TransMR: Data-Centric Programming Beyond Data Parallelism

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

Abstract

MapReduce and related data-centric programming models have proven to be effective for a variety of large-scale distributed computations, in particular, those that manifest data parallelism. The fault-tolerance model underlying these programming environments relies on deterministic replay, which makes data-sharing (side-effects) across computations harder to support. This significantly limits the application scope of MapReduce and related models. This paper: (i) investigates data sharing (side-effects) in programming models operating on distributed key-value stores, specifically, the inconsistencies between the fault recovery mechanisms in execution and storage layers; (ii) defines semantics for a novel programming model, TransMR (Transactional MapReduce), which addresses these inconsistencies; and (iii) demonstrates broad application scope and enhanced performance through data-sharing across computations for a prototype implementation of the proposed semantics.

1 Introduction

Data-centric programming models like MapReduce [7] and Dryad [8] have received considerable attention over the past few years. The success of these models can be attributed to the simplicity of the underlying programming models, support for fault tolerance, and scalable performance. MapReduce adopts deterministic replay for fault-tolerance — compute elements that fail are simply re-executed. In the absence of side-effects, re-executed compute elements produce the same outputs, thus providing clearly specified semantics.

Fault-tolerance through deterministic replay, however, does not work in the presence of side-effects (e.g., writes to persistent storage or communication over the network) or non-deterministic operations (e.g., using a random number generator). Consider a map function writing to the underlying distributed file system. If this instance is replayed (in case of a fault), the re-execution is oblivious of the previous write and hence rewrites the data. Both of these writes are, however, visible to external processes leading to non-deterministic behavior. For this reason, side-effects are not well-supported within the MapReduce framework.

The application scope of MapReduce, and related models can be extended significantly by allowing communication/ data-sharing across computations. Data-sharing through side-effects on shared address space (e.g., a shared disk-resident key-value store) enables speculation and task-parallelism in applications. Consider an illustrative example of finding the minimal spanning tree of a large graph using Boruvka's algorithm. Each iteration (operating on distinct nodes) coalesces a node and its closest neighbor. Iterations in which node-coalescing does not cause conflicts, can be executed in parallel. However, these conflicts can be detected only at runtime, since it depends on the input graph. This form of parallelism is known as speculative-parallelism or amorphous data-parallelism [13]. Exploiting this form of parallelism requires communication across computations to detect and resolve potential conflicts. Further, communication through mutable shared-data helps develop scalable online and streaming applications, such as online aggregation, which need immediate change-propagation.

Towards this goal, we propose effective mechanisms for supporting side-effects over a shared address space. As a model, we use a distributed key-value store (Bigtable [4]) as the underlying storage for MapReduce — the input, output, and side-effects are stored in this fault-tolerant key-value store. Bridging the disparate fault-tolerance mechanisms adopted by the storage (persistence through replication) and computation (deterministic replay) layers presents significant technical challenges relating to definition of semantics, efficient implementations, and application integration.

In this paper, we propose semantics for transactional execution of computations (map/reduce functions) over distributed key value stores, using primitives adapted from Software Transactional Memory (STM) literature. By restricting side-effects only to the key-value store, we derive effective mechanisms for avoiding the consistency problems associated with deterministic replay. In our model, results of one computation (writes to the global key-value store) become atomically visible to other computations, and to other concurrent jobs, upon successful completion of the computation. Though we discuss our semantics in the context of MapReduce and HBase, our proposed semantics apply more generally to all data-centric models over shared address spaces. We support our claims of performance and enhanced application scope in the context of diverse speculative-parallel applications such as Boruvka's minimum spanning tree algorithm and maximum flow calculation using Push-Relabel algorithm. Note that these algorithms cannot be expressed in the current MapReduce framework.

