Breaking the MapReduce stage barrier

**Abhishek Verma · Brian Cho · Nicolas Zea ·
Indranil Gupta · Roy H. Campbell**

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. Hence, 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 job completion times than a traditional MapReduce framework. This is due primarily to the interleaving of I/O and computation, and forgoing disk-intensive work. We achieve a reduction in job completion times that is 25% on average and 87% in the best case.

A. Verma (✉) · B. Cho · N. Zea · I. Gupta · R.H. Campbell
Department of Computer Science, University of Illinois
at Urbana-Champaign, 201 North Goodwin Avenue, Urbana,
IL 61801-2302, USA
e-mail: verma7@illinois.edu

B. Cho
e-mail: bcho2@illinois.edu

N. Zea
e-mail: nicolas.zea@gmail.com

I. Gupta
e-mail: indy@illinois.edu

R.H. Campbell
e-mail: rhc@illinois.edu

1 Introduction

Inspired by the *map* and *reduce* primitives present in functional languages, Google proposed MapReduce [7]. 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 [7]. 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 a 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 MapReduce framework divides the program execution into a Map and a Reduce stage. In the first stage, each machine in the cluster executes a Map function on a distinct portion of the input data. The Map execution produces records that consist of a key and value. Map output is transferred to Reducers according to key. For each key, the complete set of records produced across multiple machines in the Map stage are transferred to the Reducer assigned to that key. In current implementations of MapReduce, a *barrier* exists that prevents the Reduce stage from starting until all records from the Map stage have been transferred to their respective Reducers. The barrier ensures that all relevant records are available when computing the Reduce function.

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.

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. Our experience with implementing various barrier-less algorithms shows there is minimal additional programmer effort.
5. 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 job completion times.
6. By reviewing the job execution, we observe that the improvement is due to interleaving of I/O and computation, as well as forgoing disk-intensive work.

In this paper, 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 will also benefit from the techniques described in our paper.

The rest of the paper is structured as follows. We first examine the role of the MapReduce barrier in Sect. 2. In Sect. 3, we then discuss the new design of Hadoop MapReduce framework after breaking the barrier. We observe that this improves performance in many cases, but can present a memory management problem. In Sect. 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 Sect. 5 we develop new techniques that are able to manage memory for general applications while breaking the barrier. In Sect. 6, we present experimental results that show a significant improvement in Hadoop performance when these techniques are applied. We then discuss related work in Sect. 7 and finally conclude in Sect. 8.

2 Motivation: MapReduce barrier

The execution of a MapReduce program is divided into a Map stage and a Reduce stage. 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 (e.g., Hadoop), the Shuffle stage contains a distributed barrier.

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.

However, despite these apparent advantages, we argue that removing the barrier is, in many practical cases, much more efficient. When we remove the barrier, execution of our Reduce function no longer needs to wait for all records to be remotely read and grouped by key. Instead, the Reduce function can be immediately invoked on each input entry, as it becomes available. This relaxation can significantly improve the efficiency of the Reduce task execution.

More concretely, by removing the barrier, we are able to perform work at Reducers during 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. Instead of waiting for remote reads to finish, our framework interleaves the network I/O with the computation of the Reduce function. Our framework forgoes the disk-intensive operation of sorting records altogether for most algorithms. Both of these waiting intervals are sensitive to heterogeneity that is inherent in clusters. Clusters with commodity hardware often show differences in performance between machines, and they have oversubscribed links between machines. 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. This can further extend the first interval of waiting. The time consumed by disk-intensive operations in the second interval is sensitive to the relative disk speeds across each node. The barrier-less model removes these intervals, thus improving performance.

As a consequence of 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 Sect. 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

Table 1 Sort and Memory requirements of MapReduce Jobs. *Records* and *keys* denote the total number of records and keys (executed at a single Reducer)

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	$O(1)$

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 Sect. 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 batched 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 by transferring batches of records from Mappers to Reducers. Each Mapper buffers its processed records. When all local processing is finished the Mapper sorts the records by key, a step that partitions the records to be transferred to each Reducer. If the buffer grows larger than the memory available, it is spilled to disk and later merge-sorted. Each Reducer retrieves the batch of records from each Mapper only after it has completed. When all records from every Mapper are received by the Reducer it merge-sorts them. This step of sorting groups all records together.

For barrier-less MapReduce, we pipeline the record transfer and Reduce execution at the single record level, instead of batching. There are two potential benefits to pipelining. First, we make it possible to interleave network I/O of the record transfer with CPU computation. Second, we avoid the need to spill the intermediate data to disk when the size of batched records grows beyond memory.

In our implementation, the Reducer uses one asynchronous thread per Mapper to retrieve records, as soon as they are available. These records are stored into a single buffer, and 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 key/value 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 slightly changes the way applications are implemented, as we show with an example in Sect. 3.2.

However, these changes do not affect other aspects of the execution of Hadoop. In other words, assignment of tasks, fault-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* that makes calls to the Reduce function. In the original Hadoop, the run function invokes the Reduce function once per key. For barrier-less Hadoop, the run function invokes the Reduce function once per entry, and 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

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

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

   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

form 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 Reduce and run 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 implies that the Reducer must maintain partial results for every key it has received. For our purposes, we use the TreeMap data structure from `java.util`, which is internally implemented as Red-Black trees [13]. 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 1 shows the system-wide progress of the Word-Count program with and without a barrier on the same cluster. (Details of the experimental setup are provided in Sect. 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 function invocations 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

(a)

