ABSTRACT

MapReduce parallel programming model has seen wide adoption in data center applications. Recently, lightweight, fast, in-memory MapReduce runtime systems have been proposed for shared memory systems. However, what factors affect performance and what performance bottlenecks exist for a given program, are not well understood. This paper builds an analytical model to capture key performance factors of shared memory MapReduce and investigates important performance trends and behavior. Our study discovers several important findings and implications for system designers, performance tuners, and programmers.

Our model quantifies relative contribution of different key performance factors for both map and reduce phases, and shows that performance of MapReduce programs are highly input-content dependent. Our model reveals that performance is heavily affected by the order in which distinct keys are encountered during the Map phase, and the frequency of these distinct keys. Our model points out cases in which reduce phase time dominates the total execution time. We also show that data-structure and algorithm design choices affect map and reduce phases differently and sometimes affecting map phase positively while affecting reduce phase negatively. Finally, we propose an application classification framework that can be used to reason about performance bottlenecks for a given application.

1. INTRODUCTION

Technology scaling has made multicore architectures commercially prevalent. Parallel programming has become necessary for taking advantage of the parallelism provided by multicore platforms [2]. However, programmers often find managing concurrency a daunting task due to data races, deadlocks and other side effects [4, 9, 22]. There are efforts to make parallel programming easier by allowing programmers to express parallelism at a higher level, and providing automatic management of concurrency. An example of such an effort is MapReduce runtime system [8].

MapReduce, originally proposed by Google for cluster computing, has enjoyed a widespread commercial adoption by various companies, including Amazon, Facebook, and Yahoo [1, 3, 25]. With MapReduce runtime systems, programmers provide two functions: map() and reduce(), which are executed as separate phases. During the map phase, the map function accepts input data to produce intermediate output in the form of a list of <key,value> pairs. In the reduce phase, the list of key-value pairs are read by the reduce function, and all values having the same key are aggregated. Programmers are attracted to MapReduce programming model due to increased programmer productivity, low barrier to entry, high level programming abstraction, the lack of data races and deadlocks, automatic management of threads, synchronization, and fault tolerance.

There are two versions of MapReduce runtime systems. Cluster- or disk-based MapReduce (e.g. Hadoop) relies on a file system to store the intermediate output. Shared memory MapReduce (e.g. Phoenix), on the other hand, relies on storing the intermediate output in the memory shared by all threads. Shared memory MapReduce is more appropriate for computation that fits in the main memory, but due to avoiding I/O, it outperforms disk-based implementation (e.g. Hadoop) significantly [7, 12]. Therefore, it is possible that future MapReduce programs may use a hybrid approach, e.g. in-memory version in a shared memory node, and disk-based version across shared memory nodes. This paper focuses on shared memory MapReduce.

Shared memory MapReduce’s scalability has been demonstrated for a range of applications [16, 18]. However, the general understanding of what factors affect performance and what performance bottlenecks exist for shared memory MapReduce programs is still at a rudimentary level. Little is understood beyond an observation that computation that can naturally be expressed using key-value pairs tends to perform well [8, 18]. Indeed, there are factors that are not identified yet that significantly affect performance.

Figure 1(a) shows the execution time of word count (WC) application for three equally-sized inputs on an Intel Xeon system using Metis [16], a shared memory MapReduce system. The input text documents contain identical words (Input A), all distinct words (Input B), and mixed (Input C). The figure shows that scalability of the application is also affected by the order in which distinct keys are encountered during the Map phase, and the frequency of these distinct keys. Our model reveals that performance is heavily affected by the order in which distinct keys are encountered during the Map phase, and the frequency of these distinct keys. Our model points out cases in which reduce phase time dominates the total execution time. We also show that data-structure and algorithm design choices affect map and reduce phases differently and sometimes affecting map phase positively while affecting reduce phase negatively.

Finally, we propose an application classification framework that can be used to reason about performance bottlenecks for a given application.

Figure 1(b) shows the speedup of WC for up to eight threads, normalized to each input’s single thread case. The figure shows that scalability of the application is also affected by the input content. These observations demand a deeper understanding into the performance characteristics of MapReduce. Therefore, the goal of this paper is to identify and analyze key performance factors of shared memory MapReduce. Developing such an understanding is challenging because the intricate interaction between application’s characteristics, the input type and content, number of threads, MapReduce runtime system implementation, and machine charac-
teristics. However, a deeper understanding is necessary if programmers are to write efficient code with shared memory MapReduce.

To reach our goal, we develop a simple yet powerful parametric analytical performance model to establish mathematical relationship among key performance factors of shared memory MapReduce. The model relies on a mixture of Markov process and algorithm complexity analysis, augmented with empirically-derived parameters. The model is validated against results from real applications running on a real system. To the best of our knowledge, this is the first analytical model for understanding shared memory MapReduce performance. Our study discovers several important findings, among them are:

- Shared memory MapReduce performance is highly affected by the number of intermediate output key-value pairs relative to the number of distinct keys. Our model also discovers that the order in which keys are encountered during the Map phase affects execution time significantly. Inputs of which the content differs in these metrics produce significantly different performance, captured quantitatively by our model.

- Execution time of reduce phase is highly affected by task queue overhead, and this overhead may offset the benefit from increasing the number of reduce worker threads.

- The size and choice of data structure for storing intermediate output affects map and reduce phases differently, and sometimes affecting map phase positively while affecting reduce phase negatively. Our model enables us to quantify this interesting behavior. Furthermore, we show that different input content favors different choices for data structure and algorithms in the map phase, sometimes favoring non-intuitive choices.

Our model quantifies the relative contribution of different components/bottlenecks in the map and reduce phases, and how they are affected by program characteristics and input content. Finally, we propose an application classification framework that can be used to predict likely performance bottlenecks of applications. Given the knowledge about the program and input characteristics, such bottleneck analysis can be performed even prior to coding in some cases.

With the increasing prevalence of multicore systems and increasing adoption of MapReduce programming model, understanding the key factors affecting shared memory MapReduce will become increasingly critical for programmers, performance tuners, and system designers. We believe that our study is an important contribution towards that goal.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 provides an overview to shared memory MapReduce programming model. Section 4 describes our analytical model while Section 5 uses the model to explore how performance is affected by various factors. Section 6 describes evaluation platform. Section 7 shows model validation and evaluation results on real systems. Section 8 describes our application classification framework. Finally, Section 9 concludes the paper.

2. RELATED WORK

MapReduce programming model was proposed by Google [8], and has been adopted by various companies, including Amazon, Yahoo, and Facebook. The typical MapReduce system used is cluster-based (or disk-based) Hadoop and its variants [1, 3, 25]. MapReduce has also been extended to run on various platforms, including GPUs [10, 20, 11], FPGAs [19], and IBM Cell BE [13].

