MapReduce with Communication Overlap (MaRCO)

Faraz Ahmad, Seyong Lee, Mithuna Thottethodi and T. N. Vijaykumar

ABSTRACT

MapReduce is a programming model from Google for cluster-based computing in domains such as search engines, machine learning, and data mining. MapReduce provides automatic data management and fault tolerance to improve programmability of clusters. MapReduce's execution model includes an all-map-to-all-reduce communication, called the shuffle, across the network bisection. Some MapReductions move large amounts of data (e.g., as much as the input data), stressing the bisection bandwidth and introducing significant runtime overhead. Optimizing such shuffle-heavy MapReductions is important because (1) they include key applications (e.g., inverted indexing for search engines and data clustering for machine learning) and (2) they run longer than shuffle-light MapReductions (e.g., 5x longer). In MapReduce, the asynchronous nature of the shuffle results in some overlap between the shuffle and map. Unfortunately, this overlap is insufficient in shuffle-heavy MapReductions. We propose MapReduce with communication overlap (MaRCO) to achieve nearly full overlap via the novel idea of including the reduce in the overlap. While MapReduce lazily performs reduce computation only after receiving all the map data, MaRCO employs eager reduce to process partial data from some map tasks while overlapping with other map tasks' communication. MaRCO's approach of hiding the latency of the inevitably high shuffle volume of shuffle-heavy MapReductions is fundamental for achieving performance. We implement MaRCO in Hadoop's MapReduce and show that on a 128-node Amazon EC2 cluster, MaRCO achieves 23% average speedup over Hadoop for shuffle-heavy MapReductions.

INDEX TERMS

Cloud computing, parallel computing, MapReduce, performance optimization.

1 INTRODUCTION

The explosion of information on the Internet and the World-wide Web has led to commercially-important problems in the Internet- and Web-computing domains such as search engines, machine learning, and data mining. Algorithms for solving these problems process massive amounts of data (e.g., terabytes) and exhibit abundant and simple parallelism. As such, the algorithms are suitable for inexpensive large-scale clusters of commodity computers.

While the cluster-based approach achieves cost-effective performance, data management (distribution and movement), and fault tolerance performed manually by the programmer degrade programmability. Faults (software bugs or hardware failures) are not uncommon because most of the problems run for a long time even on a large cluster (e.g., a few hours on a 1000-node cluster).

MapReduce [11], the celebrated new programming model from Google inspired by functional languages' *map* and *reduce*, addresses these two programmability issues by providing automatic data management among the cluster nodes and transparent fault detection and recovery. MapReduce programs have a fixed structure where the input data is *mapped* into a set of <*key, value*> tuples and then the values for each key are *reduced* to a final value. Because of parallelism within map and within reduce computations, multiple map and reduce tasks are run in parallel. However, the programmer merely specifies the map and reduce functions, whereas task management, data management, and fault tolerance are handled automatically. Despite the fixed structure of its programs, MapReduce captures many important application domains in Internet and Web computing. Despite recent debate on the relative benefits of MapReduce over databases [19], MapReduce remains attractive for enterprises that handle large amounts of *fast-changing* data including Google [11], Yahoo [13], Microsoft [16],[24] and Facebook [15], as discussed in Section 4.

While MapReduce's *programming model* has two phases — map and reduce, MapReduce's *execution model* has four phases. In the first phase, the map computation operates over input data and emits the <*key,value*> tuples. This phase is completely parallel and can be distributed easily over a cluster. The second phase performs an all-map-to-all-reduce personalized communication [17] in which all the tuples for a particular key are sent to a single reduce task. Because there are usually many more unique keys than reduce tasks, each reduce task may process more than one key. The third phase sorts the tuples on the key field essentially grouping all the tuples for the same key. This grouping does not occur naturally because each reduce task receives tuples from all the map tasks and the tuples for the different keys meant for a reduce task may be jumbled. Finally, the fourth phase of reduce computation processes all the tuples for the same key and produces the final output for the key.

Because of the abundant parallelism within map and within reduce, and because of reduce's dependence on map, map tasks occupy the entire cluster followed by reduce tasks instead of space-sharing the cluster with reduce tasks. Consequently, the all-map-to-all-reduce personalized communication, called *shuffle* [11], amounts to all-nodes-to-all-nodes communication which crosses the network bisection. The shuffle moves data from the map nodes' disks rather than their main memories, through the cluster network, to the disks of the reduce nodes, incurring both disk and network latencies. While *shuffle-light* MapReductions move only small amounts of data (e.g., a small fraction of the input data), *shuffle-heavy* MapReductions move large amounts of data (e.g., as much as the input data), stressing the disk and network bisection bandwidths. We observe that a MapReduction's functionality fundamentally impacts its shuffle volume. Shuffle-light MapReductions correspond to data summarization tasks (e.g., counting occurrences) which naturally produce much

less output than input, whereas shuffle-heavy MapReductions correspond to data re-organization (e.g., inverted indexing, sorting) which tend to output as much as or more than the input. Shuffle-heavy MapReductions incur considerable performance overhead (e.g., 30-40%) due to the high data volume transferred using affordable disk and network bisection bandwidths of commodity parts. Specifically, network bisection for large clusters (e.g., 55 Mbps per node for a 1800-node cluster [22]) is scarcer than local disk bandwidth (e.g., 800 Mbps). Optimizing shuffle-heavy MapReductions is important because (1) they include key applications such as inverted-index in search engines and k-means in machine learning and (2) they take long to complete due not only to long shuffle times but also to long reduce times for processing the large shuffle data (e.g., *sort* runs five times longer than *grep* [11]). In contrast, shuffle-light MapReductions are short-running with much less need and opportunity for optimization.

Many straightforward options for this problem are not effective. (1) Reducing the overhead via brute-force approaches to improve bisection bandwidths are inherently not scalable. While specialized disk drives and networks may deliver higher bandwidths, commodity hardware [8] is more cost-effective, a key consideration for MapReduce. (2) In MapReduce's parallel execution, a node's execution includes map computation, waiting for shuffle, and reduce computation, and the critical path is the slowest node. To improve performance, the critical path has to be shortened. However, the natural overlap of *one* node's shuffle wait with *another* node's execution does not shorten the critical path. Instead, the operations within *a single node* need to be overlapped. (3) Simultaneously running multiple MapReductions may hide one job's shuffle under another job's computation and improve cluster utilization, however, our goal here is to improve mapreduction latency and not data center throughput because (a) latency is important in timing critical production runs which often chain multiple dependent mapreductions many of which are shuffle-heavy and (b) reducing latency also improves throughput (as a good side effect) whereas multitasking multiple jobs usually requires more memory to hold multiple program state. (4) In MapReduce [11,13], the asynchronous nature of the shuffle combined with the fact that there are multiple map tasks available at every node (to exploit within-map parallelism) results in some overlap between the shuffle and map (i.e., earlier map tasks' communication with later map tasks' computation). Unfortunately, overlap with map computation is insufficient in shuffle-heavy MapReductions because shuffle time is longer than map computation time (shuffle time is 30-40% of *all* computation but is longer than map computation *alone*). (5) Alternately, overlapping multiple map tasks' communications with each other (i.e., earlier map tasks' communication with later map tasks' communication) is not viable because the network bisection is saturated.

Instead of the above unviable options, we propose *MapReduce with communication overlap (MaRCO)* based on the novel idea of overlapping the shuffle with the reduce computation. MaRCO achieves nearly full overlap of the shuffle and is based on the following four key observations: (1) Because reduce computation is long in shuffle-heavy MapReductions, MaRCO achieves better overlap than that with map alone. (2) Because reduce generates no network traffic, MaRCO is not constrained by network saturation. (3) Though reduce is dependent on shuffle data, reduce can start operating on *partial* data without waiting for *all* the data. Because typical reduce functions are commutative and associative, their partial results can be re-reduced easily without any ordering constraints to produce the correct final output. Accordingly, MaRCO breaks up reduce into many smaller invocations on partial data from some map tasks while overlapping with other map tasks' shuffle latency. Thus, MaRCO's *eager* reduce overlaps with the shuffle while the original MapReduce execution's *lazy* reduce starts only after receiving *all* the shuffle data. We note that while overlapping independent operations is common in computer systems, overlapping dependent operations by exploiting commutativity and associativity as done by MaRCO is rarer. (4) In MaRCO, a final reduce step re-reduces all the partial reduce outputs to produce the final output. Fortunately, the partial reduce closely resembles the original reduce and the final reduce is often identical to the original reduce. Therefore, the extra programming effort under MaRCO compared to that under MapReduce is minimal. Thus, MaRCO maintains MapReduce's programmability advantages. Furthermore, MaRCO improves performance on commodity cluster hardware without increasing the hardware cost or complexity. Finally, blindly applying partial reduce may introduce overheads for which we employ several control mechanisms.

We evaluate MaRCO by extending the public-domain MapReduce implementation from Hadoop [13] and we evaluate eleven MapReductions including a mix of shuffle-heavy and shuffle-light MapReductions. Using a 128-node Amazon EC2 cluster, we show that on average MaRCO achieves 23% and 14% speedups over Hadoop on shuffle-heavy and all MapReductions, respectively. We provide arguments and experimental evidence that our speedups will likely scale to larger clusters.

In summary, the key contributions of this paper are:

- identifying the problem of exposed shuffle in shuffle-heavy MapReductions;
- addressing the problem via the novel idea of fully overlapping the shuffle with the reduce;
- employing control mechanisms to avoid overheads due to this overlap; and
- significantly speeding up shuffle-heavy MapReductions in a complete MapReduce implementation running on a real-world cluster.

