Design Patterns for Efficient Graph Algorithms in MapReduce

Jimmy Lin and Michael Schatz
University of Maryland, College Park
{jimmylin,mschatz}@umd.edu

ABSTRACT

Graphs are analyzed in many important contexts, including ranking search results based on the hyperlink structure of the world wide web, module detection of protein-protein interaction networks, and privacy analysis of social networks. Many graphs of interest are difficult to analyze because of their large size, often spanning millions of vertices and billions of edges. As such, researchers have increasingly turned to distributed solutions. In particular, MapReduce has emerged as an enabling technology for large-scale graph processing. However, existing best practices for MapReduce graph algorithms have significant shortcomings that limit performance, especially with respect to partitioning, serializing, and distributing the graph. In this paper, we present three design patterns that address these issues and can be used to accelerate a large class of graph algorithms based on message passing, exemplified by PageRank. Experiments show that the application of our design patterns reduces the running time of PageRank on a web graph with 1.4 billion edges by 69%.

1. INTRODUCTION

Large graphs are ubiquitous in today's information-based society. Two examples include the hyperlink structure of the web spanning many billion of pages (commonly known as the web graph) and social networks that connect hundreds of millions of individuals. With perhaps the exception of expensive large shared-memory systems, graph algorithms at scale are beyond the capabilities of individual machines, thus necessitating a distributed approach involving many machines in a cluster.

Distributed computations are inherently difficult to organize, manage, and reason about. With traditional programming models such as MPI, the developer must explicitly handle a range of system-level details, ranging from synchronization to data distribution to fault tolerance. Recently, MapReduce [4] has emerged as an attractive alternative: its functional abstraction provides an easy-to-understand model for designing scalable, distributed algorithms. The open-source Hadoop[1] implementation of MapReduce has provided researchers a powerful tool for tackling large-data problems in areas of machine learning [3, 16, 20], text processing [1, 5, 11], and bioinformatics [10, 18], just to name a few.

MapReduce provides an enabling technology for large-scale graph processing. However, there appears to be a paucity of knowledge on designing scalable graph algorithms. Lin and Dyer's [12] recent book begins to fill this void, and there have been a few relevant papers as well (e.g., [7, 8]). However, for the most part, information on MapReduce graph algorithms is scattered throughout informal sources on the web, including the slides and video recordings of MapReduce courses sponsored by Google.

In this paper, we recapitulate current best practices in designing large-scale graph algorithms in MapReduce and identify significant inefficiencies in those designs. We propose a set of enhanced design patterns applicable to a large class of graph algorithms that address many of those deficiencies. Using PageRank as an illustrative example, we show that the application of our design patterns can substantially reduce per-iteration running time (in our experiments, by up to 69%).

The remainder of the paper is organized as follows: in Section 2, we provide an overview of the MapReduce programming model. Section 3 discusses the class of graph algorithms that is the focus of this paper, exemplified by PageRank. Section 4 describes standard best practices for large-scale graph processing using MapReduce. Section 5 presents our enhanced design patterns for graph algorithms in MapReduce, and Section 6 evaluates their performance on a large web graph with 1.4 billion links. Finally, in Section 7 we summarize our findings and describe future directions for improvements.

2. MAPREDUCE

MapReduce builds on the observation that many information processing tasks have the same basic computational design: a computation is applied over a large number of records (e.g., web pages, vertices in a graph) to generate partial results, which are then aggregated in some fashion. Taking inspiration from higher-order functions in functional programming, MapReduce provides an abstraction for programmer-defined "mappers" (specifying per-record computations) and "reducers" (specifying result aggregation), that both operate in parallel on key-value pairs as the processing primitives.

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

MLG '10 Washington, DC USA

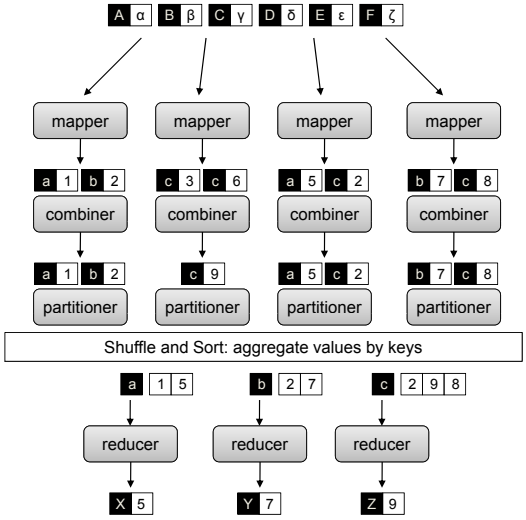

Figure 1: Illustration of MapReduce: mappers are applied to input records, which generate intermediate results that are aggregated by reducers. Local aggregation is accomplished by combiners, and partitioners determine to which reducer intermediate data are shuffled.