Shared memory MapReduce has also been developed as an alternative to cluster/disk-based MapReduce. In shared memory MapReduce, the intermediate output is stored directly in memory (rather than in the file system), shared by all threads. Shared memory MapReduce is more suitable for computation that fits in the main memory, but due to avoiding I/O, it outperforms disk-based implementation (e.g. Hadoop) significantly [7, 12]. This paper focuses on shared memory MapReduce.

The first shared memory MapReduce runtime system (Phoenix) was developed by Ranger et al. [18]. They demonstrated Phoenix’s performance scalability on contemporary CMP systems. Yoo et al. [23] further optimized Phoenix runtime system, which has been used in various other studies [5, 7, 12, 14, 15, 24]. Concurrent to our work, Talbot et al. [21] developed Phoenix++ runtime system, C++ version of Phoenix, that provides flexibility to adjust some parameters to suit the workload characteristics. Another shared memory MapReduce runtime system, Metis, was developed by Mao et al. [16], to overcome some of the performance problems associated with Phoenix. While the performance and scalability of shared memory MapReduce have been demonstrated, detailed understanding of what factors affect shared memory MapReduce performance has not been discovered, which is the focus of this paper.

3. BACKGROUND

3.1 MapReduce Programming Model

MapReduce program execution is divided into two phases: map and reduce. In the map phase, input records are distributed across map workers in a data-parallel manner. Each map worker performs computation by calling a user-defined map function. The map function produces output in the form of a list of <key, value> pairs, and the output is stored in the intermediate file or buffer. The reduce phase reads from the intermediate file or buffer to perform reduction on the output of the map phase. Each reduce worker is assigned a key or a group of keys and calls a user-defined reduce function, which aggregates all values having the same key. The output of the reduce function are <key, aggregate-value> pairs, which are then sorted to produce the final output. Thus, map workers achieve data parallelism, while reduce workers perform parallel reduction.

An example of MapReduce application is word count, which counts the number of occurrences of each distinct word in a text document. In word count, each map worker takes a chunk from the text document input, and for each word that it encounters, it produces a <key, 1> pair as intermediate output. At the completion of the map phase, the reduce phase is started. Each reduce worker works on a specific key (a distinct word) to sum up the list of values (counts) associated with the key.

Programmers are attracted to MapReduce due to its high level abstraction and simplicity. Programmers define the map and reduce functions, and the MapReduce runtime system automatically manages concurrency and fault tolerance, such as creating, dispatching, and scheduling map/reduce threads, performing synchronization, sorting output, managing locality, restarting crashed threads, etc.

3.2 Shared Memory MapReduce

Shared memory MapReduce keeps the intermediate buffer as a data structure stored in the main memory, where any map worker thread can write to, and any reduce worker thread can read from. Without shared memory, map workers and reduce workers must communicate through the file system on disks and rely on remote procedure calls for managing the workers, which are expensive. As a result, shared memory MapReduce has been shown to outperform the disk-based MapReduce by up to an order of magnitude on com-
temporary multicore machines [7, 12]. The drawbacks of shared memory MapReduce are that it is more suitable for smaller computation that fits in the main memory, and fault tolerance cannot be achieved as easily without relying on non-volatile storage. Therefore, it is possible that future MapReduce programs may use a hybrid approach, e.g. disk-based version across shared memory nodes providing fault tolerance on persistent storage, with each computation on a node further divided into finer granularity tasks to exploit in-memory computation [27].

Currently, there are two shared memory MapReduce runtime systems available publicly: Phoenix [18, 23] and Metis [16]. While Phoenix has been studied more extensively [5, 7, 12, 14, 15, 24], Metis was said to be designed to improve key shortcomings of Phoenix. In addition, while both Phoenix and Metis share an intermediate buffer data structure (2-dimensional array of hash buckets), they differ significantly in what data structure is used for each bucket, and in the key algorithms used in the map and reduce phases. Therefore, we design our model to be parametric, such that a single model can capture both Phoenix and Metis, and future variants.

In both Phoenix and Metis, input data is split into equal-sized chunks that form tasks, and tasks are queued and assigned dynamically to map worker threads. Figure 2 illustrates a generic intermediate data structure (used in both Phoenix and Metis), which is a two dimensional array with number of rows equaling the number of map workers, and number of columns equaling the number of hash buckets. Each thread works independently on a row of the matrix and stores its intermediate output in hash buckets in that row. When a thread produces a key-value pair, the key is hashed into a bucket using a hash function, and is added to the bucket. Each bucket maintains a list of distinct keys in a sorted order (to facilitate fast key searching), with each key having a list of values associated with it. Since different threads use different matrix rows to store their intermediate output, there is no data sharing or synchronization among map worker threads.

In the reduce phase, each column of bucket forms a task, and tasks are queued and assigned dynamically to reduce worker threads. Each reduce worker performs reduction by aggregating values that share a common key, both within a bucket and across buckets in the same column. Since reduce workers work on separate columns of buckets, they do not share data. However, map workers and reduce workers share data through the intermediate buffer. After aggregating values, a reduce worker produces \(<\text{key}, \text{aggregate-value}>\) pair for each distinct key assigned to it, and participates in a merge sort to combine the output from all reduce workers so that the output in the final output buffer is sorted based on key order.

4. ANALYTICAL MODEL FORMULATION

4.1 Assumptions and Scope

The goal of our modeling is not to predict execution time accurately. Rather, it is to capture mathematical relationships of how various factors are related and affect performance, and quantify their relative magnitude of impact on performance.

The scope of our study is shared memory MapReduce performance. Hence, we do not model I/O and network performance. The aggregate effects of processor and cache performance are not explicitly quantified, but are captured as parameters in our parametric model. The values of these parameters can be empirically derived on a target machine.

We assume a standard MapReduce programing model where the reduce phase starts after all map tasks have fully completed. Some variants of MapReduce break up the map task execution and interleave (but not overlap) it with the reduce phase [7]. While this model may not be readily applicable to such a variant, it can be adapted for it if desired.

We instantiated our model for Phoenix and Metis, two shared memory MapReduce libraries that are publicly available at the time of this study. Taking into account the need for the model to be extensible for future shared memory MapReduce systems, we designed the model to be parametric, such that minor variations only differ in parameter values. For example, most of the differences between Phoenix and Metis are contained in the parameter values that are input to the model.