The rest of the paper is organized as follows. We provide background on MapReduce in Section 2. We describe MaRCO in Section 3, some related work in Section 4, and our experimental methodology in Section 5. We present our results in Section 6 and conclude in Section 7.

2 MAPREDUCE: BACKGROUND

We briefly describe MapReduce with an illustrative example. In addition to the sequencing of the various computation and communication steps in MapReduce, the description also points out the sources of communication delays and the limited map-shuffle overlap in MapReduce ([11,13]).

Consider the following computation expressed as a MapReduction. The input is the directed graph *G* in Figure 1(a) specified as a set of edges (each edge represented as a <*u*,*v*> tuple of two vertices as shown in Figure 1(b)). The goal is to arrive at the adjacency list representation of the reverse graph *G'* (i.e., *G'* contains edge

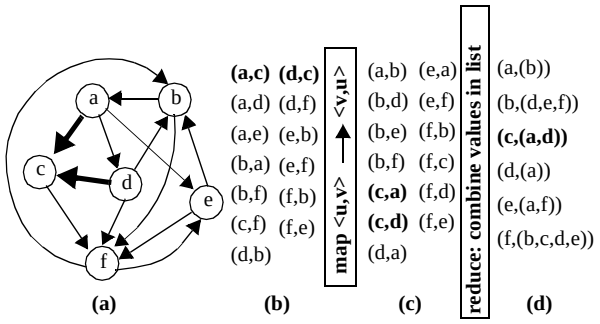

FIGURE 1: MapReduce Example

<*v*,*u*> if *G* contains <*u*,*v*>).

In the two-phase MapReduce programming model, the above computation can be specified as (1) a Map function that emits a tuple <*v*,*u*> (key is *v* and value is *u*) for every edge <*u*,*v*> in *G* which results in the intermediate data shown in Figure 1(c) and (2) a reduce function that creates a combined list of all the unique values associated with a key and emits the <*v, list(u)*> as the final output, as shown in Figure 1(d). The example highlights (shown in bold) the MapReduce functionality on the edges incident on node *C*.

To illustrate the execution of the above example on a cluster, we consider a cluster with two machines ($N = 2$) and we assume four map tasks ($M = 4$) and two reduce tasks ($R = 2$). (In practice, the choice of the number of tasks depends on a number of factors including task-granularity for load balance and re-execution granularity for failure tolerance.) Figure 2 illustrates the execution timeline on each of the two nodes. The activity on each node is further divided into two rows with the top row corresponding to map task execution and the bottom row corresponding to reduce task execution.

Because the map function is, by definition, an SPMD program that is applied to all input data, the input data is partitioned into *M* pieces each of which is processed by a map task (*M1* through *M4* in Figure 2). To facilitate scheduling of the *M* map tasks to the *N* nodes, the input data is initially placed on a global file system which makes all input data available to all nodes. However, because the global file system is typically implemented by using the local disks of the machines in the cluster [3,12], access times may be non-uniform. Consequently, to maximize locality, the runtime system attempts to schedule map tasks to nodes where the data is local. Such a schedule is not always guaranteed and there may be cases where a map task has to read data from a remote node. These remote accesses are one source of network communication, though the runtime scheduling often succeeds making this communication uncommon.

Each map task's output data must eventually be communicated to the various reduce tasks such that two conditions are satisfied. First, tuples with the same key must be sent to the same reduce task. Second, the load across reduce tasks must be balanced. To satisfy the above two conditions, each output tuple is assigned to one of *R* buckets (one per reduce task) using a hash function on the key. For our example in Figure 2, each map task creates two hash buckets ($^iR_1$ and $^iR_2$ for the i^{th} map task) because $R = 2$. Because all map tasks use the same hash function, tuples with the same key are mapped to the same reduce task. Because the hash-function is expected to map evenly across all *R* tasks, the load (in terms of number of tuples) assigned to reduce tasks is more-or-less balanced. The intermediate data (i.e., all the buckets) is held in the local disk and is not written to the global file system due to a fault-tolerance-related trade-off discussed at the end of this section. After each map task is complete, reduce tasks must pull the relevant hash-bucket of intermediate data from each map task. This all-map-to-all-reduce communication, called *shuffle*, includes disk reads at the map tasks' nodes, the network traversal, and disk writes at the reduce tasks' nodes. As mentioned in Section 1, because of the abundant parallelism within map and reduce, and because of reduce's dependence on map, the map tasks occupy the entire cluster followed by the reduce tasks instead of space-sharing the cluster with reduce tasks. Consequently, the shuffle amounts to all-nodes-to-all-nodes communication which crosses the network bisection. Thus, the shuffle is a significant problem for MapReductions that produce a lot of intermediate data. Because MapReduce requires the shuffle to complete before sort and reduce computation begins, the shuffle lies on the critical path.

To reduce the volume of the shuffle, MapReduce uses an optimization called *combining* which performs a reduce operation on each of the *R* hash buckets of a map task's intermediate data. The reduce operation used for combining is often, but not always, the same as the programmer-specified reduce function. While the requirement to specify the combiner may seem like an additional burden on the programmer, in most cases the combiner is intuitive and provides reasonable performance benefits. Figure 2 illustrates the combiner acting on the *R* buckets of intermediate data emitted by each map task and re-emitting the same number of buckets with potentially fewer tuples in each bucket.

The shuffle is asynchronous in nature which results in some overlap between the shuffle and map tasks. Because reduce tasks may not be scheduled to run when map tasks complete, it is not possible for map tasks to directly push the shuffle data to reduce tasks. Due to its pull-based nature, the shuffle is effectively asynchronous from the perspective of the map task because subsequent map tasks (*M3* and *M4*) can begin execution without waiting for the shuffle of the prior map tasks (*M1* and *M2*) to complete, as illustrated in Figure 2. This asynchrony leads to overlap of map computation with the shuffle. While the figure corresponds to our example with 2 map tasks per worker node ($M/N = 4/2 = 2$), in general there are multiple map tasks per node. Because the asynchrony exists irrespective of whether a node's map tasks are run sequentially or in some concurrent manner, the map-shuffle overlap exists independent of map tasks being sequential or concurrent. Nevertheless, if the map execution time is less than the shuffle time (typical for shuffle-heavy MapReductions), there remains some exposed shuffle which grows as the number of map tasks per node grows (shaded regions in the shuffle rectangles in Figure 2). As mentioned in Section 1, one caveat about Figure 2 is that the map-shuffle overlap within a node improves performance but the natural map-shuffle overlap *across* nodes does *not* because the critical path through a *single* node determines overall execution time.

After the shuffle is complete, each reduce task sorts all its data to group tuples with the same key. In the example, reduce task *R1* sorts (by the key) its input data ($^iR_1$ for $i = 1$ thru *4*). In addition to grouping tuples with the same key, sorting has the added advantage that the output data can be searched easily — an important consideration when MapReduce is used for indexing. Hadoop performs some in-memory pre-sorting of intermediate data while waiting for the rest of the intermediate data to be received at the

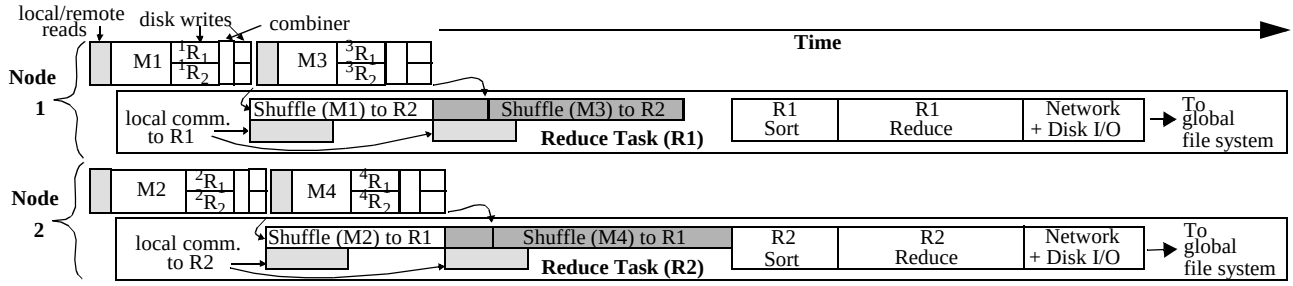

FIGURE 2: MapReduce Execution Time Line (M=4, R=2, N=2)

reduce task. Further, Hadoop opportunistically performs in-memory merging of pre-sorted intermediate data at each reduce task. Both these optimizations essentially place a part of the sort computation in parallel with map computation, allowing the critical path to be shortened. Finally, after sorting the intermediate data, reduce computation is performed on each such set of tuples with the same key. Another reason for the long execution times of shuffle-heavy MapReductions (other than the shuffle bottleneck) is that the volume of intermediate data directly translates to long sort and reduce times. The final reduce task output is written to the global file system which involves communication over the network to remote nodes. Because the final output is buffered and the actual communication occurs whenever the buffers are full, the communication is interleaved with final reduce computation. Using asynchronous writes can overlap the interleaved output communication and reduce computation which would be beneficial for shuffle-heavy MapReductions whose outputs are large. Therefore, we modified Hadoop to employ asynchronous writes.

