Breaking the MapReduce Stage Barrier*

Abhishek Verma, Nicolas Zea, Brian Cho, Indranil Gupta, Roy H. Campbell
{verma7, nzea2, bcho2, indy, rhc} @ illinois.edu
University of Illinois at Urbana-Champaign

ABSTRACT

The MapReduce model uses a barrier between the Map and Reduce stages. This provides simplicity in both programming and implementation. However, in many situations, this barrier hurts performance because it is overly restrictive. Thus, we develop a method to *break* the barrier in MapReduce in a way that *improves efficiency*. Careful design of our barrier-less MapReduce framework results in equivalent generality and retains ease of programming. We motivate our case with, and experimentally study our barrier-less techniques in, a wide variety of MapReduce applications divided into seven classes. Our experiments show that our approach can achieve better performance times than a traditional MapReduce framework. We achieve a reduction in job completion times that is 25% on average and 87% in the best case.

1. INTRODUCTION

Inspired by the map and $reduce$ primitives present in functional languages, Google proposed MapReduce [9]. The MapReduce framework simplifies the development of large-scale distributed applications on clusters of commodity machines. It has become widely popular, e.g., Google uses it internally to process more than 20 PB per day [10]. Yahoo!, Facebook and others use Hadoop, an open-source implementation of MapReduce [1].

The MapReduce model has become popular because a programmer can harness the processing power of large data centers for very large parallel tasks in a simple way. The programmer only needs to write the logic of a Map function and Reduce function. This eliminates the need to implement fault-tolerance and low-level memory management in the program; the MapReduce framework takes care of these concerns for general programs.

The execution of a MapReduce program across a datacenter is illustrated in Figure 1. The framework itself divides the program execution into a Map and Reduce stage, separated by the transfer of data between machines in the cluster. In the first stage, each machine in the cluster executes a Map function on a distinct region of the input data. The Map execution produces records that consist of a key and value. Each record is stored on the local machine it was created on. The records for any given key, which are spread out on many machines, are aggregated at each Reducer for the Reduce stage. This involves a remote data transfer between the machines in the cluster.

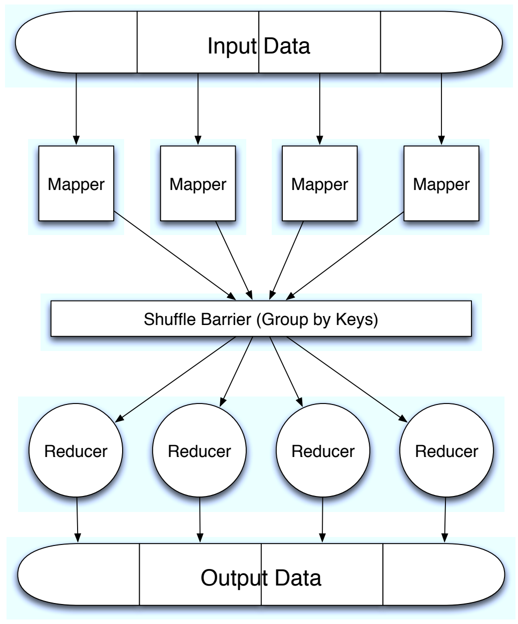


Figure 1: Illustration of MapReduce stage barrier.

In current implementations of MapReduce, the two stages are separated by a barrier. This prevents the Reduce stage from progressing until all the data from the Map stage has been remotely transferred to the appropriate machine. The barrier ensures that all relevant input data is available to the Reduce function before it proceeds.

In this paper, we break the barrier between stages in MapReduce. The result is a *barrier-less* version of MapReduce, which can have significantly improved performance. At the same time, we take special care to maintain the simplicity and generality of the MapReduce framework. To this end, we investigate a broad set of categories of MapReduce programs, differing in the structure and the memory usage of the Reduce function. Based on these observations, we develop memory management techniques that are general and yet require minimal additional effort by the MapReduce programmer.

9*This paper was supported in part by NSF grant IIS 0841765

Our main contributions are as follows:

1. We present techniques for supporting general purpose applications in a barrier-less MapReduce framework.
2. For seven different categories of MapReduce algorithms, we show how they can be converted to their barrier-less forms.
3. We identify and address the memory management concerns that arise from removing the barrier.
4. We experimentally evaluate the benefit of converting algorithms to their barrier-less version. Our results show an average improvement of 25% (and 87% in the best case) in the job completion times, with minimal additional programmer effort.

In this paper, for concreteness, we focus on the Hadoop framework because of its open source nature. However, our contributions are not limited to this particular instance. The technique of using write-local read-remote data transfer with a stage barrier is also used in Google's MapReduce, as well as related parallel processing frameworks such as Dryad [16]. Barrier-less implementations of these frameworks using our techniques should be able to reap benefits similar to the ones in this paper.

The rest of the paper is structured as follows. We first examine the role of the MapReduce barrier in Section 2. We then discuss our design for breaking the barrier in Section 3. We observe that this improves performance in many cases, but can present a memory management problem. In Section 4, we investigate the memory usage patterns of MapReduce applications, and produce a categorization based on the structure of the Reduce function. Based on these observations, in Section 5 we develop new techniques that are able to manage memory for general applications while breaking the barrier. In Section 6, we present experimental results that show a significant improvement in Hadoop performance when these techniques are applied. We then discuss related work in Section 7 and finally conclude in Section 8.

2. MOTIVATION: MAPREDUCE BARRIER

The execution of a MapReduce program is divided into a Map stage and a Reduce stage, as we illustrated in Figure 1. The MapReduce framework writes the Map output locally at each machine and then aggregates the relevant records at each Reducer by remotely reading from the Mappers. This process of transferring data is called the Shuffle stage.

In current open source MapReduce implementations (i.e., Hadoop), the Shuffle stage contains a distributed barrier. Figure 2 shows a step-by-step illustration of the Shuffle process from the perspective of a single Reducer. In this example, the Reducer works on records with three different keys. Each record is represented as a shape indicating its key. First, the Reducer reads the relevant records from many Map nodes. These entries are not in a sorted order, and are buffered at the Reducer. The barrier is reached when the Reducer has received all Map output. The Reducer then sorts the buffered entries, effectively grouping them together by key. Finally, the Reduce function is applied to each group of entries with the same key, one by one.