4.2 Modeling Approach

Table 1 shows parameters and notations we use in our model. During the map phase, all map tasks are executed and the intermediate output is stored into the intermediate buffer. Therefore, we can divide the map phase into two sub-phases: computing the map task (map-comp), and storing the intermediate output (map-output). The map-comp phase is inherent to the algorithm, and its performance is relatively independent of whether it is executed with MapReduce or other models (e.g. PThreads, OpenMP). However, the map-output phase is unique to MapReduce and is an important determinant of performance. Hence, we will focus on modeling map-output performance.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N_t )</td>
<td>Number of Input records</td>
</tr>
<tr>
<td>( N_{map} )</td>
<td>Number of map worker threads</td>
</tr>
<tr>
<td>( N_{redt} )</td>
<td>Number of reduce worker threads</td>
</tr>
<tr>
<td>( C )</td>
<td>Chunk size (#input records grouped as a task)</td>
</tr>
<tr>
<td>( P )</td>
<td>Number of intermediate output (&lt;k, v&gt;) Pairs</td>
</tr>
<tr>
<td>( D )</td>
<td>Number of Distinct keys</td>
</tr>
<tr>
<td>( B )</td>
<td>Number of hash Buckets (columns)</td>
</tr>
<tr>
<td>( w_1, w_2, \ldots, w_5 )</td>
<td>Map execution time component weights</td>
</tr>
<tr>
<td>( w_6, w_7, w_8 )</td>
<td>Reduce execution time component weights</td>
</tr>
</tbody>
</table>
During the reduce phase, all reduce tasks are computed by aggregating values from a common key (reduce-comp) and the output from the reduce-tasks are put into final buffer in key-sorted order (reduce-output).

### 4.3 Map Phase

To simplify the discussion, we will start with a simple case where there are $P$ key-value pairs to be inserted into a single hash bucket in the intermediate buffer. Of those $P$ key-value pairs, there are $D$ unique keys. We will later expand the model to take into account multiple buckets.

#### 4.3.1 Modeling map-output Time

Suppose that an intermediate output with key-value pair $<k, v>$ is generated and the bucket already has $d$ keys prior to this point. The map-output phase needs to determine whether the key $k$ is already present in the bucket. If it is, $v$ is appended to the list of values for key $k$. Otherwise, a new entry for key $k$ is inserted into the bucket, while keeping all keys in the bucket sorted.

Determining whether a key $k$ is already present in the bucket requires searching. Scalable searching algorithm, such as binary search, requires $O(\log d)$ time, assuming that keys are already sorted. Inserting a value to an existing list of values takes a constant time ($O(1)$), because values do not need to be kept sorted. Finally, the complexity of inserting a new key into a sorted collection of keys may be $O(\log d)$ in the best case (e.g., keys are kept in a balanced binary tree, as in Metis) or $O(d)$ in other cases (e.g., keys are kept sorted in an array and shifted to make room for a new one, as in Phoenix). While keeping keys in a balanced tree (e.g., B+ Tree in Metis) gives the best asymptotic complexity, searching and insertion are expensive, as they involve pointer de-referencing, recursion, and potentially tree rebalancing. Sorted array provides much faster key searching and insertion, at the expense of worse asymptotic key insertion time. Choosing the right data structure is not trivial since there is a fundamental trade-off that is not easy to reconcile.

Let us further assume that the average key searching, key insertion, and value insertion take $w_1, w_2,$ and $w_3$ on average. Processing $<k, v>$ in the map-output phase takes $w_1 \log d + w_3$ if the key was in the bucket, or $w_1 \log (d-1) + w_2 d + w_3$ if the key was not in the bucket. Assuming that the probability of each of $D$ distinct keys appearing is equal, we can capture map-output time using the following Markov model.

Let $(d)$ be a state in the Markov model, representing that currently $d$ out of $D$ distinct keys have been added to the bucket. Subtracting the last $<k, v>$ that was processed, the previous state is either $(d-1)$ if $k$ was a new key, or $d$ if $k$ was not a new key. Since all keys have an equal probability of occurring, the probability of seeing one not among $d-1$ keys is $\frac{D-d}{D-1}$, while the probability of seeing one of the $d$ keys that have occurred is $\frac{d}{D}$. The resulting Markov model is shown in Figure 3. The bold circle shows the current state, and both circles show possible previous states. Each edge in the Markov model shows the transition probability between two states. The transition probabilities do not add up to one because the transitions originate from different states.

Algebraically, the probability of reaching a state $(d)$, or $Pr(d)$, is:

$$Pr(d) = \begin{cases} 1 & \text{if } d = 0 \\ 0 & \text{if } d > D \\ \frac{d}{D} Pr(d) + \frac{D-d+1}{D} Pr(d-1) & \text{otherwise} \end{cases} \quad (1)$$

By unrolling the recursion in the Markov model, we can find the expected total time needed to complete the map-output phase ($T_{m_o}$). Unfortunately, computing this expression is not practical as $P$ is typically a very large number. Thus, we focus on two special cases derived from the model: worst case and best case map-output time. The worst case occurs when all keys occur early, which implies that all subsequent intermediate outputs will have to search the maximum number of keys (all $D$ keys). We refer to this case as skewed key occurrence (SKO). On the other hand, the best case occurs when keys occur late. Since we assume that each key has an equal chance of appearing, each key will occur exactly $\frac{w_3}{P}$ times before the next distinct key appears. We refer to the latter case as uniform key occurrence (UKO). SKO and UKO provide lower and upper bounds for $T_{m_o}$, under the assumption of uniform key occurrence frequency.

SKO and UKO are also significant special cases because they represent real scenarios. For example, for machine learning and scientific applications [6], typically all keys are emitted in the first map task, and remaining tasks only insert values as all the keys are already inserted into the bucket by the first map task. Such applications exhibit SKO. Other applications with relatively small number of keys also show SKO behavior. UKO is rarer, but may occur in real situations, such as when the input records are already sorted. For example, in word count application, a document may already be sorted in alphabetical order, and in password check application, user IDs and passwords may already be sorted. All other applications can be expected to show map-output performance between those of SKO and UKO.

With a sorted key array under SKO (Phoenix), when keys are initially inserted, we incur a cost of $w_2 + (w_1 \log 1 + w_2) + \ldots + (w_1 \log (D-1) + w_2 D) = w_1 \log (D-1)! + w_2 D(D+1)/2$. Once all keys are inserted, there are $P-D$ intermediate records left to emit. Each one of them takes $w_1 D$ key searching time, bringing the total to $(P-D)w_1 L D$. Therefore, total cost of key searching can be written as $w_1 \log D! + (P-D-1)w_1 L D$. Finally, in each of the intermediate output, value insertion takes $w_3$ time, bringing the total to $w_3 P$.