Handling failures is one of the key motivations for MapReduce. To that end, fault tolerance for input and final output data is achieved via replication in the underlying global file system. To achieve fault tolerance for computation and intermediate data (from map tasks), MapReduce uses re-execution. When a map task fails without completing, the task is re-executed on another machine. Because the input data on the global file system is replicated (and is not lost due to the failure), re-execution on other machines is always an option. Note that the intermediate data, which is present only on the local file system becomes unavailable when a machine fails necessitating re-execution of even completed map tasks. This design choice, previously mentioned in this section, is made in recognition that re-execution of map task is less expensive than replication of the intermediate data because replication cost is incurred each time whereas re-execution cost is incurred only in case of failures. Alternate designs have been proposed in other contexts where the decision to replicate or not may depend on whether re-execution is more expensive than replication [9]. Finally, MapReduce uses re-execution to handle the tasks on slow nodes that have not failed (i.e., the nodes respond to the master node, but their map/reduce tasks lag far behind similar tasks on other nodes). In such cases, *back-up tasks* are launched pro-actively assuming the original tasks are stuck due to some machine-specific performance glitch and results from the earliest completing task are used.

3 MaRCO

The goal of MaRCO is to overlap the significant exposed shuffle delays (for MapReductions with large amounts of intermediate data) with useful computation. As mentioned in Section 1 and Section 2, the asynchronous nature of pull-based shuffle results in some shuffle-map overlap, but the overlap is insufficient if a map computation time is less than the shuffle delay. To increase overlap and reduce the critical path on individual nodes in the cluster, MaRCO eagerly executes reduce computation which offers more work to overlap with the shuffle (Observation 1 from Section 1). Because reduce computation operates on sorted data, our eager reduce involves some eager sorting, called as *partial sort*. Eager execution of sort and reduce is possible because intermediate data begins arriving at the reduce task soon after the first map task completes (and continues as subsequent map tasks complete). An added advantage of overlapping the shuffle with partial sort and reduce is that, unlike map tasks, partial reduce do not generate any additional network traffic (Observation 2 from Section 1).

Figure 3 illustrates the execution time line of MaRCO on a two-node cluster with map tasks shown in the top row and reduce tasks shown in the bottom row for each node. Similar to Figure 2, the rectangle of each map task contains (in order) the delay of reading input data which may be local or remote (cross-hatched rectangle), the map computation, the emitting of *R* hash buckets of intermediate data, the operation of a combiner and the emitting of *R* combined hash buckets of intermediate data. The partial reduce tasks at every node overlap with the shuffle (and run concurrently with the node's on-going map tasks). Because of the work done by the partial sort and reduce tasks, the final sort and reduce tasks complete earlier than in MapReduce (Figure 2), resulting in the shortening of the critical path of execution.

We now describe MaRCO's implementation of eager partial sort (Section 3.1) and partial reduce (Section 3.2) which goes significantly beyond the limited forms of eager sort/reduce (i.e., combiners, presorting, in-memory merging) that exist in MapReduce as described in Section 2.

3.1 Eager Partial Sort

Google's MapReduce implementation, as described in [11], lazily sorts the intermediate data after all the data is received. Hadoop achieves a limited form of eager in-memory sorting and merging before writing sorted runs to disk. These sorted runs can span multiple map tasks' intermediate data depending on the size of the in-memory buffer. The creation of sorted runs converts the final sort to a merge of the multiple sorted runs.

In contrast, MaRCO's eager sorting approach further merges the disk-resident sorted runs (created by the in-memory merge) without waiting for all intermediate data. This merging creates longer and fewer sorted runs, which decreases the amount of work to be done in the final sort. Further, MaRCO's eager sorting benefits from caching

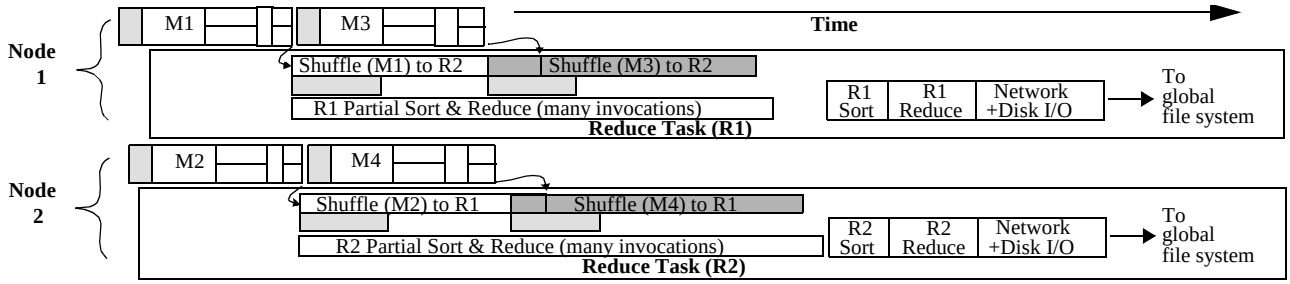

FIGURE 3: MaRCO Execution Time Line (M=4, R=2, N=2)

effects when eager partial sort may find the sorted runs in memory. Because MaRCO uses both eager partial sort and partial reduce in conjunction, partial sort is invoked only when a partial reduce is invoked. The precise policy that controls when, and over what data, partial reduce (and thus partial sort) operates upon is explained below in Section 3.2. Note that the merging performed by eager partial sort is similar to what an external merge sort would achieve in Hadoop, except that the merging is staggered over the duration of the shuffle.

3.2 Eager Partial Reduce

Similar to partial sorting, reduce functionality can also be applied without waiting for all intermediate data to be received. Because the partial reduce operates on subsets of tuples, the functionality of the partial reduce in MaRCO must be commutative and associative and there must be a final reduce operation to re-reduce all the partially-reduced data. This requirement is not a problem in practice as common reduce functions are often commutative and associative, as mentioned in Section 2. In such cases, the partial reduce, the combiner and the final reduce could, in theory, be the same. In practice, however, because the combiner operates on only one map task output while the partial reduce combines several map task outputs, the combiner is light-weight (e.g., duplicate elimination or summation) whereas the partial reduce is more heavy-weight (e.g., set union, list building) to match their respective opportunities. Indeed, the partial reduce and the combiner functions are different for *most* of our shuffle-heavy MapReductions. For instance, in our benchmark term-vector, combiner combines n <word, 1> into <word,n> but partial-reduce builds tuples of the form <host, {$word_1$:$count_1$, $word_2$:$count_2$, ..., $word_k$:$count_k$}> (See Table 1). When the reduce function is non-commutative or non-associative, an alternate partial reduce function can be specified in MaRCO. The additional burden on the programmer to extract and specify the appropriate commutative and associative partial reduce function from the full (and potentially non-commutative or non-associative) reduce function is only slightly higher than the burden imposed by the use of combiners which MapReduce implementations already support (Observations 3 and 4 in Section 1).

Beyond the difference in functionality, combiners and the partial reduce differ at a more fundamental level. First, the combiners are most effective in shuffle-light MapReductions which summarize data and hence are amenable to combining. However, because data reorganization does not lend itself to combining, the combiners achieve only marginal reduction in the shuffle volume in shuffle-heavy MapReductions, even if applied to tuples of multiple map tasks. Therefore, the partial reduce's ability to hide the latency of the inevitably high shuffle volume is fundamental for achieving performance in shuffle-heavy MapReductions. Thus, the partial reduce is a new tool to improve MapReduce's performance for the important class of shuffle-heavy MapReductions. Second, while invoking the combiner is straightforward, the partial reduce introduces overhead (both CPU and disk I/O) that can hurt performance if the invocation and scheduling are not controlled properly (e.g., uncontrolled partial reduce incurs 11% average slowdown over MapReduce). There are three types of such overhead which MaRCO controls by employing three mechanisms.

3.3 Controlling Partial Reduce Overhead

First, unlike eager partial sort where there was no increase in the total amount of work compared to lazy sorting (any merging of sorted runs achieved by the partial sort eliminates an equivalent merge that must occur at the final sort), the partial reduce introduces overheads. Specifically, the writing of the partially reduced output to disk and reading that data in the final reduce operation are both overheads that do not occur in the lazy reduce version. Accordingly, we observe that for the partial reduce to do useful work and decrease the burden on the final reduce, the partial reduce must operate on multiple tuples with the same key and emit a single reduced tuple. Reading tuples and emitting them back unchanged does not decrease the amount of work for the final reduce. Instead, given that the combiners already eliminate duplicate keys in the output of a *single* task, tuples with the same key must be aggregated across the intermediate data of *multiple* map tasks. To this end, we impose a minimum threshold, called *start_threshold*, the number of map task outputs that must be received at a reduce task before a partial reduce can commence. The choice of the threshold is driven by a trade-off between the overheads and the exposed shuffle. On one hand, waiting for the intermediate data of a large number of map tasks (i.e., setting a high value for *start_threshold*) results in wasted opportunity where the exposed shuffle is not overlapped with the partial sort/reduce. But setting a high value for *start_threshold* is likely to decrease overheads because it is more likely that multiple tuples with the same key will be encountered when larger sets of intermediate data are considered. Current MapReduce implementations can be interpreted as one extreme of this trade-off, effectively minimizing overhead but incurring the penalty of all the exposed shuffle. On the other hand, beginning the partial reduce operations on data from only a few map tasks increases the overlap of the partial reduce with the shuffle communication of early map tasks but decreases the amount of useful work accomplished in the partial reduce because there may be fewer tuples with the same key. We experimentally determined the appropriate threshold *start_threshold* to be *8*. Finally, we observed that shuffle

data is initially slow (because all map tasks do not finish at the same time). To accommodate this slow-start phenomenon, we set *start_threshold* to *4* for the first two partial reduce invocations and *8* for subsequent partial reduce invocations. While Figure 3 shows the case of *start_threshold* = 0 where *R1* (and *R2*) *partial sort and reduce* start before receiving any intermediate data, a non-zero value of *start_threshold* would imply that *R1* (and *R2) partial sort and reduce* would start later — after enough data is received.

