Transactional Support in MapReduce for Speculative Parallelism

Naresh Rapolu* Karthik Kambatla* Suresh Jagannathan Ananth Grama
Department of Computer Science, Purdue University
{nrapolu, kkambatl, suresh, ayg}@cs.purdue.edu

∗ — Student authors; Eligible for best student paper award.

Abstract

MapReduce has emerged as a popular programming model for large-scale distributed computing. Its framework enforces strict synchronization between successive `map` and `reduce` phases and limited data-sharing within a phase. Use of key-value based persistent storage with MapReduce presents intriguing opportunities and challenges. These challenges relate primarily to semantic inconsistencies arising from the different fault-tolerant mechanisms employed by the execution environment and the underlying storage medium. We define formal transactional semantics for MapReduce over reliable key-value stores. With minimal performance overhead and no increase in program complexity, our solutions support broad classes of distributed applications hitherto infeasible in MapReduce.

Specifically, this paper (i) motivates the use of key-value stores as the underlying storage for MapReduce, (ii) defines transactional semantics for MapReduce to address any inconsistencies, (iii) demonstrates broader application scope enabled by data sharing within and across jobs, and (iv) presents a detailed evaluation demonstrating the low overhead of our proposed semantics.

1. Introduction

The need to support diverse distributed data processing applications, combined with rapid advances in inexpensive commodity storage and networking solutions, motivate techniques and infrastructure for effective and reliable management of large volumes of data. On the storage side, systems such as Bigtable [12], Dynamo [16], and Cassandra [26] offer reliable, efficient and scalable storage for clients requiring key-value access. These systems are primarily differentiated by their consistency guarantees and the API. Open-source implementations of Bigtable (`HBase` [6]) and Cassandra [2] have been deployed on commodity hardware at data-centers and in cloud computing environments, providing terabytes of fault-tolerant storage to diverse applications. On the compute side, effectively processing large datasets requires scalable and fault-tolerant distributed execution environments. MapReduce [15] – its open source implementation `Hadoop` [4], Dryad [21], and All-Pairs [30], among others, address these needs to varying degrees. They support a diverse application base, spanning data-warehousing, data-mining/analytics [33], and scientific computing. Widespread use of these systems is motivated by their support for scalable distributed applications without burdening developers with node-level scheduling, load-balancing, and fault tolerance.

Integration of current key-value based storage and key-value based distributed execution engines presents significant opportunities and challenges. These challenges relate to definition of suitable semantics, efficient implementations, and application integration. Effective integration allows distributed coarse-grained computations to have a shared view of the data being processed by other tasks, enabling dependency tracking, pipelining, and speculative parallelism. This broadens the scope of MapReduce applications beyond traditional data-parallel computations. Furthermore, since many important data processing applications span long durations, leveraging the fault-tolerance capabilities of the storage infrastructure to store intermediate results potentially offers significant savings in re-computations.

The disparate fault-tolerance mechanisms adopted by the storage system (which are primarily replication and recovery-based) and the execution engine (which rely on deterministic replay of coarse-grained computations) present technical challenges for effective integration. We investigate these challenges in the context of MapReduce (`Hadoop` implementation) for the distributed execution environment and Bigtable (`HBase` implementation) as the distributed storage infrastructure. Our results, though, are more general and apply to other systems as well. We specifically examine the semantics of when and how the results of a computation (`map`), as reflected in changes to the global store, should be made visible to other computations. Of particular concern is the effect of failures of compute and storage nodes on performance of protocols for proposed semantics. The semantics should support application optimizations and high performance within the context of current systems.

To address this important problem, we propose transactional execution of computations (`maps`/`reduces`). The results of one computation (writes to the global key-value store) become atomically visible to other computations and to other concurrent jobs (only) upon successful completion of the computation. This allows data sharing across `maps`/`reduces`, speculative parallelism for computations with potential dependencies, and several other performance optimizations. We propose robust protocols and implementations for these semantics,

that introduce minimal overhead, while being resilient to failures. We support our claims of performance and enhanced application scope in the context of a variety of applications – maximum flow, classification, minimum spanning tree, and online aggregation. All of these computations involve dynamic data dependencies across computations, making them ill-suited to conventional MapReduce environments. By storing results from completed computations in persistent storage, our protocol also reduces overheads associated with replay, in the event of a failure.

In Section 2, we present necessary background on MapReduce and Bigtable, and motivate the need for their integration, along with associated technical considerations. In Section 3, we identify desired attributes of a failure-transparent model and transactional execution engine, and use these to highlight the specific contributions of our work. Section 4 describes (i) proposed semantics for the global store and distributed computations operating on the store, (ii) an efficient protocol implementing our semantics, and (iii) how the protocol can be realized within MapReduce, operating on Bigtable. Section 5 demonstrates the enhanced application scope enabled by our transactional semantics using a diverse set of illustrations, including modified WordCount, semi-supervised classification in support vector machines, Boruvka's Minimum Spanning Tree algorithm, Maximum flow calculation using the Push-Relabel algorithm, and applications requiring online aggregation. Computations associated with these applications involve dynamic data dependencies, which can not be trivially specified within the traditional MapReduce framework. We conclude by putting our contributions in the context of related work in Section 6.

2. Background and Motivation

In this section, we provide the systems context for our proposed methods and motivate the usage of key-value based storage for MapReduce using illustrative examples. We also discuss inconsistencies arising from naive integration of current key-value stores and compute engines, and describe potential solutions.