Therefore, the total map-output time with SKO is:

$$T_{m_o}^{SKO} = w_1 \log D! + (P-D-1)w_1 L D + w_2 D(D+1)/2 + w_3 P \quad (2)$$

Under UKO, each distinct key appears as late as possible, hence keys are uniformly spaced at a distance of $\frac{P}{D}$. There are as many key insertions and value insertions as in SKO, so the difference is only the amount of time spent for searching keys. Since key insertion is uniformly spaced, $\frac{P}{D}$ intermediate outputs after the first pair incur $w_1 L 1$ key searching time, next $\frac{P}{D}$ intermediate outputs incur $w_1 L 2$ key searching time, and so on. The last $(P-1)$ intermediate outputs incur $w_1 L D$ key searching time. Therefore, total key searching time can be written as $w_1 (\frac{P}{D} \log 1 + \frac{P}{D} \log 2 + \ldots + (\frac{P}{D} - 1) \log D)$. This is equivalent to $w_1 (\frac{P}{D} \log D! - \log D)$. Therefore, the total map-output time in UKO is:

$$T_{m_o}^{UKO} = w_1 (\frac{P}{D} \log D! - \log D) + w_2 D(D+1)/2 + w_3 P \quad (3)$$

Asymptotically, $O(x^2) > O(\log x)$, or $O(x)$. Furthermore, with Stirling’s approximation [26], $\ln x! \approx x \ln x - x$, which means that for a large $D$, $\log D! \approx \log D - \frac{D}{2}$, now let us consider Equation 2 and 3. When $P = D$ (all intermediate output pairs have distinct keys), then key insertion time dominates, i.e. $T_{m_o} = O(D^2)$. However, when $P > D$ (key insertions are rare), key searching time dominates. For SKO, $T_{m_o} = O(P \log D)$. For UKO, $T_{m_o} = O(\frac{P}{D} \log D) = O(P \log D)$.

With key management using B+ Tree data-structure (Metis), key insertion takes a $\log$ of the number of keys already inserted in the
tree. Hence, Equations 2 and 3 change slightly to:

\[ T_{sko}^{mo} = w_1 \left( \log D + (P - D - 1) \log D \right) + w_2 \log D! + w_3 P \quad (4) \]

\[ T_{uko}^{mo} = w_1 \left( \frac{P}{D} \log D! - \log D \right) + w_2 \log D! + w_3 P \quad (5) \]

This leads to the following observation:

**Observation 1.** The map-output time highly depends on the number of distinct keys D and the number of intermediate output pairs P. With a sorted array, if \( \frac{P}{D} \rightarrow 1 \), then \( T_{mo} = O(D^2) \) for both SKO and UKO. If \( \frac{P}{D} \rightarrow 0 \), then \( T_{mo} = O(P \log D) \) for both SKO and UKO. With a binary tree, \( T_{mo} = O(D \log D) \) if \( \frac{P}{D} \rightarrow 1 \), or \( T_{mo} = O(P \log D) \) if \( \frac{P}{D} \rightarrow 0 \), for both UKO and SKO.

**Significance:** The significance of this observation is that it points out that a key performance factor in shared memory MapReduce is the ratio of the number of distinct keys to number of intermediate output pairs. Since the ratio is affected by the inherent computation algorithm, input content, and chunk size, the observation explains why shared memory MapReduce program performance is sensitive to these factors. In contrast, a program written in PThreads or OpenMP is sensitive to data layout and memory access pattern, rather than input content. Performance tuners can use this insight to improve the performance, e.g. decreasing the number of pairs by increasing the map-task size in some cases.

Furthermore, the observation points out avenues for optimizing shared memory MapReduce. For example, when keys are comparable to total output pairs (D≠P), key insertion dominates the execution time. One way to reduce the dominance of key insertion is to use multiple hash buckets so that the number of keys per bucket decreases. In addition, Metis’ use of B+Tree produces better performance under a large number of distinct keys, but produces worse performance under a small number of distinct keys, and incurs significantly more time per instance of key searching and insertion. This interesting trade-off will be quantified in and discussed in Section 7.1.

Now let us contrast the map-output time for SKO and UKO (Equation 2 and 3). Using Stirling’s approximation and taking the difference between them, we obtain:

\[ T_{sko}^{mo} - T_{uko}^{mo} = w_1 \left( \log D! + (P - D - 1) \log D - \frac{P}{D} \log D! - \log D \right) \]

\[ = \frac{w_1}{\ln 2} (P - D) \quad (6) \]

which leads to the following observation:

**Observation 2.** Another key performance factor is the order in which keys occur. The gap in performance between the worst case, where all keys occur early in the intermediate output bucket, and the best case, where key occurrence is uniformly spaced, grows with the number of intermediate output pairs but decreases with the number of distinct keys, i.e. proportional to \( P - D \).

**Significance:** The significance of this observation is providing a link between the performance of shared memory MapReduce program and the order in which keys appear. The link may be new to programmers, as input content is not typically a major performance factor in PThreads or OpenMP programs. Using this information, performance tuners and programmers can optimize their programs, for example by rewriting the input or programs such that keys are emitted in a more uniform order. We also note that the observation is only dependent on the ordering of keys, which is a program/input characteristic, but is independent of the hash bucket data structures.

### 4.3.2 Impact of Multiple Buckets and Multithreading

The previous section assumes a single hash bucket. Using multiple hash buckets reduces the number of distinct keys per hash bucket, which in turn reduces map-output time, assuming everything else remains the same. If we use a large number of buckets such that only one key maps to a single bucket, the map-output time is reduced to \( O(P) \). However, in practice, using too many buckets increases cache and memory footprint and reduces cache locality, hurting performance by increasing the cost of each key searching and insertion. Therefore, the number of buckets is a critical performance factor.

Finding an optimum number of buckets is difficult. However, we can estimate at which number of hash buckets \( B \) the probability of key collisions becomes small. To solve this specific question, we note that the probability of finding two keys that map into the same bucket is a birthday paradox problem. Given a particular key, the probability of another key mapping to the same bucket is \( \frac{1}{2} \). Since there are \( \left( \frac{D}{B} \right) \) possible pairs, the probability to find any two keys mapping to a single bucket is \( \frac{1}{2} \left( \frac{D}{B} \right) \). Suppose that we want to keep the probability \( \leq \frac{1}{2} \). Rearranging the inequality, the minimum number of buckets is \( B \geq \frac{D^2}{2} - D \).

However, caution is needed. In some cases, we have no apriori knowledge of how many distinct keys there would be in the program, hence there is a risk of choosing too many buckets. Too many buckets not only lead to lower cache locality, but as we will demonstrate in Section 4.4, also increase the reduce phase time.

Finally, we want to model the multithreading effect on the map phase. As discussed in Section 3.2, map worker threads do not share data or synchronize until the end of the map phase. Thus, there are only two potential effects of using multiple map threads: the cost of fetching tasks from a shared task queue, and the impact of false sharing in hash bucket structures between threads. The latter is easily avoided through padding key-value pairs so that they fit multiples of cache line size (employed by both Phoenix and Metis). The former depends on how map tasks are assigned to map worker threads: statically or dynamically. While static task assignment requires no synchronization, it leads too easily to load imbalance. Dynamic assignment incurs synchronization but ensures load balance. Hence, both Phoenix and Metis adopt dynamic task assignment. When map threads simultaneously attempt to fetch map tasks from the task queue, mutual exclusion between threads is required for correctness, which is achieved using a lock or atomic instructions. Since mutual exclusion implies sequentiality, task fetching time is simply number of tasks multiplied by the average time to fetch a task.

