

Relaxed Synchronization and Eager Scheduling in MapReduce

Karthik Kambatla Naresh Rapolu Suresh Jagannathan Ananth Grama

Department of Computer Science, Purdue University
{kkambatl, nrapolu, suresh, ayg}@cs.purdue.edu

Abstract

MapReduce has rapidly emerged as a programming paradigm for large-scale distributed environments. While the underlying programming model based on `maps` and `reduces` has been shown to be effective in specific domains, significant questions relating to performance and application scope remain unresolved. This paper targets questions of performance through relaxed semantics of underlying `map` and `reduce` constructs in iterative MapReduce applications like eigenspectra/pagerank and clustering (k-means). Specifically, it investigates the notion of partial synchronizations combined with eager scheduling to overcome global synchronization overheads associated with `reduce` operations. It demonstrates the use of these partial synchronizations on two complex problems, illustrative of much larger classes of problems.

The following specific contributions are reported in the paper: (i) partial synchronizations combined with eager scheduling is capable of yielding significant performance improvements, (ii) diverse classes of applications (on sparse graphs), and data analysis (clustering) can be efficiently computed in MapReduce on hundreds of distributed hosts, and (iii) a general API for partial synchronizations and eager `map` scheduling holds significant performance potential for other applications with highly irregular data and computation profiles.

1. Introduction

Motivated by the large amounts of data generated by web-based applications, scientific experiments, business transactions, etc., and the need to analyze this data in effective, efficient, and scalable ways, there has been significant recent activity in developing suitable programming models, runtime systems, and development tools. The distributed nature of data sources, coupled with rapid advances in networking and storage technologies naturally lead to abstractions for supporting large-scale distributed applications. The inherent heterogeneity, latencies, and unreliability, of underlying platforms makes the development of such systems challenging.

With a view to supporting large-scale distributed applications in unreliable wide-area environments, Dean and

Ghemawat proposed a novel programming model based on `maps` and `reduces`, called MapReduce [5]. Programs in MapReduce are expressed as `map` and `reduce` operations. The `map` phase takes in a list of key-value pairs and applies a programmer-specified function independently on each pair in the list. The `reduce` phase operates on a list indexed by a key, of all corresponding values, and applies the `reduce` function on the values; and outputs a list of key-value pairs. The inherent simplicity of this programming model, combined with underlying system support for scheduling, fault tolerance, and application development, made MapReduce an attractive platform for diverse data-intensive applications. Indeed, MapReduce has been used effectively in a wide variety of data processing applications. The large volumes of data processed at Google, Yahoo, and Amazon, stand testimony to the effectiveness and scalability of the MapReduce paradigm. Hadoop MapReduce¹, the open source implementation of MapReduce, is in wide deployment and use.

A majority of the applications currently executing in the MapReduce framework have a data-parallel, uniform access profile, which makes them ideally suited to `map` and `reduce` abstractions. Recent research interest, however, has focused on more unstructured applications that do not lend themselves naturally to data-parallel formulations. Common examples of these include sparse unstructured graph operations (as encountered in diverse domains including social networks, financial transactions, and scientific datasets), discrete optimization and state-space search techniques (in business process optimization, planning), and discrete event modeling. For these applications, there are two major unresolved questions: (i) can the existing MapReduce framework effectively support such applications in a scalable manner? and (ii) what enhancements to the MapReduce framework would significantly enhance its performance and scalability without compromising desirable attributes of programmability and fault tolerance?

This paper primarily focuses on the second question – namely, it seeks to extend the MapReduce semantics to support specific classes of unstructured applications on large-scale distributed environments. Recognizing that one of the key bottlenecks in supporting such applications is the global

¹Hadoop. <http://hadoop.apache.org>

synchronization between the map and reduce phases, it introduces notions of partial synchronization and eager scheduling. The underlying insight is that for an important class of applications, algorithms exist that do not need global synchronization for correctness. Specifically, while global synchronizations optimize serial operation counts, violating these synchronizations merely increases operation counts without impacting correctness of the algorithm. Common examples of such algorithms include, computation of eigenvectors (pageranks) through (asynchronous) power methods, branch-and-bound based discrete optimization with lazy bound updates, and other heuristic state-space search algorithms. For such algorithms, a global synchronization can be replaced by disjoint partial synchronizations. However, this partial synchronization must be combined with suitable locality enhancing techniques to minimize the adverse effect on operation counts. These locality enhancing techniques take the form of min-cut graph partitioning and aggregation in graph analyses, periodic quality equalization in branch-and-bound, and other such operations that are well known in the parallel processing community. Replacing global synchronizations with partial synchronizations also allows us to schedule subsequent maps in an eager fashion. This has the important effect of smoothing load imbalances associated with typical applications. This paper combines partial synchronizations, locality enhancement, and eager scheduling, along with algorithmic asynchrony to deliver distributed performance improvements of 100 to 200% (and beyond in several cases). The enhanced programming semantics resulting in the significant performance improvement do not impact application programmability.

We demonstrate all of our results on the wide-area Google cluster in the context of *PageRank* and clustering (*K-Means*) implementations. The applications are representative of a broad set of application classes. They are selected because of their relative simplicity as well as their ubiquitous interaction pattern. Our extended semantics have a much broader application scope.

The rest of the paper is organized as follows: in Section 2, we provide a more comprehensive background on MapReduce, Hadoop, and motivate the problem; in Section 3, we outline our primary contributions and their significance; in Section 4, we present our new semantics and the corresponding API; in Section 5, we discuss our implementations of *PageRank* and *K-Means* clustering and analyze the performance gains of our approach. We outline avenues for ongoing work and conclusions in Sections 8 and 9.

2. Background and Motivation

The primary design motivation for the functional MapReduce abstractions is to allow programmers to express simple concurrent computations, while hiding the messy details of parallelization, fault-tolerance, data distribution and load balancing in a single library [5]. The simplicity of

the API makes programming extremely easy. Programs are expressed as sequences (iterations) of map and reduce phases with intervening synchronizations. The reduce phase must wait for all the map tasks to complete as it requires all the values corresponding to each key. Once the reduce phase terminates, the next set of map tasks can be scheduled. Fault tolerance is achieved by rescheduling maps that fail.