Second, the partial reduce computation must remain hidden under the shuffle without extending the execution time. Typically, the partial reduce operations that begin when the shuffle is *almost* complete may continue execution even *after* the shuffle is complete. Avoiding such non-overlapped partial reduce is important because such partial reduce extends overall execution time irrespective of whether the partial reduce does useful work or not. In cases where the partial reduce achieves little useful work (i.e., there is little decrease in the number of tuples after the partial reduce due to the nature either of the partial reduce function or of the data), the non-overlapped partial reduce obviously extends execution time. Even in cases where the partial reduce does useful work, the final reduce is a more efficient way to perform the computation because the final reduce has no overheads. MaRCO addresses this problem by prohibiting further partial reduce invocations after a fraction, called *stop_fraction*, of the shuffle data is received. However, the on-going invocations are not aborted.

Third, the partial reduce must not hinder map computation by contending for CPU and disk resources. MaRCO employs two schemes to address CPU contention. (1) MaRCO assigns low priority to the partial sort/reduce task, ensuring that the partial sort/reduce is scheduled only when map tasks are not runnable. (2) While our description of MaRCO assumed a single core per node and a single reduce task per node, in general there may be multiple reduce tasks per node and each node may have a multicore processor. In such cases, MapReduce can run as many reduce tasks per node as the number of cores (say *C*). Subsequent reduce tasks, if any, are started as and when reduce tasks complete such that at most *C* reduce tasks run at any given time. However, for MaRCO, each reduce task also adds partial reduce computation which often contends with map tasks for CPU resources. To decrease the contention, we begin execution with only one reduce task (and its associated partial reduce computation) instead of all *C* reduce tasks. By the time the first reduce task receives all its shuffle data, all map tasks (on all nodes) must be complete. Consequently, the remaining *(C-1)* reduce tasks can be scheduled after this time without any possibility of interfering with the node's map tasks. The sole purpose of this optimization, called *reduce staggering*, is to avoid contention from partial reduce computation. Therefore, reduce staggering is not applicable to (and does not improve) MapReduce.

MaRCO cannot eliminate disk I/O contention, unlike CPU contention. However, because disk I/O bandwidth is less of a bottleneck than network bisection bandwidth, disk contention results in a modest increase in map execution time which is more than compensated by the overlapping of useful work with the shuffle.

We end our description with three observations. First, though our description of MaRCO is limited to a single level of partial reduce operations before the final reduce, it is possible to have multiple levels of partial reduce where each upper level partial reduce operates on the outputs of lower-level partial reduce to further increase the amount of computation that is available for overlapping with the shuffle. We did not explore this option as a single level of partial reduce was adequate to overlap all the shuffle in our benchmarks.

Second, one may think that MaRCO reduces MapReduction execution time by increasing CPU and I/O utilization due to partial sort/reduce (including overheads) and that such increased utilization may adversely affect the throughput of other applications that may be sharing the cluster. However, because the partial reduce tasks are deprioritized, they do not steal CPU or I/O from other applications when resources are heavily utilized. On the other hand, when the cluster is underutilized, MaRCO exploits idle resources to improve the latency of shuffle-heavy MapReductions.

Finally, MaRCO *does not make any changes* to MapReduce's basic re-execution based fault-tolerance mechanism. On the input end of the partial sort/reduce, MaRCO does not modify the shuffle implementation. As such, any map task failures are handled transparently by the shuffle implementation, which automatically causes map task re-execution if any intermediate data cannot be retrieved. Note that node failures after intermediate data has been retrieved do not cause any re-execution of map tasks. On the output end, our design choice of emitting partial sort/reduce output to local disk implies that the work done by partial sort/reduce is lost when a machine fails. Recall that this choice is driven by the same rationale that keeps map tasks' intermediate data in local disks. Thus, entire map tasks and (partial and final) reduce tasks re-execute on failures *exactly* as in MapReduce.

4 RELATED WORK

There have been some follow-on papers on MapReduce. Map-reduce-merge [23] extends MapReduce to include a merge phase after reduce to enable database-join operations. Another paper [25] reduces the unnecessary launch of back-up tasks in heterogeneous clusters, which have fast and slow machines, by observing that tasks on slow machines would lag behind others even in the absence of any performance glitches. **Mantri[26] explores the various causes of laggards in further depth, and develops cause- and resource-aware techniques to act on outliers more intelligently and earlier in their lifetime.** These ideas are orthogonal to MaRCO which overlaps the shuffle with partial sort and partial reduce. ~~**Finally, a recent paper argues that databases perform better than MapReduce [19]. However, the paper concedes that a) while MapReduce provides fault tolerance for both data and computation, databases provide fault tolerance only for data; and b) because loading data is significantly slower in databases than in MapReduce which uses raw data dumps, MapReduce is a better candidate when data changes frequently as is the case for Web data.**~~

Dryad [16] and DryadLINQ [24] offer a framework that is more general than MapReduce with design features that enable (a) efficient database joins, and (b) automatic optimizations within and across MapReductions using techniques similar to query execution planning. A Dryad-based MapReduce implementation can include automatic combining at the node-level across multiple map tasks' output to reduce the shuffle volume. As discussed before, partial reduce's latency hiding is more important and effective

than such combining for shuffle-heavy MapReductions. Such latency-hiding techniques may be extended to other MapReduce implementations including Dryad-based implementations.

Hadoop proposes to execute the combiner optionally at the reduce side [27] to prune the data that goes to reduce. However, the combiner is usually too light-weight to achieve sufficient overlap with the shuffle, and running the combiner at the reduce side is helpful only if the combiner's functionality is a major portion of the reduce functionality. Moreover, combiners are not effective (and hence not applied) for shuffle-heavy MapReductions where much of the shuffle is exposed (details in Section 3.2). Condie et al. [28] propose MapReduce Online where reduce operates on partially received data and emits partial results to the user. Although this scheme makes partial results available to the user earlier in time and could be useful to estimate the output, it is not applicable to a majority of MapReductions where complete result (final reduction) is desired by the user. Verma et al. [29] propose breaking the map output synchronization barrier by omitting the sort phase (i.e., instead of sorting all incoming tuples in one shot, insert each tuple into its matching, unsorted list). However, many applications require sorted output (e.g., for analysis purposes). Further, this approach's match and insert is significantly less computationally efficient than sorting (i.e., increases reduce phase run time and memory footprint).

Finally, in other orthogonal research, MapReduce has been proposed as a viable programming model for multi-cores [10,21] and GPUs [14,18].

5 Experimental Methodology

We evaluate our ideas by modifying Hadoop's MapReduce implementation [13].

5.1 Benchmarks

Because there are only three reasonably-sized MapReductions — *binary-sort*, *word-count*, and *grep* — in the Hadoop release, we wrote eight more covering both shuffle-heavy and shuffle-light categories.

5.1.1 Shuffle-heavy MapReductions

Our shuffle-heavy applications include *binary-sort*, *term-vector*, *inverted-index*, *self-join*, *adjacency-list*, and *k-means*, which are described in Table 1. We note that the partial reduce is different from the combiner for *k-means*, *inverted-index*, *self-join*, and *adjacency-list*, as suggested in Section 1. We use the same light-weight combiner in both Hadoop and MaRCO. One may think that Hadoop would perform better by using the heavy-weight partial reduce as the combiner. However, we found that such usage results in worse performance for Hadoop.

We summarize the input data sizes, dataset descriptions, and benchmark characterization in Table 2. The input is split among multiple map tasks each of which is given is 50 MB of input data (or less if the input is not a multiple of 50 MB). This input split size follows the recommendation in [11] and remains the same for Hadoop and MaRCO, and across the benchmarks. We also show the run times of the base case (described later) to give an idea of how long our benchmarks run. Partial reduce for *binary-sort* is pure overhead (last column) because partial reduce simple emits the tuples without performing any of the final reduce's work. In *term-vector*, partial reduce is only partly useful because the final reduce discards infrequent words some of which are processed by the partial reduce. For the other shuffle-heavy MapReductions, partial reduce is useful and long (because the original reduce is long).

These benchmarks have substantial communication, providing significant opportunity. Though some of that opportunity is hidden under map computation in Hadoop, map computation alone is insufficient to fully hide communication. MaRCO hides most of this communication under the useful work done by partial (in-built) sort and the partial reduce, with the exception of *binary-sort* where the partial reduce is all overhead but the partial sort is useful.

All the benchmarks write their final outputs to the replicated file system which adds some run-time overhead despite the writes being asynchronous. Unfortunately, this work cannot be done in partial reduce and hence cannot be overlapped with the shuffle.

5.1.2 Shuffle-light MapReductions

Our shuffle-light MapReductions include *word-count*, *classification*, *grep*, *histogram-movies*, and *histogram-ratings*, which are described in Table 3.