The mapper is applied to every input key-value pair to generate an arbitrary number of intermediate key-value pairs. The reducer is then applied to all values associated with the same intermediate key to generate an arbitrary number of final key-value pairs as output. This two-stage processing structure is illustrated in Figure 1.

Under the MapReduce programming model, a developer needs only to provide implementations of the mapper and reducer. On top of a distributed file system [6], the execution framework (i.e., "runtime") transparently handles all other aspects of execution on clusters ranging from a few to a few thousand cores. It is responsible, among other things, for scheduling (moving code to data), handling faults, and the large distributed sorting and shuffling problem between the map and reduce phases whereby intermediate key-value pairs must be grouped by key.

As an optimization, MapReduce supports the use of "combiners", which are similar to reducers except that they operate directly on the output of mappers; one can think of them as "mini-reducers". Combiners operate in isolation on each node in the cluster and cannot use partial results from other nodes. Since the output of mappers (i.e., intermediate key-value pairs) must eventually be shuffled to the appropriate reducer over the network, combiners allow a programmer to aggregate partial results, thus reducing network traffic. In cases where an operation is both associative and commutative, reducers can directly serve as combiners, although in general they are not interchangeable.

The final component of MapReduce is the "partitioner", which is responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers. The default partitioner computes the hash value of the key modulo the number of reducers. Although with any reasonable hash function the partitioner will divide up the intermediate key space roughly evenly, this does not guarantee good load balance because the distribution of values associated with the same key may be highly skewed, nor does it provide any locality between related keys.

3. GRAPH ALGORITHMS

This paper assumes a standard definition of a directed graph $G = (V, E)$ consisting of vertices V and directed edges E, with $N^+(v_i) = \{v_j | (v_i, v_j) \in E\}$ and $N^-(v_i) = \{v_j | (v_j, v_i) \in E\}$ consisting of the set of all successors and predecessors of vertex v_i. Undirected graphs are also implicitly supported by replacing each undirected edge with two reciprocal directed edges. Both vertices and edges may be annotated with additional metadata: as a simple example, in a social network where vertices represent individuals, there might be demographic information (e.g., age, gender, location) attached to the vertices and type information attached to the edges (e.g., indicating type of relationship such as "friend" or "spouse").

We focus on a large class of iterative graph algorithms on sparse, directed graphs, where, at each iteration:

1. computations occur at every vertex as a function of the vertex's internal state and its local graph structure; and

2. partial results in the form of arbitrary messages are "passed" via directed edges to each vertex's neighbors; and, finally

3. computations occur at every vertex based on incoming partial results, potentially altering the vertex's internal state.

Typically, such algorithms iterate some number of times, using graph state from the previous iteration as input to the next iteration, until some stopping criterion is met.

A prototypical example of the above class of graph algorithms is PageRank [2, 15], a well-known algorithm for computing the importance of vertices in a graph based on its topology. For each vertex v_i in the graph, PageRank computes the value $\text{PR}(v_i)$ representing the likelihood that a random walk of the graph will arrive at vertex v_i. The likelihood value of a node is primarily derived from the topology of the graph, but the computation also includes a damping factor d, which allows for periodic random jumps to any other node in the graph. PageRank can be computed algebraically for small graphs, but is generally computed iteratively over multiple timesteps t using the power method:

$$\text{PR}(v_i; t) = \begin{cases} 1/|V| & \text{if } t = 0 \\ \frac{1-d}{|V|} + d \sum_{v_j \in N^-(v_i)} \frac{\text{PR}(v_j; t-1)}{|N^+(v_j)|} & \text{if } t > 0 \end{cases} \quad (1)$$

The algorithm iterates until either a user defined maximum number of iterations is reached, or the the values sufficiently converge. One common convergence criterion is:

$$\sum |\text{PR}(v_i; t) - \text{PR}(v_i; t-1)| < \epsilon \quad (2)$$

PageRank was originally developed to rank the importance of web pages based on the hyperlink structure of the web, but can be applied to rank vertices by their topology within any graph. It also forms the basis for many significant graph analysis algorithms [9].

In this paper, we primarily focus on PageRank as an exemplar of the class of graph algorithms we are interested in. In particular, the power method as formulated in (1) requires only that the local topology and the uniform damping factor are considered at each step, making it especially well suited to parallel computing. However, it is important to recognize that our techniques are equally applicable to a large number of algorithms that take the form discussed above: specific examples include parallel breadth-first search, label propagation, other topology-based vertex ranking algorithms such as HITS [9], a number of graph-based approaches for DNA sequence assembly [17, 21], and the analysis of protein-protein interaction networks [13, 14].