As may be expected, for many applications, the dominant overhead in the program is associated with the global synchronizations. When executed in wide-area distributed environments, these synchronizations often incur substantial latencies associated with underlying network and storage infrastructure. In contrast, partial synchronizations (over a limited set of computing nodes) take over an order of magnitude less time. The obvious question that follows from these observations is whether we can decrease the number of global synchronizations, perhaps at the expense of increased number of partial synchronizations (we define a partial synchronization to imply a synchronization/reduction only across a subset of maps). The resulting algorithm(s) may be suboptimal in serial operation counts, but may be more efficient and scalable in a MapReduce framework. A particularly relevant class of algorithms where such tradeoffs are possible are iterative techniques applied to unstructured problems (where the underlying data access patterns are unstructured). This broad class of algorithms underlies applications ranging from *PageRank* to sparse solvers in scientific computing applications and clustering algorithms.

We seek to answer the following key questions relating to the application scope and performance of MapReduce in the context of applications that tolerate algorithmic asynchrony: (i) what are suitable abstractions (MapReduce extensions) for distributed asynchronous algorithms? (ii) for an application class of interest, can the performance benefits of localization, partial synchronization, and eager scheduling of maps overcome the sub-optimality in terms of serial operation counts, and (iii) can this framework be used to deliver scalable and high performance over wide-area distributed systems?

3. Technical Contributions

To address the cost of global synchronization and granularity, we propose and validate the following solutions within the MapReduce framework:

- We provide support for global *and* partial synchronizations. Partial synchronizations are enforced only on a subset of maps. Global synchronizations are identical to reduce operations, enforced on all the maps.
- Following partial synchronizations, the subsequent maps are scheduled in an eager fashion, *i.e.*, as soon as the partial synchronization operation completes.

- We use partial synchronizations and eager scheduling in combination with a coarser grained, locality enhancing allocation of tasks to `mapS`.
- We validate the aforementioned techniques to compute *PageRank* on sparse unstructured (power-law type) real web graphs and to cluster high-dimensional data using unsupervised clustering algorithms, namely *K-Means*. These applications are illustrative of a broader class of asynchrony-tolerant algorithms. We show that while the number of serial operations (the sum of partial and global reductions) is much higher, the reduction in number of global reductions yields performance improvements of over 200% compared to traditional MapReduce implementation on a 460-node cluster provided by IBM-Google consortium as part of the CluE NSF program.

To alleviate the overhead of global synchronization, we propose a two level scheme — a *local reduce* operation applies the specified reduction function to only those key-value pairs emanating from the preceding `map`(s) at the same host. A *global reduce* operation, on the other hand, incurs significant network and file system overheads. We illustrate the use of these primitives in the context of *PageRank* and *K-Means* algorithms. Consider *PageRank*, where the rank of a node is determined by the rank of its neighbors. In the traditional MapReduce formulation, in every iteration, `map` involves each node pushing its PAGERANK to all its outlinks and `reduce` accumulates all neighbors’ contributions to compute PAGERANK for the corresponding node. These iterations continue until the PAGERANKS converge. An alternate formulation would be where the graph is partitioned, and each `map` now corresponds to the local *PageRank* computation of all nodes within the sub-graph(partition). For each of the internal nodes (nodes that have no edges leaving the partition), a partial reduction accurately computes the rank (assuming the neighbors’ ranks were accurate to begin with). On the other hand, boundary nodes (nodes that have edges leading to other partitions) require a global reduction to account for remote neighbors. It follows therefore that if the ranks of the boundary nodes were accurate, ranks of internal nodes can be computed simply through local iterations. Thus follows a two-level scheme wherein partitions (`mapS`) iterate on local data to convergence and then perform a global reduction. It is easy to see that this two-level scheme increases the serial operation count. Moreover, it increases the total number of synchronizations (partial + global) compared to the traditional formulation. However, and perhaps most importantly, it reduces the number of global reductions. Since this is the major overhead, the program has significantly better performance and scalability.

Indeed optimizations such as these have been explored in the context of traditional HPC platforms as well with some success. However, the difference in overhead between a partial and global synchronization in relation to the intervening useful computation is not as large for HPC platforms. Con-

sequently, the performance improvement is significantly amplified on distributed platforms. It also follows thereby that performance improvements from MapReduce deployments on wide-area platforms, as compared to single processor executions are not expected to be significant unless the problem is scaled significantly to amortize overheads. However, MapReduce formulations are motivated primarily by the distributed nature of underlying data and sources, as opposed to the need for parallel speedup. For this reason, performance comparisons must be with respect to traditional MapReduce formulations, as opposed to speedup and efficiency measures more often used in the parallel programming community. This is further reinforced by the fact that in cloud computing environments, it is difficult to even estimate the resources available to a program.

While most of our development efforts and all of our validation results are in the context of *PageRank* and *K-Means*, concepts of partial reductions combined with locality enhancing techniques and eager `map` scheduling have wider applicability. In general, our semantic extensions apply to this wider class of applications that admit asynchronous algorithms – algorithms for which relaxed synchronization impacts only performance, and not correctness.

4. Proposed Semantic Extensions

As mentioned earlier, numerous applications use MapReduce iteratively. Hence, improving the performance of iterative MapReduce is very important, specifically because of the high overhead due to synchronization and starting/stopping of MapReduce jobs (input and output data handling).

In this section, we present the semantics and API that we propose for iterative MapReduce to alleviate synchronization overheads, preserving the programming simplicity.