Because the shuffle and the reduce work are small in these benchmarks, there is little opportunity for MaRCO. Table 2 includes input data sizes, dataset descriptions, and benchmark characterization for these benchmarks. The partial reduces for these MapReductions are either pure overhead (as in *binary-sort*) or are useful and short (because the original reduce is short). We note that the shuffle-heavy MapReductions run much longer than the shuffle-light ones, indicating the importance of optimizing shuffle-heavy MapReductions.

We note that the difference in shuffle volumes between shuffle-light and shuffle-heavy MapReductions arises from the fundamental nature of the MapReductions. Shuffle-light MapReductions correspond to data summarization tasks (e.g., counting, classifying and binning) which naturally produce a lot less output than input, whereas shuffle-heavy MapReductions correspond to data re-organization (e.g., sorting, indexing, and clustering) which tend to output as much as or more than the input, as mentioned in Section 1.

5.2 Implementation

Hadoop implements MapReduce as a run-time system to be linked in with the user-supplied Java classes for map, combiner (optional), and reduce functionality. Hadoop uses a single global manager thread for the whole cluster. The global manager orchestrates the MapReduction execution by (1) assigning map/reduce tasks to per-node local manager threads, (2) monitoring the health of nodes via time-outs, and (3) re-assigning tasks to fault-free nodes upon node failure. Hadoop uses remote procedure call for the shuffle. All disk I/Os use in-memory buffering (75 MB default).

We implement MaRCO which takes user-provided Java methods for map in map class and combiner, partial reduce and final reduce methods in reduce class. To hide the shuffle, MaRCO launches one reduce thread at the start of the map phase. While Hadoop++ concurrently runs 4 map threads and 2 reduce threads on one node (2 CPUs), MaRCO runs 2 map threads and 2 (low-priority) reduce threads using reduce staggering (Section 3.3). MaRCO runs fewer map threads to make room for the partial reduce's

Table 1: Shuffle-heavy benchmarks

Binary-sort is based on NOWsort [7] for sorting <*binary key*,*value*> tuples on the binary keys. The map task is identity function which simply reads the tuples. Because sorting produces as many output records as input records, there is no combiner (Section 2). The sorting occurs in MapReduce's in-built sort while reduce tasks simply emit the sorted tokens. In MaRCO, partial sorting occurs in the built-in sort while partial reduce merely outputs the partially-sorted tokens which are merged by the final reduce. Because the partial reduce does not perform any of the final reduce's work, partial reduce is pure overhead (though small). To distinguish between the *binary-sort* application and the in-built sort, we will refer to them as *binary-sort* and in-built sort, respectively.

Term-vector determines the most frequent words in a host and is useful in analyses of a host's relevance to a search. The map tasks emit <*host, termvector*> tuples where *termvector* is itself a tuple of the form <*word, 1*>. The combiner combines *n* tuples for the same word from *one* map task into one <*word, n*> tuple. The reduce task discards the words whose frequency is below some cut-off and outputs a list of the rest of the words and their counts as a tuple of the form <*host, {word*$_1$*:count*$_1$*, word*$_2$*:count*$_2$*, ..., word*$_k$*:count*$_k$*}*>. In MaRCO, the partial reduce adds up partial counts for a given word at a host and builds partial lists which are merged by the final reduce. Because the final reduce discards some of the (infrequent) words processed by the partial reduce, some of the work done by partial reduce is useless.

Inverted-index takes tuples of the form <*word:file, n*> where *n* is the number of appearances of *word* in *file* and generates lists of files containing a given word in decreasing order of frequency of appearance. *Inverted-index* is similar to constructing a reverse web-link graph for identifying documents containing a given URL instead of a given word. The map tasks produce <*word, {n:file}*> tuples. Because the input already specifies the count for a given word in a file, there is no opportunity for combining counts here. Reduce task builds a list of all the files that contain a given word and the number of occurrences in each file, and produces tuples of the form <*word:{n*$_1$*:file*$_1$*, n*$_2$*:file*$_2$*, ..., n*$_k$*:file*$_k$*}*>. In MaRCO, the partial reduce tasks build partial lists of the above form and the final reduce merges the partial lists.

Self-join is similar to the candidate generation part of the *a priori* data mining algorithm to generate association among *k+1* fields given the set of *k*-field associations [5]. Map tasks receive *k*-sized candidate lists of the form {*element*$_1$*, element*$_2$*, ..., element*$_k$} in alphanumerically sorted order. The map tasks breaks the lists into <*{element*$_1$*, element*$_2$*, ..., element*$_{k-1}$*}*, {*element*$_k$*}*> tuples. The combiner simply removes duplicates within one map task's output. Reduce prepares a sorted list of all the map values for a given key by building <*{element*$_1$*, element*$_2$*, ..., element*$_{k-1}$*}*, {*element'*$_1$*, element'*$_2$*, ..., element'*$_j$*}*> tuples. From these tuples, *k+1-sized* candidates can be obtained by appending consecutive pairs of map values *element'*$_i$*, element'*$_{i+1}$ to the *k-1-sized* key. By avoiding repeating *k-1-sized* key values for every pair of map values in the list, the tuples are a compact representation of the *k+1-sized* candidates set. In MaRCO, partial reduce produces partial sorted lists of a subset of map values and the final reduce merges the partial lists.

Adjacency-list is similar to search-engine computation to generate the adjacency and reverse adjacency lists of nodes of a graph for use by PageRank-like algorithms. Map tasks receive as inputs graph edges <*p,q*> of a directed graph that follows the power law of the World-wide Web. For the input, we assume the probability, that a node has an out-degree of *i*, is proportional to $1/(i^2)$ with an average out-degree of 7.2. Map tasks emit tuples of the form <*q, from_list{p}:to_list{}*> and <*p, from_list{}:to_list{q}*>. The combiner simply removes duplicate tuples within one map task's output. For a given key, reduce generates unions of the respective lists in the *from_list* and *to_list* fields, sorts the items within the union lists, and emits <*x, from_list{sorted union of all individual from_list}:to_list{sorted union of all individual to_list}*> tuples. In MaRCO, partial reduce produces partial sorted unions which are merged together by the final reduce.

k-means is a popular data mining algorithm to cluster input data into *k* clusters[1]. k-means iterates to successively improve the clustering. We classify movies based on their ratings using Netflix's movie rating data [2] which is of the form <*movie_id, list{rater_id, rating}*>. We use random starting values for the cluster centroids. Map computes the cosine-vector similarity of a given movie with the centroids, and determines the centroid to which the movie is closest (i.e., the cluster to which it belongs). Map emits <*centroid_id, (similarity_value, movie_data)*> where *movie_data* is (*movie_id, list{rater_id, rating}).* While *movie_data* increases shuffle volume and is not needed for reduce, the data is needed for the next iteration of *k-means*. Because there is no sum or list involved, there is no opportunity for a combiner. Reduce determines the new centroids by computing the average of similarity of all the movies in a cluster. The movie closest to the average is the new centroid and reduce emits the new centroid's and all movies' tuples to be used in the next iteration. The algorithm iterates until the change in the centroids is below a threshold. In MaRCO, the partial reduce computes partial averages of similarity of a subset of movies in a cluster and the final reduce computes the final averages, and identifies and emits the new centroids.

Table 2: Benchmark Characteristics (* relative to total time)

Benchmark	Input size	Input data	#maps & #reduces	Base runt-ime	Shuf fle vol-ume	Map time*	Red uce time *	Partial reduce
binary-sort	85GB	synthetic, random	1500 & 30	3045 s	high	short	long	pure overhead
term-vector	15GB	Project Gutenberg	300 & 30	3082 s	high	very long	short	useful & short
inverted-index	22GB	Project Gutenberg	330 & 30	1013 s	high	long	long	useful & long
self-join	15GB	synthetic, $k = 5$	300 & 30	977 s	high	long	long	useful & long
adjacency-list	30GB	synthetic	600 & 30	1856 s	high	long	long	useful & long
k-means	15GB	Netflix data, $k = 6$	300 & 6	1550 s	high	long	long	useful & long
word-count	15GB	Project Gutenberg	300 & 30	1448 s	little	very long	short	useful & short
classification	15GB	Netflix data, $k = 6$	300 & 6	308 s	little	very long	short	pure overhead
grep	15GB	Project Gutenberg	300 & 1	203 s	little	very long	short	pure overhead
histogram-movies	15GB	Netflix data	300 & 8	195 s	little	very long	short	useful & short
histogram-ratings	15GB	Netflix data	300 & 5	374 s	little	very long	short	useful & short

CPU utilization. The reduce thread launches child threads which perform partial sort and partial reduce tasks when the number of map outputs exceeds the *start_threshold* (4 for the first two launches and 8 for the rest). To prevent the partial reduce from delaying the final reduce after the map phase is complete (Section 3.2), the child threads are not launched once reduce threads receive 90% (*stop_fraction = 0.9*) of map outputs.

We experimentally determined the best number of map threads to be run concurrently on one CPU to hide one map task's disk I/O under another's computation (Hadoop's default is 1 map thread). The best number varies between 2 and 4 for most of our benchmarks. We clarify that this number affects only the map-disk-I/O overlap and not map-

Table 3: Shuffle-light benchmarks

Word-count counts the occurrences of words in the input and is similar to determining the frequency of URL occurrences in a document. In Hadoop's *word-count*, each map task emits <*word, 1*> tuples. As in *term-vector*, the combiner adds up the count for the same word from *one* map task. The reduce tasks simply add up the counts for a given word from *all* the map tasks and output the final count. In MaRCO, the partial reduce tasks add up a word's partial counts from multiple map tasks.