Key-Value Storage: Bigtable Bigtable [12] is a scalable storage system for sparse, semi-structured data built on top of GFS [17]. Data is indexed by row, column and timestamp. Columns are grouped together into column families, and data items corresponding to a single column family are stored together, since they are likely to be accessed together. Keys (rows) are distributed across storage nodes. Fault-tolerance is delegated to the underlying file system, GFS, which replicates each block. In case of a failure, a new replica is created from other known replicas. The open source implementation of Bigtable, `HBase`, is built on top of `HDFS`, and provides several useful features, including transactional commits to multiple rows. Our proposed development builds on `HBase` to store key-value pairs operated on by MapReduce.

Key-Value Processing: MapReduce Dean and Ghemawat proposed MapReduce [15] to facilitate development of highly-scalable, fault-tolerant, large-scale distributed applications. The MapReduce runtime system divests programmers of low-level details of scheduling, load balancing, and fault tolerance. The `map` phase of a MapReduce job takes as input a list of key-value pairs, $< key, value >: list$, and applies a programmer-specified (`map`) function, independently, on each pair in the list. The output of the `map` phase is a list of keys and their associated value lists – $< key, value : list >: list$, referred to as intermediate data. The `reduce` phase of the MapReduce job takes this intermediate data as input, and applies another programmer-specified(`reduce`) function on each pair of key and value list. The MapReduce runtime supports fault-tolerance through a deterministic replay mechanism, described below.

In the open source implementation of MapReduce – `Hadoop` on top of HDFS [5], `map` and `reduce` phases are split into multiple tasks operating on parts of the input, with each task executing on a separate node. When a node fails, the corresponding tasks are re-executed on another node. The output of `map` tasks is sorted locally and stored, and upon completion of the `map` phase, the `reduce` tasks pull the data that they are supposed to operate on from the corresponding `map` tasks. The `reduce` phase waits for the completion of the `map` phase since all the values corresponding to each key are needed by the corresponding `reduce` function.

The rigid semantics of MapReduce are well-suited to regularly structured data-parallel applications, since they do not allow data-sharing across computations within, or across jobs. MapReduce does not naturally sup-

```
public void map(key, value, context) {
  StringTokenizer itr =
  new StringTokenizer(value.toString());
  while (itr.hasMoreTokens()) {
    word.set(itr.nextToken());
    context.write(word, one);
  }
}
public void reduce(key, values, context){
  int sum = 0;
  for (IntWritable val : values)
    sum += val.get();
  result.set(sum);
  context.write(key, result);
}
```

Figure 1. WordCount implementation from `Hadoop` src

```
public void map(key, value, context) {
  StringTokenizer itr =
  new StringTokenizer(value.toString());

  while (itr.hasMoreTokens()) {
    outTable.incrementColumnValue(
    Bytes.toBytes(itr.nextToken()),
                  family,
                  qualifier, 1, true);
  }
}
```

Figure 2. WordCount implementation using `HBase`

port task-parallel jobs, jobs exploiting optimistic parallelism (through speculative parallel execution) [23, 34], jobs with data dependencies [29], jobs that build a model over iterations on the same input [35], jobs leveraging pipelining-based workflows such as online aggregation, *etc.*. To broaden the scope of the MapReduce framework, many system level modifications, such as inherent pipelining and extreme checkpointing, similar to those discussed in the recent *MapReduce Online* proposal [14], are being investigated. Dryad [21] achieves some of this generality by incorporating extra features and intelligence into the underlying execution environment. However, this is not a general-purpose solution since applications with diverse data-flow characteristics necessitate modifications to the execution environment, which is hard to provide.

Application Scope One way to allow data-sharing in the MapReduce framework without compromising fault-tolerance or significantly altering the programming interface is to allow applications to operate directly on a (persistent) key-value store, making all the writes immediately visible to all subsequent computations with well-defined semantics.

Consider an illustrative application adapted from the original wordcount example [15] to compute the k most frequent words in a large corpus. A simple implementation would create two jobs — a WordCount MapReduce job to compute the word-counts, and another job to find the k most frequent words from the word-counts. The implementation of WordCount in `Hadoop` source is shown in Figure 1. The input to the `map` function is a line (or block) of text. For each word in the line, the `map` function emits the word and the number 1. The `reduce` function receives a word and a list of counts (1's in this case), which it sums up and computes the total number of occurrences of each word. For a large corpus, the WordCount job, and the `map` phase in particular, might take a long time to execute. Other jobs that rely on this output (*eg.*, that finds the k most-frequent words) must wait until the entire WordCount application completes.

Consider an alternate implementation of WordCount shown in Figure 2. For every word the `map` function encounters, a corresponding counter is incremented in `HBase` (persistent storage), and the updated count is instantly visible to other applications. Even if the WordCount job runs for a long time, the partial counts and frequencies for the corpus parsed thus far can be made visible. Interestingly, the implementation does not require a `reduce` function, since the aggregation is done implicitly by `HBase`. Using these counts, one can compute frequencies of words in an online fashion. Such online aggregation (asynchronous `reduce`) is enabled by the fact that the persistent store makes data visible as soon as it is computed.

Pingali and Kulkarni [23] recently introduced the notion of amorphous data parallelism in irregular applications. This form of parallelism is manifested in parallel computations in which most of the operations can occur in parallel; however, a few of the operations conflict with others. As an example, if we define an operation to be acting on a node and its neighbours in a graph, operations on non-neighbor nodes can execute in parallel, while those on the neighbors need to be serialized. Interestingly, in many graph algorithms, data dependencies

cannot be detected statically as the neighbors of a node change dynamically. Hence, these dependencies must be tracked and resolved dynamically at runtime. Optimistic parallel execution allows exploitation of amorphous data parallelism, by speculatively executing concurrent operations and rolling back if the runtime detects a conflict. The technical challenge of efficiently accomplishing this in a distributed environment forms the focus of this paper. Using shared-persistent storage facilitates such applications, since concurrent operations (`map` and `reduce` functions) can transparently operate on shared data-structures. The desired semantics of such concurrent manipulations are not completely defined for MapReduce programs. These semantics must support broad classes of applications, while admitting efficient implementations within the programming framework.