Execution of original word count

(b)
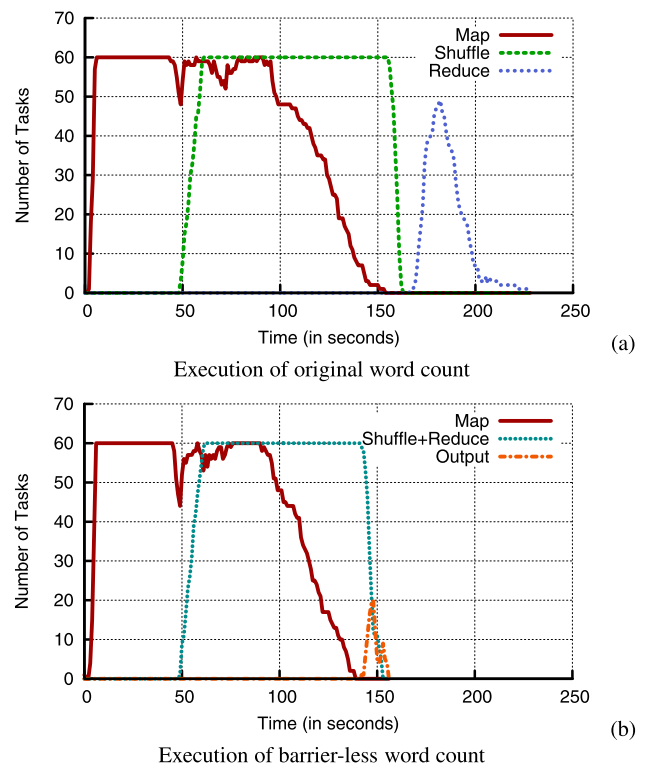
Execution of barrier-less word count

Fig. 1 Progress of MapReduce which performs a word count on a 3 GB Wikipedia data set

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.

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 and interleaved, 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 non-invasive, 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 and we address this in Sect. 5.

4 Classifying reduce operations

In order to understand the 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 [7]; machine learning benchmarks [5]; statistical machine translation [4], [9]; optimization algorithms [24]; finance algorithms [3]; and similarity scoring [10].

We classified the Reduce operations performed in these applications. 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.

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 [7]. The Map function emits a line of text if it matches a pattern. The Reduce function is merely used to write 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 100 TB of data [19].

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

We classify aggregation operations as those that are commutative and distributive—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 Sect. 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, while sorting keys the framework can also allow the developers to sort values as well (e.g., Hadoop's secondary sort). Given such an ordering, the Reduce function can trivially select the values of interest. For example, when finding a minimum, the Reducer can select and output the first value for every key.

In our barrier-less implementation, we bypass the sort operation, so these operations must be performed on a *running* basis. For example, a running minimum (or minimum k values) can be 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 processed.

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 [12] and is often used in statistical analysis applications, such as finding pairwise similarity [10].

The distance between an experimental value (E) and the training value (T) is defined as the absolute value of their difference ($|E - V|$). It is necessary to compare each experimental value to every training value. The barrier version's Map function emits a tuple ($E, |E - V|$) for the key, and an integer T for the value. A secondary sort is performed, sorting by the $|E - V|$ value in the key, but grouping by E. 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 E as the key and a tuple ($T, |E - V|$) as the value. There is no need to emit a tuple as before, since a sort is not performed. Now, for each key, the Reducer maintains a size-k ordered linked list, and decides if the most recently received ($E, |E - V|$) tuple belongs in the list, based on the $|E - V|$ 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 E values), 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 applied 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 [26]. 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 [26] 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 [24]. Each individual (I) is represented as a key and the Mapper computes the fitness (F) of each individual and emits the tuple (I, F). 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 ([3, 11]). 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:

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

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

$$= \sqrt{\frac{1}{N}\left(\sum_{i=1}^{N}x_i^2\right) - \bar{x}^2}$$

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 Sect. 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 Sect. 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, the line "In-memory TreeMap" in Fig. 2 shows the amount of heap space used by a Reducer in a MapReduce job which performs a WordCount on a 16 GB dataset. The used memory increased, until the Reducer ran out of available heap space. An OutOfMemory exception was thrown and the job was killed around 80 s.

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

5.1 Disk spill and merge

In the disk spill and merge scheme, we define a memory threshold. If memory usage for partial results reaches this
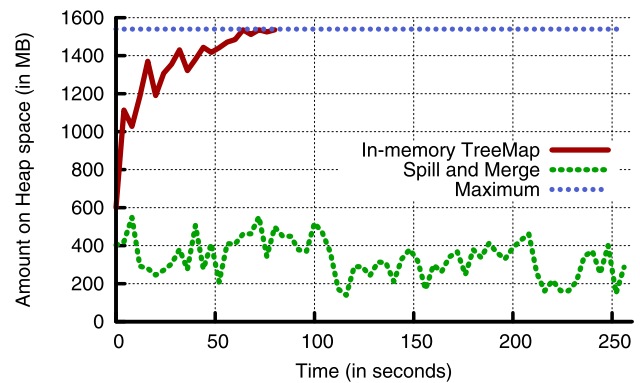

Fig. 2 WordCount over a 16 GB dataset with 10 Reducers. Having the complete TreeMap in memory leads to *out of memory* error. Spill and merge uses less memory and the job completes successfully. Partial results threshold is 240 MB

threshold, it is spilled to a newly created file on disk. When all records have been processed, the spilled files are merged using a merge function defined by the programmer.

During the course of execution at a Reducer, the memory threshold may be reached multiple times, creating many spill files. Thus, partial results for a single key may be spilled onto multiple different spill files. After all invocations of the Reduce function at a Reducer are finished, all the partial results for a single key must be merged together.

The disk spill and merge scheme is designed to allow the merge to be performed efficiently. 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, similar to an external merge-sort. 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. 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.

The method of merging results of each key depends on the application and the data structure that stores the partial results. Thus, the programmer supplies the merge function. This function is functionally similar to the combiner method of MapReduce (as specified in [7]), but may be customized if more complex methods are required for maintaining partial results.

The effect of using Disk Spill and Merge is shown in Fig. 2. In our implementation, we maintain a count of records in the partial result data structure to estimate the memory usage. We spill to and read from disk by using Java's serialize/deserialize interface. The line "Spill and Merge" shows that the partial results data structure is kept below the memory threshold.

