Ripple: Improved Architecture and Programming Model for Bulk Synchronous Parallel Style of Analytics

Mike Spreitzer
IBM Research
mspreitz@us.ibm.com

Malgorzata Steinder
IBM Research
steinder@us.ibm.com

Ian Whalley
IBM Research
inw@whalley.org

***Abstract*—**
We present Ripple, an architecture and a programming model for a broad set of data analytics. Ripple builds on the ideas of iterated MapReduce and adds two innovations. First it has a richer programming model, including more ideas from the Bulk Synchronous Parallel (BSP) model of computation and others. By doing so, Ripple creates a flexible and higher-level platform that is easier for both application programmers and platform implementors. Second, Ripple is based on a limited interface for key/value storage making it portable among many different key/value store implementations. By building on these two ideas Ripple improves the scope, performance, and openness of the data analytics platform. We evaluate Ripple using three representative, and non-trivial, data analysis scenarios requiring iterative computation. Using these examples, we show how Ripple achieves clear performance advantages over iterated MapReduce.

***Index Terms*—Distributed databases, Distributed programming**

I. Introduction

In recent years, there has been considerable interest in very scalable platforms for data storage, querying, and analytics — such as MapReduce and many NoSQL data stores/bases. Even though they offer less sophisticated data processing capabilities than other technologies (e.g., RDBMS), their relative simplicity and scalability make them attractive in many applications. The combination of a write-once-read-many large-file filesystem with MapReduce computation, as in Google GFS [9] plus MapReduce [7] and in the HDFS plus MapReduce of Apache Hadoop [1], has proven very popular.

The programming model of MapReduce was inspired by Valiant's work on Bulk Synchronous Parallel (BSP) computing [17] — which was intended to be an intermediate abstraction between parallel hardware and software, to facilitate independent development of each. While BSP is explicitly *not* a programming model, many programming models are inspired by BSP.

A BSP computation — called here a *job* — is spread out in both space and time. Temporally, the computation is divided into a sequence of *steps* (this a simplification of BSP, for brevity). Spatially, a job is divided into a number of *components*. Each component has private local state, and components can send messages to each other (and themselves) across steps; each message is received in the step following the one in which it was sent. This is illustrated in Figure 1. A job execution is an alternation between synchrony and parallelism. During one step the components run independently, each doing only local work. Between steps there is a global synchronization barrier, across which all the messages flow.

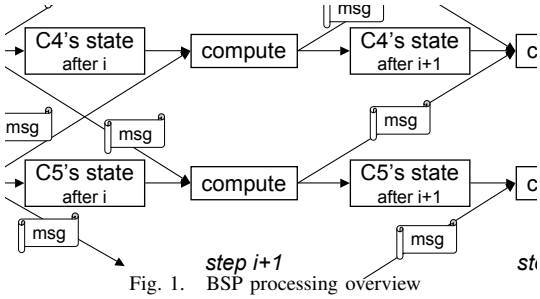

Fig. 1. BSP processing overview

MapReduce uses a simplified version of BSP: a MapReduce job has exactly two steps — map and reduce — and components carry no private local state between the two steps — all the information is carried by the messages.

Many important problems (e.g., PageRank™ [6]) are currently solved by iterating the same MapReduce couplet to successively refine the same dataset. Iteratively invoking MapReduce costs two synchronizations per iteration: one between map and reduce, the other between iterations. Such an iteration typically writes the whole dataset to the distributed filesystem at the end of every iteration, and reads it all again at the start of the next.

Those costs can be cut by using more of the BSP ideas. Typically a MapReduce iteration needs only a straight-line connection from reduce to the following map — which is efficiently realized if those two are fused into the one component compute function of BSP. Iterating MapReduce gives the platform two looks at everything per iteration: once while shuffling messages from map to reduce, once while storing data between reduce to the next map. Existing MapReduce platforms use just one of those two as the failure recovery point; eliminating that one does not preclude failure recovery.

We also advocate two additional related programming model enhancements that are powerful individually and in combination. One, also seen in Pregel [13] for example, is that it is not necessary to invoke every component in every

1063-6927/13 $26.00 © 2013 IEEE
DOI 10.1109/ICDCS.2013.67

step. This can save a tremendous amount of work for analytics that do not need it. More broadly, in opens up the scope of the platform beyond computations that are based on complete scans of large files/tables. Another interesting special case is analytics that do not need the synchronization between steps. This is recognized in a fairly general way in the original BSP work. In Ripple there is a simple all-or-nothing switch: a job either has the synchronization barriers between steps or it does not. Even this simple switch is very useful; we later show its use in dense matrix multiplication. Together these two enhancements really open up the scope: rather than only supporting big slow analytics based on complete scans and lots of synchronization, Ripple can support quick analytics that touch a tiny fraction of the data without waiting for anything.

Both Google's MapReduce and Hadoop's (1) are dedicated to storing intermediate data (that output by map and input by reduce) on disk and (2) have their design centered on storing map input and reduce output in the distributed filesystem (supporting other possibilities, but not as well). Both use a key-value data model and handle each map-reduce couplet independently. In contrast, Google's Pregel — while also inspired by the BSP model of computation — is dedicated to storage in memory, a graph-based data model, and taking advantage of the opportunities that arise from recognizing the iterative structure of many computations. Each of thse platforms supports distinct ecosystems of higher level platforms.

Building and maintaining different architectures for different kinds of analytics has disadvantages related to the extra development and code maintenance effort, and the management and resource costs of maintaining different deployments. We see considerable value in developing a more general framework, which can be tailored to various special usage scenarios.

We argue for an architecture for distributed data analytics that permits various styles of analytics in the same platform and on the same data. Our architecture stipulates a more generic programming model for distributed data processing and a data model and data storage technology on which such analytics should be performed.