2 TransMR Programming Model

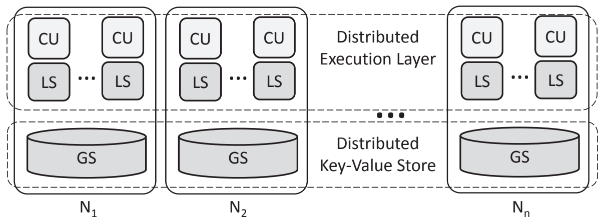


Figure 1: System Architecture

The TransMR (Transactional MapReduce) programming model defines the semantics for transactional execution of computations over shared address spaces. The system architecture, shown in Figure 1, describes interactions between various components. The computation and storage layers span a cluster of nodes. We propose the use of distributed key-value store for the shared global store (GS). The contents of the global store (GS) are accessible to all computation units (CU), albeit through a private local store (LS). Reads/writes from within a CU are served from/to its local store (write buffer). If the local store does not have the data corresponding to a `read`, it fetches the (key, value) pair from the global store. All writes are buffered in LS.

Upon execution of a computation unit CU_i, its write buffer (present in its local store) is validated against the global store for any concurrent accesses of the same data by other CUs. In the absence of such conflicts, the buffered writes are safely written to the global store. However, in case of conflicts, the computation unit (CU_i) is re-executed. Software Transactional memory (STM) systems achieve this behavior by defining transactional execution scope through TM_{BEGIN} and TM_{END} statements. In the TransMR model, each map/reduce function is treated as a computation unit, resulting in their transactional execution. The model supports serializability as the consistency guarantee during validation and commit of conflicting CU transactions.

$$
\begin{array}{rcll}
LocalStore & := & \{\Sigma_1, ..., \Sigma_m\} & (1)\\
GlobalStore & := & \{\Gamma\} & (2)\\
\sigma \in \Sigma & = & L \to Z & (3)\\
\gamma \in \Gamma & = & L \to Z & (4)\\
Fn & := & \{f_m, f_r\} & (5)\\
f \in Fn & := & Atomic\{Op^*\} & (6)\\
Op & := & Get\ k | Put\ (k, v) | Other & (7)\\
b \in Boolean & := & \{True, False\} & (8)\\
k, v \in Values & := & \{b, UnObservable\} & (9)\\
l & := & [v_1, ..., v_n] & (10)
\end{array}
$$

(a) Syntax

$$l, \sigma \implies \sigma(l) \qquad (\text{LOCAL})$$

$$l, \gamma \implies \gamma(l) \qquad (\text{GLOBAL})$$

$$\frac{\textbf{map}\ f_m\ \bar{l}, \gamma \implies \bar{l}'', \gamma'' \qquad \textbf{fold}\ f_r\ \bar{l}'', \gamma'' \implies \bar{l}', \gamma'}{\textbf{TMR}\ f_m\ f_r\ \bar{l}, \gamma \implies \bar{l}', \gamma'} \qquad (\text{TMR})$$

$$\frac{\begin{array}{c}\textbf{if}\ (k \notin domain(\sigma))\ \textbf{then}\ \sigma' = \sigma[k \mapsto \gamma(k)]\\ \textbf{else}\ \sigma' = \sigma\\ k, \sigma' \implies v\end{array}}{Get\ k, \sigma, \gamma \implies v, \sigma', \gamma} \qquad (\text{GET})$$

$$\frac{\sigma' = \sigma[k \mapsto v]}{Put\ (k, v), \sigma, \gamma \implies True, \sigma', \gamma} \qquad (\text{PUT})$$

$$\frac{}{Other, \sigma, \gamma \implies UnObservable, \sigma, \gamma} \qquad (\text{OTHER})$$

$$\frac{\begin{array}{c}Op_1, \sigma, \gamma \implies v_1, \sigma_1', \gamma\\ Op_2, \sigma_1, \gamma \implies v_2, \sigma_2', \gamma\\ ...\\ Op_n, \sigma_{n-1}, \gamma \implies v_n, \sigma_n', \gamma\\ \forall k_i \in domain(\sigma) \qquad m = |\sigma|,\\ \gamma' = \gamma[k_1 \mapsto \sigma(k_1), ..., k_i \mapsto \sigma(k_i), ...k_m \mapsto \sigma(k_m)]\end{array}}{Atomic(\ Op_1,\ Op_2, ...,\ Op_n), \gamma \implies v_n, \gamma'} \qquad (\text{FN})$$

(b) Semantics

Figure 2: Transactional MapReduce: Operational Semantics

Semantics. Figure 2a describes the syntax of our proposed model. Each computation unit has a private local store(Σ) in addition to the shared global store(Γ). We define lookup functions (mapping keys to values), σ and γ, for local and global store lookups, respectively. For each computation unit, σ is empty to begin with; subsequent reads/ writes add mappings. A computation unit is defined as a sequence of operations — read/ write from/ to the store (Get /Put) or a thread local operation (Other) with no side-effects.