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 [18], Tokyo Cabinet [23] and MongoDB [17]. Among these, BerkeleyDB (Java Edition) exhibited the highest raw read and write throughput in terms of operations per second. Hence, we chose it as the key/value system to run our experiments. 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 pre-fetching of common entries, in order to minimize reading from disk, and can therefore exploit temporal locality. We compare these approaches quantitatively in Sect. 6.5.

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 of applications described in Sect. 4.

We performed our experiments on 16 nodes from the Cloud Computing Testbed (CCT) [15] running 64 bit Cent OS 5.4 operating system. Each node has dual Intel Quad cores, 16 GB RAM and a single 1 TB hard disk. The nodes are connected together with a Gigabit Ethernet switch. 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 64 MB. 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 Sect. 4. (We omit the Identity class because the original and barrier-less versions are identical.) Figure 3 shows the job completion times for various benchmarks with and without barrier.

6.1.1 Sort

Our barrier-less sort is implemented in a similar manner to our WordCount implementation, in order to avoid consuming memory for duplicate keys. We use a Red-Black tree implementation (Java TreeMap) to store a per-key count value. This count value is incremented, when a duplicate key is encountered. When we output the results, we emit the duplicates of the key *count* number of times.

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, up to 9% in the 8 GB case, and going down to 2% for the 16 GB 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 benchmark, we observed that the barrier-less approach results in an average of 15% decrease in job completion times. 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.
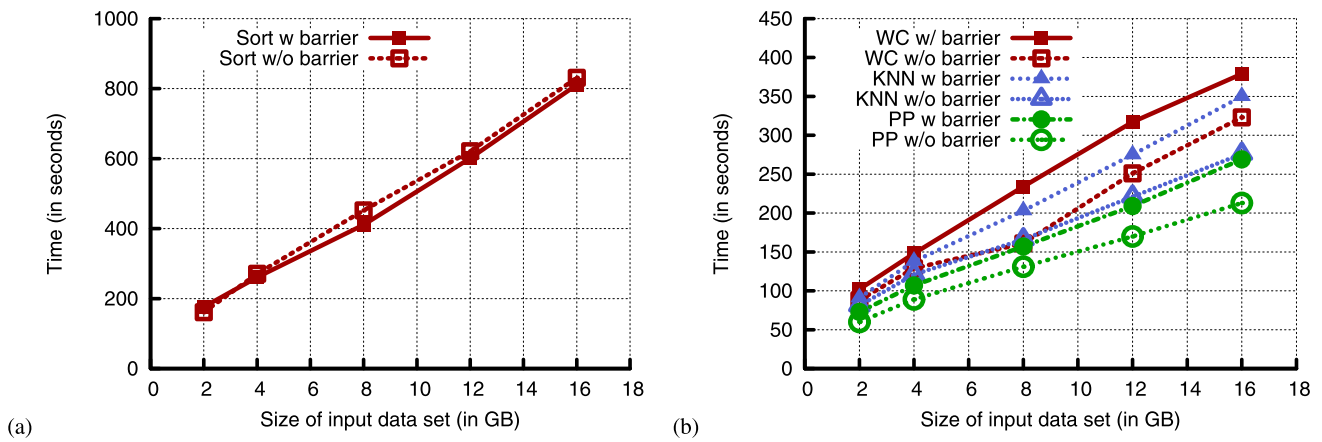
Fig. 3 Comparison of different benchmarks: Sort, WordCount (WC), *k*-Nearest Neighbor (KNN) and Post Processing (PP) with and without barrier, with increasing dataset sizes

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, 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. 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
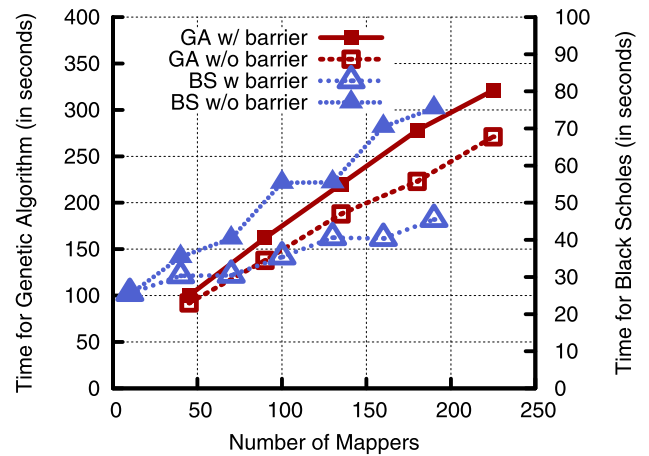

Fig. 4 Genetic Algorithms (GA) and Black Scholes (BS) with and without barrier, varying the number of Mappers

versions only need to maintain $O(window\_size)$ keys, since each key is independent of the others.

In this experiment (Fig. 4), 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 4 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.