4.1 Semantics for Iterative MapReduce

Figure 1 describes the formal syntax of our proposed language. Lists form an interesting value in our language as MapReduce operates on lists and not on its elements. Any operation on an element of the list has to be defined in terms of an abstraction, so that our MapReduce constructs can evaluate them on the elements to compute new lists. We define **L**, **G**, and **I** as the local, global, and iterative versions of MapReduce; one invoked from another. These operators evaluate on a tuple $\langle \text{condition}, \text{map function}, \text{reduce function}, \text{list} \rangle$. Programs in the language are defined as applications of our iterative MapReduce applied to functions and an input list.

To capture salient behavior of the system, we assume a set of processors and a set of associated stores. We define look up functions, σ (local) and λ (global). We also define two different evaluation rules for MapReduce — \implies_g for the evaluation rules that change the global state, and \implies_l to describe the evaluations that change the local state.

	$\lambda \in \Lambda$	=	$L \rightarrow Z$	(6)
	$f \in Fn$:: =	$\lambda x.e$	(7)
	$l \in List$:: =	$[v_1, \dots, v_n]$	(8)
	$e \in P$:: =	f	(9)
			$Apply(\mathbf{I}, \langle e, f_m, f_r, l \rangle)$	(10)
$v \in Value$				(1)
$p \in Processors$	=		$\{P_1, \dots, P_m\}$	(2)
$\Sigma \in LocalStore$	=		$\{\Sigma_1, \dots, \Sigma_m\}$	(3)
$\Lambda \in GlobalStore$	=		$\{\Lambda\}$	(4)
$\sigma \in \Sigma$	=		$L \rightarrow Z$	(5)

Figure 1. Iterative Relaxed MapReduce: Syntax

$l, \sigma \Rightarrow \sigma(l)$ (LOCAL-LOOKUP)	$l, \lambda \Rightarrow \lambda(l)$ (GLOBAL-LOOKUP)
$Apply(\mathbf{I}, \langle e, f_m, f_r, l \rangle) \Rightarrow_g \mathbf{I} e f_m f_r l$ (APPLY-ITER)	
$\frac{\mathbf{while}(cond_g) \mathbf{G} cond_l f_m f_r l_g, \lambda \Rightarrow_g l_g', \lambda'}{\mathbf{I} cond_g f_m f_r l_g, \lambda \Rightarrow_g l_g', \lambda'}$ (ITER-MAPRED)	
$\frac{\mathbf{while} cond \mathbf{map} (\mathbf{L} f_m f_r) \bar{l}_l, \sigma \Rightarrow_l \bar{l}_l', \sigma' \mathbf{agg} \bar{l}_l', \sigma, \lambda \equiv l_g', \sigma, \lambda' \mathbf{fold} f_r l_g', \lambda \Rightarrow_g l_g'', \lambda'}{\mathbf{G} cond f_m f_r l_g, \lambda, \sigma \Rightarrow_g l_g'', \lambda', \sigma'}$ (MAPRED-GLOBAL)	
$\frac{\mathbf{map} f_m l_l, \sigma \Rightarrow_l l_l'', \sigma'' \quad \mathbf{fold} f_r l_l'', \sigma \Rightarrow_l l_l', \sigma'}{\mathbf{L} f_m f_r l_l, \sigma \Rightarrow_l l_l', \sigma'}$ (MAPRED-LOCAL)	
$ch l_g \equiv \bar{l}_l \equiv \{l_l \mid l_l \subset l_g \ \& \ \cap_{l_l, l_l \in \bar{l}_l} l_l = \phi\}; \forall l_i, \sigma_i [l_i \mapsto \lambda(l)]$ (CHUNKIFY)	
$agg \bar{l}_l \equiv l_g \equiv \cup_{l_l \in \bar{l}_l} l_l; \forall l_i, \lambda [l \mapsto l_i]$ (AGGREGATE)	

Figure 2. Iterative Relaxed MapReduce: Semantics

Figure 2 describes the semantics. A typical program in the language is an application of `IterMapReduce` (**I**) to a tuple consisting of a termination condition, map and reduce functions and the list of data. The computation rules [Apply-Iter] evaluate to running global `IterMapReduce`. **I**, the iterative `MapReduce` takes `cond`, `map`, `reduce` and a list to operate on. The `cond` function operates on the global heap until the termination condition is met. Iterative `MapReduce` calls global `MapReduce` iteratively till this condition is met.

[Mapred-Global] describes the behavior of global `MapReduce`. This operation uses another condition to determine the termination of the local `MapReduce` and operates on data local to the `MapReduce` it gets associated with. The main feature is to have a mechanism of synchronization across the local and global stores. This is achieved by our `Chunkify` and `Aggregate` functions. `Chunkify`(`ch`) partitions the global list into sequence of multiple lists each made available to the local `MapReduce` and copies these chunks to the respective local stores. `Aggregate` takes care of aggregating the data from the multiple local `MapReduce` computations and

copies the changes to global store. During the evaluation of global `MapReduce`, we first `Chunkify` the data. We then have the required values in the local store. We subsequently apply the standard `map`, to a curried version of local `MapReduce` — (**L** `fm fr`) — and the local list, iteratively until it terminates. Only the local store is modified during this stage. At the end of these iterations, we need to synchronize globally, requiring us to copy data back to the global store, this task performed by the `Aggregate` function.

[Mapred-Local] controls the behavior of local `MapReduce`. This is very close in structure to the original `MapReduce` model, with the main difference being that it operates locally, with no global store changes, as explained in the semantics.

The semantics use **while**, **map**, and **fold** constructs (bold faced in the semantics) which carry their usual functional definitions. From a systems perspective, **map/fold** functions process the data given to them in parallel, and collect the output.

In addition to the inputs to the regular MapReduce, our iterative version requires a termination condition. Such a termination condition has to be thought of even for iterating over traditional MapReduce. Also, our iterative MapReduce reduces to traditional MapReduce if the local condition is set to one iteration. Our semantics do not increase programming complexity.

4.2 API

The rigorous semantics advocate a source-to-source translator for iterative MapReduce with relaxed synchronization and eager scheduling. We formulated a simple-to-implement API to validate our claims, as discussed below —

In the traditional MapReduce API, the user provides *map* and *reduce* functions along with the functions to split and format the input data. In addition to these, for iterative MapReduce applications, the user must provide functions for termination of global and local MapReduce iterations, and functions to convert data into the formats required by the local *map*, *reduce* functions. The *Chunkify* and *Aggregate* functions discussed in the semantics are built using these conversion functions.