4. BASIC IMPLEMENTATION

The original MapReduce paper [4] described several data-intensive applications for the programming model, including word count, distributed grep, and inverted index construction, but unfortunately did not discuss graph algorithms. To our knowledge, the first reasonably detailed explanation of MapReduce graph algorithms can be traced to lecture slides and video recordings of courses sponsored by Google in 2007.[2] The materials described implementations of parallel breath-first search and PageRank: these have become the *de facto* best practices for MapReduce graph processing. In this section, we provide an overview of those methods, which we will refer to as the basic implementation.

4.1 Message Passing

Computations in MapReduce operate on input key-value pairs; both keys and values can be primitive types (integers, strings, etc.) or arbitrarily complex records (e.g., tuples with nested structure). For graph processing, it is most natural to adopt an adjacency list representation: graphs are serialized into key-value pairs using the identifier of the vertex as the key, and the record comprising the vertex's structure as the value. Typically, the value would include the adjacency list $N^+(v_i)$ (possibly with metadata attached to the edges), metadata attached to the vertex, as well as the vertex's internal state (e.g., its current PageRank value). By virtue of the underlying distributed file system, the key-value pairs comprising the graph will be divided into blocks and spread across the local disks of nodes in the cluster. Vertices are assigned arbitrarily to different blocks using the default HashPartitioner, but the physical layout of the graph structure on disk can be controlled by using a custom partitioner or labeling scheme. We return to this point in Section 5.3.

MapReduce is well suited to the class of graph algorithms discussed in Section 3: the shuffle and sort phase can be exploited to propagate information between vertices using a form of distributed message passing. The canonical approach is to map over the key-value pairs comprising the graph, corresponding to Step (1) where computations occur at every vertex using the local graph structure, vertex metadata, and additional vertex state contained in the serialized graph vertices. The results of the computation are arbitrary messages to be passed to each vertex's neighbors. This is accomplished by having mappers emit intermediate key-value pairs where the key is the destination vertex id and the value is the message. This corresponds to Step (2), using the shuffle and sort phase of MapReduce to perform the routing of the messages. In the reducer, all messages that have the same key (i.e., same destination vertex id) arrive together, and another computation is performed, which corresponds to Step (3).

[2] `http://code.google.com/edu/parallel/`

```
1: class Mapper
2:     method Map(id n, vertex N)
3:         p ← N.PageRank/|N.AdjacencyList|
4:         Emit(id n, vertex N)
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(id m, value p)
1: class Reducer
2:     method Reduce(id m, [p1, p2, ...])
3:         M ← ∅
4:         for all p ∈ [p1, p2, ...] do
5:             if IsVertex(p) then
6:                 M ← p
7:             else
8:                 s ← s + p
9:         M.PageRank ← s
10:        Emit(id m, vertex M)
```

Figure 2: Pseudo-code for simplified PageRank in MapReduce. In the map phase we evenly divide up each vertex's PageRank mass and pass each piece along outgoing edges to neighbors. In the reduce phase PageRank contributions are summed up at each destination vertex. Each MapReduce job corresponds to one iteration of the algorithm.

There is one critical detail necessary for the above approach to work: the mapper must also emit the vertex structure (i.e., the input value) with the vertex id as the key. This passes the vertex structure to the reduce phase, where it is reunited with messages destined for that vertex—so that the reducer can update the vertex's internal state and write out the revised graph to disk. Without this step, there would be no way to perform multiple iterations, since we would have lost the graph structure. Thus, there are two distinct data flows in the basic implementation of graph algorithms in MapReduce: one corresponding to the flow of messages from source to destination vertices along graph edges, and the other corresponding to the shuffling of the graph structure itself.

As a concrete example, pseudo-code for a MapReduce implementation of PageRank is provided in Figure 2. This is a simplified implementation that does not handle the damping factor and dangling nodes (i.e., nodes without neighbors), but suffices to illustrate the key points. For interested readers, Lin and Dyer [12] provide the full implementation. In line (4) of the mapper we pass along the graph structure. In lines (5) and (6) of the mapper we distribute an equal share of the vertex's current PageRank value to its neighbors; messages are floating point values, representing PageRank mass contributions.

The reducer receives a number of values associated with the same key. It must differentiate the messages (PageRank mass contributions) from the vertex structure, which is accomplished in line (5) of the reducer pseudo-code.[3] PageRank contributions from incoming edges to a vertex are

[3] In practice, we create a complex value that contains an indicator variable, specifying its status either as a message or the vertex structure.

```
1: class Combiner
2:     method Combine(id m, [p1, p2, ...])
3:         M ← ∅
4:         for all p ∈ [p1, p2, ...] do
5:             if IsVertex(p) then
6:                 Emit(id m, vertex p)
7:             else
8:                 s ← s + p
9:         Emit(nid m, value p)
```

Figure 3: Combiner pseudo-code for PageRank in MapReduce. The combiner aggregates partial PageRank contributions by destination vertex and passes the graph structure along.

summed, and in line (9) of the reducer pseudo-code, the vertex's PageRank is updated. Finally, the updated graph is serialized and written to the distributed file system. This completes one iteration of PageRank. The output is then ready to serve as input to another MapReduce job representing the next iteration. Typically, a driver program examines results between iterations to check for convergence.