Let \( w_4 \) denote the average time taken to hash a key, and \( w_5 \) denote the average time to fetch a map task. The total contribution of hashing and map task fetching on map-output time are \( w_1 P \) and \( w_5 \left( \frac{N_1}{C} \right) \), respectively, where \( N_1 \) is total number of input records and \( C \) is map task size (Table 1). Taking into account the effects of the number of hash buckets and task fetching, Equations 2 and 3 become:

\[ T_{sko}^{mo} = w_1 \left( B \log \frac{D}{B} + (P - D - 1) \log \frac{D}{B} \right) + w_2 B \log \frac{B(N + 1)}{2} + (w_3 + w_4) P + w_5 \left( \frac{N_1}{C} \right) \quad (7) \]

\[ T_{uko}^{mo} = w_1 \left( \frac{P}{D} \log \frac{D}{B} - \log \frac{D}{B} \right) + w_2 B \log \frac{B(N + 1)}{2} + (w_3 + w_4) P + w_5 \left( \frac{N_1}{C} \right) \quad (8) \]

### 4.4 Reduce Phase

The Reduce phase consists of reduce-comp and reduce-output sub-phases. In reduce-comp, each worker thread calls `reduce()`
Recall from Figure 2 that a map worker thread works on buckets across a row. In contrast, a reduce worker thread works on buckets across a column. Each column forms a reduce task, and reduce tasks are dynamically assigned to reduce worker threads. Each column consists of many buckets containing the output of map worker threads. With $B$ columns and $N_{redt}$ reduce threads, if load is balanced, each thread will be assigned $\frac{BN_{mapt}}{N_{redt}}$ columns to compute. Therefore, the total number of buckets assigned to a reduce worker thread is equal to the number of columns assigned to a reduce worker thread multiplied by the number of map threads, i.e., $\frac{BN_{mapt}}{N_{redt}}$. With dynamic task assignment, task fetching requires mutual exclusion, and the synchronization overhead scales with the number of tasks $B$ and the number of reduce threads $N_{redt}$.

Let us denote reduce-comp time for a key-value pair as $w_k$, and the time to fetch a task from the task queue as $w_\tau$. Assume key-value pairs are distributed evenly across all buckets, then each bucket will contain $\frac{P}{BN_{mapt}}$ key-value pairs. Therefore, the reduce-comp time is:

$$T_{rc} = \frac{P}{BN_{mapt}} (w_k + \frac{BN_{mapt}}{N_{redt}}) + w_\tau B$$

$$= w_k \frac{P}{N_{redt}} + w_\tau B$$

leading to the following observation:

**Observation 3.** The asymptotic complexity of the reduce-comp time is $O(\frac{P}{N_{redt}} + B)$. It increases with the number of intermediate output pairs and hash buckets, but decreases with the number of reduce worker threads.

**Significance:** Equation 9 and Observation 3 show that the relative contribution of $w_\tau$ on $T_{rc}$ increases when more buckets and more reduce threads are used. This creates a dilemma: on one hand increasing the number of buckets reduces map-output time, but on the other hand it increases reduce-comp time. Therefore, keeping the reduce task queue overhead low is very important as it enables the map phase to use a larger number of buckets and run faster.

In the reduce-output sub-phase, all values with a common key are aggregated. The final output, consisting of a list of key and aggregate value pairs, may or may not be sorted. Phoenix and Metis use variants of parallel merge sort algorithm which have a asymptotic complexity of $O(D \lg D)$, hence the reduce-output time is:

$$T_{rc} = w_k \frac{D \lg D}{N_{redt}}$$

where $w_k$ is the average time to perform one merge sort comparison and data movement.

### 5. MODEL-DRIVEN STUDY

In this section, we will use the model derived in the previous section to gain insights about how various factors affect shared memory MapReduce performance, first using asymptotic analysis and then quantitative analysis based on empirically-derived parameter values.

#### 5.1 Asymptotic Analysis

Table 2 shows the asymptotic time of various shared memory MapReduce phases for three cases: small number of buckets and equal number of keys and key-value pairs (Phoenix on first row, Metis on second row), small number of buckets and number of keys hugely exceeding number of key-value pairs (third row), or a large number of buckets (fourth row). For now, let us assume that $N_{mapt} = N_{redt}$. The table reveals several interesting insights. First, when the number of buckets is small, map-output time dominates the total execution time when $D = P$, because $O(D^2) > O(D \lg D) > O(P)$, and also when $D \ll P$ because $O(P \lg D) > O(D \lg D)$ and $O(P)$.

However, when the number of buckets is large, i.e. $B \approx D^2$, the opposite occurs. Reduce-comp now dominates execution time because $T_{rc} = O(P + D^2) \gg T_{map} = O(P)$, and also $T_{rc} = O(P + D^2) > T_{rc} = O(D \lg D)$. Only if the number of pairs is in the order of the square of the number of keys, i.e. $P = O(D^2)$, then the map-output time has the same asymptotic complexity as the reduce-comp time. Therefore, we can conclude that:

**Observation 4.** Increasing the number of hash buckets reduces map-output time complexity from $O(P \lg D)$ to $O(P)$, but increases reduce-comp time, to a point where the reduce-comp time may dominate execution time.

So far, our analysis has focused on asymptotic behavior, which is unable to take into account the effect of relative parameter values. Next, we will rely on quantitative analysis to determine which time components become dominant and which factors produce performance bottlenecks.

#### 5.2 Quantitative Analysis

##### 5.2.1 Map Time

Figure 4(a) shows map-output time for both SKO and UKO for Phoenix-based model as the number of keys increases along the $x$-axes, while the number of intermediate output key-value pairs stays fixed at 1 million. Both axes are shown in a log scale, and the curves are normalized to the map-output time for UKO. The curves are obtained from Equation 3 and 2 using empirically-derived parameter values: $w_1 = 110, w_2 = 2.95, w_3 = 29, w_4 = 47, w_5 = 2500$ (Section 7.1). Note that parameter values are specific to the platform we used, hence the value of quantitative analysis lies in the performance trends, not the exact performance numbers.

From the figure, we can observe that for both UKO and SKO cases, map-output time increases with a mild slope from 1 to 100,000 keys, then increases with a steep slope from 100,000 to 1 million keys. From our model, we know that the mild slope portion of the curve is when the map-output time is dominated by key searching, while the steep slope portion is when the map-output time is dominated by key insertion. These portions correspond to the two cases stated in Observation 1: one where $D$ is much smaller than $P$, and another where $D$ is approaching the value of $P$.