The contribution of this paper is to identify an architecture that combines better abstraction over storage with a better programming model for iterated computations and show that this combination has better performance, scope, and openness than iterating MapReduce.

The rest of this paper is organized as follows. Section II describes the basic programming model in Ripple, which we call key/value extended bulk synchronous parallel (K/V EBSP) programming. Section III describes the interface to the lower layer (storage, compute, communication). Those two sections finish the discussion of the improvements in scope and openness. Section IV outlines our current implementation. Section V shows how some particular analytics benefit from the Ripple architecture in comparison to iterated MapReduce, establishing the improvement in performance. Section VI discusses related work. We conclude in Section VII.

```
public interface Job<Key, State, Message,
                     JobOutputKey, JobOutputValue>
{
  List<String> getStateTableNames();

  Compute<Key, State, Message,
        JobOutputKey, JobOutputValue> getCompute();

  List<String> getAggregators();
  ComputeAggregate getComputeAggregate(String name);

  String getReferenceTable();

  Set<Loader<Key,Message>> getLoaders();
  List<Exporter<Key,State,?>> getWriters();
  Exporter getDirectOutputter();
}
```

Listing 1. Job

```
public interface Compute<Key, State, Message,
                         JobOutputKey, JobOutputValue>
{
  boolean compute(ComputeContext<Key, State, Message,
       JobOutputKey, JobOutputValue> ctx);

  Message combine2msgs(BaseContext ctx, Key key,
             Message m1, Message m2);

  State combine2states(BaseContext ctx, Key key,
             State s1, State s2);
}
```

Listing 2. Compute

II. Ripple distributed processing

The central application programming concept in the K/V EBSP programming model is known as a *job*. As in BSP, a single Ripple job can express an iterated computation and components can have private local state.

In K/V EBSP a component is identified by a key, a message is a pair of key (indicating destination) and value, and a component's state is the values (if any) associated with the component's key in each of a list of key/value tables specified by the job.

Ripple allows component state to be factored into multiple tables; typically some are only read while others are also updated. Recognizing this reduces I/O and facilitates application integration (running a new analysis need not involve changing existing data, it could use new tables). Recognizing this also increases the set of concurrent job scenarios that are easy to handle. More subtly, Ripple does not require a component to always have any actual entry in any of the job's state tables; state table entries may be created and deleted as a job runs, and a component is said to exist when it has *either* state table entries or input messages. This increases the scope to more kinds of computations.

To specify a job, a Ripple client provides an implementation of the `Job` interface, partially shown in Listing 1[1].

The central attribute of a job is an object that implements `Compute`, shown in Listing 2, which provides the `compute` method for component execution. The `Compute` object is

[1]Listings 1, 2, and 3 Copyright 2013 IBM

```
interface ComputeContext<Key, State, Message,
                         JobOutputKey, JobOutputValue>
{
  int getStepNum();

  Key getKey();

  State readState(int tabIdx);
  void writeState(int tabIdx, State s);
  State readWriteState(int tabIdx);

  void deleteState(int tabIdx);
  void createState(int tabIdx, Key key, State state);

  Iterator<Message> getInputMessages();
  void outputMessage(Key key, Message msg);

  void aggregateValue(String aggregatorName, Object value);
  Object getAggregateValue(String name);

  Object getBroadcastDatum(Object key);

  void directJobOutput(JobOutputKey key, JobOutputValue
       value);
}
```

Listing 3. ComputeContext

mobile code — it will be distributed by Ripple and invoked near its data.

Temporally, the computation is divided into a series of *steps*. During one step, components execute a user-defined functon with essentially the following signature:
compute: (previous state, incoming messages) ! (new state, outgoing messages, continue signal)

That compute function is expressed as the `compute` method, which fetches its inputs from and delivers its outputs to a context parameter of type `ComputeContext`, shown in Listing 3. When invoked during a step, a component may (among other things): *a*) read/write/delete its own local state; *b*) request creation of a new component's state (by supplying an identifier and initial local state of the new component); *c*) handle messages sent to it in the previous step; and/or *d*) send messages to arbitrary components that will be delivered in the following step.

During a given step only some of the components are invoked, namely the *enabled* ones. As it returns from an invocation, a component returns a binary *continue signal* that indicates whether the component wishes to be enabled in the following step. A component is actually enabled in a step if and only if either: *a*) it was invoked and returned the positive continue signal in the previous step, or *b*) it was sent a message by some component invoked in the previous step.

Between steps there is a synchronization barrier, and all communication flows only across such barriers. In other words, any component's c's computation in step $i+1$ can only start when all components have completed their computation in step i and all messages sent to c are available at c.

If there are multiple messages destined for a given component in a given step, the platform may combine some of them by one or more invocations (at arbitrary times and places) of a pairwise *message combiner* provided by the job. A job's message combiner is expressed as a method in the `Compute` interface.

The function that merges conflicting new component states is also a method in the `Compute` interface.

As in Pregel, a job can have a set of individually defined *aggregators*. Each aggregator has a name and an aggregation technique. Each `compute` invocation can give input values to aggregators identified by name. The results of the aggregations are available to be read, again by aggregator name, in the following step.

A job can also have immutable *broadcast data*. The client supplies this data and the platform is expected to make that data cheaply available to all component invocations.

A job also has the option to do *direct job output*, which is a distinct set of key-value pairs output by compute invocations and handled in a client-specified way.

Via the implementation of the `Job` interface the client also (a) provides access to aggregator objects and (b) names the table containing broadcast data.

A job's initial condition includes: initial local component states, a set of incoming messages, initial aggregator states, and a designation of which additional components are enabled. For the initial message set (if any) or populating a key/value table, the client implements one of a few supported interfaces (all of which extend a marker interface named `Loader`) to prescribe how the desired key/value pairs are computed from a supported source. A client can implement its own `Loader` or use one provided in the Ripple library. Each of those interfaces includes a method that the client implements to compute the desired key/value pairs from the relevant source. When initializing a job this method also has options *a*) to identify, by key, additional components to enable, and *b*) to supply input to individual aggregators.