The barrier is useful for several reasons, most prominent being to provide simplicity and efficiency by allowing the Reduce function to atomically operate on all records for a particular key. This in turn means that once a key is processed all partial results for that key can be disposed of and the output may be written. For example, Figure 2(d) shows a snapshot of the Reduce function being executed,

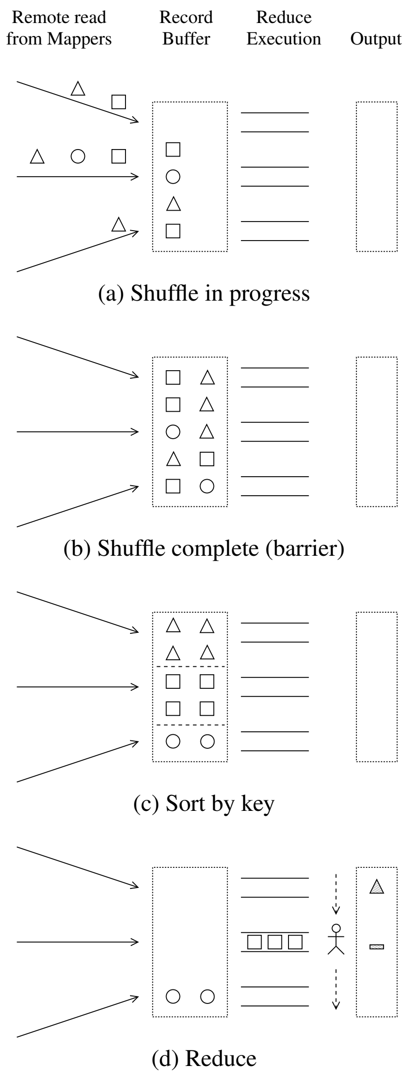


Figure 2: MapReduce Shuffle and Reduce stages as seen at a Reduce node. Each record is shown as a shape representing its key. This shape is also used to show the key for the Reduce output. Partial output is represented with a small portion of the shape.

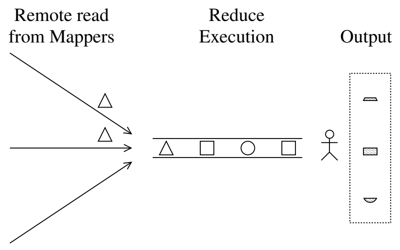


Figure 3: Barrier-less Reduce stage in-progress. Each record and output is shown as a shape, representing its key. Partial output is represented with a small portion of the shape.

where records of the square key are currently being processed. The triangle results have already been output, and therefore do not need to be maintained. The circle records have not yet been operated on, so there are no results to be maintained. Only the square results need to be maintained at this point.

However, despite these apparent advantages, we argue that removing the barrier is, in many practical cases, much more efficient. Figure 3 illustrates what the Reduce stage may look like in a barrier-less execution. The Reduce operation no longer needs to wait for all records to be remotely read and grouped by key (steps (a)-(c) in Figure 2). Instead, the Reduce operation can be immediately invoked on each input entry, as it becomes available. This relaxation can significantly improve the efficiency of the Reduce function execution.

More concretely, by removing the barrier, we eschew two waiting intervals before the Reduce operations are executed: (1) the time interval between remote read of the first and last records, and (2) the time taken for sorting the records. The time it takes to read all records depends on the relative speed between the Mapper nodes and the speed of data transfer from the Mapper nodes to Reducer nodes. Datacenters with commodity hardware often show differences in performance between machines, and they have oversubscribed links between machines. This can further extend the first interval of waiting. The barrier-less model removes these intervals, thus improving performance.

By removing both waiting intervals, the records in the barrier-less model are no longer sorted in key order. Our investigation of seven classes of MapReduce applications summarized in Table 1 (detailed in Section 4) reveals that, in practice, a significant number of applications do not require the full key sorting provided by the MapReduce framework. The main role of sorting by key is to group records with the same key together. Grouping is necessary in traditional MapReduce, because it requires all records for a key to be present when the Reduce function is executed. The barrier-less approach removes this requirement, and the Reduce function is run on records one by one. This approach raises an important problem: *partial results* for each key must be maintained.

Our investigation shows that the number of partial results that must be maintained differs widely across MapReduce applications (see column "Size of partial results" in Table 1). Thus, for the barrier-less model to work with general MapReduce applications, we require techniques for maintaining and updating these partial results. Before we develop these techniques in detail (see Section 5), we first describe the structure and usage of our basic framework for barrier-less MapReduce.

3. BREAKING THE BARRIER

In this section, we describe our implementation of barrier-less MapReduce, and illustrate how to modify an existing MapReduce application to be used in this framework.

3.1 Barrier-less Hadoop Implementation

We implemented barrier-less MapReduce by modifying the open-source Hadoop implementation. The original Hadoop implementation employs a barrier as described in the previous section. In order to break the barrier, we had to incorporate two primary design decisions: (1) bypass the sorting mechanism, and (2) modify the invocation of the Reduce function so that it can be called with a single record (instead of a key and all values corresponding to it).

Hadoop's Shuffle stage is implemented to work in an efficient and asynchronous manner, since it is critical to exit the stage as fast as possible. To avoid I/O interference at the Reducer, Hadoop designates an asynchronous thread and local buffer for each Mapper.

Application (Reduce Classification)	Key sort required	Size of partial results
Distributed Grep (Identity)	No	$O(1)$
Sort (Sorting)	Yes	$O(records)$
Word Count (Aggregation)	No	$O(keys)$
k-Nearest Neighbors (Selection)	No	$O(k * keys)$
Last.fm unique listens (Post-reduction processing)	No	$O(records)$
Genetic Algorithms (Cross-key operations)	No	$O(window\_size)$
Black Scholes (Single Reducer Aggregation)	No	$O(1)$

Table 1: Sort and Memory requirements of MapReduce Jobs. $Records$ and $keys$ denote respectively, the total number of records and keys, executed at a single Reducer.

Algorithm 1 Original WordCount

```
function map(key, value):
// key: document name
// value: document contents
for each word in value do
    Emit intermediate (word, 1)
end for

function reduce(key, values, context):
// key: a word
// values: a list of counts
result ← 0
for each v in values do
    result ← result + v
end for
Write (key, result) to context

function run():
while context has more keys do
    key ← current key from context
    values ← current values from context
    reduce(key, values, context)
end while
```

The thread polls its Mappers for new records, and when the records are available fetches them to the local buffer. When all threads have retrieved their data from their Mappers, the barrier has been reached and the local buffers are merge-sorted. Finally, each key and all its values are passed into the Reduce function, one by one.

For barrier-less MapReduce, the remote read of records and the Reduce execution on these records are pipelined. Therefore, the impact of I/O interference is minimized. In our barrier-less Hadoop implementation, we use one asynchronous thread per Mapper as in the original Hadoop, but retrieve the records into a single buffer instead. A separate thread executes the Reduce function on the records in the buffer in a first-in first-out manner. The Reduce function called in this manner is only passed a single record, as opposed to a key and all its corresponding values in the original Hadoop. This subtle difference in the framework compared to original Hadoop changes the way applications are implemented slightly, as we show with an example in Section 3.2.

However, these changes do not affect other aspects of the execution of Hadoop. In other words, assignment of tasks, fault-

Algorithm 2 Barrier-less WordCount
Changes made to Algorithm 1 are ***boldfaced and italicized***.

```
function map(key, value):
// key: document name
// value: document contents
for each word in value do
    Emit intermediate(word, 1)
end for

function reduce(key, values, context):
// key: a word
// values: a list of counts
result ← 0
for each v in values do
    result ← result + v
end for
Insert (key, result) in the TreeMap

function run():
Create a new TreeMap
while context has more keys do
    key ← current key from context
    values ← current values from context
    if TreeMap does not contain key then
        Insert (key, 0) in the TreeMap
    end if
    reduce(key, values, context)
end while
// After all the reduce invocations are done
for each (key, value) in TreeMap do
    Write (key, value) to context
end for
```

tolerance, scheduling, etc., are handled in the same way as original Hadoop.