An example of how such an infrastructure could be used is Boruvka's algorithm for finding the minimal spanning tree of a graph [31]. In the sequential version of the algorithm, each iteration (on different nodes) coalesces a node and its closest neighbor. Storing the graph as an adjacency matrix in a key-value store allows execution of these iterations concurrently (`map` phase). However, two iterations might involve conflicting operations (two nodes may attempt coalescing with the same node) and may need to be serialized. In this paper, we propose novel transactional semantics for MapReduce over key-value storage to resolve such conflicts and to enable broad classes of applications hitherto inaccessible within current MapReduce frameworks.

Inconsistencies Naively accessing a global key-value store from within a `map` or `reduce` function potentially leads to inconsistencies —

Failures MapReduce implements fault-tolerance through deterministic-replay. A `map`/`reduce` task is re-executed with the same input if the original task fails. Such replay does not roll back any side-effects of the failed execution of the task, and can thus result in inconsistencies arising from writes to persistent storage from partially completed (and replayed) `maps`.

Conflicts Optimistic parallel execution of applications with dependencies might lead to conflicting concurrent operations compromising isolation and serializability. Isolation is violated when a computation's commit to the global store is only partially witnessed by another computation's reads.

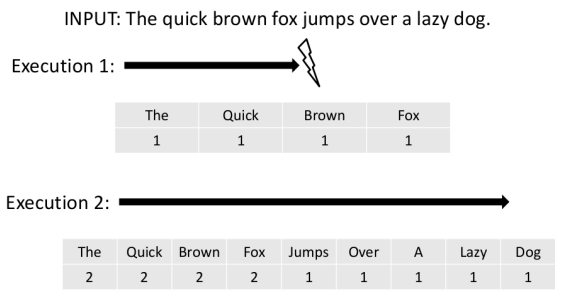


Figure 3. A sample execution of `map` function of WordCount from Figure 2

Figure 3 demonstrates the inconsistencies due to failures that might arise in our modified implementation of WordCount described earlier. The `map` function takes as input a line, l, denoted as input in the Figure. Execution 1 corresponds to an execution of the `map` task whose first line is l, which crashes during the execution of our `map` function. In this execution, the `map` function parses the words – `the-quick-brown-fox`, and crashes while parsing the next word. The word-counts corresponding to these words are updated in the `HBase` table as shown in the figure. These updates are made persistent, and the only way to roll-back is to revert to old values (or delete). Since MapReduce employs a deterministic-replay for failure recovery, the `map` task is re-executed, say execution 2, which finishes successfully. During the re-execution, the `map` function increments the word-counts of previously encountered words again, resulting in the incorrect counts, shown in the second table in the Figure.

The source of this inconsistency lies in the different fault-tolerance mechanisms employed at the computation and storage layers. Each computation must be executed atomically to prevent such inconsistencies. In case of optimistic parallel execution (as in Boruvka's), all but one of the conflicting operations must be rolled back and re-executed to guarantee isolation and serializability.

This paper defines transactional semantics for MapReduce over global key-value stores to ensure atomic, isolated, serializable execution of computations. Such semantics allow applications to exploit parallelism (even in cases with data dependencies) through data-sharing over the global store. The paper discusses our

implementation of these semantics in the context of `Hadoop` operating over `HBase`. Furthermore, our approach inherently makes intermediate data persistent, which is desirable for long-lived applications that benefit from intermediate checkpointing (e.g., multistage MapReduce jobs as found in Pig [32]) and applications where intermediate data also contains valuable information.

3. Contributions

This paper makes the following specific contributions:

- It defines transactional semantics for MapReduce computations operating on a global key-value store. The proposed semantics ensure atomic, isolated execution of the computations in a failure-transparent manner.
- It provides the design and implementation of a protocol that implements proposed semantics, through the use of global and computation-collocated stores.
- It demonstrates enhanced application scope of the extended semantics using four sample applications – maximum flow, minimum spanning tree, classification, and online aggregation. Data-sharing and resolution of runtime dependencies in these applications is enabled by our proposed semantics. In each case, it shows significant performance improvements (speedups) on commercial cloud environments.
- It demonstrates that the overhead of the proposed integrated system for traditional workloads (those with no dependencies, supported by MapReduce) is minimal.

4. Semantics

In this section, we define the semantics of a Global Store (GS) and a computation unit (CU), along with a protocol describing their interactions. A computation unit is a sequence of operations executed transactionally. In MapReduce, the `map` and `reduce` functions are abstracted as computation units. We represent MapReduce using this protocol so that the computation and storage layers are aware of each other's fault-tolerance mechanisms. Both storage and computation are assumed to be distributed, and span multiple nodes. A node refers to a set of resources — a processor, local memory, and local disk space.

4.1 Global Store

The Global Store (GS) is a set of global mappings of keys to values. This mapping is stored in a distributed fashion on multiple nodes. The keys are distributed across nodes and all accesses to a key are directed to the associated node. The underlying file system, on top of which the store is built, provides reliability and persistence.