To provide more flexibility to the programmer and to allow better exploitation of application-specific optimizations, we propose four new functions — *gmap*, *greduce*, *lmap* and *lreduce* (two different sets of map and reduce for the local and global versions of the operations). *Global map* uses thread pool(s) to schedule *lmap* and *lreduce* to exploit available parallelism. Functions *Emit()* and *EmitIntermediate()* support data-flow in traditional MapReduce. We introduce their local equivalents — *EmitLocal()* and *EmitLocalIntermediate()*. *Aggregate* iterates over the data emitted by *EmitLocal()* and sends it to the global *reduce* through the *EmitIntermediate()* function.

Our API has been validated in our implementation of *PageRank* and *K-Means* algorithms used in the evaluation of these relaxed semantics.

5. Evaluation

To evaluate the performance benefits from our proposed semantics, we consider two applications — *PageRank* and *K-Means*. We compare general MapReduce implementations of these applications to their modified implementations that exploit relaxed synchronization and eager scheduling; *PageRank* is more generally, computing the eigenspectra of a matrix and *K-Means* represents clustering algorithms).

Table 1. Measurement testbed, Software

Clue Cluster Machines - 460	Intel(R) Xeon(TM) CPU 2.80GHz 4 GB RAM, 2x 400 GB hard drives
VM	1 VM per host
Software	Hadoop 0.17.3, Java 1.6
Heap space	1 GB per slave

In our attempt to model our experiments in a real-life setting, we use the 460 node cluster provided by IBM-Google consortium as part of the CluE (Cluster Exploratory) NSF program.

Table 1 lists the physical resources, software and restrictions on the cluster. Since this is a shared cluster, there is the potential that several jobs are concurrently executing during our experiments. Nonetheless, our results strongly validate our insights.

5.1 Implementation

For both applications, we implemented base versions which conform to the popular MapReduce implementations known for these applications. We also implement modified versions; relaxed synchronization and eager scheduling realized by rewriting the *global map* function (*gmap*) to have a local MapReduce (*lmap* and *lreduce*) as described by the following pseudo-code -

```

gmap(xs : X list) {
  while(no-local-convergence-intimated) {
    lmap(x); // emits (lkey, lval)

    lreduce();
  }

  for each value in lreduce-output{
    EmitIntermediate(key, value);
  }
}

```

When a regular map phase is given an input of $\langle \text{key}, \text{value} \rangle$ list, we give each map task a smaller list (instead of an element of the list).

The argument to *gmap* is a $\langle \text{key}, \text{value} \rangle$ list(*xs*). We run a *local* MapReduce with each element of *xs* as input to *lmap*. We use a hashtable to store the intermediate and final results of the local MapReduce.

Applying this method to any application translates to knowing the local termination condition. The local termination can be convergence (as in *PageRank* and *K-Means*) or a particular number of iterations.

5.2 PageRank

The *PageRank* of a node is the scaled sum of the pageranks of all of its neighbors. Mathematically, the *PageRank* of a node is given by the following expression:

$$PR_d = (1 - \chi) + \chi * \sum_{(s,d) \in E} s.pagerank / s.outlinks \quad (11)$$

where χ is the damping factor, *s.pagerank* and *s.outlinks* correspond to the pagerank and the out-degree of the source node, respectively.

For both the implementations, the input is a graph represented as adjacency lists. We start with a pagerank of 1 for all nodes. The pageranks converge to their correct values after a few iterations of applying the above expression for each

node. We define convergence as the pageranks of all nodes not changing by more than a particular threshold (10^{-5} in our case) in subsequent iterations.

5.2.1 General PageRank

The general MapReduce implementation of *PageRank* iterates over a map task that emits the pageranks of all the source nodes to the corresponding destinations in the graph, and a reduce task that accumulates the pagerank contributions from various sources to a single destination.

In the actual implementation, the map function emits tuples of the type $\langle \textit{destination-node}, \textit{pagerank contributed to this destination node by the source} \rangle$. The reduce task operates on a destination node, which gathers the pageranks from its incoming source nodes and computes a new pagerank for itself. Thus, after every iteration, the nodes have renewed pageranks which propagate through the graph in subsequent iterations till they converge. One can observe that even a small change in the pagerank of one node is broadcast to all the nodes in the graph in successive iterations of MapReduce, incurring a potentially significant global synchronization cost.

Our baseline for performance comparison is a MapReduce implementation for which maps correspond to complete partitions, as opposed to single node updates. We use this as a baseline because the performance of this formulation was noted to be on par or better than the adjacency-list formulation where the update of a single node is associated with a map. For this reason, our baseline provides a more competitive implementation.

5.2.2 Eager PageRank

We begin our description of Eager *PageRank* with an intuitive description of how the underlying algorithm accommodates asynchrony. In a graph with a power-law type distribution, one may assume that each hub is surrounded by a large number of spokes, and that inter-hub edges are comparatively infrequent. This allows us to relax strict synchronization on inter-hub edges until the subgraph in the proximity of a hub has relatively self-consistent pageranks. Disregarding the inter-hub edges does not lead to algorithmic inconsistency since, after every few local iterations of MapReduce calculating the pageranks in the subgraph, there is a global synchronization (following a *global map*) leading to a dissemination of the pageranks in this subgraph to other subgraphs via inter-hub edges. This propagation reimposes consistency on the global state.

Consequently, we update only the (neighboring) nodes in the smaller subgraph. We achieve this by a set of iterations of *local* MapReduce as described in the semantics. Evidently, this method leads to improved efficiency if each map operates on a hub or a group of topologically localized nodes. Such topology is inherent in the way we collect data as it is crawler-induced. One can also use one-time graph partition-

ing using tools like Metis [11]. We use Metis as our synthetic data set is not inherently partitioned.