3.2 Barrier-less WordCount

As previously mentioned, when executed with barrier-less Hadoop, the Reduce function does not have the guarantee of atomically receiving all records for a given key. Therefore, the application must be modified to handle records one by one. To do this, a programmer must code, in addition to the Map and Reduce function, a custom *run function*. In the original Hadoop, the run function invokes the Reduce function once per key. For barrier-less Hadoop, the programmer additionally specifies in this function how partial results are stored and reused across Reduce invocations.

In the rest of this section, we present a concrete example of the difference between an application coded for the original and barrier-less MapReduce frameworks. For this, we use the WordCount application provided with the Hadoop distribution. The original program is shown in pseudo-code in Algorithm 1. In the Map function, each output entry is simply a word and a count of 1. In the Reduce function, the number of output entries with the same key is counted. The run function, which is part of the Hadoop framework, ensures that the Reduce function is called once for each key, with all the values as input.

To run WordCount without a barrier, the programmer has to modify the run and Reduce functions as presented in Algorithm 2. The run function calls Reduce on each entry that is received. In other words, *the Reduce function no longer assumes that all values for a key are passed in at once*. This means that the Reducer must maintain partial results for every key it has received. For our purposes, we use the Java implementation of Red-Black trees [15] called TreeMap. A TreeMap can quickly access partial results while maintaining key ordering. As a record (which is a key/value pair) arrives, the run function reads the previous partial result, and passes it to the Reduce function. The Reduce function performs the computation, and stores the new result back into place. Once there are no more records and all the Reduce invocations have completed, the output is generated by the run function.

Figure 4 shows the system-wide progress of the WordCount program with and without a barrier on the same datacenter. (Details of the experimental setup are provided in Section 6.) The y-axis represents the number of CPU cores executing at each stage. In the original MapReduce, we can see the barrier in the delay between the Map tasks finishing at 155 seconds and the Reduce tasks beginning at 170 seconds. In the barrier-less version though, the combined Shuffle and Reduce stage begins at 50 seconds, when the first mappers begin to complete. We refer to the time gap between when the first mappers complete and when the shuffle stage completes as the *mapper slack*. It is indicative of the extra time taken by the buffering and sorting parts of the Shuffle stage (see Figure 1).

In barrier-less MapReduce, there is no distinct barrier between Shuffle and Reduce. Instead, each Reducer works on individual records as the Shuffle process pulls them in. Because these two stages are combined, we see an improvement in job completion time. We observe that the job finishes within 160 seconds, or only 10 seconds after the final Map task completes. This is a 30% improvement in the job completion time for WordCount. This benefit arises because we can perform meaningful work in the form of Reduce operations during the *mapper slack* time, in which the barrier version is performing the shuffle/sort operation. At the same time, since our modifications were idempotent, the correctness and the completeness of the MapReduce execution is not compromised.

Finally, we observe that depending on the application, the amount of memory consumed at each Reducer by partial results may vary. In the worst case the number of partial results may become very large and cause the Reducer to run out of memory. This motivates the development of new memory management techniques that can prevent overflows. We address this in Section 5.

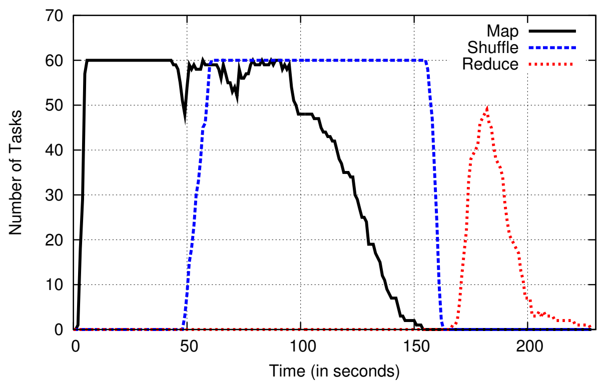
4. CLASSIFYING REDUCE OPERATIONS

In order to understand the general implications of breaking the barrier in the general case, we need to understand concrete MapReduce applications. Hence, we performed a case study of a wide variety of published MapReduce applications and investigated how to break the barrier for each of them. The applications we studied were the following: MapReduce example benchmarks [9]; machine learning benchmarks [7]; statistical machine translation [6, 11]; optimization algorithms [20]; finance algorithms[5]; and similarity scoring [12].

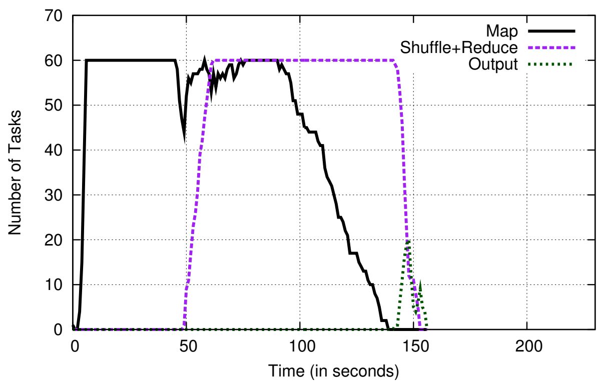
We classified these applications based on the type of Reduce operations they perform. The result is a list of seven types: Identity, Sorting, Aggregation, Selection, Post-reduction processing, Cross-key operations and Single reducer aggregations. This information is summarized in Table 1. In the rest of this section, we present our classification. For each type, we discuss a representative application and how partial results must be stored and updated during execution in barrier-less MapReduce. Note that applications may combine one or more Reduce operations.

4.1 Identity

Identity operations are Reduce operations that perform little to no explicit work. An example of an Identity operation is a Distributed Grep application [9]. The Map function emits a line of text if it matches a pattern. The Reduce function is merely used to write



(a) With barrier



(b) Without barrier

Figure 4: Progress of MapReduce which performs a word count on a 3GB Wikipedia data set

the final output.

Identity operations are the simplest kind of Reduce operation. They do not require the Reduce input to be sorted by key. There is also no need to keep partial results for any keys, because the results are written immediately as final output. Hence, there is no difference between implementing this operation for original and for barrier-less MapReduce.

4.2 Sorting

This is the only prominent kind of operation we found that requires a strict ordering on the output keys. For sorting operations, the Reduce operation must write output that is in a sorted order. This is a popular application, e.g., a sort implemented in Hadoop holds the record for the fastest sort of 100TB of data [18].

The implementation of a sorting operation is dependent on whether or not a barrier is present. With a barrier, the implementation of a sorting operation is identical to an Identity operation. The MapReduce framework itself, rather than the Reduce operation, does the job of sorting the output by key. If sorting by value is also required, a secondary sort operation is easily performed using custom grouping and comparison operations. However, this is not the case when the barrier is broken.

Effect of no barrier: To implement a sorting operation without a barrier, the data must be sorted in the Reduce function, typically through the use of an ordered data structure like a Red-Black tree. None of the partial results can be emitted until all the values have been seen and completely sorted. Thus, in the worst case each Reducer must maintain a data structure of size $O(records)$, the total number of records executed at the Reducer.

4.3 Aggregation

Aggregation operations are commutative and distributive operations – they include addition and multiplication. They perform an operation on all the values associated with a key, and emit an aggregated value as output. Since the operations are commutative, the ordering of the keys is not required.