The proposed semantics of the GS formalize operations supported by Bigtable and its open-source version `HBase`. They can be extended to any key-value store based on the consistency guarantees they offer. As described in Chang *et.al.* [12], Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes. The timestamp can be interpreted as an update counter that monotonically increases on each write to the value in that row and column. We use the term update-counter instead of timestamp, since the notion of time has other connotations in distributed systems. The mapping for a GS is given by: $mapping = \{(row : col : update - counter, value)\}$. For a write request to the GS with an unspecified update-counter, the value is appended with an update-counter higher than the highest existing update-counter in the GS. One can scan all the values associated with a given row:col pair with monotonically increasing update-counters. Similarly in a read request, if the update-counter is not specified, `HBase` returns the value with the highest update-counter. For the sake of simplicity, in this paper, we assume that the key for read/write access to GS is of the form row:column (without the update-counter). Note that the terms GS and `HBase` are used interchangeably in this paper.

The following consistency guarantees are provided by the current version of `HBase`. The global store supports atomic commits on single-row operations, by acquiring a lock on the row. This can be used to perform atomic read-modify-write sequences on the data stored under a single row key. Building on this atomic row

commit, the global store also allows a transactional commit on multiple rows, using a two-phase commit protocol [10] to lock and atomically update each row.

4.2 Computation Unit

A *Computation Unit(CU)* is an abstraction of a sequence of computations on a certain input. GS serves as source and sink for a CU's input and output, respectively. Fault-tolerance is achieved through deterministic replay. If the node executing the CU fails, the CU is re-executed, from the beginning, on a different node. In the proposed model, a CU is executed as a transaction using the protocol described below.

4.3 CUGS protocol

The proposed CUGS protocol (Computation Unit over Global Store) attempts to integrate the execution of CU over the GS in a failure-transparent manner. It provides transactional semantics to a CU. We define *Computation Store(CS)* as a temporary buffer for storing the key-value pairs accessed by the CU during its execution. It serves as a read-write list of a CU, to be used for validation and concurrency control of the CU transaction.

read(Row:Col) During a read(Row:Col), the fetched value from GS is first stored in the Computation Store(CS) along with its key (serves as a read-list for this transaction) and then returned to the CU. The most recent update-counter for this Row:Col pair is also stored alongside. The value is fetched from GS only if there does not already exist a copy of Row:Col in the CS. Otherwise, the value present in CS is returned (to provide read-after-write consistency). This technique is usually referred to as optimistic-reads in software transactional memory (STM) literature.

write(Row:Col, value) A write request is always fulfilled by the Computation Store. If a read on the same key is performed before the write, then the CS would have a copy of it, which gets updated by this write; otherwise, a new copy is created in the CS. GS is never immediately accessed in a write request. All updated key-values in the CS form its write-list. This technique is generally termed write-buffering in STM literature.

The validation, atomic commit, and fault-tolerance of this transaction clearly depend on the physical placement of the Computation Store. There are two design choices here. The Computation Store can reside on the same node on which CU executes (client-side buffering) or it may reside on the nodes hosting the GS (server-side buffering).

Each region, potentially hosted at a different node, in `HBase` spans a set of rows. In server-side buffering of CS, the read/write lists of the CU corresponding to a row are placed at the node containing its corresponding region. Since all read/write lists of concurrent transactions are present at the same node, they can be validated against each other to detect conflicts during an atomic commit. A transaction would be committed only if all the regions successfully validate. Every transaction increases an atomic counter before its validate-and-commit phase. The read/write lists of a transaction t_i, are validated against those transactions that committed their writes, between the start of t_i and the time it increased the atomic counter. This approach is similar to the well-known optimistic concurrency control method used in most databases [25]. The atomic commit of all the writes is executed using the two-phase commit protocol [10]. To handle CU failures using server-side buffers, each CU acquires a lease on the region before the start of its transaction. The lease is periodically renewed as the transaction proceeds. If the CU fails and the lease is not renewed for a long time, the region in `HBase` automatically revokes the lease and re-claims the CS buffers for the specific transaction.

Consider the case in which a CU executes many read/write operations during its execution. Since reads/writes get buffered in the Computation Store present at the server, each read/write operation incurs heavy network latency. One solution to this is to have the CS reside at the node where the CU executes (client-side buffering). The read/write semantics of the transaction, mentioned above, are affected locally, in-memory, leading to better performance. Once the CU executes, during transactional commit, the entire read/write lists on the local CS can be transferred to the server for validation and atomic commit. Handling CU failures is less complex in client-side-buffering. In this method, if the node executing the CU crashes, the entire Computation Store present on

the node is also lost. Deterministic replay of CU would result in the creation of a new local store. Failure during the transactional commit is handled by the server-side lease and two-phase commit, as described above.

While client-side-buffering appears superior, it has a few inherent disadvantages. If the amount of data being accessed by the CU from GS is large, buffering all of it at a single client node is not feasible. This overhead is significant if the number of reads is considerably larger than writes. If many CU's execute on the same node, this can lead to severe memory overheads. As mentioned above, server-side buffering is not feasible for applications with a large number of writes to the GS. Consequently, the choice is dependent on the data access patterns of the CU and the physical location of its execution. We test our applications with both server-side buffering and client-side buffering. For applications, involving few reads and writes to the GS, there is no significant difference in performance.

4.4 Guarantees provided by the protocol

Definitions

Schedule is an interleaving of successful completions of the computation units(CU) within a job.

Mutation is an insert/update to the global store.

Mutation order of a job is the order in which mutations occur in the job's execution.

The execution of a job is *faulty* if failures occur during the execution. Otherwise, it is *faultless.*

The CUGS protocol mentioned in the paper ensures the following invariant.

Invariant 1. For every mutation order of a job, J, under faulty execution, there exists a faultless execution of J with the same mutation order.