In the Eager *PageRank* implementation, the map task operates on a sub-graph. *Local* MapReduce, within the *global map*, computes the pagerank of the constituent nodes in the sub-graph. Hence, we run the *local* MapReduce to convergence.

Instead of waiting for all the other *global map* tasks operating on different subgraphs, we eagerly schedule the next *local map* and *local reduce* iterations on the individual sub-graph inside a single *global map* task. Upon local convergence of the subgraphs, we synchronize globally, so that all nodes can propagate their computed pageranks to other sub-graphs. This iteration over *global* MapReduce runs to convergence. Such an Eager *PageRank* incurs more computational cost, since local reductions may proceed with imprecise values of global pageranks. However, the pagerank of any node propagated during the *global reduce* is representative, in a way, of the sub-graph it belongs to. Thus, one may observe that the *local* and *global reduce* functions are functionally identical.

Note that in Eager *PageRank*, the *local reduce* waits on a *local* synchronization barrier, while the *local maps* can be implemented on a thread pool on a single host in a cluster. The local synchronization does not incur any inter-host communication delays. This makes the *local* overheads considerably lower than the *global* overheads.

5.2.3 Input data

Table 2. Input graph properties

Input graphs	Stanford webgraph	Power-law
Nodes	280,000	100,000
Edges	3 million	3 million
Damping factor	0.75	0.85

Table 2 describes the two graphs that are considered for our experiments on *PageRank*, both conforming to power law distributions. Our primary validation dataset is the Stanford web graph from the Stanford WebBase project [14]. It is a medium-sized crawl conducted in 2002 of 280K nodes and about 3 million edges. To study the impact of problem size and related parameters, we also rely on synthetic graphs with power-law distributions, generated through preferential attachment [4] in *igraph* [7]. The algorithm used to create the synthetic graph is described below, along with its justification.

Preferential attachment based graph generation The graph is generated by adding vertices one at a time — connecting it to *numConn* vertices already in the network, chosen uniformly at random. For each of these *numConn* vertices, *numIn* and *numOut* of its inlinks and outlinks are chosen uniformly at random and connected to the joining vertex. This is done for all of the newly connected nodes to the incoming vertex. This method of creating a graph is closely

related to the evolution of the web. This procedure increases the probability of a highly reputed site getting linked to new sites, since it has a greater probability of being in an inlink from other randomly chosen sites.

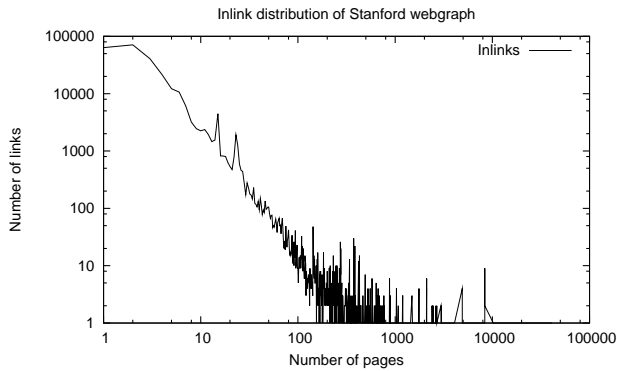


Figure 3. Inlink distribution of the Stanford webgraph

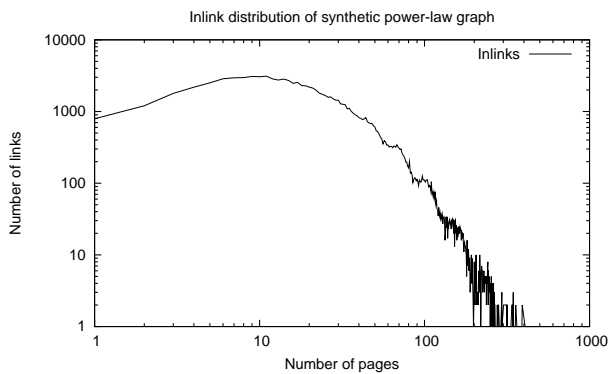


Figure 4. Inlink distribution of the synthetic power-law graph

Figures 3 and 4 show the distribution of inlinks for the two input graphs. The best line fit gives the power-law exponent for the two graphs showing their conformity with a hubs-and-spokes model. Very few nodes have a very high inlink value, emphasizing our point that very few nodes require frequent global synchronization. More often than not, even these nodes (hubs) mostly have spokes as their neighbours.

Crawlers inherently induce locality in the graphs as they crawl neighborhoods before crawling remote sites. As we also use synthetic data in addition to the Stanford Web Graph, we partition it using Metis. A good partitioning algorithm that minimizes edge-cuts has the desired effect of reducing global synchronizations as well. Metis, though not tailored for power-law graphs, does a decent partitioning of the graph. This partitioning is performed off-line (only once) and takes about 5 seconds which is negligible compared to the runtime of *PageRank*, and hence is not included in the reported numbers.

5.2.4 Results

To show the dependence of performance on global synchronizations, we vary the number of iterations of the algorithm by altering the number of partitions the graph is split into. Fewer partitions result in a smaller number of larger sub-graphs. Each `map` task does more work and would normally result in fewer global iterations in the relaxed case. The fundamental observation here is that it takes fewer iterations to converge for a graph having already converged subgraphs. The trends are more pronounced when the graph follows the power-law distribution more closely. In either case, the total number of iterations are lesser than the general case. For Eager *PageRank*, if the number of partitions is decreased to one, the whole graph is given to one *global map* and its local MapReduce would compute the final pageranks of all the nodes. If the partition size is one, each partition gets a single adjacency list, Eager *PageRank* becomes regular *PageRank*, because each `map` task operates on a single node.

Figures 5 6 show the number of iterations taken by the relaxed and general implementations of *PageRank*, on the Stanford and synthetic webgraphs that we use for input, as we vary the number of partitions. The number of iterations does not change in the general case as each iteration performs the same work irrespective of the number of partitions and partition sizes.