An example is the WordCount application from Section 3. For each key, the entries that contain the count of the key are summed up into the aggregate word count. In the original version, the Reduce function is invoked with a key and all of its associated values. Hence, it can aggregate them and emit the final count immediately.

Effect of no barrier: For the barrier-less version, a running aggregate result must be maintained for each key. Thus, the Reducers must maintain $O(keys)$ state for storing the partial results. The Reducer outputs the results only when all the keys and their associated values have been processed.

4.4 Selection

Selection operations are those that select a subset of the values associated with a key. Examples include finding the max, min, median, or top k values.

With a barrier present, implementations typically take advantage of key/value ordering by doing a secondary sort, using the desired metric for ordering. This allows the Reducer to finish after having processed only those values scoring highest (or lowest) on the metric. For example, when finding a minimum, the metric will be distance, and thus the Reducer can finish after processing the first value for every key.

Without a barrier, a sort operation would remove the benefit of being barrier-less, so these operations must be performed on a *running* basis. For example, a running minimum (or minimum k values) is kept and updated as new values arrive. Therefore, the barrier-less version of a selection operation maintains a per-key context with the currently selected values, and emits the final output once all values have been received.

To investigate selection algorithms, we implemented a $k-$Nearest Neighbors algorithm. This is a classic algorithm that reads in two sets of data, a training set and an experimental set, and finds the k values in the training set closest to each value in the experimental set. It was first presented in [14] and is often used in statistical analysis applications, such as finding pairwise similarity [12].

The distance between an experimental value (exp_value) and the training value ($train\_val$) is defined as the absolute value of their difference. It is necessary to compare each experimental value to every training value. The barrier version's Map function emits a tuple $(exp\_value, distance)$ for the key, and an integer $train\_value$ for the value. A secondary sort is performed, sorting by the $distance$ value in the key, but grouping by exp_value. Then, in the Reducer, the first k values are emitted.

Effect of no barrier: The barrier-less version maintains a k-value-per-key context, stored as a TreeMap (a Red-Black tree implementation in Java) of linked lists. The Mapper emits an integer exp_value as the key and a tuple $(train\_value, distance)$ as the value. This is done because no secondary sort is being performed, so there is no need to emit a tuple as the key. Now, for each key, the Reducer maintains a size-k ordered linked list, and decides if the most recently received $(exp\_value, distance)$ tuple belongs in the list, based on the $distance$ value within the tuple. If this is the case, it is inserted into the appropriate location within the ordered linked list, evicting the tuple with the largest distance if the linked list size exceeds k. Once all value tuples have been processed for a key, the contents of the linked list (namely the top k exp_value's), are emitted.

4.5 Post-reduction processing

In post-reducer processing operations, the Reduce operation works in two steps. First, the entries with a key are processed and inserted into a temporary data structure. When all the entries for a key have been processed, a post-processing operation is done on the temporary data structure to get the final output for the key.

An example application is the one used at Last.fm to track the number of unique users that listen to each track of music in the service [21]. Entries of the input data consists of a userId and trackId (and other information). The trackId is the key of the record. The number of unique users per track is counted in two steps. In the processing stage, the userId of each record is added into a data structure that does not hold duplicate values e.g., the code presented in [21] uses a Java Set. Then the post-processing step counts the total number of entries in the data structure.

Effect of no barrier: With a barrier, the temporary data structure will grow with the maximum number of records with a certain key. This in itself could be a large amount of data. However, when the barrier is broken, the structure can grow even larger. The temporary data structure for each key must be maintained, in a partial result structure such as a TreeMap. The total amount of partial results can grow to $O(records)$.

4.6 Cross-key operations

Typically a Reduce function processes its keys independent of the other Reduce functions. However, in cross key operations, the Reduce function can depend on other keys, for example the previous k keys. This can be implemented by maintaining a window of k previously seen keys, operating over them and emitting the final output. Since Reduce does not depend on other keys, it can terminate after emitting its output.

To investigate cross-key operations, we use the example of genetic algorithms; in particular we use [20]. Each individual is represented as a key and the map computes the fitness of each individual and emits the tuple $(individual, fitness)$. The Reducer maintains a window of previously seen individuals and when the window is full, performs the selection and crossover operations of the genetic algorithm and finally emits the individuals as output.

Effect of no barrier: Only partial results for the window containing the previous k keys need to be maintained. When a partial result is removed from the window, it is written as a final result. Thus, the memory requirement for storing partial results is $O(k)$.

4.7 Single reducer aggregation

Single reducer aggregations involve the use of a single Reducer to aggregate the outputs from multiple mappers. This is generally used for determining measures of central tendency or dispersion where global knowledge of all the map outputs is required.

We study single reducer aggregations through a Monte Carlo simulation that computes the Black-Scholes option pricing value [5, 13]. Each mapper performs complex floating point operations like exponentiation according to the Black-Scholes formula and the Reducer computes the average and standard deviation of all the values computed by the mappers.

Effect of no barrier: The average operation can be incrementally computed by maintaining a running sum of the values and performing a division at the end. In order to calculate the standard deviation along with the average, the mapper emits the square of the value along with the value itself. The Reducer maintains a running sum of the squares of the values along with a running sum and a count of the values. Let $x_1, x_2, \ldots, x_N$ be the values whose mean is $\bar{x}$. The standard deviation is computed as follows:

$$\begin{aligned}
\sigma &= \sqrt{\frac{1}{N}\left(\sum_{i=1}^{N}(x_i-\bar{x})^2\right)} \\
&= \sqrt{\frac{1}{N}\left(\sum_{i=1}^{N}x_i^2 - 2\bar{x}\sum_{i=1}^{N}x_i + N\bar{x}^2\right)} \\
&= \sqrt{\frac{1}{N}\left(\sum_{i=1}^{N}x_i^2\right) - \bar{x}^2}
\end{aligned}$$

As only the running sums have to be saved, only $O(1)$ memory is required for storing the partial results at the Reducer. Since summations are commutative operations, ordering of the keys is not required.

5. MANAGING MEMORY OVERFLOWS

As noted in Section 3, an important change in our barrier-less MapReduce framework is the need to manage the storage of partial results. Depending on the category of the Reduce operation involved (see Section 4), the partial result memory complexity can be up to $O(records)$, growing to the number of records executed at the Reducer. For large datasets, which MapReduce caters to, this can quickly overflow the in-memory capacity at a server. For instance, Figure 5(a) shows a plot of the amount of heap space used by a Reducer in a MapReduce job which performs a wordcount on a 16GB dataset. The memory grew until the Reducer ran out of available heap space before an exception was thrown and the job was killed.

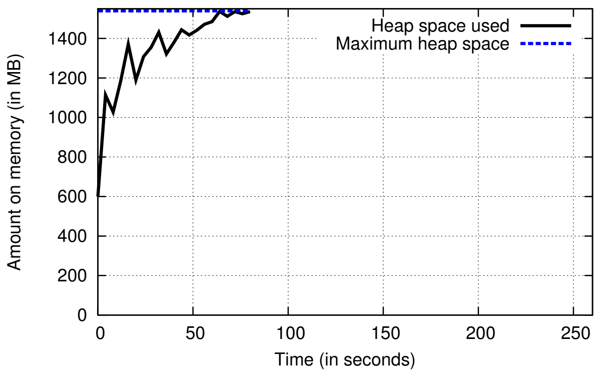
In order to address these memory overflow problems, we explore two possible memory management solutions: a *disk spill and merge* scheme (result: Figure 5(b)) and an off-the-shelf *disk-spilling key/-value store*.