Proof. The mutation order of a job depends on its schedule. Upon successful completion, each computation atomically writes its output to the global store. The set of mutations corresponding to a single computation unit are written atomically. This is achieved in the GS by using the atomic counter and two-phase commit protocol for mutations. Given a mutation order, one can decipher the schedule and vice-versa. Hence, it suffices to prove that for every schedule in a faulty execution, there exists a faultless execution with the same schedule.

Let C be the set of computation units, and S be the set of all possible schedules for C. Clearly, S is the set of all permutations of elements of C. Consider a particular faultless schedule, $s_i \epsilon S$. When s_i's execution encounters a fault (a particular computational unit c_i fails), it re-executes resulting in a different schedule s_j. Though $s_i \neq s_j$, $s_j \epsilon S$, since s_j is still a valid permutation of elements of C. □

The above invariant assures that, the order of CU execution is serializable. With respect to MapReduce, let us consider `map` as a CU which operates on elements of a list. If the elements are unordered, i.e., the application admits any order of their execution, then the above invariant assures that the protocol will guarantee a serializable execution even in fault-prone environments.

4.4.1 Consistency and Reliability

We show that the protocol guarantees the following properties, even in failure-prone conditions.

- *Deadlock Freedom.* Note that the CU transaction is based on optimistic reads. No locks are acquired during reads. The protocol allows stale reads during the CU execution, which is verified only at commit time. Since reads go through without waiting, there cannot be a deadlock on reads. Furthermore, writes happen only during the transactional commit. As mentioned in the protocol, during the atomic commit of writes, the CU first acquires a lease on all the regions, and increases an atomic counter. All the writes get committed only in the order of this counter. If a transaction in this list crashes, GS forcibly revokes the lease after a time-out. This allows for other subsequent transactions in the commit-pending queue to acquire the lease and transactionally commit. This time-out mechanism eliminates the possibility of deadlocks during writes. The same holds even if multiple nodes hosting Computation Units crash in the system. The two-phase commit protocol handles

the case for consistency of writes during GS failure. During writes, if a node hosting a region of GS fails, none of the writes go through. This failure of transactional commit, will force any pending transaction to revoke the lease.

- *Livelock Free* The transactions at the regions hosting the GS, during atomic commits, are stored in FIFO order. If a transaction fails during its transactional commit (either due to read-write dependencies or due to the corresponding CU crash), a time-out mechanism forcibly removes it from the list. Consequently, every transaction commits in bounded time.
- *Progress* A failed Computation Unit is re-executed, assuring Progress. In case of a crash failure, `Hadoop` affects re-execution. In case of a failure due to conflict, the CU is queued for later re-execution. On the storage front, `HBase` uses the distributed file system for replication of its data. Consequently, when one of its regions fail, `HBase` re-spawns it at another node using the replicated data.

4.5 Realizing CUGS protocol in MapReduce

We now demonstrate how MapReduce can utilize the CUGS protocol to provide well-defined semantics and high performance on applications with potential data dependencies. To realize the CUGS protocol in MapReduce, we treat the `map` and `reduce` function instantiations as Computation Units (CU). The `map` and `reduce` functions are executed transactionally and upon successful completion, their outputs are stored in persistent storage. Thus, the output becomes visible to other `map`/`reduce` functions of the same job scheduled after the current function, and also to other jobs – MapReduce or otherwise.

Speculative or optimistic parallelism requires serializability to address data-dependencies. Traditional MapReduce supports only data-parallel applications; it has no provision for taking data dependencies into account while scheduling `map`/`reduce` tasks. However, the CUGS protocol keeps track of the accesses made by each concurrent computation, and serializes their execution in case of conflicts. If there are no conflicts, all computations run through to completion. Hence, we spawn and execute the `map`/`reduce` functions as in traditional MapReduce. In case of a conflict, all but one of the conflicting functions are queued for later re-execution assuring serializability.

4.5.1 Fault-tolerance model

Our proposed approach to fault-tolerance is markedly different from that adopted by `Hadoop`. In `Hadoop`, to achieve coarse-grained control over re-execution and data-movement, `map` functions are grouped into a `map task`. Similarly, `reduce` functions are grouped into a `reduce task`. A task executes its corresponding functions iteratively. Failure during the execution of any one function leads to the re-execution of the entire task. Re-execution of a reduce task is more complicated, since the input to the `reduce` tasks is stored locally on the nodes which executed the `map` tasks. In order to re-execute the `reduce task`, the input is fetched from the corresponding `map` tasks, which potentially involves re-execution of some `map` tasks whose nodes might have crashed since the last pull.

In our approach, we can not re-execute the entire `map task`, since we make the output of every function persistent upon completion. Also, this persistence allows us to execute only the partially executed and unexecuted map instances, as opposed to the entire `map task`. We maintain a system table in GS, which stores the number of functions successfully committed by all tasks. The keys in this system table correspond to task-ids, which are unique ids assigned by `Hadoop`. Furthermore, the id for a re-executing task is obtained by appending the attempt number to the original task-id. Before starting iterative re-execution of the map functions, we parse the map-task-id to get the original id and determine the number of functions successfully executed during prior runs, and start executing from last failure. Note that the system table is updated atomically, during the transactional commit, at the end of a function, avoiding inconsistent behaviour during faults. The `reduce task` also can be re-executed similar to a map task, on the intermediate data stored in the GS, maintaining its transactional nature. Upon completion of the job, intermediate data can be deleted from the GS by a background cleanup job.