The operational semantics, shown in Figure 2b, capture the behavior of the model. The semantics use **map**, **fold**, **if-then-else** constructs, which carry their usual functional definitions. A Transactional MapReduce job (**TMR**) takes an input list, along with map/reduce functions. The job involves applying the computation units (map/reduce functions) on appropriate elements in the input list *atomically*. The possible constituent operations (Get, Put, and Other) are executed in the context of both local and global stores. A `Put`(k, v) operation modifies the local store adding the new key-value pair, (k, v), to the map. A `Get`(k) operation first copies the value from global store to local store if it does not already exist ($k \notin \mathit{domain}(\sigma)$), and subsequently returns the value. Upon successful completion of all operations in the computation, the local store is copied to the global store atomically through a *two-phase commit protocol*. The functional definitions of **map** and **fold** capture the serialized application of functions to list items. In practice, validation protocols are used to achieve this serialization. Our implementation for MapReduce over Bigtable uses optimistic concurrency control to guarantee serializability among concurrent transactional executions of computation units.

3 Design of TransMR Framework

The transactional semantics mentioned above are general enough to be realized using conventional MapReduce-based execution environments (Hadoop[1], Dryad, Pig, *etc.*) operating on typical key-value stores (HBase[2], Cassandra, *etc.*). This section discusses various design considerations and our implementation of the proposed programming model. The TransMR framework uses Hadoop and HBase as the execution and storage engines, respectively. The framework treats map/reduce functions as computation units (CU) executing over the global store, HBase. The map and reduce functions are executed transactionally and upon successful completion, their outputs are stored in HBase. These outputs are visible to other map/ reduce computations of the same MapReduce job, and also to other jobs. As long as the key of a map function output forms the key of HBase table, all values with the same key are versioned using timestamps and stored together. They are implicitly sorted using insertion sort. Thus, a reduce function can directly read its input from HBase, through a scan of all the versioned values for any particular key, avoiding the expensive shuffle phase in Hadoop.

Concurrency Control. The validation-and-commit phase of the CU transaction uses optimistic concurrency control [9]. At the start of its validate-and-commit phase, a transaction increments atomic counters on those GS nodes hosting keys involved in that transaction. The read/write sets of a transaction T_i , are validated against the sets of those transactions that committed their writes, between the start of T_i and the time it increased the atomic counter; the start of T_i is noted by saving the state of the atomic counters at the beginning of the transaction.

In our implementation, the choice of optimistic concurrency control (optimistic reads and write-buffering), as opposed to pessimistic locking, must be noted. This choice is motivated by the nature of clients in data-centric models. Typically, a client can execute at any node (potentially, the slowest) in a heterogeneous distributed environment. Furthermore, the duration of a transaction may be potentially long. In such a scenario, pessimistic locking of rows prevents parallel execution of other transactions with data dependencies. In case of crash failures, these transactions must wait for the system to release all the locks held by the failed transaction. Thus, in fault-prone environments, pessimistic locking could impact the performance significantly. In optimistic concurrency control, reads do not require locks (eager reads) and writes are buffered (lazy writes). During commits, only those concurrent transactions that have conflicts are considered [6]. As no locks are acquired, the possibility of a deadlock is avoided.