6.1.7 Comparing improvement across applications

Black-Scholes showed the largest improvement, due to the $O(1)$ memory overhead and output dataset size. 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.
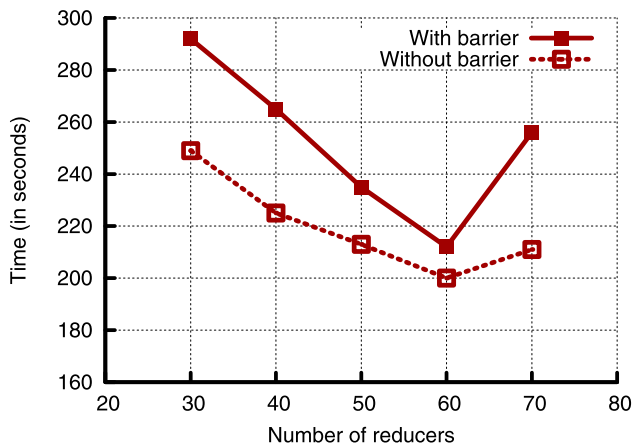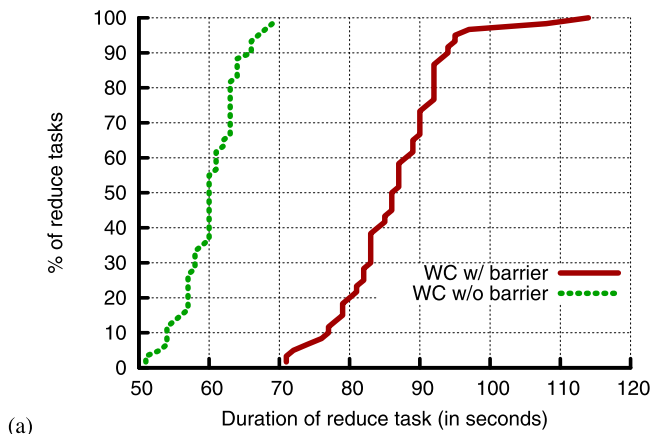

Fig. 5 Genetic Algorithms with and without barrier, varying the number of reducers

6.2 Performance improvement breakdown

In order to better understand the reason behind the improvement in overall completion time for barrier-less MapReduce, we take an in-depth look at the execution of two of the applications, WordCount and Genetic Algorithm. We chose these applications because they showed improvements closest to the average.

In Fig. 6, we plot a CDF of the duration of each task in the Reduce stage, for both original MapReduce and barrier-less MapReduce. The task duration denotes the duration from when the first byte is received at the Reduce task from a Map task, to when the Reduce task emits its final byte of output. We make two observations about the improvement in overall completion time. First, the median Reduce task completion time is smaller for barrier-less MapReduce. For WordCount, up to the 95th percentile, the gap is between 20 to 30 seconds, while for Genetic Algorithm, up to the 90th percentile, the gap is between less than 5, to around 10 seconds. Secondly, the original MapReduce contains outlier Reduce tasks that have significantly longer completion times than other Reduce tasks. In contrast, we do not see any significant outliers in barrier-less MapReduce. Due to this, there is a large gap between the Reduce tasks with the longest completion times: the gap for WordCount is more than 40 seconds, while for Genetic Algorithm it is almost 25 seconds.

These factors are also visibly present in Fig. 7, where we plot the end time for each Reduce task, i.e., when the final byte of output is emitted, counting from the start of the entire MapReduce job.

Figures 8 and 9 plot the start and duration for each individual Map and Reduce task. For original MapReduce, the Reduce stage is separated into Shuffle, Sort (not to be confused with the Sort application), and Reduce function execution. Shuffle execution is dominated by network I/O, Sort

(a)
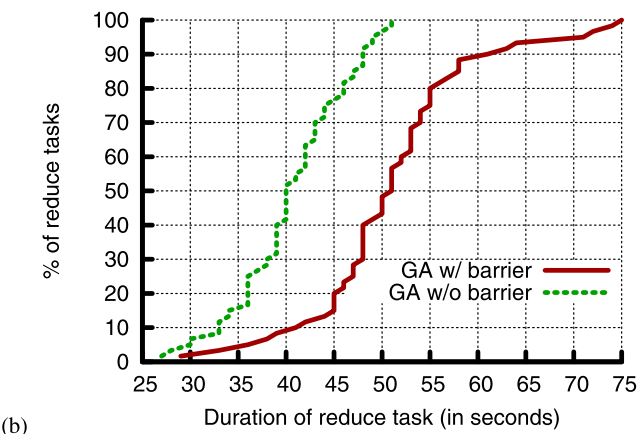
(b)