The execution consists of step after step until there is nothing left to do — which is when no components are enabled. A job can optionally supply a function called an *aborter* that will cause early termination; invoked between steps it returnes a boolean indicating whether execution should be stopped immediately.

The execution of a job yields these results: final component states (which are available to the client through the K/V store), direct job output (which the client directs), and final aggregator results & the number of steps taken (which the client supplies a callback to consume). For each state table's final contents and for direct job output the client can independently supply an instance of `Exporter` — which specifies what to do with each key-value pair.

A. Special Cases

Some jobs have properties that allow more efficient execution techniques. We have identified nine relevant job properties, and five possible areas of optimization in the execution that are enabled by certain combinations of those properties. The properties are as follows:

no-agg– the job has no individual aggregators.

no-client-sync
: the job has no aborter.

needs-order
: the job requires that collocated `compute` invocations be ordered by key.

no-continue
: the compute method always returns the negative completion signal.

one-msg
: for a given destination key and step, there is at most one message.

rare-state
: the bandwidth of state access is much less than the bandwidth of messaging.

no-ss-order
: the compute method invocations for a given key do not need to be in step order.

incremental
: the messages for a given component can be delivered in any order and grouping, with no regard for steps, provided that messages from a given sender to a given receiver are delivered in the order in which they are sent.

deterministic
: the `compute` function is deterministic

The first two properties can easily be detected by Ripple before it starts actually running the job; the others must be explicitly declared by the job.

We now examine the implications of combinations of these properties on how the job can be executed.

(: **needs-order**)) **no-sort** – In this case the implementation does not need to sort.

one-msg ^ no-continue) no-collect – In this case, as there is never more than one message, Ripple does not collect together multiple messages for delivery to a particular component in a particular step.

no-collect ^ rare-state) run-anywhere – In this case the implementation can freely engage in work-stealing, for example to balance load. As the work done by a given component in a given step requires little access to its associated state, there is little penalty to performing this work at a location distant from the state. As there is at most one message per key and step, there is no need to pin a compute invocation to a rendezvous point for multiple messages.

(**no-collect ^ no-ss-order _ incremental**) **^ no-agg ^ no-client-sync) no-sync** – In these cases there is no need for a synchronization barrier between steps. The essence of the conditions is that there are no computations that inherently involve step boundaries (i.e., individual aggregators or aborters), and the incoming messages for a given key can be divided among compute invocations in any way that preserves ordering per (sender,receiver) pair. An example is matrix multiplication according to the SUMMA pattern (see Section V-B), which involves pipelined multicasts interleaved with local computations.

deterministic – In this situation Ripple optimizes failure recovery to speed up completion using techniques known in the literature.

III. Interfaces to the Lower Layer

Figure 2 outlines the design of Ripple. At the core of our system is a platform for extended BSP style processing of key/value data — K/V EBSP, which is introduced in Section II. Other propgramming models may be easily provided above K/V EBSP, e.g., MapReduce, Iterated MapReduce, or Graph EBSP.

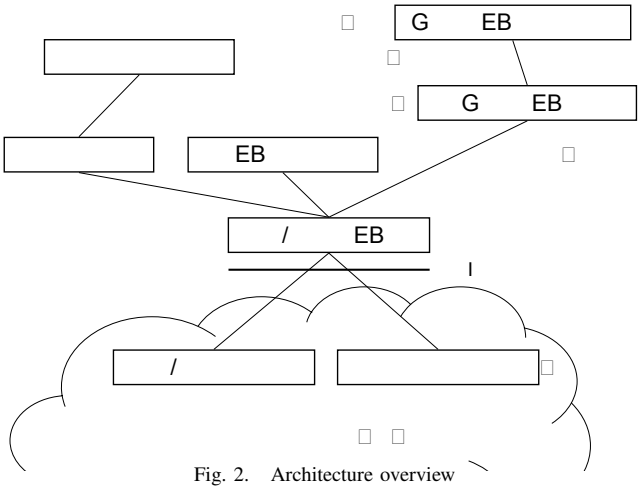

Fig. 2. Architecture overview

Underlying the core K/V EBSP layer is a foundational layer in two parts: the K/V store itself, and message queuing. Ripple does not require particular implementations, but instead works with System Programming Interfaces (SPIs) through which the implementations are provided. We use those to wrap a particular key-value store implementation to make the rest of Ripple implementation store independent. The SPIs (for the messaging and key/value store layers) are kept narrow to maximize the set of possible stores and communication infrastructures that can (with only modest adapter code) satisfy those interfaces. While beyond the scope of this paper, note that this opens the architecture to other uses of the K/V store.

The K/V Store abstraction is not dedicated to storage in memory, nor is it dedicated to storage on disk; it covers both and also other approaches (note the diversity of our implementations mentioned later). While a filesystem can be held in memory, a key-value abstraction is a better interface for in-memory storage and a better abstraction for this architecture since it removes an inherent "impedence mismatch" between the analytics and the store. The finer grained data model is also better for concurrency control.

A. The Key/Value Store

The key/value store holds input and/or output, and it holds the intermediate data not handled by messaging. Additionally, Ripple moves responsibility for placing computation from the analytics layer to the storage layer, making it a fundamental storage+compute layer. This association makes a better division of labor: since the storage layer is certainly in charge of placing data, having it also place computation enables a thinner interface between the two layers and gives the lower layer

more options. While the inclusion of collocated computation is not (yet) common among key/value stores, it is present in some of the more mature ones (such as IBM ® [2] WebSphere ® eXtreme Scale [5] and HBase [3]) and we suspect will be showing up in more over time.