Fault Tolerance Model and its Implications on CAP. The client (the process executing the computation units) may be fault-prone and also fault-tolerant in itself. This fault-prone nature is directly implied by the general characteristics of MapReduce based execution environments — run on commodity clusters or virtual machines in the cloud and susceptible to hardware/software faults. The client's fault-tolerant nature implies that, even if the client fails during its execution, its replay mechanism makes the client recover and process all its records. Since the availability of the client is itself in question, expecting high availability from the storage servers is unreasonable. Further, MapReduce based applications demand strict consistency of data to ensure correctness of the algorithm's execution. The above two considerations motivate our choice of Consistency(C) over Availability(A), while accounting for Network Partitions(P) inside a datacenter, during execution of distributed transactions (*i.e.,* choosing C and P in the CAP Theorem [2]). For the duration of the network partition, the system does not allow affected distributed transactions to succeed. Thus, by weakening availability, the framework assures strict consistency of data. In the wake of crash failures, the leases held by compute nodes and storage nodes timeout, leading to replay-based recovery measures for compute nodes and replica-based recovery for storage nodes.

Prototype Implementation. The prototype was implemented by modifying and integrating various parts of Hadoop and transactional HBase. It primarily involved integrating their disparate fault-tolerance mechanisms during execution followed by validation of the CU transaction. Consider the following known corner case in the two-phase-commit protocol: a participant crashes after sending its own vote, but before receiving the commit/abort decision from the coordinator. Upon recov-

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

[2]HBase. http://hbase.apache.org

ery, the participant faces ambiguity over committing the logged read/write sets of the successful validation phase. In our prototype, to resolve this ambiguity, the transaction manager(coordinator) writes its decision to Abort or Commit in the *Global CU log* (a table in HBase), before sending the decision to the nodes involved in the commit. The entry in the *Global CU log* can be identified by a unique *transaction-id*. The recovering storage node uses this id to look up the final decision and complete the write-ahead log, which is later used to regain the consistent state of the failed node. Further, the entry in the *Global CU log* is duplicated to be accessible by using the unique *computation-unit-id*; this helps Hadoop verify the successful execution of a map/reduce function and thereby avert re-execution of it, due to faults or speculative execution. Thus, the *Global CU log* forms a key element in dealing with arbitrary failures of computation and storage, by integrating their disparate fault-tolerance mechanisms.

4 Evaluation

The TransMR programming model allows speculative parallel execution of tasks with potential data dependencies. We demonstrate results on two such applications in this section — Boruvka's minimum spanning tree algorithm, and Preflow Push-Relabel maximum flow computations. We run our experiments on 16 Amazon EC2[3] extra large instances (`c1.xlarge`; each instance has 8 cores and 7 GB RAM).

Boruvka's Minimum Spanning Tree (MST). The sequential version of Boruvka's MST algorithm iterates over nodes in the graph. Each iteration — operating on a node (u) — involves 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); every iteration coalesces two components. The resulting component gives the minimal spanning tree of the input graph.

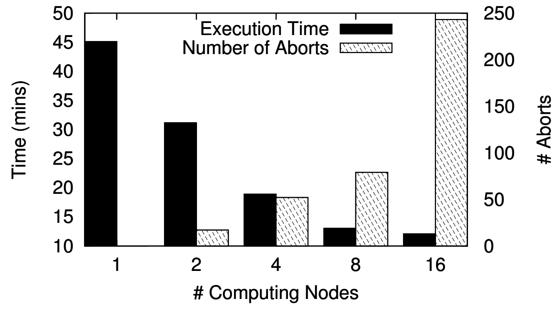
Parallelizing these iterations involves detecting runtime conflicts in case two distinct nodes (u_1, u_2) attempt to coalesce the same node v. Such a formulation is infeasible in traditional MapReduce. In the the TransMR formulation of Borvuka's algorithm, we store the input graph as well as coalescing information, as different column families in HBase. Each row corresponds to one graph node, the adjacency list, and the *node-id* of its parent in the component tree. Each map function, with a single row being its input, parses the adjacency list of a node u, and the adjacency lists of other nodes in its component (obtained by traversing its component tree) to find the closest node v. It then coalesces u's component tree with v's component tree by making one the parent of the other. The algorithm does not need a reduce phase. Instantiations of the same map function on different nodes in the graph might conflict when they both try to coalesce the same component; in this case, the consistency guarantee — serializability among conflicting instantiations — provided by the runtime, is necessary and sufficient for the correctness of algorithm's execution. From a programmer's perspective, the algorithm fits within the regular MapReduce programming model, except that the system needs to handle runtime data-dependencies; the TransMR programming model provides this guarantee to the programmer.