Fig. 6 CDF of Reduce task duration
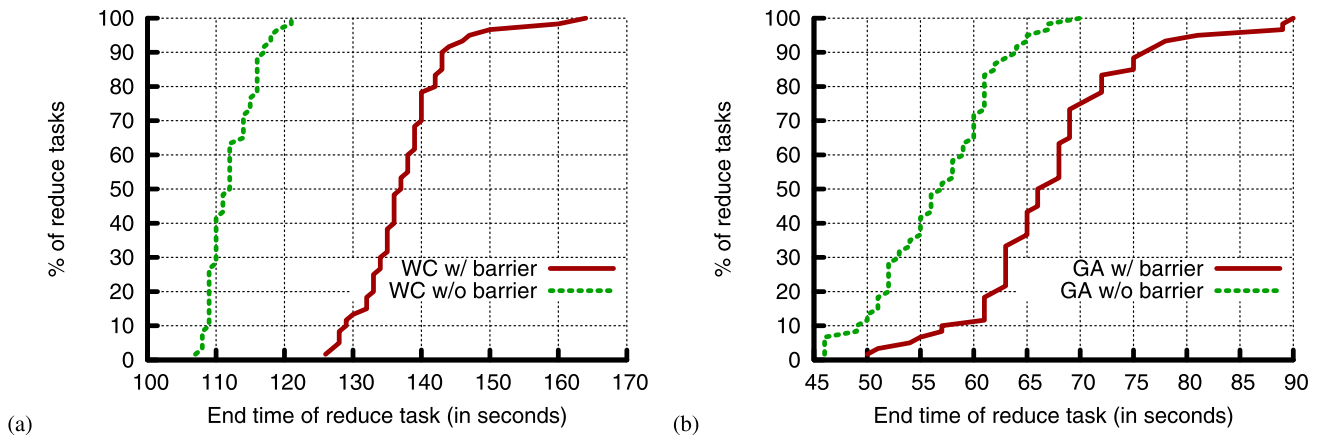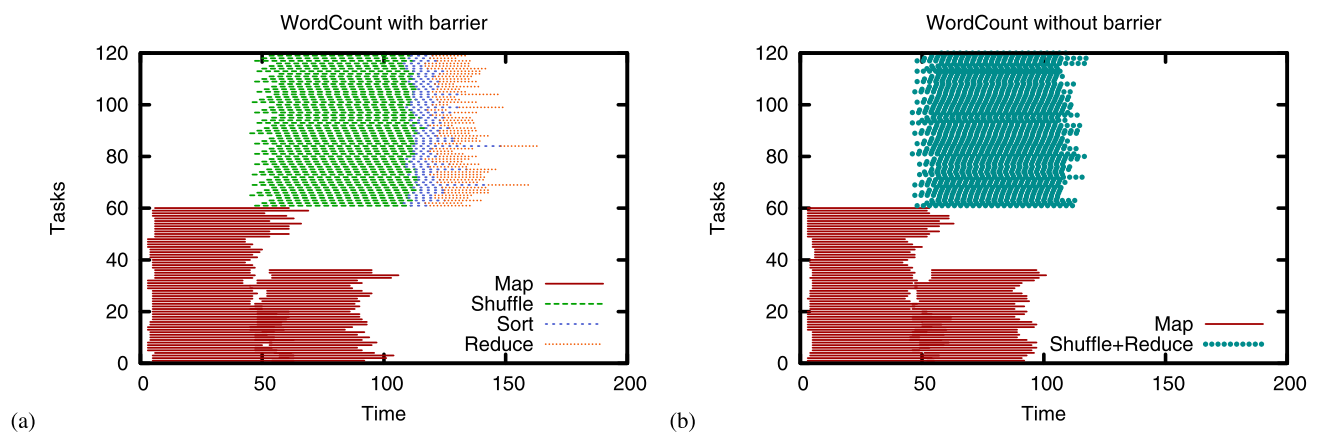
Fig. 7 CDF of end times of Reduce tasks
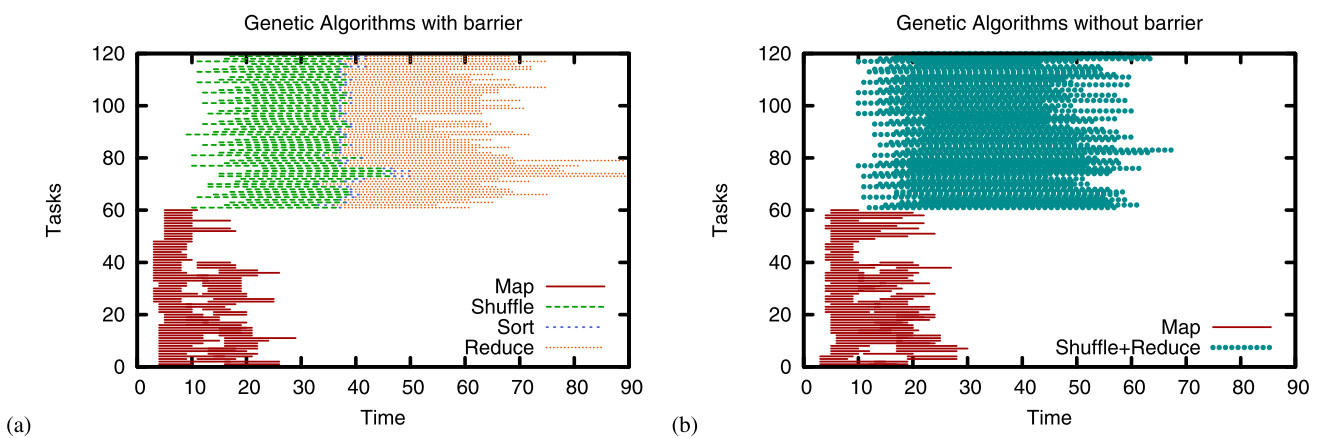

Fig. 8 Lanes plot of WordCount

Fig. 9 Lanes plot of Genetic Algorithms

is dominated by disk I/O, and Reduce is dominated by CPU computation. From the plot, we observe two main causes of the dissimilarity in Reduce task duration times. First, the contrast in median completion time is because barrier-less

MapReduce is able to successfully interleave both disk I/O and CPU computation together with network I/O. Thus, the duration of the entire barrier-less Reduce task is close to the duration of Shuffle for the original Reduce task. Sec-
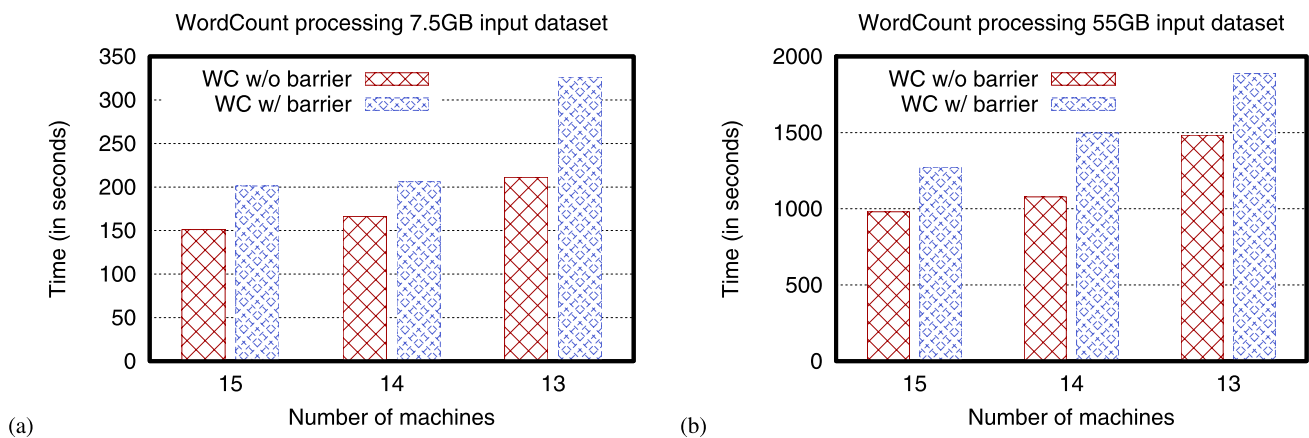
WordCount processing 7.5GB input dataset

WordCount processing 55GB input dataset

(a)

(b)

Fig. 10 WordCount performance on heterogeneous machines processing 7.5 GB (*left*) and 55 GB input dataset (*right*)

ond, Sort appears to be the main cause for the outliers of the original Reduce—this is likely caused by heterogeneity of disk I/O performance within the cluster.