4.2 Local Aggregation

Although it is not always the case, mapper and reducer computations for the class of graph algorithms we are interested in are often very simple. In PageRank, for example, the mappers perform a simple division to "divy up" the PageRank mass, and the reducers sum incoming PageRank contributions. Therefore, the algorithm running time is dominated by shuffling large amounts of data across the network between the map and reduce stages of processing: both messages passed along graph edges and the graph structure itself. Because of this, any reductions in the amount of data shuffled across the network increases the speed of a MapReduce algorithm.

Combiners in MapReduce are responsible for performing local aggregation (i.e., a partial reduce on map output), which reduces the amount of data that must be shuffled across the network. Clearly, they are only effective if there are multiple key-value pairs with the same key computed on the same machine that can be aggregated. In practice, combiners often yield dramatic reductions in algorithm running time due to decreased network traffic. The combiner for PageRank is shown in Figure 3. If it encounters messages destined for the same vertex, it sums up those partial PageRank values and emits the aggregate, while the graph structure is simply "passed along" to the reducer. For PageRank, combiners are especially effective for reducing the number of messages passed to vertices with high in-degrees. This has the additional effect of reducing the skew in the running time of reducers. PageRank is typically run on graphs whose vertex in-degrees follow power law distributions (e.g., the web graph): since reducer computations are proportional to the vertex's in-degree (i.e., number of incoming messages), some vertices take much longer to process than others. Combiners significantly cut down on the number of messages destined for vertices with high in-degrees.

5. ALGORITHM OPTIMIZATIONS

The previous section recapitulates existing best practices for designing large-scale graph algorithms in MapReduce.

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(id n, vertex N)
5:         p ← N.PageRank/|N.AdjacencyList|
6:         Emit(id n, vertex N)
7:         for all id m ∈ N.AdjacencyList do
8:             H{m} ← H{m} + p
9:     method Close
10:        for all id n ∈ H do
11:            Emit(id n, value H{n})
```

Figure 4: Mapper pseudo-code for PageRank in MapReduce that implements the in-mapper combining design pattern.

Building on this, we present three enhanced design patterns that address significant inefficiencies in the basic implementation: (1) costs associated with materializing intermediate key-value pairs when using combiners, (2) costs associated with shuffling the graph structure from the mappers to the reducers, and (3) costs associated with topology-oblivious hash partitioning of vertices.

5.1 In-Mapper Combining

Although combiners provide a mechanism for local aggregation in MapReduce, there are two major disadvantages with using them. First, combiner semantics is underspecified in MapReduce. For example, Hadoop makes no guarantees on how many times the combiner is applied, or that it is even applied at all. The combiner is provided as a semantics-preserving optimization to the execution framework, which has the *option* of using, perhaps multiple times, or not at all. Such indeterminism may be unacceptable.

Second, combiners reduce the amount of intermediate data shuffled across the network, but don't actually reduce the number of key-value pairs that are emitted by the mappers in the first place. With Hadoop combiners, intermediate key-value pairs are materialized in an in-memory buffer and then "spilled" to local disk. Only in subsequent merge passes of on-disk key-value pairs are combiners executed. This process involves unnecessary object creation and destruction, and furthermore, object serialization and deserialization.

To address these downsides, Lin and Dyer proposed the "in-mapper combining" design pattern [12], which exploits the fact that mappers can preserve state across the processing of multiple input key-value pairs and defer emission of intermediate data until all input records have been processed. Understanding this pattern requires a few additional details on the life cycle of mappers in Hadoop. A mapper object is created (by the execution framework) to process a block of input, and the object's Map method is repeatedly called to process input key-value pairs. Hadoop also provides two API hooks, which we refer to as Initialize and Close, that allow user-specified code to execute before any key-value pairs are processed and after all key-value pairs are processed, respectively.

Figure 4 illustrates the in-mapper combining pattern applied to the PageRank algorithm. Prior to processing input, the mapper initializes an associative array (i.e., map) for accumulating partial PageRank scores, with the destination vertex id as the key. When mapping over graph vertices, this associative array is updated with partial PageRank contri-

butions. Intermediate key-value pairs are not emitted until all inputs have been processed. This design pattern is so called because, in effect, we are moving the functionality of the combiner inside the mapper itself. We eliminate multiple messages with the same destination vertex that would have been otherwise emitted by the mapper, and instead only emit a single message that contains an aggregated value.

5.2 Schimmy

In the basic graph algorithm implementation described in Section 4, there are two separate dataflows from mappers to reducers: (1) messages passed from source to destination vertices and (2) the graph structure itself. By emitting each vertex's structure in the mapper, the appropriate reducer receives messages destined for that vertex along with its structure. This allows the reducer to perform a computation and to update the graph structure, which is written to disk for the next iteration.

Shuffling the graph structure between the map and reduce phases is highly inefficient, especially since the algorithms we are interested in are iterative and require multiple MapReduce jobs. Because the graph structure includes metadata and other state information, it is frequently much larger than the messages that are passed along graph edges. As we previously discussed, network traffic dominates the execution time, so reductions in the amount of intermediate data are highly desirable. Furthermore, in many algorithms the topology of the graph and associated metadata do not change (only each vertex's state does), making repeated shuffling of the graph structure even more wasteful. To address this significant shortcoming of the basic implementation, we introduce a novel design pattern called "schimmy" that addresses this inefficiency. As far as we know, this is the first elucidation of this general approach to graph processing for MapReduce that we are aware of.

The intuition behind the schimmy design pattern is the parallel merge join, which is a well known join technique in the database community [19]. Let's say we wish to join two relations, S and T, and the tuples in both relations were sorted by the join key. If this were the case, we can perform a join by scanning through both relations simultaneously. This process can be parallelized by partitioning and sorting *both* relations in the same way. For example, suppose S and T were both divided into ten files, partitioned in the same manner by the join key. Further suppose that in each file, the tuples were sorted by the join key. In this case, we simply need to merge join the first file of S with the first file of T, the second file with S with the second file of T, etc.; these merge joins could happen in parallel.

This method can be applied in graph processing as follows: suppose the input key-value pairs representing the graph structure were partitioned into n files (i.e., parts), such that $G = G_1 \cup G_2 \cup \ldots \cup G_n$, and within each file, vertices are sorted by vertex id. Now let us use the same partition function as the partitioner in our MapReduce graph algorithm, and set the number of reducers equal to the number of input files (i.e., parts). This guarantees that the intermediate keys (vertex ids) processed by reducer R_1 are exactly the same as the vertex ids in G_1; the same for R_2 and G_2, R_3 and G_3, and so on up to R_n and G_n. Furthermore, since the MapReduce execution framework guarantees that intermediate keys are processed in sorted order, the corresponding R_n and G_n parts are sorted in exactly the same manner.