For evaluation, we run Boruvka's algorithm on a 100 thousand node graph with an average degree of 50 generated using the forest fire model of iGraph[4]. The sequential implementation is the same program run on a single node without any speculative parallelism (all maps executed sequentially). Figure 3a plots the average execution time and the number of aborts due to conflicts against the number of machines used. Due to the large number of vertices, the average number of conflicts detected amount to less than 0.5 percent of total executions. We observe upto 3.73 times speedup on 16 nodes. In the initial stages of the algorithm, almost half of the nodes can coalesce with their nearest neighbors without conflicts, leading to abundant parallelism. The available parallelism reduces significantly as the computation progresses. Considering the algorithm's inherent sequential nature due to dependencies, the observed performance gains are significant.

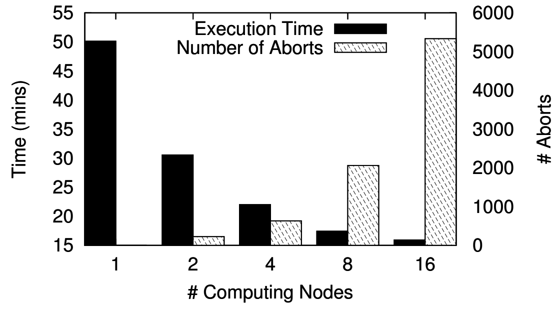
Preflow Push-Relabel. The Preflow Push-Relabel algorithm computes the maximum flow possible through a flow network. The algorithm maintains a preflow — a flow function with the possibility of excess at the vertices — terminating when there is no positive excess. The *Push* operation increases the flow 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 excess preflows. 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. Conflicting executions must be detected at runtime, and hence traditional MapReduce cannot exploit this parallelism. A trivial concurrent implementation is to lock the entire neighborhood of a node before operating on it. This involves significant serialization overhead. An alternate approach is to speculatively execute the operations on all the nodes; detect and resolve con-

[3] Amazon EC2. http://aws.amazon.com/ec2/

[4] iGraph. http://igraph.sourceforge.net



(a) Boruvka's MST



(b) Preflow Push-Relabel

Figure 3: TransMR: Application Performance

flicts at runtime by serializing their execution. The latter approach is adopted by the TransMR programming model.

In the TransMR formulation, each map function operates on one node whose adjacency list is stored as one row in an HBase table. Depending on the neighborhood constraints, the function executes a *Push* or a *Relabel* operation on the node. A *Relabel* operation simply involves increasing the height of the node, if it is less than all the neighboring nodes. In a *Push* operation, a residual edge is chosen from the node's 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 aborted and the corresponding map function is re-executed from the beginning; this ensures serializability. The programmer merely specifies concurrent transactions (maps) and not consider conflict detection or resolution, thus adding *no additional complexity* to programs. The job is iteratively executed until there are no feasible *Push* or *Relabel* operations.

The input flow network is generated using the Washington network generator[5]. The network is a 1000 x 1000 grid with the source connected to all the nodes in the first column and the sink connected to all the nodes in the last column. Every node in a column randomly connects to three other nodes in the next column. The edge weights are randomly generated. The sequential implementation is the same program run on a single node without any speculative parallelism — a single map task executing all the map functions sequentially. Note that the algorithm can only be executed sequentially in the regular MapReduce setup, without any transactional support. Figure 3b shows the average times and associated aborts over a window of 40 iterations of *Push* or *Relabel* operations on the feasible nodes. On the 16 node cluster, the number of aborts (re-executions) amount to about 4% of the total executions. Further, we observe 4.5x speedup on 16 nodes. As before, speculative execution enables a meaningful performance gain, as compared to the baseline case where no parallelism could be exploited.