Classification classifies the input into one of *k* pre-determined clusters (unlike *k-means*, the cluster centroids are fixed). Similar to *k-means*, *classification* uses Netflix movie rating data which is of the form <*movie_id, list{rater_id, rating}*>. Similar to *k-means*, map computes the cosine vector similarity of a given movie with the centroids, and determines the centroid to which the movie is closest (i.e., the cluster to which it belongs). Map emits <*centroid_id, movie_id*>. Unlike *k-means*, the details of movie ratings are not emitted because there are no further iterations which may need the details. There is no opportunity for a combiner. Reduce is identity function which collects all the movies in a cluster and emits <*centroid_id, movie_id*>. In MaRCO, the partial reduce is identity function.

Grep searches for a pattern in a file and is a generic search tool used in many data analyses. Map outputs lines containing the pattern as <*line, 1*> tuples. There is no opportunity for a combiner. Reduce is identity function which just outputs the tuples from map. In MaRCO, partial reduce is identity function as is the final reduce.

Histogram-movies generates a histogram of input data and is a generic tool used in many data analyses. We use the Netflix movie rating data. Based on the average ratings of movies (ratings range from 1 to 5) we bin the movies into 8 bins each with a range of 0.5. The input is of the form <*rater_id, rating, date*> and the filename is *movie_id*. Map computes the average rating for a movie, determines the bin, and emits <*bin, 1*> tuples. The combiner combines the tuples for the same bin from one map task. Reduce collects all the tuples for a bin and outputs a <*bin, n*> tuple. In MaRCO, the partial reduce adds up partial counts of tuples for the same bin.

Histogram-ratings generates a histogram of the ratings as opposed to that of the movies based on their average ratings. The input is same as that for *histogram-movies*. Here, we bin the ratings of 1-5 into 5 bins and map emits <*rating, 1*> tuple for each review. The combiner combines tuples with the same rating. Reduce collects all the tuples for a rating and emits a <*rating, n*> tuple. In MaRCO, the partial reduce adds up partial counts of tuples for the same rating.

shuffle overlap which occurs irrespective of the number of concurrent map tasks, as mentioned in Section 2. Because setting the number to be different for each benchmark would be hard to do in practice, we use 2 map threads, which gives the best average performance for all the benchmarks. In addition, we modified Hadoop to use asynchronous writes for the final reduce output (Section 2). In shuffle-heavy MapReductions the final output is large and the asynchronous writes hide much of the write latency. The Hadoop variant with 2 map threads and asynchronous writes, called *Hadoop++*, serves as our base case for performance comparisons with MaRCO. Because asynchronous write is a well-known optimization that is orthogonal to MaRCO, we include asynchronous writes in both *Hadoop++* and MaRCO. To ensure that *Hadoop++* is a strong base case, we compared *Hadoop++* with the default Hadoop. We found that *Hadoop++* achieves 6-19% and 2-4% speedups on our shuffle-heavy and shuffle-light MapReductions, respectively.

5.3 Platform

We use a 128-node Amazon EC2 cluster to evaluate MaRCO. Each node is a Xen-based virtual machine with a virtual core of 2-3 GHz Opteron or Xeon, 1MB L2 cache, 1.7 GB RAM, 160GB SATA disk drives rated at 2400 Mbps peak bandwidth, running Linux 2.6.16. The experiments for this paper ran for more than 1000 hours and buying that much time on the Amazon cluster is expensive. Consequently, we show speedups on the Amazon cluster for a representative subset of our benchmarks and use our self-owned 16-node cluster to show more detailed results. In our 16-node cluster, each node is a 2.8-GHz dual-core Xeon with 2 MB L2 cache and 4GB RAM running Linux 2.6.9 and SATA150 disk drives rated at 1200 Mbps peak bandwidth and 600-800 Mbps observed bandwidth. Later in this section we provide some arguments and experimental results on why the results from our small cluster would hold for much larger clusters. In our cluster, one of the nodes runs the global manager which manages the entire cluster, and this node does not run any worker (i.e., MapReduce computation) threads which run on the rest of the 15 nodes.

Because of real system artifacts such as differences in disk seek times and OS scheduling variations, the execution time for the same job can vary across runs. To account for this effect, we repeat each run 6 times to achieve a confidence level of 95%. We determined the number of repetitions using standard statistical calculations based on the observed variance and Student's *t* distribution [4,6].

The cluster network is a gigabit ethernet which combined with the relatively small size of our cluster results in a much higher per-node bisection bandwidth than that available in a typical large cluster. To simulate limited bisection bandwidth, we use the network-utility tools *tc* and *iptables* to limit the bandwidths from one quarter of the cluster to another to some parameter value without limiting the within-quarter bandwidth. We vary this parameter between 75-25 Mbps (typical per-node bisection bandwidth for large clusters) in our experiments. Because limiting the bandwidths of *all* possible bisections is hard to do, our method approximates by limiting the bandwidths of only some bisections and not all. This approximation only makes our results conservative by reducing our opportunity.

Though we simulate limited bisection bandwidth, there still remains the question of how our speedups would scale to larger clusters. *In general*, speedups may not scale due to inherent bottlenecks either in the applications such as load imbalance and synchronization overhead, or in the hardware such as network bandwidth. Except for the shuffle, there are no other synchronizations in MapReductions which are well load-balanced and have abundant parallelism. Thus, the only bottleneck that can prevent speedups from scaling to larger clusters is the shuffle due to larger clusters' lower per-node bisection bandwidths.

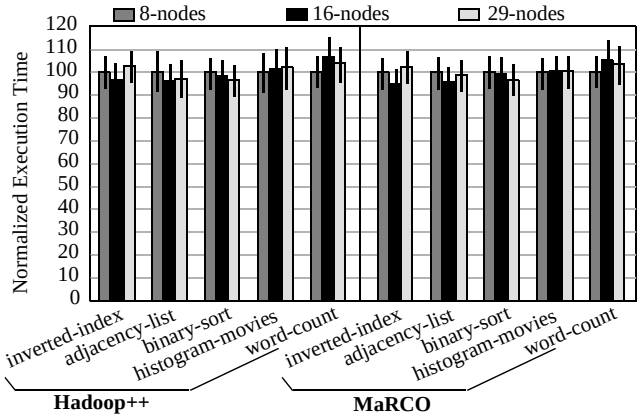

FIGURE 4: Scaling

By limiting this bandwidth in our small cluster to the amounts typically seen in larger clusters, we claim that the run times on a small cluster with small input would be close to those on a larger cluster with larger input if the following two conditions are met. (1) The per-node workload (per-node map and reduce workload) remains the same as the input data size is scaled up for larger clusters. (2) The node hardware characteristics (e.g., CPU speed and local disk bandwidth) are the same for the small and larger clusters. With the run times of the small and larger clusters being close, MaRCO's speedups would hold as the cluster size is increased. As evidence, we scale our cluster size from 8 nodes to 29 nodes and the input data size accordingly, while keeping the per-node bisection bandwidth constant at 50 Mbps and the per-node workload constant. In Figure 4, we show Hadoop++'s and MaRCO's run times on the different cluster sizes normalized to their respective run times on 8 nodes. We show the run times' statistical range using lines on top of the bars. The run times remain practically the same (i.e., within statistical error) as the cluster size is increased, implying that our small-cluster results will likely hold for larger clusters. Because we could not get long-term access to the 29-node cluster, we use our 16-node cluster for the rest of the experiments.

6 Experimental Results

We start with quantifying MaRCO's performance improvements over Hadoop's assuming no faults for simplicity, followed by a breakdown of execution time to explain the improvements. We then present performance in the presence of faults. Finally, we present results for our improvements' sensitivity to the network bisection bandwidth.

6.1 Performance (no faults)

We show results for our 16-node cluster first and then for a 128-node Amazon EC2 cluster.

6.1.1 Performance on 16-node cluster

We compare MaRCO with *Hadoop++* which, recall from Section 5.2, runs two map tasks per processor for better overlap between the shuffle and map tasks and employs asynchronous writes for the final reduce output. For MaRCO, we run one map thread and one partial-sort-and-reduce thread (deprioritized) per processor, use *start_threshold* = 8 (4 for the first two invocations) and *stop_fraction* = 0.9, and employ reduce staggering (Section 3.2). We show two variants of MaRCO, the first one is called *MaRCO-no-partial-reduce*, which overlaps only partial sort with the shuffle and does not perform any partial reduce. The second variant is *MaRCO-no-control* which does not employ any of the control mechanisms of Section 3.3. In MaRCO-no-control, the partial reduces are not deprioritized, *start_threshold* = 1 and *stop_fraction* = 1.0 (further partial reduces are not invoked once all the map data is received).

In Figure 5, the Y-axis shows the performance of MaRCO, MaRCO-no-partial-reduce, and MaRCO-no-control normalized to that of Hadoop++. The X-axis shows our benchmarks, grouped as shuffle-heavy and shuffle-light) in the order of decreasing performance improvements of MaRCO to show clearly the trends across our benchmarks. We show the statistical range across the runs using lines on top of the bars. The numbers below the bars show the *percent* speedups for an ideal case whose runtime is Hadoop++'s runtime after removing the *minimum* of shuffle time (disk + network) and final sort + reduce time. Thus, the ideal case represents the shuffle being hidden to the maximum extent possible.