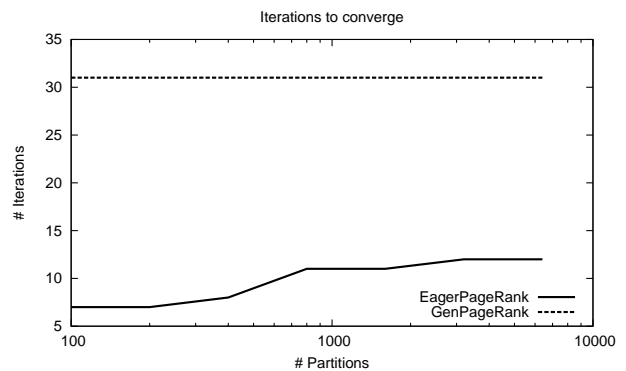


Figure 5. Number of Iterations to converge (on y-axis) for different number of Partitions (on x-axis) for the Stanford webgraph for a damping factor of 0.75

The results for Eager *PageRank* are consistent with our predictions. The number of global iterations decrease with the number of partitions. For both graphs, the number of iterations to converge increases monotonically with the number of partitions.

The time to solution depends strongly on the number of iterations but is not completely determined by it. It is true that the global synchronization costs would decrease, but when we reduce the number of partitions significantly, the work to be done by each `map` task increases. This increase potentially results in increased cost of computation, even more than the benefit of decrease in communication. Hence,

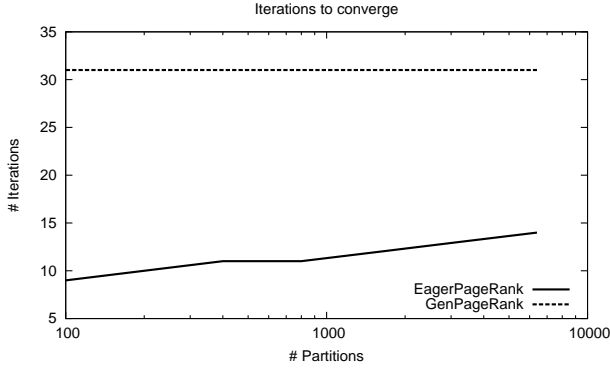


Figure 6. Number of Iterations to converge(on y-axis) for different number of Partitions(on x-axis) for the synthetic webgraph for a damping factor of 0.85

there would be an optimal number of partitions for every application on a given cluster. For a well-partitioned graph we expect an inverted bell-curve.

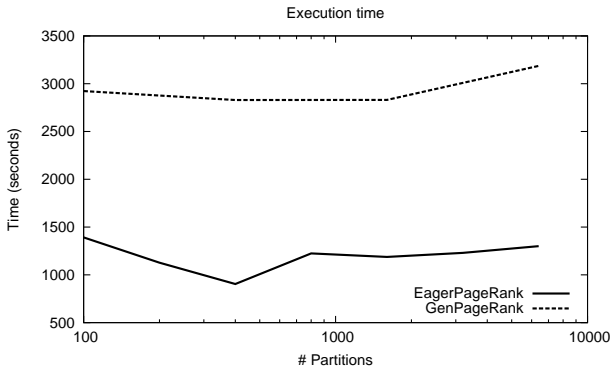


Figure 7. Time to converge(on y-axis) for various number of Partitions(on x-axis) for the Stanford webgraph for a damping factor of 0.75

Figures 7 and 8 show the runtimes for the relaxed and general implementations of *PageRank* on the stanford and synthetic webgraphs with varying number of partitions. One may notice the significant performance gains the relaxed case achieves over the general case for both graphs. On an average, we see 2x to 3x improvement in runtimes. The improvement in the case of synthetic graph is expected to be more uniform and consistent, which can be attributed to its close conformance to a power-law distribution, leading to more accurate partitioning. This in turn leads to a better load balancing on the `map` tasks along with faster convergence inside the subgraph. This is observed consistently in our experiments.

Finally, given that both graphs exhibit power-law characteristics, we notice their respective points of optimality, 400 partitions for the Stanford webgraph and 200 for the synthetic webgraph. We also observe that the smaller the graph, the lower this optimal number.

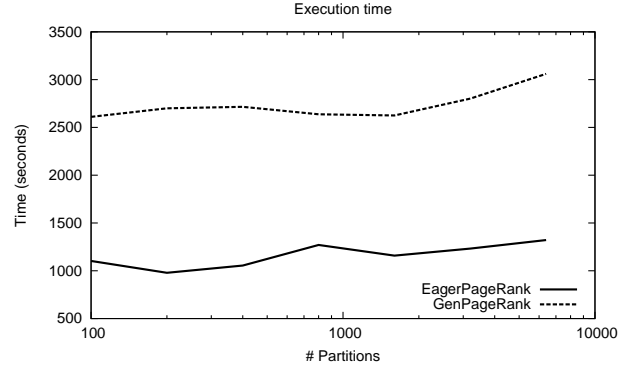


Figure 8. Time to converge(on y-axis) for various number of Partitions(on x-axis) for the synthetic webgraph for a damping factor of 0.85

5.3 *K-Means*

K-Means is a popular technique used for unsupervised clustering. Implementation of the algorithm in the MapReduce framework is straightforward as shown in [15, 3]. Briefly, in the `map` phase, every point chooses its closest cluster centroid and in the `reduce` phase, every centroid is updated to be the mean of all the points which chose the particular centroid. The iterations of `map` and `reduce` phases continue until the centroid movement is below a given threshold. Euclidean distance metric is usually used to calculate the centroid movement.

In the proposed Relaxed MapReduce framework, each *global map* handles a unique subset of the input points. The *local map* and *reduce* iterations inside the *global map*, cluster the given subset of the points using the common input-cluster-centroids. Once convergence is achieved in the local iterations, the *global map* emits the input-cluster-centroids and their associated updated-centroids. The *global reduce* calculates the final-centroids which is the mean of all updated-centroids corresponding to a single input-cluster-centroid. The final-centroids form the input-cluster-centroids for the next iteration. These iterations continue until the input-cluster-centroids converge.

