has been implemented on top
of Phoenix, a shared memory implementation of map-reduce developed at Stanford [18].

A. The System API

The current implementation of our system is written in C. The system includes one set of system functions that are transparent to the programmers, in addition to the two sets of APIs, which are summarized in Table I. The first API set are the functions that are simply invoked in the application code to initialize the system, perform the computation, and produce the final output. The second API set includes the functions that have to be defined by the programmers, specific to an application.

The most important operation is the reduction function that processes one split of the data and updates the reduction object. Besides, the programmers can also define application-specific splitter function to split the data before each stage. Note that the splitter, reduction, and combination functions can be different for each iteration, if necessary. In other words, multiple such functions may be defined. Also, to maintain the computation state, the reduction object after each iteration’s processing is stored as intermediate results and can be reused by calling the get_intermediate_result function in future stages. The descriptions of other functions are listed in Table I. Note that C pointers can provide flexibility in handling different data types. Thus, the function arguments are declared as void pointers at several places to utilize this feature.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Fields</td>
<td></td>
</tr>
<tr>
<td>Input_data</td>
<td>Data pointer to the input dataset</td>
</tr>
<tr>
<td>Data_size</td>
<td>Size of the input dataset</td>
</tr>
<tr>
<td>Data_type</td>
<td>Type of the data instance</td>
</tr>
<tr>
<td>Stage_num</td>
<td>Computation-stage number</td>
</tr>
<tr>
<td>Splitter</td>
<td>Pointer to the Splitter function</td>
</tr>
<tr>
<td>Reduction</td>
<td>Pointer to the Reduction function</td>
</tr>
<tr>
<td>Finalize</td>
<td>Pointer to the Finalize function</td>
</tr>
<tr>
<td>Optional Fields</td>
<td></td>
</tr>
<tr>
<td>Unit_size</td>
<td># of bytes of one data instance</td>
</tr>
<tr>
<td>L1_cache_size</td>
<td># of bytes of L1 data cache</td>
</tr>
<tr>
<td>Model</td>
<td>Shared-memory parallelization model</td>
</tr>
<tr>
<td>Num_threads</td>
<td>Max # of threads for Reduction workers</td>
</tr>
<tr>
<td>Num_procs</td>
<td>Max # of processor cores available</td>
</tr>
</tbody>
</table>

Apart from the API functions, the data structure used to communicate between the application code and the runtime system is of type scheduler_args_t. Table II shows the basic and optional fields of this structure. The basic fields should be appropriately set in the application code since they provide the data pointers to both the input/output data buffers and the user-defined functions. The optional fields are used for scheduling and performance tuning.

B. Runtime System

As stated above, our system is built on Phoenix runtime, which in turn is based on P-threads [18]. We use the same scheduling strategy, where the data is partitioned and the splits are assigned to threads dynamically.

Execution Overview: Fig. 4 gives an overview of the execution of an application. Some of the key aspects are as follows:

Scheduler Initialization: To initialize the scheduler, the programmer needs to specify all required data pointers and functions. Particularly, the programmer should pass the input data pointers and implement the splitter, reduction, and combination functions. Also, the programmer has to declare a reduction object.

Reduction: After initialization, the scheduler checks the availability of processor cores, and for each core, it creates one worker thread. Before the Reduction phase starts, the splitter function is used to partition the data into equal-sized splits. Each worker thread will then invoke the splitter to retrieve one split and process it with the reduction function.

To avoid load imbalance, we try to assign Reduction tasks to worker threads dynamically instead of statically partitioning the data. Using this approach, each thread worker will repeatedly retrieve one split of the data and then process it as specified in the reduction function until no more splits are available. The split size, however, must be set appropriately to balance the lower overheads (few larger splits) and the load balancing (more smaller splits). To make use of temporary locality, by default, the runtime adjusts the split size such that the input data for a Reduction task can fit in the L1 cache. The programmer can also vary this parameter to achieve better performance given the knowledge of a specific application.

The reduction object is updated correspondingly in the reduction function as specified by the user code. Each thread will repeat this step until all data instances have been processed. At this point, the Reduction stage is over. The scheduler must
TABLE I
DESCRIPTIONS OF THE FUNCTIONS IN THE SYSTEM API.