The key/value data are organized into *tables*, each of which may be partitioned into *parts* (identified by successive integers starting at 0), and a given table's parts may be replicated. We currently suppose the way a table is partitioned does not change during a K/V EBSP job.

In addition to the standard tables described above, the key/value store interface exposes the concept of *ubiquitous tables*. By contract, a ubiquitous table is one that is quick to read and of limited size (the contents of a ubiquitous table is expected to fit into each location where it will be used). An expedient implementation of this contract is for a ubiquitous table to have a single part, which is fully replicated to all locations.

The `KVStore` interface provides methods to create, drop, and lookup tables, represented as `Table` objects. We do not suppose every table is partitioned and placed in the same way. However, the `KVStore` provides a method to create a table in a manner that guarantees its consistent partitioning with other tables, which may be used by the same computation.

A `Table` supports get/put/delete by key, where a key and its associated value are general objects. The table client can control the assignment of keys to parts by controlling the hash values of its keys.

A `Table` can enumerate its parts and key-value pairs. When enumerating the parts of a `Table`, the table client provides a call-back object of type `PartConsumer` with methods to process a part and combine results of computation in multiple parts. When enumerating the key/value pairs of a `Table`, the table client provides a call-back object `PairConsumer` with a method to `consume` the key-value pair, which returns a `boolean` indicating whether the enumeration should stop after processing that pair. A `PairConsumer` also has a per-part setup method and a per-part finalize method that returns the result that is combined with its peers.

B. Message Queuing

Having delegated the placement of computation to the lower layer, it makes sense to also ask the lower layer to provide a communication facility that the parts of the distributed computation can use to talk to each other. Ripple thus includes an adjunct lower level interface to a simple message queuing functionality.

The message queuing abstraction is centered on the idea of a *queue set*. The queuing client can create and delete queue sets. A queue set is placed like some given key/value table — there is a queue per part of the table. A queue set can run a piece of mobile client code in each part, and that client code can read (with a timeout) from the local queue of the set. Messages can be put into a given queue of a queue set from anywhere in the system.

[2]IBM and WebSphere are registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml. Other trademarks are property of their respective owners.

IV. Implementation

A. K/V EBSP Implementation

For a job that uses synchronization, the K/V EBSP implementation executes a series of steps. BSP messages are transported in batches called *spills*. Our prototype implementation uses a table, called the *transport table*, to move the spills between parts. Each spill from part S to part D is written to the transport table with a new unique key that is constructed to be located in part D. Between steps the spills are read and deleted from that transport table, and the (key, value list) pairs are constructed in an appropriate local table, where the value list is the concatenation/combination of messages destined to the component identified by the key. This local table is ordered when the job needs sorting, a hash table otherwise. An enumeration of the (key, value list) table then drives the `compute` invocations of the following step. In the no-collect special case the construction of value lists is not needed, and if key ordering is not requested then there is no need to construct additional local tables. The above message handling is analogous to the shuffle in MapReduce.

The implementation of the continue signal transforms a positive one into a special kind of BSP message. Thus, the basic mechanism is driven purely by BSP messages.

Our implementation of individual aggregators starts with partial aggregations done independently in each part as the components are invoked. If the number of individual aggregators is modest, the partial aggregates are returned to the table client for final aggregation and redistribution. For a large number, this is instead done through a couple of auxiliary tables and another round of enumeration.

When synchronization is not needed, the job is instead executed in one dispatch of EBSP implementation code to a queue set, where its instances invoke components and exchange messages until there is no more work to do[3].

The tables and/or queue sets used by the implementation of EBSP concepts are private to that implementation.

While a full discussion of fault tolerance techniques is beyond the scope of this paper, here is an outline of how to efficiently recover from faults for a job that uses synchronization and not direct job output, supposing the key/value store allows (as does WXS) an ACID transaction over all the entries in a shard of co-placed replicated tables. Add a table that maps shard ID to completed step number, and commit transactions in the right order; recover from primary shard failure by deleting writes done by the failed shard(s) and retry.

B. Store, Compute, Communicate

We have three key/value store implementations: an adapter that plugs IBM WebSphere eXtreme Scale (WXS) [5] into

[3]We detect distributed termination essentialy by Huang's algorithm [11].

Ripple, an adapter for HBase [3], and a debugging implementation. From the point of view of this work, WXS offers an elastic in-memory key/value store supporting data partitioning, replication, and the ability to execute mobile code adjacent to the data.

Our current implementation uses a generic implementation of the message queuing interface based on an private extension in the `Table` interface. Each new queue set is implemented by such a new table.

V. Evaluation

We evaluate the Ripple architecture using three practical applications. Comparing a Ripple-based solution with one based on iterated MapReduce would compound the effects of several architectural differences. Instead we tease apart the contributions of those differences. For each application we use a pair of implementations that differs in just one or a few of the architectural issues we have discussed.

- PageRank — we show that fusing reduce with the following map can speed up computation by replacing two rounds of I/O with one and replacing two synchronization steps with one.
- SUMMA-style matrix multiplication — we show how an alternate execution strategy that omits the synchronization can greatly benefit applications that do not need synchronization.
- An on-line graph algorithm (maintaining the distance from a single source) — we show how selective enabling of components can greatly speed up the computation.

A. PageRank

PageRank™ [6] is a method of assigning a rank to every vertex in a directed graph (V, E). It is parameterized by a damping factor d (in the range $(0, 1)$) and defined by a recursive set of equations. The PageRank R_v of a vertex v is defined as

$$R_v = \frac{1 \quad d}{jVj} + d \sum_{u \in V} R_u A'_{u,v}$$

where A' is a modified adjacency matrix.

$$A'_{u,v} = \begin{cases} \frac{1}{W_u} & \text{if } W_u > 0 \text{ ^ } (u,v) \; 2 \; E \\ 0 & \text{if } W_u > 0 \text{ ^ } (u,v) \; \not 2 \; E \\ \frac{1}{|V|} & \text{if } W_u = 0 \end{cases}$$

where $W_u = jfv : (u, v) \; 2 \; Egj$.