```
1: class Reducer
2:     method Initialize
3:         P.OpenGraphPartition()
4:     method Reduce(id m, [p_1, p_2, ...])
5:         repeat
6:             (id n, vertex N) ← P.Read()
7:             if n ≠ m then
8:                 Emit(id n, vertex N)
9:         until n = m
10:        for all p ∈ values [p_1, p_2, ...] do
11:            s ← s + p
12:        N.PageRank ← s
13:        Emit(id n, vertex N)
```

Figure 5: Reducer pseudo-code for PageRank in MapReduce using the schimmy design pattern to avoid shuffling the graph structure.

The intermediate keys in R_n represent messages passed to each vertex, and the G_n key-value pairs comprise the graph structure. Therefore, a parallel merge join between R and G suffices to unite the result of computations based on messages passed to a vertex and the vertex's structure, thus enabling the algorithm to update the vertex's internal state. We have eliminated the need to shuffle G across the network.

With the MapReduce implementation of PageRank using the schimmy design pattern, we no longer need to emit the graph structure in the map phase of processing. The mapper remains the same as the pseudo-code in Figure 2, with the exception that line (4) is removed (or, alternatively, one could take advantage of in-mapper combining as shown in Figure 4). The corresponding reducer is shown in Figure 5. In the initialization API hook, the reducer opens the file containing the graph partition corresponding to the intermediate keys that are to be processed by the reducer. As the reducer is processing messages passed to each vertex in the Reduce method, it advances the file stream in the graph structure until the corresponding vertex's structure is found. Once the reduce computation is complete (a simple sum), the vertex's state is updated with the revised PageRank value and written back to disk. The partitioner ensures consistent partitioning of the graph structure from iteration to iteration.

When the reducer processes intermediate key-value pairs, those data are read from the local disks of the cluster nodes running the reducers. Files containing the graph structure, on the other hand, reside on the underlying distributed file system (HDFS for Hadoop). Since the MapReduce execution framework arbitrarily assigns reducers to cluster nodes, accessing vertex data structures will almost always involve remote reads (i.e., data streamed off the local disk of another cluster node). This is a potential bottleneck, but experimental results show that these remote reads are not a significant limitation.

5.3 Range Partitioning

As previously mentioned, the distributed filesystem underlying MapReduce splits large graphs into multiple blocks stored on different machines in the cluster. This allows different mappers to execute in parallel on separate, locally-stored portions of the graph. By default, the partitioning of the graph uses an arbitrary hash function of the vertex id,

effectively assigning a particular vertex to a particular block with uniform probability. The hash function is effective for ensuring the different blocks have approximately the same number of vertices, and consequently should require approximately the same amount of time to process. However, the default HashPartitioner has no consideration for the topology of the graph, so the probability that a vertex and its neighbors are in the same block is purely a function of the size of each block, even if the graph has very regular structure or multiple independent connected components.

For graph processing it is highly advantageous for adjacent vertices to be stored in the same block, so that any messages passing between them can be processed in memory or at least without any network traffic. The general problem of partitioning a graph into multiple blocks of roughly equal size such that the intra-block links are maximized and the inter-block links are minimized is computationally quite difficult, but real world graphs often have properties that can be used to effectively approximate this partitioning.