5.1 Disk Spill and Merge

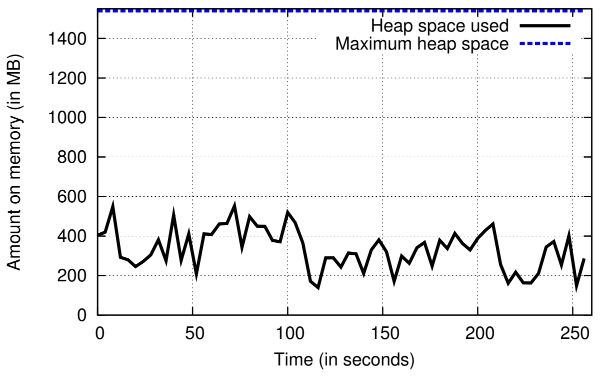
In the disk spill and merge scheme, a customized partial result storage structure is used that maintains an estimate of memory usage and supports serialization/deserializaton of the partial result contents. When the memory usage reaches a predefined memory threshold, the structure moves the partial results to a newly created local spill file on the disk.

During the course of execution at a Reducer, the memory threshold may be reached multiple times, creating many spill files. Partial results for a single key may be spilled onto multiple different spill files. When the Reduce stage is finished, the contents of these spill files must be merged together to recover the final output. Thus, we need an efficient technique to merge the partial results of the same key together.

In the spill phase, partial results are sorted by key as they are moved to a spill file. Then the merge phase merges all the partial results for each key in a straightforward manner: For every local spill file, the first partial result is loaded into the memory and stored in a buffer. Spill files containing the globally lowest key are then repeatedly read from until the lowest key's partial results are all loaded into memory. These partial results are then merged together, using a merge function. This merge function is often functionally the same as the combiner method of MapReduce (as specified in [9]),



(a) Having the complete TreeMap in memory leads to *out of memory* error and the job fails at 80 seconds.



(b) Disk spill and merge uses less memory and the job completes successfully. Partial results threshold is 240 MB.

Figure 5: Wordcount over a 16GB dataset with 10 Reducers.

but may be customized to operate on more complex methods for maintaining partial results. Once all partial results for a key have been merged, the result can be written as the final output. The next globally lowest key is found and processed in the same fashion, until all keys have been processed.

5.2 Disk spilling key/value store

Instead of flushing the entire contents of the memory to a file on the disk, the partial results can be maintained in a key/value store that has the capability of spilling to disk. Every invocation of the Reduce function fetches the previous partial result from the key/value store, processes the current input and then stores the result back into the key/value store. This read-modify-update cycle is carried out for all the inputs to the Reducer. The key/value store is capable of evicting some records out of memory and spilling to disk according to policies like Least Recently Used (LRU), whenever it runs out of memory.

We experimented with different key/value stores such as BerkeleyDB [17], Tokyo Cabinet [4] and MongoDB [3]. Among these, BerkeleyDB (Java Edition) exhibited the highest raw read and write throughput in terms of operations per second. Thus, we chose it as the key/value system to run our experiments on. We configured BerkeleyDB for performance without guaranteeing fault-tolerance of the data, because the MapReduce framework takes care of these concerns. The transaction log buffers were maintained in memory and only written to stable storage when BerkeleyDB determines that they are full or it is out of main memory.

5.3 Qualitative Comparison

The disk spill and merge approach has the advantage of avoiding the thrashing of in-memory data, unlike BerkeleyDB's caching scheme. Similarly, because it is intended specifically for managing partial result storage, it is more lightweight and efficient than a generic disk-spillable key/value store. On the other hand, it will not be able to take advantage of any prior knowledge of the distribution of keys, as it treats each of them equally. Therefore, in situations where certain keys are significantly more common than others, unnecessary spilling may occur. BerkeleyDB, like most key/value stores, performs caching and prefetching of common entries, in order to minimize reading from disk, and can therefore exploit temporal locality. We compare these approaches quantitatively in Section 6.3.

6. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance characteristics of our implementation of barrier-less MapReduce. Our implementation is based on Hadoop 0.20. We measured the improvement over the original Hadoop 0.20 for the seven classes described in Section 4.

We performed our experiments on 16 nodes from the Cloud Computing Testbed (CCT) [2]. The nodes are connected together with a Gigabit ethernet switch running 64 bit Cent OS 5.4 operating system. Each node has dual Intel Quad cores, 16GB RAM and 2TB hard disks. A single node was configured to be the JobTracker and the NameNode and the other 15 nodes were used as slaves. The replication factor of the distributed file system was set to 3 and the default chunksize was 64MB. The number of mappers and Reducers per node was set to 4, in order to utilize all the 8 cores on each node.

6.1 Improvement with Input Data Size

We experimentally evaluated the improvements in the job completion times for six applications in the following subsections. These applications correspond to the seven classes described in Section 4. (We omit the Identity class because the original and barrier-less versions are identical.) These results are summarized in Figure 7.

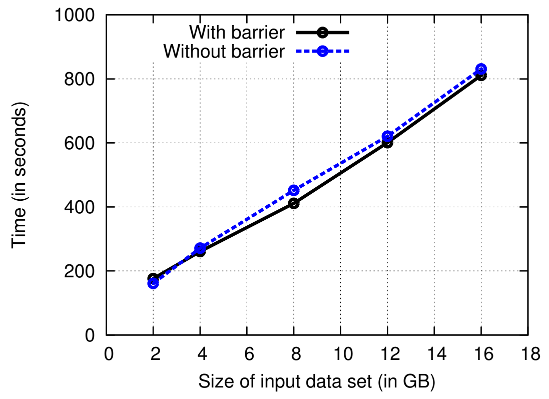
6.1.1 Sort

Our barrier-less sort implementation is similar to our WordCount implementation. We use a Red-Black tree implementation (Java TreeMap) to store a per-key count value. This count value is incremented so that duplicate values do not consume memory. Then, we emit the key *count* number of times in the end.

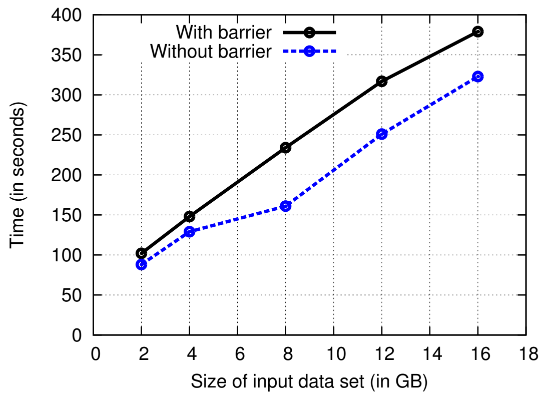
This is a degenerate case, because in the original MapReduce, both Mappers and Reducers perform no work. The comparison between the original and the barrier-less MapReduce versions becomes a competition between the two sorting mechanisms. In this case the original merge sort is faster than performing insertions into a Red-Black Tree. As a result, we observed slight slowdowns in the barrier-less version as shown in Figure 6(a), up to 9% in the 8GB case, and going down to 2% for the 16GB case.