These equations can be numerically approximated by repeatedly evaluating them, starting from any initial approximation in which the ranks sum to 1.

We implemented two variants of this technique, both using our K/V EBSP platform: a **direct** variant and a **MapReduce** variant. In each variant, a graph vertex is identified by a Java `int` or, when an independent object is needed, `Integer`.

In the direct variant, we defined a K/V EBSP job with a component per vertex, and a step per iteration of the equations. This variant puts both the ranking state and the graph structure in the BSP messages; the first step begins by reading a table holding the graph structure, and the last step replaces each entry in that table with an enhanced vertex object that holds its rank as well as its structure. The representation of structure is this: each vertex object v includes a Java `int` array holding the ID of each vertex that lies at the far end of an outgoing edge from v. An enhanced vertex object also includes a Java `double` holding the vertex's rank. When passed as a BSP message we use a further enhanced vertex object that includes both the rank last computed and another `double` that is accumulating contributions to the next iteration's rank estimate. This variant has two varieties of BSP messages; some carry a vertex's structure and ranking state forward to the next step, while others carry rank contributions (as `Double` objects) along edges. The regular computation for a vertex v combines all the incoming messages via the job's combiner, completes another iteration of the equation for v's rank, and then sends (a) v's structure and ranking state as a message to itself and (b1) sends $R_v A'_{v,u}$ messages along v's outgoing edges if $W_v > 0$ or (b2) contributes R_v/jVj to a sink rank aggregator if $W_v = 0$.

The MapReduce variant emulates the MapReduce programming model in the Ripple framework. The MapReduce variant also has one BSP component per vertex, but uses two BSP steps per iteration of the PageRank equations — one step acting like a map and the other like a reduce. This variant also puts both structure and ranking state in BSP messages — but only between a map-like step and the following reduce-like step, which implements the shuffle phase of MapReduce. The other half of the time (from reduce to following map) it eschews BSP messages and instead stores both structure and ranking state in a K/V table.

The MapReduce variant is purely inferior to the direct variant, doing strictly more work. In particular, the MapReduce variant has two synchronizations per iteration of the equations, while the direct variant has one. Also, the MapReduce variant does an additional round of I/O (to/from the K/V table holding the structure and ranking state between reduce and following map).

Table I shows the time these two variants take to rank a few graphs. In these cases the direct variant is 15–19% faster than the MapReduce variant, because it has 50% fewer I/O and synchronization rounds. Each result summarizes 11 trials of ranking the same randomly generated graph; the same graph is used for each alternative. Each graph follows a biased power-law distribution for edge attachments. For these tests we used an IBM x3550 M2 with 8 Intel R Xeon R CPUs, hyperthreaded to give the illusion of 16 CPUs, running at 2.93 GHz. On this we ran RedHat R Enterprise Linux R 5.8, and the 64-bit IBM Java SDK version 1.6.0. The `KVStore` implementation was our parallel debugging store with 6 partitions. This store approximates a distributed key-value store, all in

threads: one to handle short request-response table operations (get, put), while the other handles (one at a time) long-running requests (i.e., enumerations). Communication between emulated partitions involves marshalling and un-marshalling, while local operations do not.

Vertices	Edges	Direct Variant avg ± stddev	MapReduce Variant avg ± stddev
132000	4341659	28.5 ± 0.4	32.9 ± 0.7
132000	8683970	44.8 ± 0.5	53.2 ± 0.4
262000	8683970	55.3 ± 0.6	63.5 ± 0.7

TABLE I
ELAPSED TIME (SEC) FOR PAGERANK VARIANTS

B. Matrix Multiply by SUMMA

Next we consider matrix multiplication according to the communication/computation pattern laid out in the original SUMMA paper [18]. That describes a way to multiply matrices stored in a grid of processors, using the MPI programming model and having no global synchronization. SUMMA nicely interleaves communication and computation in a way that does not require buffering much data in any one processor at any one time (as long as there are no spurious delays). Moving it to the (extended or not) BSP model involves interesting challenges, primarily due to the introduction of synchronization. Even if BSPified SUMMA is not the best way to multiply matrices in BSP, it is an interesting example of a pipelined BSP computation that does not need synchronization and can benefit greatly from removing it.

We compute C A B as follows. Each of the three matrices is decomposed into a M N grid of blocks, with all matrices stored in the same MN components in the BSP model. Each block of A is multicast throughout its grid row, and each block of B is multicast throughout its grid column; when corresponding blocks meet at the grid location to which their product contributes, that product is computed and added into the local block of the running total for C. At the coarsest level this can be seen as a (distributed, pipelined, non-synchronized) iteration over batches of columns of A (and, simultaneously, rows of B). Original SUMMA does not use a multicast communication primitive; rather each multicast is pipelined as several point-to-point sends from one grid point to the next, interleaved with the block multiplications and additions. SUMMA prescribes a particular pattern of block sends and arithmetic that causes corresponding blocks to meet up at their destinations without much waiting and without needing any buffering to re-order arrivals.

The per-component state of BSP nicely serves to hold the running total for C; in MapReduce, for comparison, the running totals would have to be passed around in messages or explicitly I/Oed to/from the file system.