The algorithm used in the relaxed approach to *K-Means* is similar to the one recently shown by Tom-Yov and Slonim [19] for pairwise clustering. An important observation from their results is that the input to the *global map* should not be the same subset of the input points in every iteration. For every few iterations the input points need to be partitioned differently across *global maps* so as to avoid the algorithm move towards local optima. Also, the convergence condition includes detection of oscillations along with the Euclidean metric.

We use the *K-Means* implementation in the normal MapReduce framework from the Apache Mahout project [10]. Sampled US Census data of 1990 from the UCI Machine Learning repository [17] was used as the clustering data for

the comparison between the normal and relaxed approaches. The sample size is around 200K points each with 68 dimensions. For both relaxed and normal *K-Means*, initial centroids were chosen randomly for the sake of generality. Algorithms such as canopy clustering can be used to identify initial centroids for faster execution and better quality of final clusters.

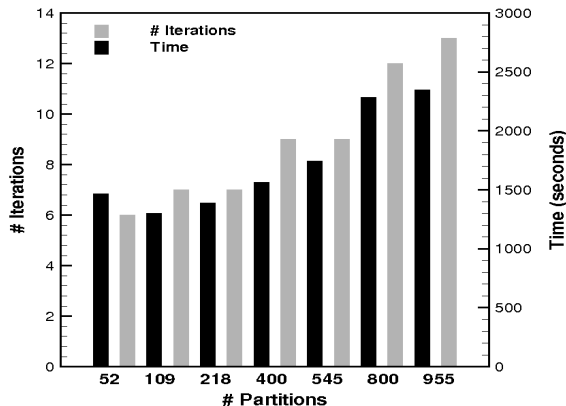


Figure 9. Iterations and Time-to-Converge for different Partitions

Figure 9 shows the number of iterations and time-to-converge with varying number of partitions (disjoint subsets of points). A *global map* applies local MapReduce clustering on its input partition. An increase in the number of partitions decreases the size of points subset. The normal *K-Means* clustering takes 18 iterations for this dataset to converge for a threshold of 0.01.

From the graph, one can observe that (1) relaxed *K-Means* is significantly faster for the partitions considered, and (2) as in Pagerank, there exists an optimal number of partitions, namely at 109 partitions.

6. Discussion

Generality Our relaxed synchronization mechanisms can be generalized to broad classes of applications. The *PageRank* application, which relies on an asynchronous mat-vec, is representative of eigenvalue/linear system solvers (computing eigenvectors using the power method of repeated multiplications by a unitary matrix). A wide range of similar applications requiring the spectra of graphs, global alignments, random walks, can be computed using this algorithmic template. For all of these applications, our methods and results are directly applicable. Algorithms such as shortest paths over sparse graphs and graph alignment can be directly cast into our framework. Beyond graphs, the asynchronous matvec forms the core of iterative linear system solvers.

Asynchronous *K-Means* clustering immediately validates the usefulness of our approach in various clustering and data-mining applications.

Finally, the goal of this paper is to examine tradeoffs of serial operation count and distributed performance. These tradeoffs manifest themselves in a much wider application class.

Programming Complexity While relaxed synchronization requires slightly more programming effort than traditional MapReduce, we argue that the programming complexity is not substantial. This is also evidenced by the simplicity of the semantics used in the paper to describe it.

Fault-tolerance While our approach relies on existing MapReduce mechanisms for fault-tolerance, in the event of failure(s), our recovery times may be slightly longer as each map task is longer and re-execution would take longer. However, all of our results are reported on a production wide-area cluster of 460 nodes, with real-life transient failures. This leads us to believe that the overhead is not significant.

Scalability In general, it is difficult to estimate the resources available to, and used by a program execution in a Cloud computing environment. We draw our assertions on scalability of our formulation by inferring resource availability from the number of graph partitions. Please note that these inferences are meant to be qualitative in nature, and not based on raw data (since the data is not made available by design).

Typically, clusters run of the order of 10 map tasks per node. Since each map handles one complete partition of the graph, for very large numbers of partitions (eg., 6400), we potentially use the entire 460 nodes in the Google cluster for the map phase. Such high node utilization incurs heavy network delays during copying and merging before the reduce phase, leading to increased synchronization overheads. Several orders of speedup obtained using our semantics for iterative MapReduce in such a large data center environments, demonstrate that our solution is indeed scalable.

7. Related work

Over the past few years, the MapReduce programming model has gained attention primarily because of its simple programming model and the wide range of underlying hardware environments. There have been efforts exploring both the systems aspects as well as the application base for MapReduce.

A number of efforts [9, 16, 1] target optimizations to the MapReduce runtime and scheduling systems. Proposals include dynamic resource allocation to fit job requirements and system capabilities to detect and eliminate bottlenecks within a job. Such improvements combined with our efficient application semantics, would significantly increase the scope and scalability of MapReduce applications. The simplicity of MapReduce programming model has also motivated its use in traditional shared memory systems [15].

A dominant component of a typical Hadoop execution corresponds to the underlying communication and I/O. This happens even though the MapReduce runtime attempts to reduce communication by trying to instantiate a task at the node or the rack where the data is present. Afrati et al.² study this important problem and proposed alternate computational models for sorting applications to reduce communication between hosts in different racks. Our extended semantics deal with the same problem but, from the application’s perspective, irrespective of the underlying hardware resources.

There have been recent efforts aimed at bridging the gap between relational databases and MapReduce [18]. These efforts propose a merge phase after the `reduce` phase to compute joins on outputs of various reductions. Olston et al [12] propose a SQL-style declarative language, Pig Latin, on top of the MapReduce model. The underlying compiler deals with the automatic creation of the procedural `map` and `reduce` functions for data-parallel operations. Such languages further enhance programmability and allow the system to optimize execution. Dryad [8], DryadLinq [20], and Sawzall [13] aim to make MapReduce an implicit low-level programming primitive. A program written in Linq can be parsed to form a DAG, which is then used by the Dryad system to schedule the data parallel portions on a distributed cluster. Most of these efforts are aimed at bringing MapReduce programming primitives into high-level languages and also to overcome the rigidity of MapReduce semantics. They are also helpful in logical separation of data-parallel and task-parallel regions in a general application.