<table>
<thead>
<tr>
<th>Functions Provided by the Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>int mate_init(scheduler_args*);</td>
</tr>
<tr>
<td>Initializes the runtime system. The arguments provides the needed functions and data pointers</td>
</tr>
<tr>
<td>int mate_scheduler(void*);</td>
</tr>
<tr>
<td>Schedules the Reduction tasks on the input dataset</td>
</tr>
<tr>
<td>int mate_finalize(void*);</td>
</tr>
<tr>
<td>Executes the Finalizing task to generate a final output if needed</td>
</tr>
<tr>
<td>void reduction_object_pre_init();</td>
</tr>
<tr>
<td>Allocates a default size of space for the reduction object</td>
</tr>
<tr>
<td>int reduction_object_alloc();</td>
</tr>
<tr>
<td>Assigns a unique object id representing a group of consecutive elements in the reduction object</td>
</tr>
<tr>
<td>void reduction_object_post_init();</td>
</tr>
<tr>
<td>Clones the reduction object for all threads in the full replication technique</td>
</tr>
<tr>
<td>void accumulate(int, int, void* value);</td>
</tr>
<tr>
<td>Updates a particular element of the reduction object by addition. The arguments are the object id, the offset, and the associated value</td>
</tr>
<tr>
<td>void reuse_reduction_object();</td>
</tr>
<tr>
<td>Reuses the reduction object for future stages. Memsets all fields of the reduction object to be zero</td>
</tr>
<tr>
<td>void process_next_iteration();</td>
</tr>
<tr>
<td>Notifies the runtime that the computation is proceeding to next stage</td>
</tr>
<tr>
<td>void* get_reduction_object(int);</td>
</tr>
<tr>
<td>Retrieves the reduction object copy belonging to one thread. The argument is the thread number</td>
</tr>
<tr>
<td>void* get_intermediate_result(int, int, int);</td>
</tr>
<tr>
<td>Retrieves a particular element of the reduction object in the combination buffer. The arguments are the stage number, the object id, and the offset</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functions Defined by the User</th>
</tr>
</thead>
<tbody>
<tr>
<td>int (<em>splitter_f)(void</em>, int, reduction_args*);</td>
</tr>
<tr>
<td>Splits the input dataset across Reduction tasks</td>
</tr>
<tr>
<td>void (<em>reduction_f)(reduction_args</em>);</td>
</tr>
<tr>
<td>The Reduction function. Each Reduction task executes this function on its input split. This function must be implemented by the user</td>
</tr>
<tr>
<td>void (<em>combination_f)(void</em>);</td>
</tr>
<tr>
<td>The Combination function. In the full replication model, this function is used to combine results from all threads</td>
</tr>
<tr>
<td>void (<em>finalize_f)(void</em>);</td>
</tr>
<tr>
<td>The Finalize function. The runtime will execute this function when all computation is done</td>
</tr>
</tbody>
</table>

wait for all reduction workers to finish before initializing the Combination stage.

Combination: The processing structure of Generalized Reduction enables us to develop general but efficient shared-memory parallelization schemes. Consider again the loop in Fig. 1. The main correctness challenge in parallelizing a loop like this on a shared-memory machine arises because of possible race conditions (or access conflicts) when multiple processors update the same element of the reduction object. Two obvious approaches for avoiding race conditions are: full replication and full locking. In the full replication approach, each processor can update its own reduction object and these copies are merged together later. In the full locking approach, one lock or latch is associated with each aggregated value.

Currently, we use the full replication approach to implement the parallelization. So a Combination phase is required to merge the reduction-objects of multiple threads when all data is processed. The full locking and its optimized schemes are being considered and will be supported in the future.

In the Combination phase, the scheduler will spawn a combination-worker thread to merge the reduction object copies of all Reduction threads. The result is a single copy of the final reduction object after every iteration. If the application involves multiple iterations, this object is stored by the scheduler and can be accessed by the future stages.

Finalize: This phase occurs after all other stages of the computation have been performed, i.e., each Combination stage has its own copy of the reduction object, which is combined from multiple threads. The programmer is then able to perform a manipulation on the reduction-objects to summarize the results specific to an application. The scheduler also needs to free the allocated space in this phase, but this is transparent to the programmer.

Buffer Management: The runtime system handles two types of temporary buffers. One is the reduction-object buffer allocated for each thread to do its own computation over different splits. The other is the combination buffer created for storing the intermediate output results of each stage. The combination buffer is of type reduction object in our implementation. The buffers are initially sized to a default value and then grown dynamically if needed.

Fault Tolerance: Fault detection and recovery have been an important aspect of map-reduce implementations, as data-intensive applications can be very long running, and/or may
need to be executed on a large number of nodes or cores. Our current support for fault-tolerance is based on the Phoenix system. It supports fault-recovery for the Reduction tasks. If a Reduction worker does not finish within a reasonable time limit, the scheduler assumes that a failure of this task has occurred. The failed task is then re-executed by the runtime. In our implementation, separate buffers are allocated for the new task to avoid data access conflicts.