The variation in Reduce execution time for Genetic Algorithm is likely due to the involved computation being more complex. We observe high variance in the duration for both Map and Reduce function execution for the original tasks. This effect is also observed in the barrier-less setting, where the end times of the Reduce tasks in Genetic Algorithm have more variance than WordCount.

Thus, we conclude that for the Genetic Algorithm and WordCount applications, any outliers arising from our setting are eliminated by using a barrier-less approach.

6.3 Improvement with number of Reducers

In order to understand the sensitivity to the number of Reducers, we varied the Reducer count in Fig. 5 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 cores), 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 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.

6.4 Cluster heterogeneity

We go a step further and induce more heterogeneity, by placing multiple Mappers and Reducers on a designated overloaded machine. This setup emulates heterogeneity caused by different loads on cluster machines, e.g. which is typical in a virtualized data center [28].

In these experiments, we look at the effect of cluster heterogeneity on barrier-less MapReduce and compare it to original MapReduce. Some heterogeneity is already present in our cluster despite it being composed of machines with originally identical specifications. We go a step further and induce more heterogeneity, by overloading machines with more Mappers and Reducers. Our 16 machine cluster has 1 master and 15 workers. We vary the number of worker machines n from 15 to 13, with the following configuration: $n - 1$ machines each have 4 map and 4 reduce slots while the nth (designated) machine has $(16 - n) * 4$ map and $(16 - n) * 4$ reduce slots. This causes the nth machine to be overloaded.

Figure 10 shows the execution time of WordCount with 15, 14, and 13 worker machines, i.e., when the designated machine is configured with 4, 8, and 12 Mappers and an equal number of Reducers respectively. This mimics, for example, a virtualized cluster where heterogeneity can be
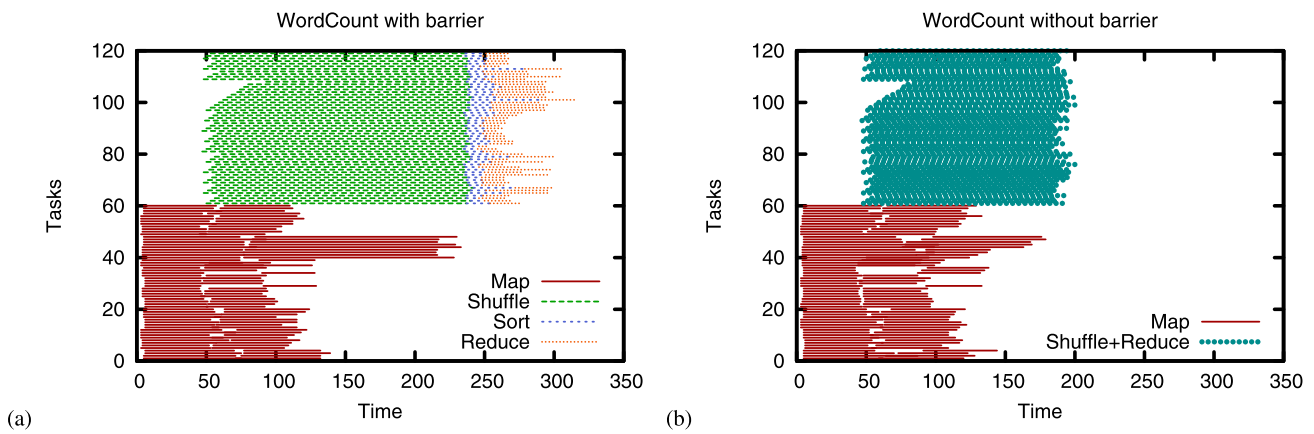
Fig. 11 Lanes plot of WordCount processing 7.5 GB on 13 heterogeneous machines

caused by many virtual machines being run on a single machine. When the input dataset processed is 7.5 GB, the entire intermediate data is kept in memory for both barrier-less and original MapReduce. We see a large difference (54%) in execution time in the case of the 13 worker machines. We observe that increasing the heterogeneity from 14 to 13 worker machines leads to 58% worse performance for original MapReduce, but only 27% worse performance for barrier-less MapReduce.

In order to understand the cause for this improvement, Fig. 11 shows the start and duration of each task when Word-Count is run on 13 worker machines. In this configuration, one of the machines is configured with 12 Mappers and 12 Reducers, while the other 12 machines are configured with 4 Mappers and 4 Reducers. We observe that the duration of the map tasks on the overloaded machines is longer in original MapReduce. This is because the overloaded machines have too little available memory per task, and thus the Mappers must buffer their records on disk. These Map outliers, coupled with the barrier, force the Reduce function execution to be delayed. In contrast, barrier-less Mappers are not required to buffer all records because they transfer records immediately. Thus, disk I/O time is saved resulting in the overloaded Map outliers finishing earlier than their original counterparts. Also, because there is no barrier, the Reduce function execution is interleaved with the network transfer, resulting in the combined Shuffle and Reduce finishing shortly after the final outlier Mapper. As a result of the combination of these factors, we observe that the entire job completes in our barrier-less MapReduce implementation before even the Map stage is completed in original MapReduce.