The move to BSP is problematic because communication happens only across synchronization barriers. We cannot do all the block arithmetic for a given batch of columns of A in one BSP step because the pipelined communication does not get all the corresponding blocks to meet up at their destinations in the same step. Here we consider one plausible way to introduce the synchronization barriers. We insist that a given component do its block sends and arithmetic in an order consistent with original SUMMA[4]. The question is how much work does a given component do in a given step before it returns to the framework to wait for the synchronization barrier. One part of the answer is that a given component will do no more than one block multiply and add in a given step. Another part is that a given component will send no more than one block in a given direction in a given step (so that blocks do not pile up, violating the SUMMA virtue of limited buffering). The last part is that a component invocation does as much work as is allowed by the other considerations. The timing of the receipt of blocks in BSP is not an available degree of freedom — BSP insists on delivering each block in the step after it was sent.

How balanced is the computation load in this approach? Consider the trivial example in which all blocks have the same size and $M = N = 3$. The nine components are logically arrayed in a three-by-three grid. Table II shows a summary of the number of block multiplications in each step. Seven steps are required, even though a given component does only three block multiplications. In over half of the steps, no more than $1/3$ of the components do a block multiply. If we measure time by the number of block multiplications done in series, introducing the synchronization required by BSP has slowed down this example by a factor of $7/3$.

Step	1	2	3	4	5	6	7
Multiplications	1	3	6	3	6	3	5

TABLE II
BLOCK MULTIPLICATIONS IN EACH STEP

This computation, even after moving to BSP, does not really need the global synchronizations. Each component is able to deal with blocks as they arrive, regardless of when they arrive. Because the components follow the SUMMA pattern, and Ripple preserves message ordering independently for each pair of sender and receiver, even without synchronization the blocks will arrive at their destinations in properly SUMMA-coordinated order. Without the unnecessary waiting for global synchronizations, the computation can finish much sooner.

We implemented the matrix multiplication technique described above, and tested it in the scenario discussed above ($M = N = 3$). For the K/V store we used WebSphere Extreme Scale, which is an in-memory K/V store, with 10 data container processes. We ran it 8 times with synchronization, and the run time average was 90 seconds with an estimated standard deviation of 0.5 seconds. We ran it 8 times without synchronization, and the run time average was 51 seconds with an estimated standard deviation of 0.5 seconds. Although not a factor of 3:7, various overheads can be expected to

[4]...slightly liberalized to allow a component to do either the horizontal or the vertical communication for a given batch first, depending on which is allowed first by all the other considerations.

lessen the improvement. Nevertheless, the result is still a worthwhile improvement clearly demonstrating the benefits of a programming framework that allows synchronization to be controlled by the programmer.

C. Incremental Single-Source Shortest Paths

Consider the following common problem in graph analytics defined on a time-varying undirected graph (V, E). A vertex $\hat{v}$ is distinguished as the source. For each of the other vertices v, we want to annotate it with the length $d(\hat{v}, v)$ of the shortest path between v and $\hat{v}$. Once the problem has been solved on some initial graph, the graph may be changed by a small batch of changes and then we want to update the annotations to hold the new values of d. We implemented two variants of a method for doing this, one, **selective enablement** taking advantage of EBSP, the other, **full-scan**, doing full scans of the graph structure, which is required with MapReduce style computations. The selective variant has a great performance advantage, even though it does extra bookkeeping to support its incrementality.

We suppose the graph may change in these primitive ways: gaining or losing a vertex that has no neighbors, and gaining or losing an edge. If the batch of changes includes no edge deletions then the solution is updated by one wave of breadth-first updates, otherwise it is two waves. In that harder case, the first wave updates to $+\infty$ all the distance annotations that dependended critically on a now-removed edge, while the other wave decreases distance annotations that are higher than are justified by neighbors' values.

The full-scan variant maintains two things at each vertex v: (1) a Java `int` holding the most recently computed value of $d(\hat{v}, v)$, and (2) a `int` array holding the ID of each neighbor vertex. Both waves are done with very similar logic. Each is done with a series of MapReduce-like K/V EBSP jobs. Each of these jobs has two steps. As in the PageRank case, we keep both structure and distance state in one K/V table — in both variants. In the full scan variant, the map job step reads from the K/V table and sends BSP messages — each vertex sends a full state-propagating one to itself, and a distance update (an `Integer`) along each outgoing edge. The full state object carries, in addition to the neighbor array, both the current distance value and the minimum distance value heard from a neighbor. This job has a combiner with an obvious implementation. The reduce step for a vertex v starts by combining all the input messages, which necessarily produces a preliminary full state for v. The new distance value is produced from the minimum neighbor's distance and, in the hard case, the previous distance value (if no remaining neighbor supports the previous distance value, the new value is $+\infty$). The reduce step finishes by writing the new structure plus distance value into the state table. We use an aggregator to count the number of vertices whose distance changed in the step; there is an external driver that invokes a series of MapReduce-like jobs until there are no more changes.

The selective enablement variant maintains more state at each vertex, which makes the incrementality possible. Each vertex object includes two Java `int` arrays of the same length — one holds the ID of each neighbor, and the other holds the distance value most recently received from each neighbor. With this, it is not necessary for a vertex to hear from every neighbor in each iteration. In this variant each distance message carries the ID of the sender as well as its current distance value. The job's combiner does not combine these messages. The compute method for a vertex starts by reading its current structure and state, then applies all the input messages to the array of neighbor distances. Then the new distance value is computed from the array of neighbor distances. If this produced a distance update then it is sent out along all the incident edges. If there was a state change then the vertex state is written back to the state table.

We compared these two variants. We used the same hardware and software as in the PageRank comparison (excepting, of course, the particular graph algorithm and its data structures). The input was randomly generated as follows. First comes the creation of 100,000 unconnected vertices; one of them is chosen uniformly at random as the distinguished source $\hat{v}$. Then about 1.8 million random edges are added. For each such edge, its source and destination are randomly chosen according to a power law distribution. The initial distance values are also computed. Then the following is repeated ten times: a batch of random edge additions and removals is generated (without regard to which already exist, so some of these changes will be no-ops) and applied, then the distance annotations are updated according to the method discussed here. The elapsed time for updating the distance values for each of the ten batches is added up to produce the time for one trial. Below we report on the statistics for 12 trials.