The alternate API implemented in the MATE system also allows a different approach for fault-tolerance. This will be a low-cost checkpointing approach, where a copy of the reduction object can be cached at another location after processing of each split. If a thread fails while processing a particular split, we can restart computation using the cached reduction object. We only need to process the current split assigned to this thread, and not the splits that had been already processed. We plan to support this approach in future versions of the MATE system.

IV. EXPERIMENTAL RESULTS

In this section, we evaluate the MATE system on multi-core machines by comparing its performance against Phoenix and Hadoop.

Since our main focus is on data mining applications, we choose three popular data mining algorithms. They are k-means clustering, apriori association mining, and principal components analysis (PCA). Among these, we described apriori association mining earlier in Section II. K-means clustering is one of the commonly used data mining algorithms [12]. The clustering problem is as follows. We consider data instances as representing points in a high-dimensional space. Proximity within this space is used as the criterion for classifying the points into clusters. Four steps in the sequential version of k-means clustering algorithm are as follows: 1) start with k given centers for clusters; 2) scan the data instances, for each data instance (point), find the center closest to it and assign this point to the corresponding cluster, 3) determine the k centroids from the points assigned to the corresponding centers, and 4) repeat this process until the assignment of points to clusters does not change. PCA is a popular dimensionality reduction method that was developed by Pearson in 1901. Its goal is to compute the mean vector and the covariance matrix for a set of data points that are represented by a matrix.

The datasets and the application parameters we used are as follows. For k-means, the dataset size is 1.2 GB, and the points are 3-dimensional. The number of clusters, k, was set to be 100. With apriori, we used a dataset that has 1,000,000 transactions, with each one having at most 100 items, and the average number of items in each transaction is 10. The support and confidence levels were 3% and 9%, respectively, and the application ran for 2 passes. In PCA, the number of rows and columns used in our experiments were 8,000 and 1,024, respectively.

Our experiments were conducted on two distinct multi-core machines. One system uses Intel Xeon CPU E5345, comprising two quad-core CPUs (8 cores in all). Each core has a clock frequency of 2.33GHz and the system has a 6 GB main memory. The other system uses AMD Opteron Processor 8350 with 4 quad-core CPUs (16 cores in all). Each core has a clock frequency of 1.00GHz and the system has a 16 GB main memory.

We executed the aforementioned three applications with both MATE and Phoenix systems on the two multi-core machines. For two of the applications, k-means and apriori, we also do a comparison with Hadoop. We could not compare the performance of PCA on Hadoop, as we did not have an implementation of PCA on Hadoop. Hadoop implementations of k-means and apriori were carefully tuned by optimizing various parameters [13].

Figures 5 through 10 show the comparison results for these three applications as we scale the number of cores used. Since our system uses the same scheduler as Phoenix, the main performance difference is likely due to the different processing structure, as we had discussed in earlier sections. Note that for Hadoop, we only report results from the maximum number of cores available on each machine (8 and 16). This is because the number of tasks in Hadoop cannot be completely controlled, as there was no mechanism available to make it use only a subset of the available cores.

Figures 5 and 6 show the comparison for k-means. Our
system outperforms both Phoenix and Hadoop on both the machines. MATE is almost twice as fast as Phoenix with 8 threads on the 8-core machine, and almost thrice as fast with 16 threads on the 16-core machine. Hadoop is much slower than both MATE and Phoenix, which we believe is due to the high overheads in data initialization, file system implementation, and use of Java for computations. As compared to MATE, it also has the overhead of grouping, sorting, and shuffling of intermediate pairs [13]. We also see a very good scalability with increasing number of threads or cores with MATE. On the 8-core machine, the speedup with MATE in going from 1 to 8 threads is about 7.8, whereas, Phoenix can only achieve a speedup of 5.4. The result is similar on the 16-core machine: our system can get a speedup of about 15.0 between 1 and 16 threads, whereas, it is 5.1 for Phoenix. With more than 100 million points in the k-means dataset, Phoenix was slower because of the high memory requirements associated with creating, storing, and accessing a large number of intermediate pairs. MATE, in comparison, directly accumulates values into the reduction object, which is of much smaller size.