4.5.2 Advantages of persistent intermediate data

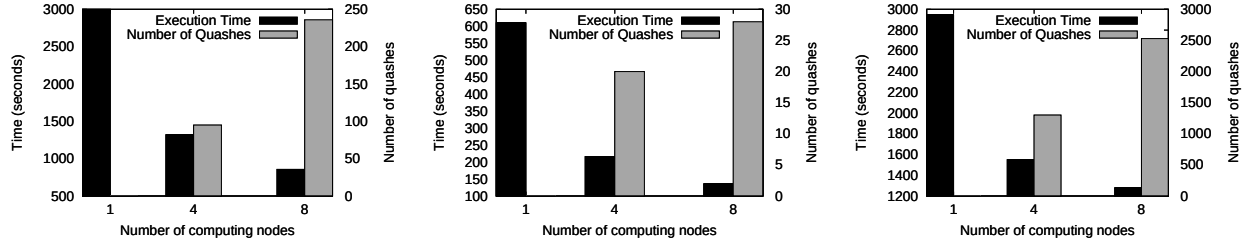
There are several advantages associated with storing intermediate data on the GS, in addition to safeguarding the transactional nature of reduce functions. As mentioned earlier, the output of the `map` functions is made visible to other maps and other MapReduce jobs as and when they are completed, instead of waiting for the entire `map` phase to finish. This allows data-sharing among applications involving partial dependencies, asynchronous and online aggregation *etc.*, explained in detail in the evaluation section. In many multi-stage dataflow frameworks such as Pig [32], intermediate data after every stage is stored in the local filesystems. The crash failure of a node at a stage can lead to expensive cascaded re-execution, involving re-execution of certain tasks from prior stages. Checkpointing intermediate data [22] is an obvious solution. However, the overhead of n-copy replication in the underlying distributed file system potentially causes significant network overhead, since this happens concurrently with the reduce tasks fetching their input from the map tasks. Ko *et.al.* [22] propose replicating the intermediate data asynchronously using TCP-Nice [37] at the end of `map` phase. Our solution allows asynchronous data transfer even during the execution of `map` phase as and when each function completes, further reducing network overhead. Persisting data during the `map` phase sorts intermediate data (`HBase` sorts the data based on key similar to insertion sort), eliminating the need for a special sorting/shuffling/merging phase. The `reduce` tasks can be spawned just like the `map` tasks.

4.5.3 Handling conflicts among concurrent Computation Units

If transactional commit of a map function fails due to conflicting transactions, it is inserted into a queue, and re-executed upon successful execution of all the other map functions in the task. This technique of backing-off reduces the possibility of another conflict, if the transaction is re-executed immediately. Re-executions continue until all map functions are successfully executed and committed. Due to the optimistic nature of concurrency control, all transactions eventually commit. The same holds true for reduce functions.

5. Evaluation

In this section, we demonstrate the performance gain from our transactional MapReduce semantics. We use the transactional client package in the `HBase` API to implement the two-phase commit, re-execution mechanism, client-side, and server-side buffering. All applications are executed on an eight-node cluster in Amazon EC2 [1] under real-world cloud latencies. Each node is an *xlarge* instance with $16GB$ of RAM and four virtual cores (32 cores in all). We demonstrate our results on four applications – push-re-label maximum flow computations, Boruvka's minimum spanning tree algorithm, generalized learning vector quantization, and online aggregation. In each case, we show how our semantic model can be used to specify underlying computations (with dependencies), and the performance enhancements achieved.



(a) Preflow *Push-Relabel* (b) Boruvka's MST (c) GLVQ classification

Figure 4. Performance of Preflow Push-Relabel, Boruvka's MST, and GLVQ classification using CUGS protocol

5.1 Push-Relabel maximum flow algorithm

The maximum flow problem is to find a feasible flow through a single-source, single-sink, flow network with the maximum capacity. The *Push-Relabel* algorithm [18] maintains a preflow – a flow function with the possibility of excess at the vertices. The *Push* operation increases the flow on a residual edge, and a height function on the vertices identifies the residual edges that can be pushed. When there are no more *Push* operations to be executed, a *Relabel* operation increases the height of the vertices, which have excesses. This sequence of operations continues until there are no more excesses on any of the vertices other than the source. It is evident that the same operation *Push* or *Relabel* cannot be applied to neighboring nodes concurrently. As long as the neighbourhoods of two nodes do not intersect, operations on them can occur in parallel. However, since this can only be determined at runtime, it is difficult to specify this in the form of `maps` (and `reduces`).

In our transactional MapReduce formulation, each `map` function operates on a node and its adjacency list, which is stored as one row in a `HBase` table. In the *Push* operation, a residual edge is chosen from the nodes adjacency list and its capacity is updated. Data corresponding to the other vertex connected by the edge, its values of excess, and residual capacity are updated, and both of the updated nodes (rows) are atomically committed. During the transactional commit, concurrent `map`-transactions are checked for reads or writes to the two rows being updated. If a conflict is detected, the transaction which is later in the commit-pending-queue is quashed and the corresponding `map` function is re-executed from the beginning.

The input flow network is generated using the Washington network generator [9]. The network is a 200 x 500 grid with the source connected to all the nodes in the first row and the sink connected to all the nodes in the last row. Every node in a column randomly connects to three other nodes in the next column. The edge weights are randomly generated. Figure 4(a) shows the average times taken over a window of 10 iterations of *Push* and *Relabel* operations. Speedups of up to 3.5 were observed on 8 nodes. The average number of `map`-transaction re-executions due to conflicts are under 0.8 percent implying considerable speculative execution. While the speedup may not appear large, when viewed in the context of the dynamic data dependencies underlying the application, this is significant.