In particular, web pages within a given domain are much more densely hyperlinked than pages across domains. Thus, if pages from the same domain are assigned to the same block, we can achieve a reasonably good partitioning of the web graph. This insight can be utilized in MapReduce by implementing a custom partitioner that hashes the domain name of each URL. However, this would require storing the URL of vertices, which is not desirable since an integer vertex id is far more compact. Nevertheless, we can still leverage this insight if web pages from the same domain are assigned to consecutive vertex ids, and we partition the graph into integer ranges. For example, when splitting a graph with $|V|$ vertices into 100 blocks, block 1 contains vertex ids $[1, |V|/100)$, block 2 contains vertex ids $[|V|/100, 2|V|/100)$, and so forth, using a simple function of the vertex id. In this way, a simple RangePartitioner effectively partitions the graph by domain. With sufficiently large block sizes, we can ensure that only a very small number of domains are split across more than one block.

6. RESULTS

We performed experiments on the ClueWeb09 collection, a best-first web crawl by Carnegie Mellon University in early 2009. The collection contains approximately one billion web pages in ten languages totaling around 25 terabytes. Of those, about 500 million pages are in English, divided into ten roughly equal-sized segments. Our experiments used only the first English segment, which consists of 50.2 million documents, totaling 1.53 TB uncompressed (247 GB compressed). The web graph extracted from this document collection contains 1.4 billion links; the link structure is stored as a 7.0 GB concise binary representation, using integer value vertex ids assigned consecutively by domain. The in-degree distribution of the collection follows a familiar power law, with most pages having a small number of predecessors, but a few highly connected pages with several million (see Figure 6).

We analyzed this web graph using a Hadoop cluster located at the University of Maryland with 10 worker nodes each containing 2 hyperthreaded 3.2 GHz Intel Xeon CPUs, 4GB of RAM, and 367 GB of local disk allocated to HDFS. In total, the cluster contains 20 physical cores (40 virtual cores). Each node was running Hadoop version 0.20.0 on Red Hat Enterprise Linux Server release 5.3 and connected

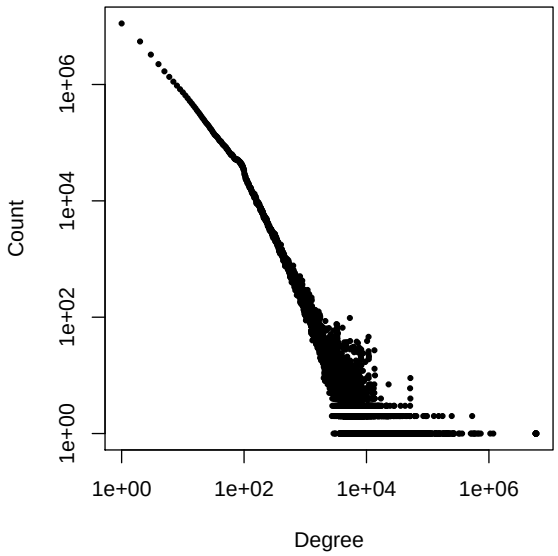

Figure 6: ClueWeb09 in-degree distribution. Most pages have relatively few predecessors, but a significant fraction have more than 100, and 15 pages have more than 1 million.

by gigabit Ethernet to a commodity switch.

We evaluated the effectiveness of the different MapReduce optimizations by computing 5 iterations of PageRank with different sets of optimization enabled. The mean and standard deviation runtimes for computing a single iteration using each configuration are displayed in Figure 7. A naïve implementation of PageRank in MapReduce uses basic (non-schimmy) message passing with the default HashPartitioner, but no combiner. A standard implementation would also include a combiner to partially sum PageRank mass contributions directed to the same vertex from the same mapper. For our web graph, the combiner reduces the number of PageRank mass messages passed by over 50% from 1.4 billion to 674 million and improves performance by nearly 20% over the naïve implementation. The basic implementation, using combiners and the HashPartitioner, characterizes current best practices and is used as the reference point for measuring subsequent performance increases.

Our enhanced design patterns consistently and dramatically improve performance beyond the basic implementation. The individual improvement of each technique is considerable: in-mapper combining improved performance by 16%; the RangePartitioner improved performance by 24%; and schimmy improved performance by 21%. Using all three techniques improved performance by nearly 70% over the existing best practice implementation. All three techniques were consistently effective at improving performance with the notable exception of the unoptimized (no combiner), basic implementation using the RangePartitioner. This particular configuration was an outlier because simple uniform range partitioning caused moderate load imbalance, and a few straggler partitions delayed the overall runtime.