5 Discussion and Related Work

Applicability. While our evaluation describes only two applications, the TransMR framework is applicable to *all* applications exhibiting speculative-parallelism [13]. Furthermore, applications suited to transactional memory systems (concurrent threads modifying a shared data-structure) and pipelined workflows, such as those present in the STAMP benchmark suite [3], can be easily formulated in the TransMR programming model. The model also suits producer-consumer based online applications needing immediate access to mutable shared data. The model trivially allows regular data-parallel applications; however, the involved setup costs for transactional support might lead to minor overheads. By implicitly executing each computation transactionally instead of explicit scope definitions (*begin, end statements*), TransMR model offers increased applicability without increasing the programming complexity. As in any data-centric programming model, the programmer only needs to specify the operation on the specific data-element without being concerned about its runtime interaction with other operations.

Performance Improvement. Distributed transactions constitute the primary overhead in the TransMR model. It should be noted that the number of keys involved in a distributed transaction is typically small, because the read-write sets of computations (map/reduce functions), where the keys come from, are small. The performance of TransMR framework can be significantly improved by using locality-enhancing storage schemes, leading to

[5]Washington max flow network generator. http://www.avglab.com/andrew/CATS/maxflow/synthetic.htm

localization of distributed transactions. To further mitigate the overhead of distributed transactions, application-specific optimizations such as relaxing consistency guarantees from serializability to snapshot isolation, as used in Percolator [12] or reducing the transaction scope to a subset of data-items, as used in Megastore [1], can be employed. While realizing these optimizations constitutes our future work, the primary goal of this paper is to advocate the transactional programming model and its benefits.

Related Work. The TransMR model supports transactional execution of distributed computations through the notion of a mutable shared state. Spark [15] and Piccolo [14] propose the use of shared state for distributed computations to achieve different goals. Spark uses read-only shared data to build working-sets for concurrent map/reduce function invocations. Piccolo proposes the use of mutable in-memory tables to store data shared by concurrent threads. Piccolo's fault-tolerance and recovery model based on periodic, user-assisted checkpointing through distributed snapshots makes it hard to realize (efficient) transactional execution. Specifically, when any of the nodes fail, all of them have to be halted and rolled back to a consistent snapshot; unless checkpointing is executed at high frequency, it is hard to reason about the transactional behavior of processes and the consequent effect-propagation through shared state.

Google proposed Pregel [10] for large-scale graph-processing, based on the bulk-synchronous parallel (BSP) programming model. Pregel (or BSP) does not support transactions and hence disallows speculative execution. Realizing Boruvka's MST application in Pregel, consequently, would involve algorithmically identifying and executing the non-conflicting operations at each stage. To avoid conflicting operations, the algorithm should be executed as a series of iterations (called steps in Pregel). Each iteration needs to compute the set of non-conflicting operations and execute them. Computing the set of non-conflicting operations is itself quite involved – making the program significantly more sophisticated.

In recent years, several systems have been proposed to increase the applicability of MapReduce. MapReduce Online [5] streams the data between map and reduce phases supporting pipelined execution, continuous queries, and online aggregation. Dryad [8] supports acyclic tasks and CIEL [11] adds support for dynamic task graphs particularly useful for dynamic-programming based applications. While these efforts have similar goals of increasing applicability, they do not address applications with multiple computational units accessing shared data-structures in a faulty environment.

6 Conclusion

In this paper, we propose TransMR programming model to enable data-sharing in data-centric programming models for enhanced applicability. We define the semantics for transactional execution of MapReduce computations over shared address space. Through a prototype implementation of the proposed semantics, we demonstrate the applicability of the TransMR programming model in the context of applications exhibiting speculative parallelism.

References

[1] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. *CIDR'11*, 2011.

[2] E. A. Brewer. Towards robust distributed systems (abstract). *PODC*, 2000.

[3] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. *IISWC*, 2008.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *OSDI*, 2006.

[5] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. *NSDI*, 2010.

[6] O. S. D. Dice and N. Shavit. Transactional locking ii. In *DISC*, 2006.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, 2004.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *Eurosys*, 2007.

[9] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, 2010.

[11] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *NSDI*, 2011.

[12] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, 2010.

[13] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *PLDI*, 2011.

[14] R. Power and J. Li. Piccolo: building fast, distributed programs with partitioned tables. In *OSDI*, 2010.

[15] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, 2010.