5.2 Boruvka's Minimal Spanning Tree (MST) Algorithm

Boruvka's MST algorithm iterates over the nodes (u) in the graph, finding the node v closest to u's component, adding the edge between these two nodes to the minimal spanning tree, and coalescing u and v. The process is initiated with as many components as nodes (each node forming a component); each iteration coalesces two components – the resulting component at the end of all iterations being the minimal spanning tree of the input graph. In its MapReduce formulation, the `map` function takes the adjacency list of a node u, finds the node v closest to the component that contains u, and coalesces these two nodes (adds u to v's component and vice-versa). Two `map` functions attempting to coalesce with the the same node must be serialized. In our implementation, we use `HBase` to store the input graph as well as information regarding coalescing nodes. Each row in the `HBase` table corresponds to a node, the adjacency list is stored as one family and the corresponding coalesced node list is stored as another family.

We run Boruvka's algorithm on a 100,000 node graph with an average degree of 20, generated using the forest fire model of iGraph [7]. Figure 4(b) shows the average time taken to compute the spanning tree with 1,4 and 8 nodes. We observe speedups of 2.82 and 4.48 for 4 and 8 nodes, respectively. Due to the graph structure and sparsity, the average number of conflicts detected amount to less than 0.5 percent of total executions. As before, considering the dynamic data dependencies, this is excellent speedup.

5.3 Generalized Learning Vector Quantization (GLVQ)

We focus on parallelizing the compute-intensive training phase of the GLVQ algorithm [19]. In its sequential version, given two sets of labeled training and reference vectors, the training phase sequentially computes distance between a training vector and all of the reference vectors. The nearest reference vector belonging to the same class as the training vector is brought closer to it, whereas the nearest vector belonging to the

other class is pushed farther. The training vectors must be processed sequentially due to a potential read-after-write dependency. The key observation here is that every training vector updates only two reference vectors. Therefore, the distance calculation phase can happen in parallel for all the training vectors. Once all the training vectors choose their nearest reference vectors, each training vector needs to re-calculate its distance from updated reference vectors to ensure that changes to other reference vectors do not affect the nearest-neighbour decision of individual training vectors. Any serializable execution of this check-and-commit operations of all the training vectors is a valid execution.

In the MapReduce formulation of the problem, each `map` function takes a training vector, calculates its distance from all reference vectors, chooses the nearest two and writes their identifiers in an intermediate table. Each reduce function operates on a training vector which scans and recalculates its distance from the reference vectors in the intermediate table; updates and atomically commits the nearest reference vectors. Each `reduce` fetches the latest values from `HBase`, before re-calculating the distances, therefore the read-after-write dependency is not violated. The transactional commit handles concurrent updates to the same reference vector. As test input to our program, we use the URL Reputation Data Set [27] from UCI Machine Learning Repository. We use 10000 training and 10000 reference vectors. Figure 4(c) shows the average times taken for the entire training phase. We observe speedups of 1.9 and 2.3 on 4 and 8 nodes, respectively. This moderate improvement in performance is mainly due to the large number of conflicts (23%) detected during in-order commit of the `reduce` phase. The number of conflicts is a function of the dataset – a dataset with better separation of classes is likely to have lower conflict rates.

5.4 Online Aggregation

Traditional MapReduce implementations provide a poor interface for interactive data analysis tasks. They do not produce any output until the job runs to completion. Online aggregation [20] has been proposed in the database literature to address this problem. Our CUGS protocol enables making the output of a `map` function visible to other jobs as they complete. Consider as an example, the task of extracting the top-k words in a document corpus. This can be implemented using a regular WordCount job and an online aggregator job. The `map` function of the WordCount job parses a line from the given document and scores each encountered word in `HBase`. The online aggregator job is spawned at regular intervals. The `map` function of this job scans a row(word) and sums up the frequencies. The `reduce` function in the aggregator job sorts the frequencies and outputs the top-k words. We test this on a 5 GB text dataset from project Gutenberg [8] is used as input. Though the WordCount job took 460 seconds to complete, the aggregator job emitted the exact top-10 words in 130 seconds.

5.5 PageRank

We run PageRank (mentioned in the original MapReduce paper [15]) on traditional and transactional MapReduce to show that CUGS protocol has very low performance overhead. PageRank takes a document-links graph and outputs the pageranks of documents, implemented as an iterative MapReduce job. In the `map` phase, each node pushes its pagerank to each of its outlinks. In the `reduce` phase, each node aggregates the contributions of each of its inlinks. We iterate until the pageranks converge. We ran PageRank on a 5 million node sparse graph. Traditional MapReduce version took 134 seconds per iteration, while the transactional MapReduce version took 146 seconds per iteration. Given the transactional execution and checkpointing of intermediate data, this overhead is very low. The main reason for the low overhead is the spreading of the network traffic over the entire MapReduce job.

It should be evident that our proposed transactional model (i) is general and applies to broad classes of applications; (ii) is capable of speculatively executing complex applications with dynamic data dependencies with well-defined semantics; and (iii) incurs low overhead, delivering high overall application performance without altering the programming API.

6. Related work

A number of research and development efforts have targeted both systems and applications aspects of MapReduce. On the application front, the MapReduce programming model has been validated on diverse application domains including, machine learning [13], scientific computing, data-mining [36], and database operations [32]. These applications are largely data-parallel, and well-suited to the MapReduce programming model. To the best of our knowledge, this paper represents the first attempt to exploit speculative or optimistic parallelism in MapReduce, while providing well-defined execution semantics.

Dryad [21] is a data-parallel programming environment supporting a more general model of acyclic data-flow graphs. This model admits pipelining among nodes in the graph. However, it is unclear how this can be adapted to a distributed environment in which the underlying storage is a distributed filesystem. In the recent *MapReduce Online* [14] proposal, `map` tasks push parts of their output to `reduce` tasks, as and when they are generated. Since communication happens over a filesystem to account for failures, tasks need to checkpoint each file that is being sent. Reduce tasks need to be aware of the possibility of `map` task failure, and consequently have rollback functionality. The protocol gets further complicated for a multi-stage pipeline, or when a large number of jobs are waiting for the `map` output. In contrast, our protocol uses an intermediate key-value store, into which `maps` push data, and from which `reduces` pull data. By making the key-value store aware of the failures in computations through transactional mechanisms, we eliminate the need for the computation to be aware of the other computations involved in the pipeline. This solution is inherently scalable and does not increase program complexity.