6.1.2 WordCount

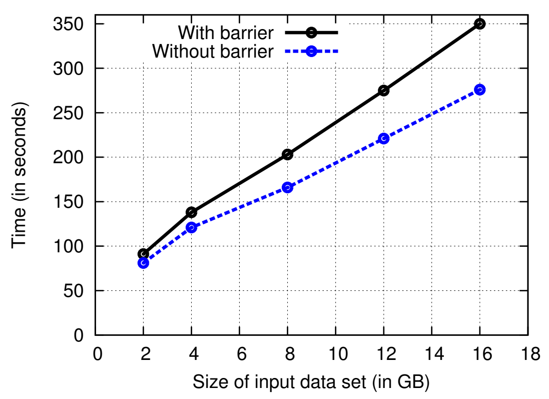
The WordCount application involves the *aggregation operation* of summing the count of the word occurrences. Despite the relatively small amount of non-sorting work performed in this bench-



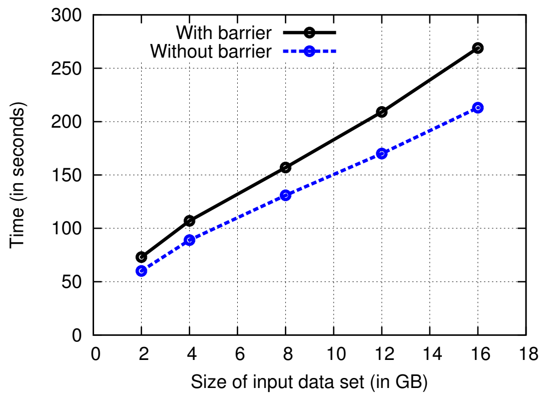
(a) Sort



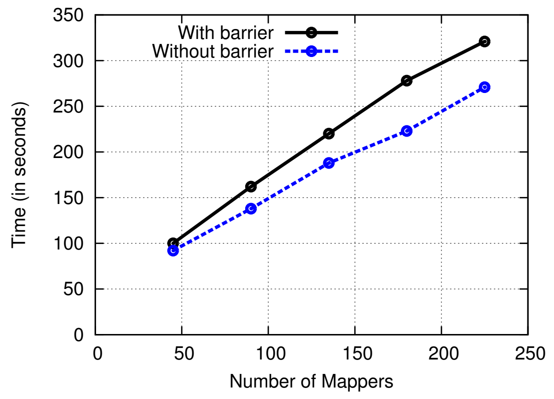
(b) WordCount



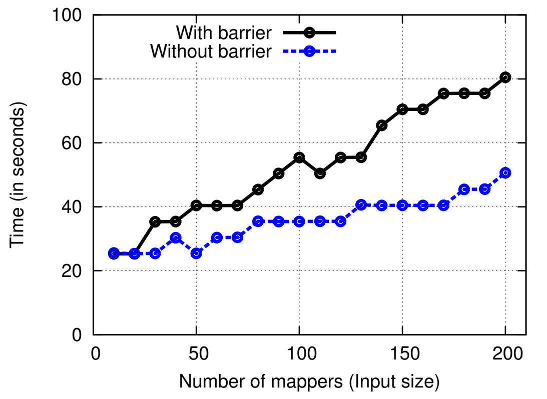
(c) k-Nearest Neighbors



(d) Last.fm Post Processing



(e) Genetic Algorithms



(f) Black-Scholes

Figure 6: Job completion times of various case studies

mark, we observed that the barrier-less approach results in an average of 15% decrease in job completion times as shown in Figure 6(b). Although the work performed in the barrier-less WordCount is essentially the same as in the barrier-less Sort, WordCount has more room for improvement due to the extra aggregation work the original version performs. This shows that, although Reducers performing no work may not see gains from our barrier-less system, even work as simple as aggregation can see notable gains. However, this improvement did not increase proportionally with the size of the dataset, since writing the output to the distributed file system is the bottleneck.

6.1.3 *k-Nearest Neighbors*

This application uses a *selection operator* which selects the top k values from the input keys. For our experiments (Figure 6(c)), we used a k value of 10. Our data values ranged from 0 to 1,000,000. The barrier-less version of k-Nearest Neighbors must perform extra work in maintaining a sorted list of the top k values, which is done automatically by the Shuffle stage in the original framework.

Nevertheless, we observed an average decrease of 18% in job completion times. This improvement slowly increased as the dataset size was increased, since the number of map rounds increased, thereby increasing mapper slack. In addition, the experimental values must be unique while training set values need not be. Therefore, the number of keys did not grow at the same rate as the number of values, resulting in less per-key data. This nature of the data affects performance as it results in relatively lesser memory overhead for the barrier-less version.

6.1.4 *Last.fm Unique Listens*

The calculation of unique listens uses *post-reduction processing*. The application counts the unique number of users that listen to a track. We ran our experiments on a dataset that generated track listens, uniformly at random across 50 users and 5000 tracks. As shown in Figure 6(d), for varying sizes of input data, we consistently observed a 20% decrease in job completion time.

6.1.5 *Genetic Algorithms*

Genetic algorithms are used to exemplify *cross-key operations*. The genetic algorithm required no change to perform barrier-less calculation, as no per-key data had to be maintained. The algorithm in both the original and the barrier-less versions only need to maintain $O(window\_size)$ keys, since each key is independent of the others.

In this experiment (Figure 6(e)), we executed a genetic algorithm with a population of 50 million individuals per mapper and varied the dataset size by increasing the number of mappers. The number of Reducers was set to 40. We observed that the performance is limited by the time spent in writing intermediate data to the local disk or the output to the distributed filesystem. This resulted in a benefit of about 15%, which stays relatively constant as the dataset size increases.

6.1.6 *Black-Scholes Options Pricing*

The calculation of options pricing using Black-Scholes involves using a *single reducer aggregation* to calculate the mean and standard deviation. In this experiment, we executed a million iterations of the Black-Scholes algorithm per mapper. Black-Scholes, similar to genetic algorithms, has a constant amount of memory in use at the Reducer ($O(1)$ with relation to the input dataset size). However, unlike genetic algorithms, the output data is also constant in size since it is just a single running average and standard deviation. Figure 6(f) shows that our approach resulted in an average benefit of about 56%, which continued to increase as the number of iterations increased. The maximum improvement in completion time observed was 87%. This is the best performance of our approach across all application classes.