The results from PCA are shown in Figures 7 and 8. PCA does not scale as well with increasing number of cores. This is because some segments of the program are either not parallelized, or do not have regular parallelism. However, we can still see that, with 8 threads on the 8-core machine and 16 threads on the 16-core machine, our system was twice as fast as Phoenix. Besides the overheads associated with the intermediate pairs, the execution-time breakdown analysis for Phoenix shows that the Reduce and Merge phases account for a non-trivial fraction of the total time. This overhead is non-existent for the MATE system, because of the use of the reduction object.

Finally, the results for apriori is shown in Figures 9 and 10. For 1 and 2 threads, MATE is around 1.5 times as fast as
Phoenix on both the machines. With 4 threads and more, the performance of Phoenix was relatively close to MATE. One of the reasons is that, compared to k-means and PCA, the Reduce and Merge phases take only a small fraction of time with the apriori dataset. Thus, the dominant amount of time is spent on the Map phase, leading to good scalability for Phoenix. For the MATE system, the reduction object can be of large size and needs to be grown and reallocated for all threads as the new candidate itemsets are generated during the processing. This, we believe, introduced significant overheads for our system. Hadoop was much slower than our system, by a factor of more than 25 on both the machines.

Overall, our experiments show that the MATE system has between reasonable to significant speedups for all the three datamining applications. In most of the cases, it outperforms Phoenix and Hadoop significantly, while slightly better in only one case.

V. RELATED WORK

The topics of data-intensive computing and map-reduce have received much attention within the last 2-3 years. Efforts underway include projects that have used and evaluated map-reduce implementations, as well as those that are trying to improve its performance and programmability in different computing environments.

Projects in both academia and industry are working towards improving map-reduce. CGL-MapReduce [7] uses streaming for all the communications, and thus improves the performance to some extent. Mars [10] is the first attempt to harness GPU's power for map-reduce. Farivar et al. introduced an architecture named MITHRA [8] and integrated the Hadoop map-reduce with the power of GPGPUs in the heterogeneous environments. Zaharia et al. [21] improved Hadoop response times by designing a new scheduling algorithm in a virtualized data center. Seo et al. [19] proposed two optimization schemes, prefetching and pre-shuffling, to improve Hadoop's overall performance in a shared map-reduce environment. Ranger et al. [18] have implemented Phoenix that we have used as the basis for our implementation.

Facebook uses Hive [20] as the warehousing solution to support data summarization and ad-hoc querying on top of Hadoop. Yahoo has developed Pig Latin [16] and Map-Reduce-Merge [4], both of which are extensions to Hadoop, with the goal being to support more high-level primitives and improve the performance. Google has developed Sawzall [17] on top of map-reduce to provide higher-level API. Microsoft has built Dryad [11], which is more flexible than map-reduce, since it allows execution of computations that can be expressed as DAGs.

Chu et al. [5] have used Google’s map-reduce for a variety of learning algorithms. Gillick et al. [9] evaluate Hadoop in a distributed environment with multi-node clusters using both a set of benchmark applications and machine learning algorithms. They also offer improvements possibilities to Hadoop’s implementation. Disco [1] is an open source map-reduce runtime which is similar to Google’s and Hadoop’s architectures, but does not support a distributed file system.

VI. CONCLUSIONS

This paper has described a system MATE, which provides an alternate API to the original map-reduce model for developing data-intensive applications. Our API takes advantage of a user-declared reduction object, which substantially reduces an overhead arising because of the original map-reduce API. These overheads are caused by high memory requirements of intermediate results and the need for data communication between Map and Reduce stages. MATE supports this alternate API on top of Phoenix, a multi-core map-reduce implementation from Stanford.

We have evaluated our system with three popular data mining algorithms, which are k-means clustering, apriori association mining, and principal component analysis. We also compared our system with Phoenix and Hadoop on two distinct multi-core platforms. Our results show that MATE achieves good scalability and outperforms Phoenix and Hadoop for all three applications. Overall, our work suggests that, despite an easy to program API support, map-reduce may not fit naturally into some data-intensive applications and results in non-trivial performance losses. An alternate API, which is based on the generalized reduction, offers better performance
and scalability while still maintaining easy programmability.

ACKNOWLEDGEMENTS

The authors thank Christos Kozyrakis and his research
group at Stanford for making the Phoenix system open source.

REFERENCES

DataBases (VLDB’94), pages 487–499, Santiago, Chile,
September 1994.
Mining association rules between sets of items in large
databases. In Proceedings of the 1993 ACM SIGMOD
[4] Hung chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and
Douglas Stott Parker Jr. Map-reduce-merge: simplified