It is worthwhile to note that our enhanced design patterns are applicable to the class of graph algorithms discussed in Section 3, although their effectiveness depends on the connectivity of the graph and the use of message passing within

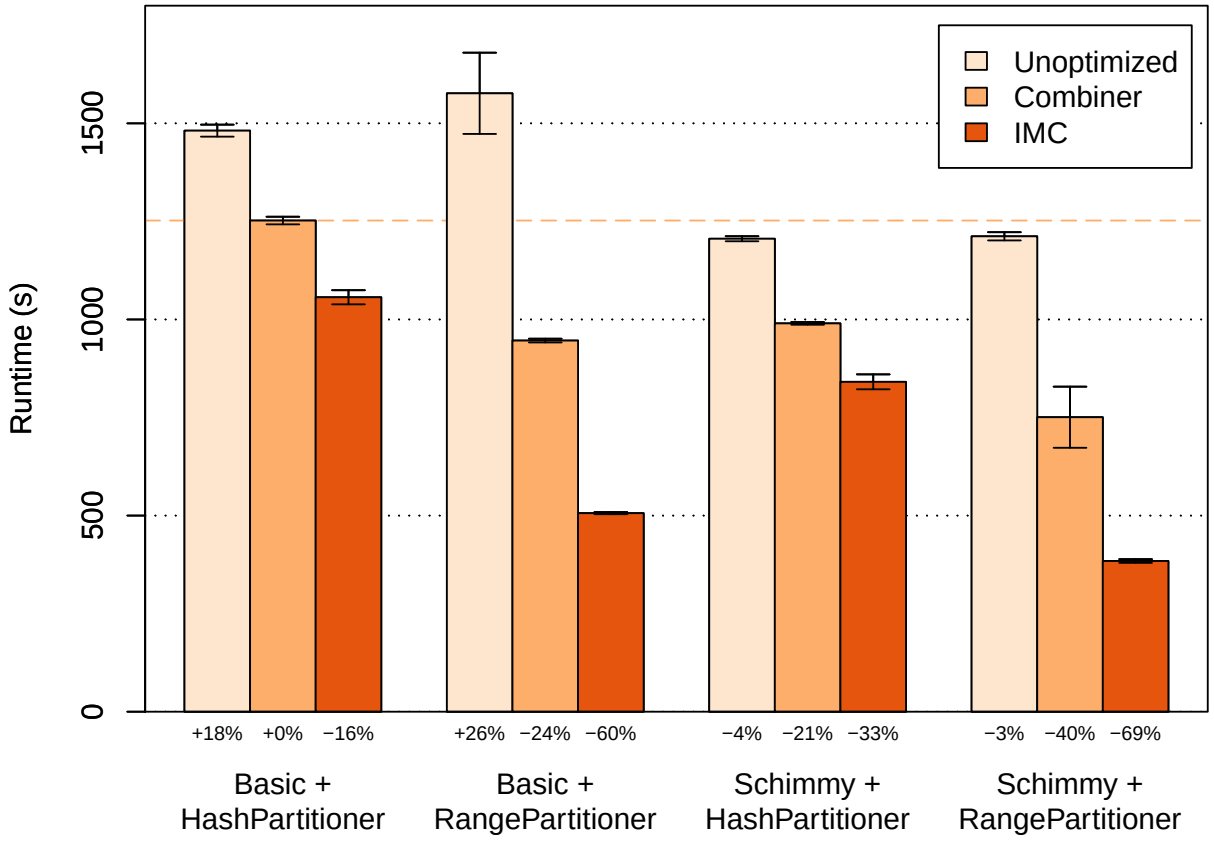

Figure 7: Evaluation of enhanced design patterns on the ClueWeb09 graph. Each bar shows the average and standard deviation runtimes across 5 iterations of PageRank using the indicated set of optimizations. The value below each bar shows the percent change in runtime relative to the baseline configuration using basic message passing (non-schimmy), HashPartitioner, and standard combiner (shown as a dashed orange line).

the algorithm. For example, in-mapper combining is effective in the web graph setting because the power law distribution of in-degrees implies that there are numerous messages destined for highly connected nodes to combine from the same mapper. Without opportunities to aggregate messages, in-mapper combining would offer no performance advantage (although neither would standard combiners). The RangePartitioner was particularly effective on this dataset because the vast majority of links are to other pages in the same domain (typical of web graphs). This enabled combiners to decrease the number of messages to 86 million, representing an almost 8-fold improvement in locality over the HashPartitioner. Such a simple clustering heuristic may not be available in all other problems. Finally, PageRank is actually a poor use case for the schimmy design pattern (in the sense of being able to derive efficiency improvements) because messages are passed along every edge in every iteration. Other algorithms that pass fewer messages relative to the size of the graph would benefit even more from the schimmy design: as the total number messages decreases, efficiency improvements from schimmy will become more pronounced since it is eliminating a greater fraction intermediate data that need to be shuffled (i.e., the graph structure). Nevertheless, our results show that the improvements using schimmy are significant on a real-world web graph.

7. FUTURE WORK AND CONCLUSIONS

This work presents three design patterns broadly applicable to MapReduce graph algorithms. In-mapper combining can be used in any setting where a standard out-of-memory combiner can be used. Schimmy is useful because it obviates the need to reshuffle the graph structure at every iteration. Range partitioning requires only that vertices within a cluster, including approximate clusters derived from some attribute of the vertex, are consecutively labeled.

Future work remains to improve these enhanced design patterns even further. Partitioning could be improved to cluster based on actual graph topology. The challenge is that the graph will, in general, be too large to store entirely in memory, so clustering must be performed in a distributed fashion (perhaps with MapReduce). The schimmy design pattern could be improved by modifying Hadoop's scheduling algorithm so that a particular key range is consistently assigned to the same machine, which would allow the graph structure to be merged from the local disk of the machine running the reducer (as opposed to a remote network read, as in the current design). This is challenging to implement without compromising the robustness of the system to hardware failures. Finally, the general problem of serialization addressed by in-mapper combining could be improved by

storing more of the graph in memory between iterations.