All of the aforementioned efforts try to improve efficiency of MapReduce by adding software layers on top of existing MapReduce semantics or by improving the underlying runtime. In contrast, we aim to extend MapReduce semantics to leverage algorithmic asynchrony in important application classes.

8. Future Work

The myriad trade-offs associated with wide range of overheads on different platforms pose intriguing challenges. We identify some of these challenges as they relate to our proposed solutions:

Generality of semantic extensions We have demonstrated the use of partial synchronization and eager scheduling in the context of a specific application class. While we have argued in favor of their broader applicability, these claims must be quantitatively established. Several task-parallel applications with complex interactions are not naturally suited to traditional MapReduce formulations. Are the proposed set of semantic extensions adequate for such applications? MapReduce tries to strike a balance between programmabil-

ity and performance. It attempts to divest the programmer of burdens of optimizing locality, scheduling, and communication. In doing so, it often compromises performance. These trade-offs must be viewed in the context of the underlying platforms and applications.

Implications for tightly coupled systems MapReduce has been shown to be an effective alternative to Pthreads even in the shared memory systems [15] for specific applications. It is important to note that the shared memory MapReduce has a much larger design space because of higher control over the spawned `map` and `reduce` tasks (thread pools). A number of performance optimizations are possible here – ranging from pipelining iterates of `map` and `reduce` operations to reordering `map` and `reduce` operations in order to aggregate `maps` and enhance computation-communication characteristics. Other optimizations involve speculative execution of `maps`, relying on promises as outputs of `maps` as opposed to the values themselves. This rich space of optimizations bears significant research investigation.

Integration of proposed semantics with high-level programming languages Our proposed semantics and its variants can be integrated with SQL-type declarative languages, namely Pig-Latin [12] and DryadLinq [20]. The resulting system would potentially improve the efficiency of MapReduce significantly, while decreasing the load on application programmers to make decisions over the distribution of data.

Optimal granularity for maps As shown in our work, as well as the results of others, the performance of a MapReduce program is a sensitive function of `map` granularity. An automated technique, based on execution traces and sampling [6] can potentially deliver these performance increments without burdening the programmer with locality enhancing aggregations.

System-level enhancements Often times, when executing iterative MapReduce programs, the output of one iteration is needed in the next iteration. Currently, the output from a reduction is written to the distributed file system (DFS) and must be accessed from the DFS by the next set of `maps`, which involves significant overhead. Using online data structures (for example, Bigtable) provides credible alternatives, however, issues of fault tolerance must be resolved.

9. Conclusion

In this paper, we propose extended semantics for MapReduce based on partial synchronizations and eager `map` scheduling. We demonstrate that when combined with locality enhancing techniques and algorithmic asynchrony, these extensions are capable of yielding significant performance improvements. We demonstrate our results in the context of the problem of computing pageranks on a web

²Foto N. Afrati and Jeffrey D. Ullman: A New Computation Model for Rack-based Computing. <http://infolab.stanford.edu/~ullman/pub/mapred.pdf>

graph, and *K-Means* clustering on US census data. Our results strongly motivate the use of partial synchronizations for broad application classes. Finally, these enhancements in performance do not adversely impact the programmability and fault-tolerance features of the underlying MapReduce framework.

References

- [1] *Improving MapReduce Performance in Heterogeneous Environments*, San Diego, CA, 12/2008 2008. USENIX Association.
- [2] R.E. Bryant. Data-intensive supercomputing: The case for disc. technical report cmu-cs-07-128, carnegie melon university. 2007.
- [3] Cheng-Tao Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multi-core. *Advances in Neural Information Processing Systems 19*, pages 281–288.
- [4] Price D. J. de S. A general theory of bibliometric and other cumulative advantage processes. *Journal of the American Society for Information Science*, Vol 27, 292-306, 1976.
- [5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Proc. of OSDI*, 2004.
- [6] Richard O. Duda, Peter E. Hart, and David G. Stork. Chapter 8. pattern classification. 2nd edition. *A Wiley-Interscience Publication*, 2001.
- [7] The igraph library. <http://igraph.sourceforge.net/>.
- [8] 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.
- [9] Karthik Kambatla, Abhinav Pathak, and Himabindu Pucha. Towards optimizing hadoop provisioning for the cloud. *1st Workshop on Hot Topics in Cloud Computing, HotCloud*, 2009.
- [10] Apache Mahout. <http://lucene.apache.org/mahout>.
- [11] METIS. <http://glaros.dtc.umn.edu/gkhome/views/metis>.
- [12] 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.
- [13] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure*, 13(4):227-298, 2003.
- [14] Stanford WebBase Project. <http://diglib.stanford.edu:8091/doc2/WebBase/>.
- [15] 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.
- [16] Thomas Sandholm and Kevin Lai. Mapreduce optimization using dynamic regulated prioritization. *SIGMETRICS/Performance '09: Proceedings of the 2009 Joint International Conference on Measurement & Modeling of Computer Systems*, 2009.
- [17] 1990. UCI Machine Learning Repository US Census Data. <http://kdd.ics.uci.edu/databases/census1990/USCensus1990.html>.
- [18] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. P. Jr. Map-reduce-merge: simplified relational data processing on large clusters. *SIGMOD '07: Proceedings of the ACM SIGMOD international conference on Management of data*, 2007.
- [19] Elad Yom-tov and Noam Slonim. Parallel pairwise clustering. *SDM'09, Proceedings of SIAM Data Mining conference*, 2009.
- [20] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, far Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadling: A system for general-purpose distributed data-parallel computing using a high-level language. *Symposium on Operating System Design and Implementation (OSDI)*, San Diego, CA, 2008.