The selective enablement variant took $0.21 \quad 0.03$ seconds to react to ten batches of 1,000 primitive changes. The full scanning variant took $78 \quad 5$ seconds to accomplish the same thing.

VI. Related Work

Ripple's programming model is that of Pregel [13], simplified from graph based data to key/value data, and extended; Pregel is dedicated to storage in memory while Ripple is more general. The functionality of Pregel can be constructed atop Ripple's K/V EBSP.

Other platforms inspired by BSP include BSPlib [10], a library that supports a BSP-based programming model expressed in C or Fortran, and the Apache project Hama [2], which seeks to do a similar thing in Java.

Microsoft's Dryad [12] and CIEL [14] also provide a low-level platform for analytics. Their central abstraction is an explicitly managed DAG of computation and communication. This is arguably a lower level, and thus more general, platform than the BSP-inspired ones. One can use Dryad or CIEL to implement a BSP-inspired platform. Since Ripple's K/V Store + EBSP platform is more abstract than Hadoop's HDFS + MapReduce, we expect it would be better/faster/easier to implement the former than the latter on top of Dryad or CIEL. Such a layered system might be a good idea because of the

popularity of BSP-style programming and, as acknowledged in the CIEL paper for example, few analytic developers want to use the DAG model directly. However, using Dryad or CIEL as a lower level platform precludes delegating computation to a key/value store that will collocate the computation with the data.

There is a crucial difference between (a) MapReduce and (b) BSPlib, Hama, Dryad, and CIEL: while the latter tend (although not exclusively) to focus on relatively heavy-weight processor-like units, MapReduce focuses on relatively lightweight units of application data. This seemingly small difference has proven to be of great practical value [8]. While the other programming models require the analytic application to explicitly multiplex its meaningful units into the model's larger processor-like units, the MapReduce application can supply some relatively small, simple, and essentially functional code — and this is sufficient to prescribe a large distributed MapReduce computation.

Dryad has the ability to do certain run-time optimizations of aggregations. A program fragment that is the implementation of an aggregation first needs to be identified and then it can be replaced by "run-time graph refinement" with an equivalent, but better, Dryad program fragment. In Ripple such rewriting is not needed because the programming model can address the real units of the analytic application and directly includes aggregation operations. Also, Ripple has even more optimization opportunities because the implementation of the aggregation is not constrained to be expressed in the original programming model.

Microsoft's Daytona [4] is also exploring the value of one of the two ideas we advocate — better supporting the iterative structure of many computations — but not the other.

Iterated MapReduce has other disadvantages in addition to those discussed earlier. Hadoop MapReduce is formulated in a way that imposes certain performance limits. Its usual treatment of intermediate data (map output, reduce input) involves an expensive strategy with much sorting and storage on disk. Each job involves launching many heavyweight "tasks". Even the most trivial job is a fairly heavyweight computation; Zaharia reports even minimal jobs taking 25 seconds (section 7.1 of [19]).

While Hadoop's central abstractions (mappers and reducers) are pure key/value, divorced from any mention of HDFS, every job also includes the critical peripheral abstractions of `InputFormat` and `OutputFormat`. It is not possible to run an assembly of jobs without the client stipulating how the data flows between the jobs — the MapReduce platform does not have the opportunity to optimize that, and there is nothing the client can say to get an efficient straight-line connection from reduce to following map in the iterative case. Also, the `InputFormat` abstraction has subtle inclusions of characteristics of HDFS and Hadoop MapReduce's execution technique. An `InputFormat` is responsible for decomposing itself into multiple instances of `InputSplit`, each of which is responsible for naming some machines — in the Hadoop cluster — that hold that split. What if we wish to process data held in some store that does not publicize the location of any particular piece of data — perhaps because it moves from time to time? What if the ideal containers for the computation are not processes launched by TaskTrackers? If you take a hard look, you see that the input and map side of Hadoop MapReduce is about running processes with nearly arbitrary application code (usually) in places that the application chooses. Indeed, this is all that some users ask of Hadoop. Yet this exposure of low-level facilities means there is little flexibility in how the Hadoop APIs are implemented or extended. Ripple limits its exposure/implication of lower level details to those in the limited generic APIs for storage, computation, and communication.

Hadoop does not normally[5] give control over where reduce work is done. As a consequence, applications do not have any control over where job output data are placed. In turn this means that a subsequent job that wishes to join data output by previous jobs can not expect convenient co-location of corresponding inputs. In contrast, systems like Ripple and Piccolo — based on a key/value store that allows the application to request consistent partitioning and placement of certain tables — *can* arrange such co-location.

M3R [16] supports iterated MapReduce with storage in memory, with some compatability with the Hadoop MapReduce API. M3R circumvents some of the problems with iterated MapReduce; full realization of M3R's benefits requires client modifications. M3R cannot accomplish selective enablement or achieve zero synchronizations per iteration.

HaLoop is derived from Hadoop, adding support for iterative structure. HaLoop, and its paper's application examples, are very concerned with the case where data to be joined are not co-located. Piccolo and Ripple solve this more elegantly by using a key/value store that offers co-placement. While you will not always have co-placement, Ripple's approach makes it much easier and more likely. When Ripple would access data remotely, caching it (which is what HaLoop does) is a detail of the fundamental store/compute layer (and good to address there). HaLoop has per-component state, but it has to be immutable and it has to be computed in the first iteration (the platform can not tell if it is a pre-existing table). HaLoop allows a series of map-reduce pairs per iteration, but to no real advantage over more iterations of a single map-reduce pair with loop counter dependent behavior. HaLoop requires the output of each iteration to be saved until the whole computation is done. Following Hadoop, HaLoop: (1) is dedicated to storage in HDFS at iteration boundaries and local disks otherwise, (2) uses a fixed execution strategy (always sorts, can not eliminate the sync between map and reduce, and so on), and (3) does not have selective enablement.