The ideal speedups are in the range of 15-40% for the shuffle-heavy MapReductions. The high speedups indicate that the shuffle introduces considerable runtime overhead. MaRCO achieves good improvements (12-28%) over Hadoop++ for the shuffle-heavy MapReductions. *Inverted-index*, *k-means*, *adjacency-list*, and *self-join* achieve 28%, 26%, 24%, and 22% improvements, respectively. These four programs have high shuffle volume and long reduce time (Table 2) that can be overlapped. *Term-vector* and *binary-sort* achieve less improvements, 13% and 12% respectively. Despite its high shuffle volume, *term-vector*'s improvement is less because the reduce computation is short and also some of partial reduce's work is useless (Table 2). Similarly, *binary-sort* also has high shuffle volume but its partial reduce does not do any useful work, leaving only partial-sort to overlap with the shuffle. Due to lack of opportunity, the shuffle-light MapReductions improve only modestly (ideal speedups 4-5% and real improvements 2-3%). MaRCO's improvements are close to the ideal speedups in many cases. MaRCO's improvements for *inverted-index*, *k-means*, *self-join* are slightly better than or equal to their ideal speedups because the partial sort and the partial reduce achieve some overlap with map disk I/O

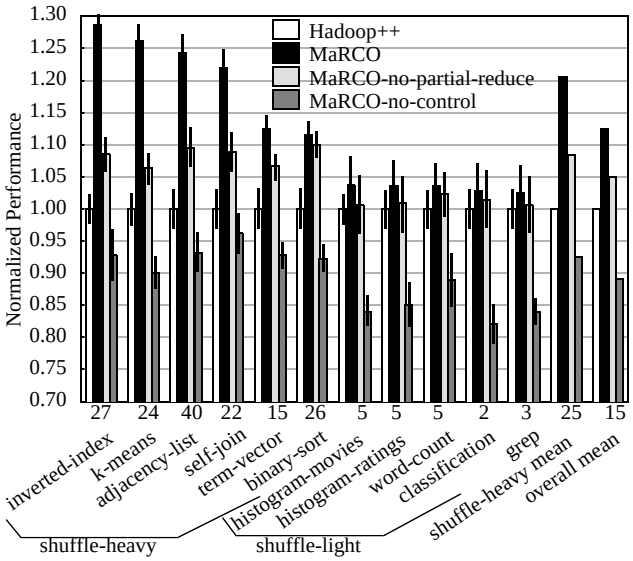

FIGURE 5: Performance on 16 nodes (no faults)

in addition to the overlap with the shuffle, whereas ideal considers overlap only with the shuffle. MaRCO's improvements for *adjacency-list* and *binary-sort* are much less than the ideal speedups because the partial reduce in *binary-sort* does not do any useful work (Table 2) and MaRCO incurs some computational overhead in *adjacency-list* which is quantified in Section 6.2.

MaRCO-no-partial-reduce improves over Hadoop++ despite not overlapping the shuffle with partial reduce. However, the limited overlap puts MaRCO-no-partial-reduce behind MaRCO by a significant margin. Finally, MaRCO-no-control performs significantly worse than Hadoop++, illustrating the importance of our control mechanisms and emphasizing the point that partial reduce invocation and scheduling need careful control.

6.1.2 Performance on 128-node cluster

Figure 6 shows the performance of MaRCO normalized to that of Hadoop++ on a 128-node Amazon EC2 cluster running a subset of our benchmarks. We keep the configuration of Hadoop and MaRCO same as in Section 6.1.1. Despite its larger size, the 128-node cluster experienced few faults during our runs. We see that the performance improvements for each of the shuffle-heavy benchmarks on the 128-node cluster are close to those on the 16-node cluster (Figure 5). As with our 16-node cluster, MaRCO does not degrade the shuffle-light benchmarks' performance on the 128-node cluster. These results support our claim that our small-cluster results hold for larger clusters. (Section 5.3 and Figure 4).

In the rest of the paper, we show detailed results on the

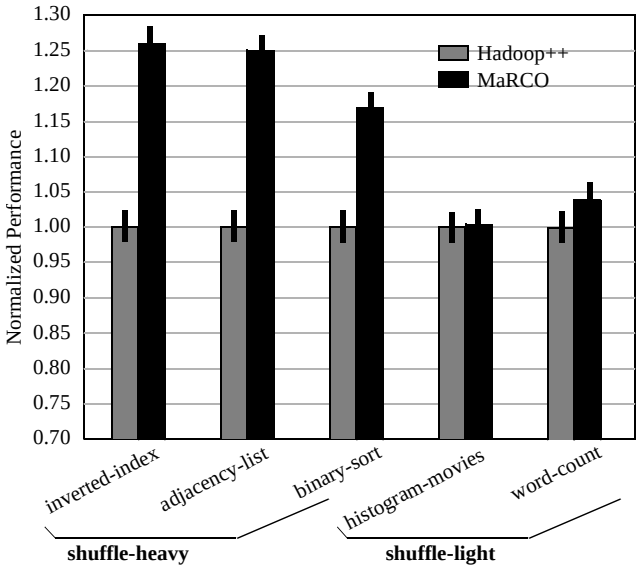

FIGURE 6: Performance on 128-node Amazon EC2

16-node cluster.

6.2 Execution Time Breakdown

We explain the above improvements using a detailed execution time breakdown of the 16-node runs. In Figure 7, the Y-axis shows the execution times of Hadoop++ and MaRCO normalized to that of Hadoop++ (which is shown at 100). The X-axis shows all the shuffle-heavy MapReductions but only two shuffle-light MapReductions, namely *histogram-movies* and *grep* due to two reasons. (1) The shuffle-light MapReductions' runtimes are uniformly dominated by map times. (2) *histogram-ratings's* and *word-count's* breakdowns resemble *histogram-movies's* and *classification*'s is close to *grep*'s. The execution time is broken up into time spent in map computation, exposed map disk I/O, exposed part of the shuffle (both disk and network), overhead (only for MaRCO), partial sort and partial reduce (only for MaRCO), final sort, and final reduce. Map disk I/O time includes only map tasks' disk accesses and not the shuffle's disk accesses which are included in the shuffle time. For Hadoop++, final sort and final reduce components show the time spent in the original sort and original reduce, respectively. The overhead component is the extra computation performed by MaRCO and is the difference between MaRCO's partial sort + partial reduce + final sort + final reduce and Hadoop++'s final sort + final reduce. Each individual component is the average across nodes. We obtain this breakdown by modifying Hadoop's logs to provide direct measurements of the components.

Hadoop++'s execution times for *inverted-index*, *k-means*, *adjacency-list*, and *self-join* get more or less equal contribution from all the components — map, map task disk I/O, shuffle, sort, and reduce. In *term-vector*, the map times are relatively long whereas the reduce times are short whereas it is the reverse in *binary-sort*. Because *term-vector* processes every one of the large number of words in the input file, the map tasks are long. Because of *Binary-sort's* large data size (Table 2), the final output to the file system increases the reduce time. Irrespective of these differences, all the shuffle-heavy MapReductions have significant exposed shuffle times because the map computation is not long enough to achieve full overlap, illustrating a key point

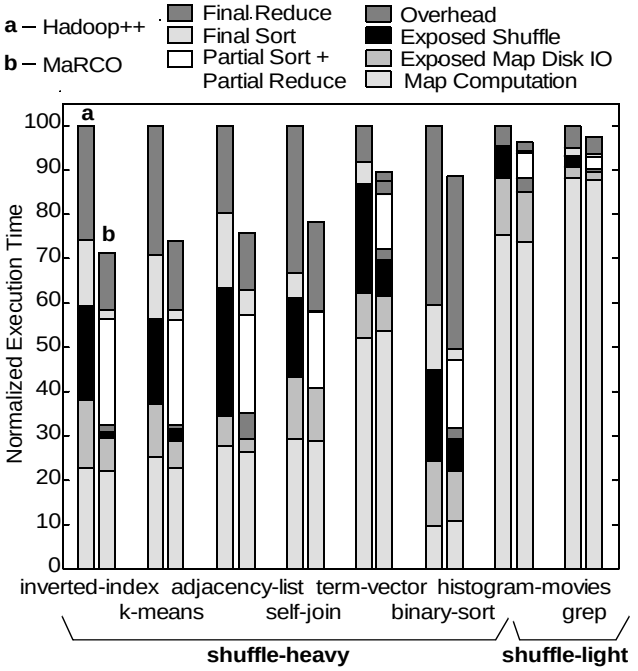

FIGURE 7: Execution time breakdown

of this paper. As mentioned before, the shuffle-light MapReductions' map times dwarf all other components.

MaRCO's partial sort and partial reduce tasks overlap with both map disk I/O and the shuffle. This overlap almost completely hides the shuffle in most benchmarks except for *term-vector* and *binary-sort* where partial reduce is useless or insufficient (Table 2). However, partial sort and partial reduce may introduce some overhead (e.g., partial reduce may write data to the local disk to be read by the final reduce whereas the original reduce does not make these accesses), as mentioned in Section 3.2. The extra disk accesses in this overhead compete with map disk I/O causing MaRCO to have some exposed map disk I/O in all the benchmarks. Also, these accesses may also cause MaRCO's partial sort + partial reduce + final sort + final reduce times to exceed Hadoop++'s final sort + final reduce times. This excess, called the overhead, occurs in *adjacency-list*, *term-vector* and *binary-sort*. The extra disk I/O and the overhead offset some of the overlapped shuffle times, explaining the difference between ideal speedups and MaRCO's improvements (Section 6.1).