When the input dataset size is increased to 55 GB, the amount of intermediate data processed per Reducer increases above the available memory limit and the disk spill and merge technique is used in the barrier-less version, while the original version uses an external MergeSort. In
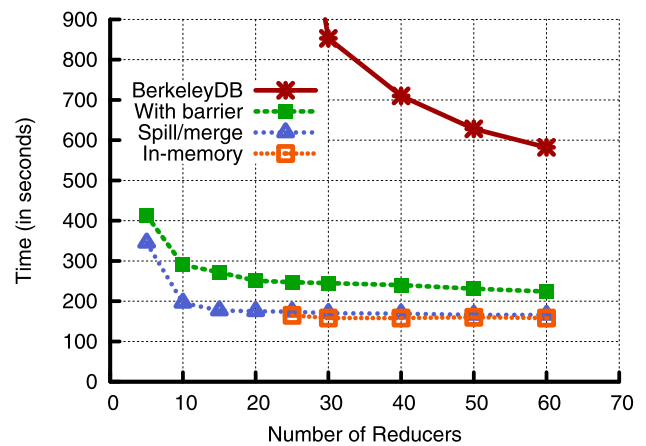

Fig. 12 WordCount with different memory management techniques with increasing number of Reducers

this case, we observe that the difference (27%) is smaller when the disk spill and merge technique is used instead of keeping the intermediate results entirely in memory, because disk I/O is required.

Thus, we conclude that our barrier-less implementation reduces variation in task completion times and thus helps in reducing the performance degradation due to some types of heterogeneity.

6.5 Memory management techniques

We compared the different memory management techniques described in Sect. 5. Figure 12 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
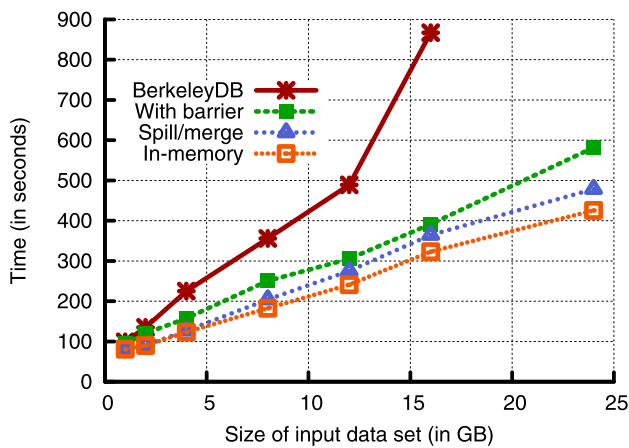
Fig. 13 WordCount with different memory management techniques with increasing dataset size

Table 2 Programmer effort, in terms of LOC required to convert MapReduce applications to their barrier-less versions

Application	Lines of code (LOC)		
	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%

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 13 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.

6.6 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 ver-

sion. 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 [5], statistical machine translation [4–9], optimization [24], finance [3], and similarity scoring [10]. 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 investigate the transition between the Map and Reduce stages. In MapReduce Online [6], data is pushed by Mappers and merge-sorted by Reducers at a finer granularity. This has two main advantages. First, this effectively overlaps (pipelines) the stages, resulting in faster runtime. Our work can similarly benefit from such an optimization. Second, this allows early partial results (snapshots) to be computed to approximate the correct result. Each snapshot must be computed independently of others and furthermore, previous snapshots do not help in computing subsequent results. In contrast, by storing partial results we are able to decrease the time needed to compute the actual final result.

Similar techniques like incremental updates of materialized views [14], hash-based joins [8], etc. have been explored by the parallel database community. However, such techniques can only be applied to a select group of operators, while our technique can be applied to many arbitrary MapReduce programs. Also, MapReduce programs benefit from scalability and fault tolerance at the cost of some performance overhead as compared to parallel DBs. There is a considerable amount of debate on MapReduce vs parallel databases [22]. Pavlo et. al [20] perform a comparison of MapReduce and parallel databases for large scale data analysis and demonstrate the overhead of materializing the intermediate data in MapReduce.

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 [27]. Other work has aimed to be completely transparent to the programmer [28]. Our work is a combination of both categories. We have preserved the baseline MapReduce API, while empowering the programmer to improve performance by relaxing assumptions in the Reduce function.

Dryad [16] is a distributed platform that has been developed at Microsoft to provide large-scale, parallel, fault-tolerant execution of processing tasks. 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.

There has been recent interest in dealing with outliers in MapReduce clusters. The Longest Approximate Time to End (LATE) scheduling algorithm [28] was designed to provide better performance in the face of outliers. The algorithm shows improved performance in virtualized, heterogeneous clusters. The Mantri system [2] is also designed to target MapReduce outliers to improve performance. The system uses real-time progress reports to detect outliers early in their lifetime, and takes corrective action depending on the cause of the outlier and the available resources in the cluster. While not the main goal, our barrier-less implementation is shown to reduce outliers and their effect on job completion time, in homogeneous conditions and under some types of heterogeneity.

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. This is because our barrier-less MapReduce framework allows the interleaving of network I/O and computation while keeping disk usage to a minimum. At the same time, 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 [21] becomes feasible in the barrier-less model.