Power and Li described Piccolo [15] as a programming model for parallel in-memory applications. Piccolo's API is like Ripple's KVStore ifc, augmented a little rather than (as Ripple does) leaving substantial enhancement to a higher layer.

[5] An administrator — not an individual application — can install an alternate scheduler to get this control, but this is non-standard and may raise issues of its own.

Piccolo provides one particular implementation of its API, while Ripple uses its KVStore interface to abstract over stores. Piccolo's API has two ways to support checkpointing, and the higher layer has to explicitly program checkpointing using those Piccolo features. Ripple has the advantage that its BSP-inspired programming model is sufficiently structured that the reliability can be handled in the KV EBSP implementation. Piccolo's programming model is lower level. Ripple's higher model (KV EBSP) is helpful to app developers who want to focus on their higher level logic as well as to system builders who, under the covers of a higher level model, have more freedom to use various execution strategies.

Spark [19] is a system that supports in-memory computations on clusters using a functional programming model embedded in Scala and centered on a distributed one-dimensional array concept called *reliable distributed datasets* (RDDs). The Spark programming model is fairly expressive, but does not dispatch general application code next to the individual K/V pairs — instead Spark provides only a modest sized library of operations on whole arrays (with pair-structured elements in some operations). When the values have significant structure and the logic manipulating them is complex, this complexity must be driven through the Spark API rather than encapsulated and dispatched to the data. The Spark work includes an outline of how the Pregel programming model could be implemented in Spark. It feels like a reconstruction of a lower level thing from a higher level thing, and does not cover all the features of Pregel — because Spark makes full cover of Pregel difficult. E.g., the fact that fine-grained manipulation of the values has to go through the Spark API makes it hard to eliminate complete scans of the arrays (hard to do selective enablement). Although the Spark API is generally more high level than KV EBSP programming model, Spark forces iterative applications to explicitly program the iteration, manage memory (materialization), and program reliability. In addition to the obvious disadvantages, this makes it hard to accomplish something analogous to Ripple's ability to not synchronize in the case of an application that does not need it. Spark does not have any public lower level interfaces; there is no architected limited generic interface to lower level storage and compute functionality. Rather than implement Ripple as an application of Spark, it may be more natural to implement Spark as an application of Ripple.

VII. Conclusions and Future Work

We presented Ripple, a middleware for distributed data analytics which permits various styles of analytics in the same platform and on the same data. We have shown some ways in which two architectural ideas, (1) indirecting through a limited generic interface to a fundamental storage+compute layer and (2) an enhanced programming model that recognizes and better supports the iterative structure of many analytics, can improve the scope, performance, and openness of a BSP-style analytics platform.

We are interested in further exploration of three types of issues. One is the issues that concern only the analytics; these amount to further refinement of the existing programming models, perhaps adding others, and improving their integration. Another type of issues is those that arise when managing multiple analytics jobs concurrently. Finally, there are also the issues that arise when EBSP shares a runtime with some other workload (such as OLTP).

References

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

[2] Apache Hama Project. http://hama.apache.org/.

[3] Apache HBase Project. http://hbase.apache.org/.

[4] Daytona. http://research.microsoft.com/en-us/projects/daytona/default.aspx/.

[5] WebSphere eXtreme Scale. http://www.ibm.com/extremescale/.

[6] Brin, S., and Page, L. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems 30*, 1–7 (1998), 107–117. Proceedings of the Seventh International World Wide Web Conference.

[7] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (2004), OSDI '04, USENIX, pp. 137–150.

[8] Foley, M. J. Microsoft drops Dryad; puts its big-data bets on Hadoop. *ZDNet blog "All About Microsoft"* (2011). http://www.zdnet.com/blog/microsoft/microsoft-drops-dryad-puts-its-big-data-bets-on-hadoop/11226.

[9] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43. http://doi.acm.org/10.1145/945445.945450.

[10] Hill, J. BSPlib - The BSP Programming Library. http://www.bsp-worldwide.org/implmnts/oxtool/bsplib.html.

[11] Huang, S.-T. Detecting termination of distributed computations by external agents. In *Distributed Computing Systems, 1989., 9th International Conference on* (jun 1989), pp. 79 –84.

[12] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (New York, NY, USA, 2007), EuroSys '07, ACM, pp. 59–72. http://doi.acm.org/10.1145/1272996.1273005.

[13] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 135–146.

[14] Murray, D. G., et al. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA, USA, 2011), NSDI '11, USENIX, pp. 113–126.

[15] Power, R., and Li, J. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–14. http://dl.acm.org/citation.cfm?id=1924943.1924964.

[16] Shinnar, A., Cunningham, D., Saraswat, V., and Herta, B. M3r: increased performance for in-memory hadoop jobs. *Proc. VLDB Endow. 5*, 12 (Aug. 2012), 1736–1747.

[17] Valiant, L. G. A bridging model for parallel computation. *Commun. ACM 33* (August 1990), 103–111. http://doi.acm.org/10.1145/79173.79181.

[18] van de Geijn, R., and Watts, J. SUMMA: Scalable Universal Matrix Multiplication Algorithm. Tech. Rep. 95-13, Dept. of CS, U. Texas, April 1995. http://www.cs.utexas.edu/ rvdg/abstracts/SUMMA.html.

[19] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Tech. Rep. UCB/EECS-2011-82, EECS Department, University of California, Berkeley, Jul 2011.