6.3 Performance (with faults)

Because one of MapReduce's key motivations is fault tolerance, and because the chance of a failure during long-running jobs in large clusters is high, showing performance improvement in the no-fault case offers insufficient indication of actual improvements in large clusters. For example, Google has previously reported that 57% of the runs of a certain MapReduce-based wrapper application experienced faults [20]. Therefore, we evaluate MaRCO's performance in the presence of faults. Because both 16-node and 128-node runs experienced few faults, we simulated faults in our 16-node cluster.

We simulate a fault by simply killing some of the MaRCO-associated threads approximately in the middle of a MapReduce job. For Hadoop++ 10 out of 90 map and reduce threads (11%) were killed, and for MaRCO 10 out

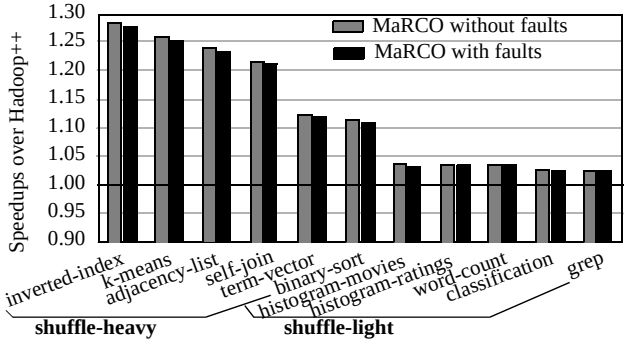

FIGURE 8: Speedups on 16 nodes (with faults)

of 60 map and reduce threads (16.6%) were killed (Section 5.2 explains why the thread counts are different). These simulated failure rates are in line with the MapReduce paper [11] which killed 200 out of 1746 threads (11.5%). In Figure 8, we show MaRCO's speedups over Hadoop++ with and without faults. We include the no-fault case from Figure 5 for comparison. We see that the faults have little impact on MaRCO's speedups.

6.4 Sensitivity to network Bisection Bandwidth

We study MaRCO's performance sensitivity to the bisection bandwidth. Using the network-utility tools *tc* and *iptables* (Section 5.3), we varied the inter-quarter bisection bandwidths in our 16-node cluster as 75 Mbps, 50 Mbps (default), and 25 Mbps. These values are close to the expected per-node bisection bandwidth available in a large cluster (e.g., 55 Mbps in a 1800-node cluster [22]). In Figure 9, we show MaRCO's speedups over Hadoop++ for these three bandwidth settings. As the bisection bandwidth decreases, MaRCO's opportunity increases and MaRCO's speedups improve, as expected. At higher bandwidths, our control mechanisms ensure that MaRCO does not incur slowdowns even for the shuffle-light MapReductions.

6.5 Sensitivity to *start_threshold* and *stop_fraction*

Finally, we study MaRCO's sensitivity to *start_threshold* and *stop_fraction*, the parameters for our partial reduce control mechanisms (Section 3.3). In Figure 10, we vary *start_threshold* (left) as 1, 8 (default), and 12, and *stop_fraction* (right) as 0.8, 0.9 (default), and 1.0 in our 16-node cluster. The Y-axis shows execution times normalized to that of the default MaRCO. In the interest of space, we show only three shuffle-heavy and two shuffle-light MapReductions. We see that the default values perform the best and that the two parameters have significant impact on execution time. As explained in Section 3.3, with *start_threshold* of 1, the partial reduce not only is less effective but also incurs overhead as it is invoked with too little data (just one map output) and hence not enough tuples with the same key. With *start_threshold* of 12, the partial reduce invocations are delayed causing the shuffle to be exposed. With *stop_fraction* of 0.8, the partial reduce invocations are discontinued too early (when 80% of map output is received) causing the shuffle to be exposed; and with *stop_fraction* of 1.0, the partial reduce invocations are discontinued too late (when all the map output is received) causing the partial reduces to extend even after the shuffle is complete and delaying the final reduce.

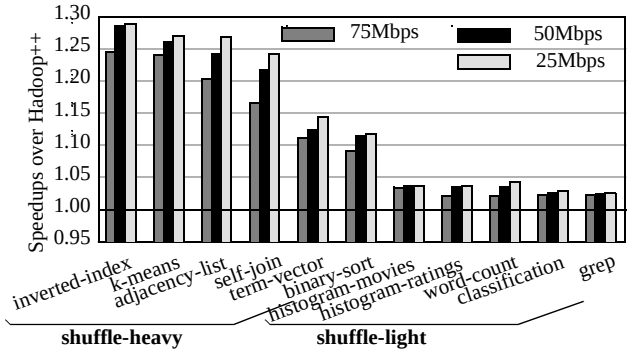

FIGURE 9: Sensitivity to bisection bandwidth

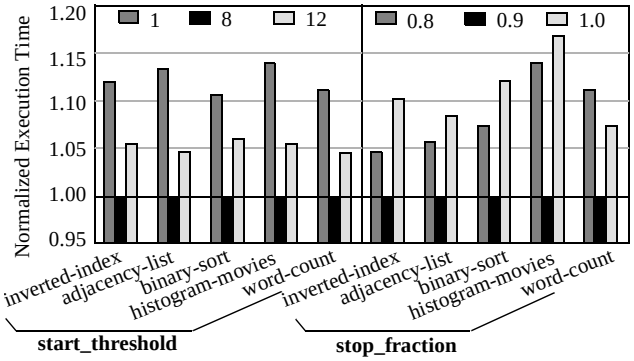

FIGURE 10: Sensitivity to start_threshold & stop_fraction

7 CONCLUSION

While MapReduce achieves some overlap between the shuffle and the map tasks, the map computation is not long enough to achieve full overlap with the long shuffle in shuffle-heavy MapReductions. We proposed *MapReduce with communication overlap (MaRCO)* to achieve nearly full overlap via the novel idea of including the sort and reduce in the overlap. While MapReduce lazily performs sort and reduce computation only after receiving all the map data, MaRCO employs eager sort and reduce to process partial data from some map tasks while overlapping with other map tasks' communication. Such overlap is a fundamental and new tool to improve performance for the important class of shuffle-heavy MapReductions. We implemented MaRCO by augmenting Hadoop's MapReduce and showed that on a 128-node Amazon EC2 cluster MaRCO achieves 23% and 14% speedups over Hadoop on shuffle-heavy and all MapReductions, respectively.

ACKNOWLEDGMENT

This work is supported, in part, by the National Science Foundation (Award Number: 0621457).

REFERENCES

[1] J.Hartigan. *Clustering Algorithms. Wiley, 1975*.
[2] Netflix movies data. *http://www.netflixprize.com/download*.
[3] The Hadoop Distributed File System: Architecture and Design. *http://lucene.apache.org/hadoop/hdfs_design.html*.
[4] J. Aczél and W. Ertel. A new formula for speedup and its characterization. *Acta Informatica*, 34(8):637–652, 1997.
[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of 20th VLDB*, 1215:487499, 1994.
[6] A. Alameldeen and D. Wood. Variability in architectural simula-

tions of multi-threaded workloads. In *Proc. of HPCA-9* 2003.

[7] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-performance sorting on networks of workstations. In *Proc. 1997 SIGMOD*, pages 243–254, Tucson,Arizona, May 1997.

[8] L. Barroso, J. Dean, and U. Holzle. Web search for a planet: The Google cluster architecture. In *Micro, IEEE*, 2003.

[9] J. Bent, D. Thain, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Explicit control in a batch-aware distributed file system. In *Proc. First NSDI, March*, 2004.

[10] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-Reduce for Machine Learning on Multicore. In *Proc. 20th NIPS*, Dec 2006.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. OSDI '04*, 2004.

[12] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. *ACM SIGOPS OS Review*, 37(5):29–43, 2003.

[13] Hadoop. http://lucene.apache.org/hadoop/.

[14] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *PACT '08: Proc. of the 17th PACT*, pages 260–269, 2008.

[15] Facebook Hive. http://hadoop.apache.org/hive.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proc. EuroSys '07*, 2007.

[17] V. Kumar, A. Grama, A. Gupta, and G. Karypis. Introduction to Parallel Computing: Design and Analysis of Algorithms. 1994.

[18] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.

[19] A. Pavlo, E. Paulson, A. Rasin, D. Abadi, D. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 SIGMOD international conference on Management of data*, 2009.

[20] R. Pike. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005.

[21] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *IEEE 13th International Symposium on High Performance Computer Architecture, 2007. HPCA 2007*, pages 13–24, 2007.

[22] D. Weld. Lecture notes on MapReduce(based on Jeff Dean's slides). *http://rakaposhi.eas.asu.edu/cse494/notes/s07-map-reduce.ppt.*

[23] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007.

[24] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of International Symposium on Operating System Design and Implementation (OSDI)*, 2008.

[25] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of OSDI 2008*.

[26] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in MapReduce clusters using Mantri. In Proceedings of the Usenix Symposium on Operating Systems Design and Implementation (OSDI),2010.

[27] https://issues.apache.org/jira/browse/HADOOP-3226.

[28] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmleegy, and R. Sears. MapReduce Online. In NSDI, 2010.

[29] A. Verma, N. Zea, B. Cho, I. Gupta, and R. Campbell, "Breaking the MapReduce stage barrier", in CLUSTER 2010, Heraklion, Crete, Greece, 2010, pp. 235-244