We also plot map-output time for Metis-derived model with empirically derived weights $w_1 = 175, w_2 = 15.90, w_3 = 29, w_4 = 47, w_5 = 2500$ (Figure 4(b)). How these parameter values are derived are discussed in Section 7.1. We notice that unlike the Phoenix model, Metis model’s map-output time increases with flatten slope as the number of keys increases. This is because...
Phoenix’s data structure has a logarithmic complexity for key insertion. Moreover, the difference in map-output time between UKO and SKO is proportional to $P - D$, as we observe in Observation 1.

Based on the observation that Metis model shows a slower increase in map-output time compared to Phoenix as the number of keys increases, Metis may seem the preferred choice. In Figure 4(c), we plot map-output time for both Metis and Phoenix under SKO, normalized to Phoenix’s map-output execution time. We notice that when the number of keys exceeds 10,000, Metis hugely outperforms Phoenix. However, the opposite is true when the number of keys is less than 10,000: Phoenix significantly outperforms Metis (magnitude not readily apparent on a log scale chart). A similar trend applies for UKO, but not shown due to space limitation. We should emphasize that what matters here is not the number of keys, but rather the ratio between number of keys and pairs. Phoenix outperforms Metis as long as $D << P$, and underperforms Metis when $\frac{D}{P} \to 1$. To summarize:

**Observation 5.** Comparing Phoenix and Metis, their choice of data structures produce the following performance difference: Phoenix’s map-output time is lower than Metis when $D << P$, but becomes higher as $\frac{D}{P} \to 1$.

**Significance:** The observation is interesting as it could not be derived using an asymptotic analysis, which would always favor Metis. The observation is also important for programmers and performance tuners because our analytical model can help in deciding which data structure to use given an algorithm with known number of keys and output pairs, even before writing code for the algorithm using different MapReduce runtime systems.

Figure 4(b) and (c) also show that initially SKO and UKO differ, but as the number of keys increases, they converge. Note, however, that we should not conclude that the difference between SKO and UKO is minor. In Figure 4(d), we plot the difference of map-output time between SKO and UKO, divided by the map-output time of UKO, as the number of keys is increased from 10 to 1 million. The figure shows that the performance gap between SKO and UKO is significant (> 10%) when the number of keys is up to 10,000, and can be as high as 60% for both Phoenix and Metis. This is consistent with Observation 2 which states that UKO and SKO performance difference is proportional to $w_1$ (average key search time) and to $P - D$. However, Metis’ performance gap is consistently higher compared to Phoenix, because of the higher performance weight factor associated with key searching.

Observation 5 is critical for performance because the number of keys and key occurrence order are algorithm-dependent characteristics and may not be changed in some cases. For example, linear regression will always have only five keys.

**5.2.2 Reduce Time**

Figure 4(e) shows reduce-comp time as a function of number of hash bucket columns (reduce tasks) for various numbers of reduce threads under Phoenix (similar trends and insights are observed for Metis but are not shown due to space limitation). The reduce-comp time is normalized to the single-threaded case with 1,000 reduce tasks. According to Equation 9, the reduce-comp time is a weighted sum of $\frac{P}{w_{map}}$ and $B$. We use $w_0 = 13$ and $P = 10^6$. At 1,000 buckets (leftmost points in the curves), the reduce-comp time is halved as the number of map threads doubles. As the number of hash buckets increases, the contribution of $\frac{P}{w_{map}}$ to reduce-comp time increases, hence all lines show an increasing trend. However, the slope is steeper for a higher number of threads because synchronization cost increases as more threads are involved (Section 7.1 shows how $w_{map}$ is affected by the number of threads), leading to the following observation:

**Observation 6.** While using more reduce worker threads decreases reduce-comp time, the benefit disappears and reverses as the number of reduce tasks (i.e., hash-buckets) increases. Thus, with a sufficiently high number of buckets, using fewer reduce worker threads yields better performance.

**Significance:** The observation opens up an avenue for tuning
MapReduce runtime systems: adapting the number of reduce threads and map threads differently (in prior studies they were always equal), and gives a guideline as to how many reduce worker threads should be used.

5.2.3 Comparing Map and Reduce Time

Figure 4(f) plots the map time, reduce time, and their sum, as the number of buckets varies. The numbers are normalized to the total execution time with 1K buckets. The figure shows that as the number of buckets increases, map time declines due to fewer hash bucket collisions, but reduce time increases due to reduce task queue contention. We observed the same trends when we use Metis model at different magnitude. This result was captured in Observation 4. Overall, in this section we have shown that a combination of asymptotic and quantitative analyses can explain key MapReduce performance phenomena and contrast two different implementations of MapReduce.

6. EXPERIMENTAL ENVIRONMENT

Machine Configurations. For empirical evaluation, we use Intel Xeon X5560 8-core processor with Linux kernel v2.6.31. This system has two chips with each chip having four cores. Each core has a 2.80GHz frequency, a 32KB private L1 cache, a private 256KB L2 cache, and an 8MB L3 cache shared by four cores. Each core also has two hardware contexts, but we do not use one of them for our experiments to avoid simultaneous multithreading effects. The system is lightly loaded for all the experiments. Only the application, regular OS processes and daemons, run. Each run was repeated ten times, the highest and lowest values were removed, and the average was taken and reported, to avoid measurement biases [17].

Applications. We use applications and inputs that come with Phoenix and Metis, which are already well-optimized by Phoenix developers [18, 23]. They include Histogram (HG) with a 400MB image input, KMeans (KM) with 20 clusters and one million points, Linear Regression (LR) with a 100MB input file, Matrix Multiply (MM) with a 8K × 8K dimension, String Matching (SM) with a 100MB input, and Word Count (WC) with 100MB input file (input C). We also generated two more inputs with the same number of words: input A (identical words) and B (all different words). We also constructed other inputs for WC for other purposes. For all the applications, the default number of hash buckets is 32K. The inputs are selected to be large enough to have a substantial execution time but without incurring much I/O activity.

7. EXPERIMENTAL EVALUATION

7.1 Microbenchmark and Validation Results

In this section, we validate the model we discussed in Section 4. All the equations in our model assume that \( w_1, w_2, \ldots, w_8 \) are parameters with constant values that give weights to various terms of the execution time. If the equations have captured all important variables correctly, then all the parameter values derived empirically should remain constant as the variables’ values vary. Figure 5 and 6 show empirically-derived parameter values in number of clock cycles, as all key variables are varied. The parameter values were derived from synthetic benchmarks, which are variants of the word-count application that we specifically designed for this purpose. We delimit the Map and Reduce phases with hardware counter collection, so that we can measure their execution time separately.

We will first discuss the methodology to infer the weight factor (parameter) values for Map phase, and the Reduce phase next. For \( w_4 \) (weight factor for hashing), we take out the hashing function and take the total execution time for hashing various number keys, and divide it by the number of pairs to get \( w_4 \). We plot \( w_4 \) values separately for the same key and for unique keys.