Study of Software Transactional Memory (STM) and transactions is an active area of research in fine-grained concurrent and parallel systems [11]. Our protocol includes concepts such as optimistic reads and pessimistic writes, which are well-known in the STM literature. Though, they are being extensively examined with respect to fine-grained shared-memory systems, the relative overheads of target distributed systems yield much better performance than fine-grained systems, where overhead of rollback negates some of the gains of reduced locking and serialization.

Recent research efforts have also focussed on understanding the performance potential of optimistic parallelism [34] and best-effort computing [29] in irregular applications. It has been shown that many commonly used algorithms can be efficiently parallelized through optimistic or speculative parallelism. Work by Pingali and Kulkarni et.al. [23, 28] on speculative parallelism has resulted in the Galois runtime environment [24] for shared-memory systems. The system maintains a working set and a pool of threads, which pick an element from the set and speculatively execute. By defining a set of commutativity rules, the system eliminates the overhead of transactions. In contrast, our protocol works in a fault-prone distributed environment with limited control over the executing threads. Furthermore, by integrating the concept of speculative parallelism in the well known MapReduce paradigm, we simplify the programming of speculatively parallel applications.

7. Conclusion and Future Work

In this paper, we demonstrated the use a global key-value store as underlying storage for MapReduce allowing data-sharing and asynchrony to extend MapReduce to support task-parallel applications with minor dependencies. By exploiting speculative parallelism, we increase the applicability and performance of MapReduce. As naively using the global store leads to inconsistencies, we proposed transactional semantics for MapReduce using optimistic reads and buffered writes for accesses to the global store from within a computation (`map` or `reduce`). We evaluate various methods of achieving the proposed semantics. Our performance evaluation showed significant speed-ups for different classes of applications – maximum flow, classification, minimal spanning tree, online aggregation. We plan to including the implementations of the above mentioned and other speculatively parallel applications into the Apache Mahout [3] open source repository for MapReduce-based machine learning and data mining applications. We are evaluating other application specific optimizations which can be leveraged in a distributed setting along with speculative parallelism.

References

[1] Amazon ec2. http://aws.amazon.com/ec2.

[2] The apache cassandra project. http://incubator.apache.org/cassandra.

[3] Apache mahout. http://lucene.apache.org/mahout.

[4] Hadoop. http://hadoop.apache.org.

[5] Hadoop distributed file system. http://hadoop.apache.org/hdfs.

[6] Hbase. http://hadoop.apache.org/hbase.

[7] Igraph. http://igraph.sourceforge.net.

[8] Project gutenberg. http://www.gutenberg.org.

[9] Washington max flow network generator. http://www.avglab.com/andrew/CATS/maxflow_synthetic.htm.

[10] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[11] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. Jun 2007.

[12] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[13] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288, 2006.

[14] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[16] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[17] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[18] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, 1988.

[19] Grana M. D'Anjou A. Gonzalez, A.I. An analysis of the glvq algorithm. *Neural Networks, IEEE Transactions on*, 6(4):1012 –1016, Jul 1995.

[20] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. pages 171–182, 1997.

[21] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[22] Steven Y. Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. On the availability of intermediate data in cloud computations. *12th Workshop on Hot Topics in Operating Systems*, 2009.

[23] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? *SIGPLAN Not.*, 44(4):3–14, 2009.

[24] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[25] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*,

6(2):213–226, 1981.

[26] Avinash Lakshman and Prashant Malik. Cassandra: a structured storage system on a p2p network. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 47–47, New York, NY, USA, 2009. ACM.

[27] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Identifying suspicious urls: an application of large-scale online learning. In *ICML '09: Proceedings of the 26th Annual International Conference on Machine Learning*, pages 681–688, New York, NY, USA, 2009. ACM.

[28] Mario Méndez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, New York, NY, USA, 2010. ACM.

[29] Jiayuan Meng, Srimat Chakradhar, and Anand Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[30] C. Moretti, J. Bulosan, P. Flynn, and D. Thain. All-pairs: An abstraction for data intensive cloud computing. *International Parallel and Distributed Processing Symposium (IPDPS)*, 2008.

[31] Nesetril, Milkova, and Nesetrilova. Otakar boruvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history. *DMATH: Discrete Mathematics*, 233, 2001.

[32] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. *SIGMOD '08: Proceedings of the ACM SIGMOD international conference on Management of data*, 2008.

[33] Spiros Papadimitriou and Jimeng Sun. Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining. In *ICDM '08: Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, pages 512–521, Washington, DC, USA, 2008. IEEE Computer Society.

[34] Keshav Pingali, Milind Kulkarni, Donald Nguyen, Martin Burtscher, Mario Mendez-Lojo, Dimitrios Prountzos, Xin Sui, and Zifei Zhong. Amorphous data-parallelism in irregular algorithms. Technical Report TR-09-05, Department of Computer Science, The University of Texas at Austin, February 2009.

[35] P.Dubey. Recognition, mining and synthesis moves computers to the era of tera. *Technology@Intel Magazine*, 2005.

[36] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor system. *Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture (HPCA), Phoenix , AZ*, 2007.

[37] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. Tcp nice: a mechanism for background transfers. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 329–343, New York, NY, USA, 2002. ACM.