MapReduce has emerged as an enabling technology for analyzing large datasets, because its generalized distributed framework facilitates many types of large-scale analysis with fewer demands on the developer relative to other paradigms. Already it has been used for a diverse set of applications ranging from social network analysis to DNA sequence alignment. However, this generality and flexibility comes at a significant performance cost when analyzing large graphs, because standard best practices do not sufficiently address serializing, partitioning, and distributing the graph across a large cluster. Our enhanced designed patterns of in-mapper combining, schimmy, and range partitioning directly address these limitations, and together improved the performance of PageRank by 69% compared to the best practice baseline. This dramatic runtime improvement has significant implications for graph processing in allowing much larger graphs to be efficiently analyzed.

8. ACKNOWLEDGEMENTS

This work was supported by the NSF under awards IIS-0836560, IIS-0844494, and IIS-0916043, and by NIH grant R01-LM006845. Any opinions, findings, conclusions, or recommendations expressed are the authors' and do not necessarily reflect those of the sponsors. We'd like to thank Nima Asadi for preparing the web graph in this study. The first author is grateful to Esther and Kiri for their loving support.

9. REFERENCES

[1] T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 858–867, Prague, Czech Republic, 2007.

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the Seventh International World Wide Web Conference (WWW 7)*, pages 107–117, Brisbane, Australia, 1998.

[3] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-Reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems 19 (NIPS 2006)*, pages 281–288, Vancouver, British Columbia, Canada, 2006.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 137–150, San Francisco, California, 2004.

[5] C. Dyer, A. Cordova, A. Mont, and J. Lin. Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce. In *Proceedings of the Third Workshop on Statistical Machine Translation at ACL 2008*, pages 199–207, Columbus, Ohio, 2008.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 29–43, Bolton Landing, New York, 2003.

[7] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. HADI: Fast diameter estimation and mining in massive graphs with Hadoop. Technical Report CMU-ML-08-117, School of Computer Science, Carnegie Mellon University, 2008.

[8] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system—implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM 2009)*, pages 229–238, Miami, Floria, 2009.

[9] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46:668–677, 1999.

[10] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for SNPs with cloud computing. *Genome Biology*, 10(R134), 2009.

[11] J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with MapReduce. In *Proceedings of the 32nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2009)*, pages 155–162, Boston, Massachusetts, 2009.

[12] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[13] S. Navlakha and C. Kingsford. The power of protein interaction networks for associating genes with diseases. *Bioinformatics*, 26:1057–1063, 2010.

[14] S. Navlakha, M. C. Schatz, and C. Kingsford. Revealing biological modules via graph summarization. *J Comput Biol*, 16:253–264, 2009.

[15] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[16] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. PLANET: Massively parallel learning of tree ensembles with MapReduce. In *Proceedings of the 35th International Conference on Very Large Data Base (VLDB 2009)*, pages 1426–1437, Lyon, France, 2009.

[17] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proc Natl Acad Sci USA*, 98:9748–9753, 2001.

[18] M. C. Schatz. CloudBurst: Highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[19] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 110–121, Portland, Oregon, 1989.

[20] J. Wolfe, A. Haghighi, and D. Klein. Fully distributed EM for very large datasets. In *Proceedings of the 25th International Conference on Machine Learning (ICML 2008)*, pages 1184–1191, Helsinki, Finland, 2008.

[21] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res*, 18:821–829, 2008.