Then we set the number of hash buckets to one, and remove the hashing function for keys in order to exclude its effect when measuring other parameter values. For \( w_3 \) (weight factor for value insertion), we emit 10 to 1 million pairs having the same key, and divide the total time by the number of pairs to get \( w_3 \).

To estimate \( w_1 \) (weight factor for key search) for a given number of keys \( D \), we first insert \( D \) keys and subsequently search for 1 million randomly-generated keys from among already-inserted keys. The total time is subtracted by the value insertion time and then divided by \( 10^7 \log D \). The value of \( w_1 \) is not affected by key insertion as it is mostly excluded and amortized over a large number of key searches. We measure \( w_2 \) (weight for key insertion) by inserting a varying number of keys and subtracting the time for key search and value insertion, and then dividing it by the number of keys (for Phoenix implementation) or \( \log \) of keys (for Metis implementation). Finally, \( w_5 \) (weight factor for map-task queue overhead) is estimated by creating 10 to 10^5 map tasks with NULL map function and executing them with 1–8 threads.

For \( w_5 \) (reduce task queue overhead factor), we vary the number of buckets from 1K to 128K, but leave the buckets empty (zero pairs and keys). This removes reduce-output (no output produced) and value aggregation part of the reduce phase. For \( w_6 \) (weight factor for aggregating values), the number of map and reduce worker thread is set to one, the number of keys and hash bucket to one, but the number of pairs is varied. In one case we schedule map and reduce threads on same core and in another case we schedule them on different cores to measure the effect of cache coherence (transferring values across caches) on value aggregation. \( w_6 \) is obtained by keeping the number of hash buckets fixed and varying the number of threads and keys. The resulting execution time is subtracted by the reduce-comp component and then divided by \( D \log D \).

Figure 5 and 6 show all weight factors for Phoenix and Metis. The values of the parameter weight factors reveal the relative importance of performance factors. We can make several interesting observations. First, for both Phoenix and Metis, \( w_1 > w_4 > w_3 > w_2 \), suggesting that a single instance of key searching is the most expensive, followed by a single instance of key hashing, value insertion, and key insertion. This is likely because of value insertion and key insertion enjoy better spatial cache locality than key searching. Key searching jumps over different elements, while key insertion only involves shifting adjacent elements, and value insertion involves adding adjacent to previously accessed elements. Second, since hashing is more expensive than value insertion, it suggests that hashing should be of higher priority for optimization than value insertion. Finally, \( w_5 \) has a large value but is multiplied by a small value (there are only 1K map tasks), hence it is not a bottleneck during the map phase.

Now, we discuss the difference between Phoenix and Metis for a given weight factor (Figure 5). Phoenix and Metis show similar weight factors for hashing, value insertion and multi-threading overhead, on our test platform. The similarity may extend to other platforms because of similar map task queue design, array data structure for storing values, and an identical hash function. However, key search and insertion are vastly different because of the choice of data structures. Phoenix uses a binary search tree implemented as a sorted array, resulting in low key search time and good key insert cache locality, at a cost of \( O(D) \) key insertion time. In contrast, Metis uses B+Tree which provides \( O(\log D) \) for both key search and key insertion, but suffers from poor cache locality from
pointer dereferencing while searching for keys and splitting buckets to balance the tree. Consequently, \( w_1 \) and \( w_2 \) for Phoenix are smaller than Metis, and Phoenix outperforms Metis until the number of keys is large enough to offset the effect of lower \( w_1 \) and \( w_2 \).

Next, we discuss weight factor for the Reduce Phase components. \( w_6 \) (weight factor for value aggregation) is low for both Metis and Phoenix, even when values must be fetched from other caches. The reason is that value aggregation inherently has good spatial locality if values are kept in an array, enabling the cache’s sequential prefetchers to be effective in bringing data early into the cache. This argues against implementing the value list using data structures that reduce spatial locality, such as linked lists. Second, \( w_5 \) is lower than \( w_4 \) but multiplied with the same variable \( P \). This suggests that improving hashing function performance is more important compared to improving value aggregation performance.

The figure shows that \( w_7 \) (weight factor for bucket overhead) is roughly constant but its value is specific to the number of threads. Since the relationship of bucket overhead and number of threads is affected by many factors (coherence misses, synchronization implementation, etc.), and is difficult to express mathematically, we simply use different \( w_7 \)'s for different numbers of threads.

We also notice that \( w_7 \) in Phoenix is significantly larger than in Metis. Analyzing and comparing Phoenix and Metis code, we found that Phoenix uses a traditional task queue implementation where each thread competes for the head of the queue using a lock. In contrast, Metis uses atomic update of a counter variable that mimics the behavior of a task queue, incurring less synchronization and fewer dynamic instructions. This difference in performance emphasizes that task queue synchronization should be avoided as much as possible.

So far, we have shown that our model capture key variables affecting performance of the Map and Reduce phases. However, one limitation of our model is that it does not model the reduce-output time very well. Instead of showing a constant value, \( w_8 \) is affected by the number of keys and the number of threads. \( w_8 \) in Metis is lower than in Phoenix because Metis uses sampling-based parallel merge sort which utilizes all cores better compared to Phoenix’s regular parallel merge sort. The higher \( w_8 \) value on a larger number of threads indicates sub-linear speedup (Metis), while a lower value on a larger number of threads indicates super-linear speedup (Phoenix). However, reduce-output time may be an optional step and is important only when number of keys is large enough such that reduce-output time becomes larger than reduce-comp time. Although our reduce-output time modeling is less accurate, we can still use it to predict the trends correctly.
7.2 Empirical Results for Real Applications

In this section, we present experimental results from running real applications using Phoenix and Metis libraries on the test platform described in Section 6. We will demonstrate the role of our model in: (1) predicting performance trends in real applications, and (2) explaining the reasons behind such trends.

7.2.1 Map Phase Performance Results

Figure 8(a) shows the SKO map-output time for Phoenix and Metis, normalized to Phoenix with 25K keys. The input to WC application is a text document containing 20 million words with varying number of distinct keys. There are two important observations. First, the figure shows that Phoenix’s map-output time increases with increasing slope with the number of keys, while Metis’s map-output time increases with decreasing slope with the number of keys. Second, Phoenix’s map-output time is lower than Metis until the number of keys reaches 125K. These two observations are both predicted (Figure 4(c)) and explained (Observation 5) by our model. They are an outcome of the asymptotic key search insertions and the choice of data structure (array vs. B+tree) for the hash buckets. Knowing these helps programmers in choosing the appropriate MapReduce library early in their project.

Next, we consider the effect of UKO vs. SKO. Figure 8(a) and (b) show the map-output time for Phoenix and Metis, respectively. The figure confirms the model’s prediction that key ordering is a significant performance factor, with UKO significantly outperforming SKO (Observation 2 and Figure 4(a) and (b)). Furthermore, the performance gap between UKO and SKO is higher in Phoenix than in Metis, again consistent with our model (Figure 4(d)).