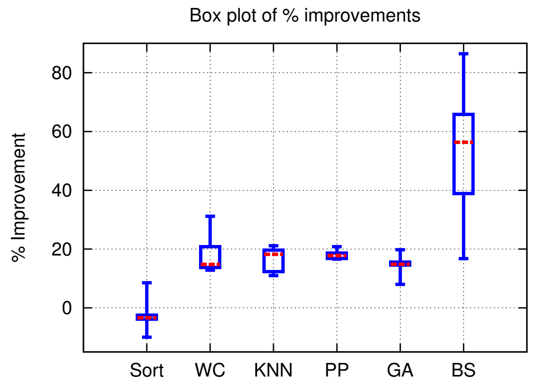


Figure 7: Relative % improvements for **Sort**; **WC**: Word Count; **KNN**: k Nearest Neighbors; **PP**: Last.fm Post Processing; **GA**: Genetic Algorithm and **BS**: Black-Scholes. The whiskers show the maximum and the minimum, the box shows the 25% and 75% quartiles and the dotted line shows the median of the values.

6.1.7 *Comparing Improvement Across Applications*

Figure 7 shows a box plot of the relative improvements of the different case studies discussed. Black-Scholes showed the largest improvement, due to the $O(1)$ memory overhead and output dataset size. In addition, because the improvement continued to increase, it had the largest variation. The other benchmarks had improvements that stayed consistent around the 20% mark, which was the common case due to the limitations imposed by mapper slack and time spent writing to disk. In addition, sort was observed to be our worst case with a small performance loss on average. It is possible that exploring the effects of heterogeneity may likely yield larger improvements, but this is outside the scope of this paper.

6.2 Improvement with Number of Reducers

In order to understand the sensitivity to the number of Reducers, we varied the Reducer count in Figure 8 and observed the improvement as the count rose from 30 to 70 (which is 10 more than the number of available CPU cores for Reducers). This illustrates the effect of applications or systems with an irregular amount of Reducers, for example if nodes fail in the middle of computation.

Our results show that although job completion time decreased as the compute utilization increased (as the number of Reducers reached the compute capacity of 60), our improvement over the barrier version decreased somewhat. When the number of Reducers surpassed the amount of compute resources available (70 Reducers running on 60 nodes), the job completion time increased, but our improvement also increased.

The reason for our scheme having a larger improvement when the system is underutilized (for example when there are only 30 Reducers), is that each Reducer has to shuffle respectively more data than in the fully utilized case. This means that the shuffle

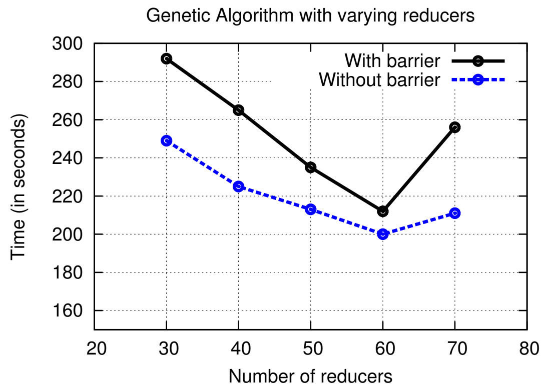


Figure 8: Varying the number of Reducers for genetic algorithms

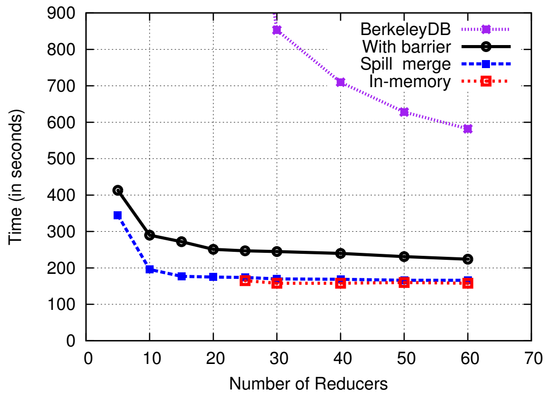


Figure 9: Wordcount with different memory management techniques with increasing number of Reducers.

time is larger, and the *mapper slack*, during which the barrier-less version can perform meaningful work, is also larger. As the utilization becomes more full, the mapper slack decreases, limiting, but not removing, the benefit gained from breaking the barrier. On the other hand, once the system becomes over-saturated (the 70 Reducer case), a new round of Reducers is needed, which must themselves undergo a shuffle stage, once again increasing the mapper slack. In other words, the benefit of switching to a barrier-less framework is closely tied to the amount of mapper slack in the runtime.

6.3 Memory Management Techniques

We compared the different memory management techniques described in Section 5. Figure 9 shows a plot of the job completion times for wordcount with and without a barrier, while varying the number of Reducers. The disk spill and merge scheme performed slightly worse than storing the partial results in memory. However, as the number of Reducers was decreased below 25, the in-memory technique resulted in an out of memory exception and the job was killed. The spill and merge technique continued to perform better than the original MapReduce. BerkeleyDB on the other hand, performed poorly on the wordcount. Even though we could observe about 30,000 inserts per second into the database, this was not enough throughput to keep up with the millions of small records handled at each Reducer. This result shows that off-the-shelf key/-value stores may not be a suitable option for MapReduce workloads.

Figure 10 shows a comparison with increasing dataset size. It can be seen that as the dataset increases, both the disk spill and merge, and the in-memory barrier-less versions, outperformed the original version. Again, the BerkeleyDB key/value store can not keep up with the high frequency of record accesses.

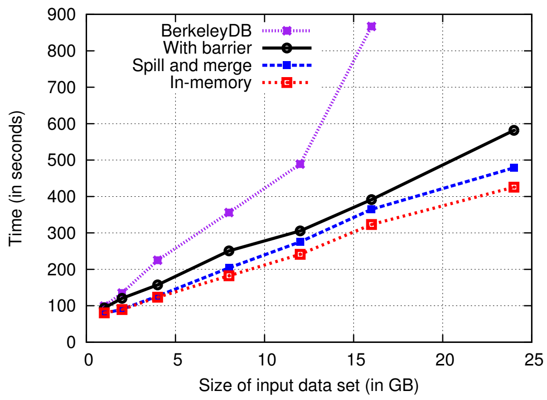


Figure 10: Wordcount with different memory management techniques with increasing dataset size.

Application	Lines of code		
	Original	Barrier-less	% increase
Sort	28	95	240%
WordCount	73	88	20%
k-Nearest Neighbors	195	208	10%
Post Processing	73	91	25%
Genetic Algorithm	532	533	0%
Black-Scholes	251	252	0%

Table 2: Programmer effort, in terms of lines of code, required to convert MapReduce applications to their barrier-less versions.

6.4 Programmer Effort

Table 2 summarizes the programmer effort required, in terms of lines of code, to convert the MapReduce applications into their barrier-less counterparts. The code for sorting in the original case is very short due to the use of the Identity Mapper and the Identity Reducer, since the framework does the job of sorting. However, we had to add more functionality in the Reduce function of the barrier-less version. WordCount, k-Nearest Neighbors and Post Processing required small changes to compute and update the partial results. For Black-Scholes and the genetic algorithm, the only change required was that a flag for barrier-less execution be turned on.

7. RELATED WORK

MapReduce has been widely used for processing large data because of its simple model that is applicable to “embarrassingly parallel” problems – such as log processing. Current research looks to push MapReduce by using it to solve harder problems. These include machine learning [7], statistical machine translation [6, 11], optimization [20], finance [5], and similarity scoring [12]. MapReduce is a logical choice because it allows the problems to be solved on a loosely coupled set of machines, with less effort than producing custom parallel processing code. However, MapReduce does not always give the most efficient parallel processing implementation. In this paper, we looked at the stage barrier in MapReduce and showed how breaking it may result in making MapReduce more efficient for general MapReduce problems.