References

1. Apache hadoop. http://hadoop.apache.org (2011)
2. Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., Harris, E.: Reining in the outliers in map-reduce clusters using mantri. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, pp. 1–16. USENIX Association, Berkeley (2010)
3. Black, F., Scholes, M.S.: The pricing of options and corporate liabilities. J. Polit. Econ. **81**(3), 637–654 (1973)
4. Brants, T., Popat, A.C., Xu, P., Och, F.J., Dean, J.: Large language models in machine translation. In: Proc. of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), pp. 858–867 (2007)
5. Chu, C., Kim, S., Lin, Y., Yu, Y., Bradski, G., Ng, A., Olukotun, K.: Map-reduce for machine learning on multicore. In: Advances in Neural Information Processing Systems (NIPS), pp. 281–288 (2007)
6. Condie, T., Conway, N., Alvaro, P., Hellerstein, Elmeleegy, K., Sears, R.: Online aggregation and continuous query support in mapreduce. In: NSDI'10: Seventh USENIX Symposium on Networked Systems Design and Implementation. USENIX Association, Berkeley (2010)
7. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 137–149 (2004)
8. DeWitt, D., Gerber, R.: Multiprocessor hash-based join algorithms. In: Proc. of VLDB Conference. Citeseer, Princeton (1985)
9. Dyer, C., Cordova, A., Mont, A., Lin, J.: Fast, easy, and cheap: construction of statistical machine translation models with mapreduce. In: Proc. of the Third Workshop on Statistical Machine Translation (StatMT), pp. 199–207. Association for Computational Linguistics, Stroudsburg (2008)
10. Elsayed, T., Lin, J., Oard, D.W.: Pairwise document similarity in large collections with mapreduce. In: Proc. of the Annual Meeting of the Association for Computational Linguistics on Human Language Technologies (HLT), pp. 265–268. Association for Computational Linguistics, Stroudsburg (2008)
11. Farivar, R., Verma, A., Chan, E., Campbell, R.: Mithra: multiple data independent tasks on a heterogeneous resource architecture. In: Proc. of IEEE International Conference on Cluster Computing and Workshops (CLUSTER), pp. 1–10 (2009). doi:10.1109/CLUSTR.2009.5289201
12. Fix, E., Hodges, J.: Discriminatory analysis, nonparametric discrimination: Consistency properties. Tech. rep., USAF School of Aviation Medicine, Randolph Field, Texas (1951)
13. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: Proc. of Symposium on Foundations of Computer Science, pp. 8–21 (1978). doi:10.1109/SFCS.1978.3
14. Gupta, A., Mumick, I.: Maintenance of materialized views: problems, techniques, and applications. Bull. Tech. Commun. **51**, 3 (2010)
15. Illinois Cloud Computing Testbed. http://cloud.cs.illinois.edu/ (2011)
16. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: Proc. of European Conference on Computer Systems (EuroSys), pp. 59–72 (2007)
17. Mongodb. http://www.mongodb.org (2011)
18. Olson, M.A., Bostic, K., Seltzer, M.: Berkeley db. In: Proc. of the USENIX Annual Technical Conference (ATEC), p. 43. USENIX Association, Berkeley (1999)
19. O'Malley, O., Murthy, A.C.: Winning a 60 second dash with a yellow elephant. http://sortbenchmark.org/Yahoo2009.pdf (2009)

20. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., Dewitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proc. of the ACM SIGMOD International Conference. ACM (2009). URL http://database.cs.brown.edu/sigmod09/benchmarks-sigmod09.pdf
21. Popa, L., Budiu, M., Yu, Y., Isard, M.: DryadInc: Reusing work in large-scale computations. In: Proc. of Workshop on Hot Topics in Cloud Computing (HotCloud) (2009)
22. Stonebraker, M., Abadi, D., DeWitt, D., Madden, S., Paulson, E., Pavlo, A., Rasin, A.: MapReduce and parallel DBMSs: friends or foes? Commun. ACM **53**(1), 64–71 (2010)
23. Tokyo cabinet. http://1978th.net/tokyocabinet/ (2011)
24. Verma, A., Llora, X., Goldberg, D.E., Campbell, R.H.: Scaling genetic algorithms using mapreduce. In: Proc. of International Conference on Intelligent Systems Design and Applications (2009)
25. Verma, A., Zea, N., Cho, B., Gupta, I., Campbell, R.: Breaking the MapReduce stage barrier. In: Proc. of IEEE International Conference on Cluster Computing (CLUSTER), pp. 235–244 (2010)
26. White, T.: Hadoop: The Definitive Guide. O'Reilly Media, Sebastopol (2009)
27. Yang, H., Dasdan, A., Hsiao, R., Parker, D.: Map-reduce-merge: simplified relational data processing on large clusters. In: Proc. of the 2007 ACM SIGMOD International Conf. on Management of data (2007)
28. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R., Stoica, I.: Improving mapreduce performance in heterogeneous environments. In: Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2008)

Abhishek Verma is currently a Ph.D. student in the Computer Science department at the University of Illinois at Urbana-Champaign. He received his M.S. in Computer Science from the same university in 2010. Prior to that, he earned his B.Tech. in Computer Science and Engineering from NIT Trichy, India in 2008. His research interests include large scale data intensive processing and performance modeling.

Brian Cho is currently a Ph.D. student in the Computer Science department at the University of Illinois at Urbana-Champaign. He received his B.S. in Computer Science from the same university in 2007. His research interests are in cloud and data center protocols and applications.

Nicolas Zea received his B.S. in Electrical and Computer Engineering from the University of Texas at Austin and his M.S. from the University of Illinois at Urbana-Champaign. His research interests lie in error resilient computer architectures and parallel programming systems, with a focus on non-shared memory parallelism across many-core architectures and distributed systems. He is currently employed as a software engineer at Google Inc. since 2010.

Indranil Gupta is Associate Professor of Computer Science at the University of Illinois (Urbana-Champaign), where he leads the Distributed Protocols Research Group. He received his Ph.D. in Computer Science from Cornell University in 2004. He is recipient of the NSF CAREER award in 2005. Indranil served as PC co-chair at ACM/IFIP/Usenix Middleware 2010, IEEE SASO 2010 and StoDiS 2005; General Chair of ACM PODC 2007; and Associate Vice Chair at ICDCS 2008.

Roy H. Campbell is Sohaib and Sara Abbasi Professor of Computer Science at the University of Illinois (Urbana-Champaign), where he leads the Systems Research Group. He received his Ph.D. in Computer Science from University of Newcastle upon Tyne in 1977. His current research projects include security assessment of SCADA networks, operating system dependability and security, active spaces for ubiquitous computing, and the design of peer-to-peer distributed operating systems.