Next, we discuss the effect of the number of hash buckets and threads on the map phase. Figure 9(a) and (b) show map-output time with varying number of buckets (shown across groups of bars as indicated by numbers on the x-axes) and number of map threads (shown across different bars in a group). For both Phoenix and Metis, all the bars are normalized correspondingly to a case where one hash bucket is used. Let us first focus on the first bar of each group, corresponding to single-thread execution (Thr 1). We can
Reduce Phase Execution Time

Figure 10: Reduce phase execution time for various number of threads and hash buckets Phoenix (a), and Metis (b)

see that increasing the number of hash buckets from one to 32K decrease the map-output time for most benchmarks (HG, KM, LR, WC-B, and WC-C). Only MM and WC-A do not see any improvement, because MM completely bypasses key insertion, while WC-A only has one unique key. On another extreme, WC-B shows more than 100× improvement from using 32K buckets compared to 1 bucket in Phoenix, hence the bars are not visible in the figure. For the remaining benchmarks, the map-output time improvement of using 32K buckets vary. Our model explains that the improvement from reduced hash collisions stops when the number of buckets is significantly larger than the number of distinct keys (e.g. $B \geq D^2$).

Next, let us compare Thr1 bars across different groups representing different number of hash buckets. WC-C (in Phoenix and Metis) and WC-B (in Metis) are the only benchmarks that enjoy map-output time reduction as the number of hash buckets increases from 32K to 128K. Again, the reason is explained by our model: these applications are the ones that have thousands to millions of distinct keys, hence the increase in number of hash buckets continues to improve performance.

Finally, let us compare bars in each group, representing the effect of increasing the number of map threads. It is straightforward to observe that increasing the number of threads reduces map-output time inverse-proportionally as the computation load is distributed across more threads.

### 7.2.2 Reduce Phase Performance Results

Now, we present experimental results for reduce phase. Figure 10(a) and (b) show the reduce phase execution time as the number of hash buckets and threads vary, normalized to a single thread case with 32K hash buckets. There are several major observations. First, comparing across groups of bars, increasing the number of hash buckets from 32K to 128K increases reduce time (notably for HG, LR, KM, and SM). This phenomena is predicted and explained in Observation 3, which shows how the number of hash buckets (multiplied by weight factor $w_i$) directly influences the reduce time. The reason is due to synchronization overheads for assigning reduce tasks to reduce worker threads. Second, let us compare bars within each group, which represent the effect of increasing number of reduce worker threads. There are two opposite outcomes from using a larger number of reduce threads: increased reduce time for some benchmarks (HG, LR, KM, and SM) or decreased reduce time for others (WC-B and WC-C). For some benchmark (HG, LR, KM, and WC-A), both outcomes appear, for example going from one to two threads in HG decreases reduce time by 12%, but going from two to four/eight threads the reduce time increases. We note that this performance trend is captured and explained by our model (Observation 6), which states that using more reduce worker threads allow more parallelism in aggregating values (during reduce-comp) and in sorting the final output (during reduce-output) but at the same time incurs a higher task queue overhead (during reduce-comp), especially when the number of hash buckets is high. This again points out to the need to choose the number of reduce worker threads and map worker threads differently (also taking into account the number of hash buckets), unlike prior studies which always kept them equal.

Finally, WC-B and WC-C enjoy a significant improvement from increased number of threads. A key reason for this is in the output merging in reduce-output phase, the sorting algorithm benefits from a larger number of threads. This is evident in the weight factor $w_i$ that decreases with the increase in number of threads (Figure 6(c) and (f)). However, due to the non-linear nature of $w_i$, we cannot capture the exact improvement precisely in our model, but knowing how $w_i$ responds to number of threads in Phoenix and Metis still helps in successfully predicting this trend.

### 7.2.3 Comparing Map and Reduce Phase

We also investigate how the number of hash buckets affects both Map and Reduce phases simultaneously. Figure 11 shows map vs. reduce time as fractions of the total execution time for Phoenix (top) and Metis (bottom).
and Metis respectively. The figure shows that the fraction reduce time increases with the increase in the number of hash buckets for many applications. Consistent with Observation 4, increasing hash buckets has the opposite effect on map time (reducing it) due to fewer hash collisions and faster key searching, versus on reduce time (increasing it) due to a higher task queue overhead. And for some benchmarks, reduce time may dominate the execution time.

We point out that the fraction of reduce time in Phoenix is larger than in Metis (e.g. HG and LR). This is due to the reduce task queue overhead factor (w_r) being higher in Phoenix than in Metis (Figure 6). However, Phoenix’s task queue structure is more generic and may be more scalable.

8. APPLICATION CLASSIFICATION

We have demonstrated that the most important performance factor for MapReduce performance is the number of intermediate key-value pairs and number of distinct keys. Interestingly, both metrics are specific to the algorithm and input. Consequently, they are already known to programmers prior to coding. Figure 12 shows a map of where applications in the Phoenix package fall (not drawn to scale). The arrows show the range of parameter values for various applications.

![Figure 12: Classification of MapReduce Applications. Not drawn to scale.](image)

At the bottom right corner are applications with a relatively large number of distinct keys but with a relatively small frequency of distinct keys. These applications will have key insertion dominating the map-output time (Observation 1) as well as a high Reduce time (Observation 3). This category includes word count (WC-B), matrix multiplication (MM), and string matching (SM). Effective tuning for these applications includes not sorting the final output, or skipping the Reduce phase altogether (e.g. in Phoenix, MM directly writes to a matrix output). Applications without natural keys can use such tuning techniques.

At the top left corner are applications with a small number of distinct keys but a high frequency of distinct keys, including word count (WC-A) and linear regression (LR). These applications’ performance will be dominated by key searching in the Map phase (Observation 1), while the Reduce phase will not be a bottleneck. Such applications should use a relatively small number of hash buckets which keeps reduce time low (Observation 3) and a small number of reduce worker threads (Observation 6) which produces similar effects.

At the top right corner are applications with a high number of keys and high frequency of distinct keys, which include HG and WC with certain inputs. For such applications, all components of execution time are stressed, hence the number of buckets should be fine tuned to keep both map and reduce time low (Observation 4 and 3).

9. CONCLUSION

We discussed a new analytical model that identifies key performance factors for a shared memory MapReduce system, and reveals how they affect performance and scalability. We demonstrated how the model can explain performance phenomena that MapReduce exhibits and predict likely performance bottlenecks, in some cases prior to coding. We have summarized several important and interesting findings, such as how the number of intermediate output pairs, number of keys, number of hash buckets, number of map and reduce threads, affect performance. We have also showed various components of MapReduce execution time, and under what circumstances each component becomes a performance bottleneck.

We have also developed an application classification framework that helps in predicting likely performance bottlenecks and methods to address them, for a given application.

10. REFERENCES