We are not the first to look at breaking the barrier. Previous work has looked at using the MapReduce programming framework for online processing [8]. Unlike batch processing, online processing applications such as event monitoring or stream processing require breaking the barrier to keep computations up-to-date. Unlike these application specific solutions, we present a general framework for breaking the barrier that can be used for both online and batch processing. Our techniques solve memory issues that can appear when processing large-scale data without a barrier, whether it is online or batched.

Improving the efficiency of MapReduce has been of recent interest to the systems community. Much of the research presented has required changes to the MapReduce API [22]. Other work has aimed to be completely transparent to the programmer [23]. Our work is a combination of both categories. We have preserved the baseline MapReduce API, while empowering the programmer to improve performance by working under relaxed assumptions in the Reduce function framework.

Dryad [16] is a distributed platform that has been developed at Microsoft to provide large-scale, parallel, fault-tolerant execution of processing tasks. It allows the execution of computations that are expressed as directed acyclic graphs. Dryad also follows the policy of writing locally and reading remotely and has a barrier to control this process. The techniques in this paper can likely be applied to break the barrier in a similar way to the MapReduce barrier. Because Dryad is a closed system, we were not able to make modifications to apply these techniques.

8. CONCLUSION

This paper demonstrated that general purpose MapReduce frameworks without a barrier are feasible, and they can result in significant performance benefits. By intelligently managing memory and identifying which forms of Reduce functions see the most benefit, our experiments with Hadoop demonstrate speedups of up to 87% for well-suited applications, and an average of 25% for more typical applications. Our approach preserves the fault tolerance of the original MapReduce model, and has similar ease of programming.

Our work opens up new avenues. Memoization, an optimization similar to DryadInc [19] becomes feasible in the barrier-less model. Exploring heterogeneity in systems and how much improvement our barrier-less framework grants in the face of that heterogeneity is another important line of investigation.

9. REFERENCES

[1] Apache hadoop. `http://hadoop.apache.org`.

[2] Illinois Cloud Computing Testbed. `http://cloud.cs.illinois.edu/`.

[3] Mongodb. `http://www.mongodb.org`.

[4] Tokyo cabinet. `http://1978th.net/tokyocabinet/`.

[5] F. Black and M. S. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–54, May-June 1973.

[6] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *Proc. of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 858–867, 2007.

[7] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. *Advances in Neural Information Processing Systems: Proc. of the 2006 Conference*, page 281, 2007.

[8] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Proc of the 6th Symposium on Operating Systems Design and Implementation*, pages 137–149, 2004.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[11] C. Dyer, A. Cordova, A. Mont, and J. Lin. Fast, easy, and cheap: construction of statistical machine translation models with mapreduce. In *StatMT '08: Proc. of the Third Workshop on Statistical Machine Translation*, pages 199–207, Morristown, NJ, USA, 2008. Association for Computational Linguistics.

[12] T. Elsayed, J. Lin, and D. W. Oard. Pairwise document similarity in large collections with mapreduce. In *Proc. of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies (HLT '08)*, pages 265–268, Morristown, NJ, USA, 2008. Association for Computational Linguistics.

[13] R. Farivar, A. Verma, E. Chan, and R. Campbell. Mithra: Multiple data independent tasks on a heterogeneous resource architecture. In *Proc. of IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER '09.*, pages 1–10, 31 2009-Sept. 4 2009.

[14] E. Fix and J. Hodges. Discriminatory analysis, nonparametric discrimination: Consistency properties. Technical report, USAF School of Aviation Medicine, Randolph Field, Texas, 1951.

[15] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 8–21, Oct. 1978.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, Lisbon, Portugal, 2007.

[17] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley db. In *ATEC '99: Proc. of the annual conference on USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[18] O. O'Malley and A. C. Murthy. Winning a 60 second dash with a yellow elephant. `http://sortbenchmark.org/Yahoo2009.pdf`, 2009.

[19] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing work in large-scale computations. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*, San Diego, CA, 2009.

[20] A. Verma, X. Llora, D. E. Goldberg, and R. H. Campbell. Scaling genetic algorithms using mapreduce. In *International Conference on Intelligent Systems Design and Applications*, Pisa, Italy, 2009.

[21] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., June 2009.

[22] H. Yang, A. Dasdan, R. Hsiao, and D. Parker. Map-reduce-merge: simplified relational data processing on large clusters. *Proc. of the 2007 ACM SIGMOD international conference on Management of data*, Jan 2007.

[23] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

APPENDIX

This section lists the source code of the Wordcount MapReduce application which counts the number of occurences of each unique word for all the files present in the input directory specified on the command line.

```
import java.io.IOException;
import org.apache.hadoop.*;

public class WordCountWithoutBarrier {
 public static class TokenizerMapper
 extends Mapper<Object,Text,Text,IntWritable>{

  private final static IntWritable one
        =new IntWritable(1);
  private Text word=new Text();

  public void map(Object key,Text value,
        Context context) throws IOException,
        InterruptedException {
   StringTokenizer itr=
        new StringTokenizer(value.toString());

   while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
   }
  }
 }

 public static class IntSumReducer extends
   Reducer<Text,IntWritable,Text,IntWritable> {
  TreeMap<Text, IntWritable> hm =
   new TreeMap<Text, IntWritable>();
  public void reduce(Text key,
    Iterable<IntWritable> values,Context context)
    throws IOException, InterruptedException {

   int sum=((IntWritable) hm.get(key)).get();

   for(IntWritable val:values){
    sum+=val.get();
   }
   hm.put(new Text(key),new IntWritable(sum));
  }

  public void run(Context context) throws
        IOException,InterruptedException {
   while(context.nextKey()){
    Text key=context.getCurrentKey();
    if(!hm.containsKey(key))
        hm.put(new Text(key),new IntWritable(0));
    reduce(key,context.getValues(),context);
   }

   try{ for(Object key:hm.keySet()){
      context.write((Text)(key),
        ((IntWritable)hm.get(key))); }
   }catch(IOException e){e.printStackTrace();}
  }
 }

 public static void main(String[] args)
        throws Exception {
  int numReducers=1;
  Configuration conf=new Configuration();
  String[] otherArgs=new GenericOptionsParser
        (conf, args).getRemainingArgs();
  if(otherArgs.length!=3) {
   System.err.println("Usage: incwordcount " +
        "<in> <out> <numReducers>");
   System.exit(1);
  }
  conf.setIncrementalReduction(true);
  numReducers=Integer.parseInt(args[2]);
  Job job=new Job(conf,"Wordcount without barrier");
  job.setJarByClass(WordCountWithoutBarrier.class);
  job.setMapperClass(TokenizerMapper.class);
  job.setReducerClass(IntSumReducer.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  job.setNumReduceTasks(numReducers);
  FileInputFormat.addInputPath(job,
        new Path(otherArgs[0]));
  FileOutputFormat.setOutputPath(job,
        new Path(otherArgs[1]));
  System.exit(job.waitForCompletion()?0:1);
 }
}
```