Blue*: A Unified Programming Model for Diverse Data-intensive Cloud Computing Paradigms

Maneesh Varshney  Vishwa Goudar

Technical Report #130005
Computer Science Department
University of California, Los Angeles
April, 2013

ABSTRACT
Several computational paradigms exist today for processing large volumes of data on a cluster of resources: batch processing, iterative, interactive, memory-based, data flow oriented, relational, structured, etc. A unified system or framework that supports all paradigms would be greatly beneficial, as it will allow running different algorithms on the same dataset, reduce the system deployment, maintenance and training costs, provide a common platform for research and enable advances to be quickly incorporated.

In this paper, we propose an approach towards such a system by introducing a unified programming model, called Blue. The model supports diverse paradigms of data-intensive computation, under the assumption that a program can be decomposed into a dependency graph of component tasks. In particular, Blue is capable of modeling iterative problems by a novel approach of unfolding cyclic graph into an unbounded acyclic graph. We illustrate the Blue model for several paradigms, such as Map-Reduce, joining datasets, Pregel, iterative algorithms such as k-Means and interactive querying. The model can be efficiently implemented by analyzing the graph and scheduling resources in manner that benefit from data and network locality. Additionally, Blue supports in-memory caching of data, explicitly by programmer or opportunistically by system, which has been shown to greatly improve the latency and throughput of interactive and iterative programs [1]. Finally, the Blue model provides simple and consistent semantics for fault-tolerance of acyclic as well as cyclic dependency graphs.

1. MOTIVATION

Data-intensive cloud computing involves several paradigms of computation for processing large amount of data over a cluster of resources, including batch-oriented (MapReduce [2], Dryad [3]), streaming (STORM [4]), interactive (Dremel [5]), iterative (HaLoop [6]), graph-based (Pregel [7]), structured (Hive [8], Tenzing [9]) and relational (MapReduce Merge [10]); and several application classes within each paradigm. These paradigms and applications are, however, mostly incompatible with each other, that is, a system or framework for one application cannot be easily or efficiently adapted for other application uses cases.

Developing a system or framework anew for each new paradigm is an overwhelming engineering initiative, and particularly inefficient since several capabilities are shared, such as distributed resource management. To address this last need, the cluster computing frameworks were divided into cluster manager layer for distributed resource management, such as Apache YARN [11] and Mesos [12], and the application layer that encapsulates the computation paradigm. Several “Applications” where ported over the cluster manager frameworks, such as MPI, Spark [13], Apache HAMA and, of course, MapReduce.

The cluster manager frameworks, such as YARN, were designed with the objective of supporting multiple application paradigms. While they are capable of abstracting the distributed resource management, they are, however, still too “low-level” to support other capabilities that are common to applications. Any cluster application, in addition to implementing the specific computation paradigm logic, typically needs to support the following capabilities:

a) Scheduling: scheduling jobs, monitoring progress, backup tasks to optimize straggler tasks, support multiple scheduling disciplines;

b) Resource Management: spawn and terminate processes, allocate and free data on disk, interact with distributed storage, optimizations for large data transfers, data pipelining;

c) Client Interaction: report progress, status and errors;

d) Fault tolerance: identify failure of processes, data, communication and nodes, restarting failed resources, skipping bad records;

e) Locality Awareness: rack awareness, data locality;

f) Sorting and merging data on disk, working with compressed data.

The motivation for this paper is to ease the development of new cluster applications, by introducing an intermediate layer (Figure 1) between resource management and applications. This layer would provide the aforementioned capabilities, and more importantly, a generic programming model upon which any arbitrary cluster application can be built. Not only will this significantly diminish the cost of developing applications, the users will be able to easily select the computation paradigm that best fits their needs.

This layer provides an abstraction for distributing computing, in form of a programming model, to the cluster applications. The challenge lies in developing an abstraction...
that is neither too low-level, such a lot of effort has to be devoted to develop a new application, or too high-level, which would be too “rigid” to support different paradigms.

In this paper, we present Blue – a framework for developing cluster applications. The framework provides a novel programming model that is both concise and complete to easily implement arbitrary batch-oriented, interactive, iterative, relational and other programs. We present an outline for implementation strategy, and highlight the key strengths of the framework: efficiency and fault-tolerance for cluster program, with a special emphasis on iterative algorithms. We have developed necessary and sufficient criteria for the job scheduler and have proven that the scheduler will always make progress if there is outstanding work in the system.

2. BLUE FRAMEWORK

Blue framework is a novel approach towards developing cluster computing applications. The framework is a layer above the distributed resource management, and abstracts the distributed computation as a graph (possibly cyclical) of communicating tasks. The layer provides the runtime support capabilities for a cluster application, as listed in previous section, and a programming model to develop such applications. In the following we enumerate the salient features and benefits of our framework:

Completeness of programming model: the programming model, discussed in Section 3, is a generic model that can be used to develop arbitrary applications, including batch-oriented, interactive, iterative, relational and so on. In the next section, we illustrate the programming model in action via several examples. In particular, we note that the programming model is suitable for developing algorithms with cyclical dependencies.

Compact and easy programming model: the application programs are simply written by reading data from ‘input queues’ and writing data to ‘output queues’ (Section 3). In particular, we do not impose a functional paradigm of programming, by requiring developers to translate their problem as a sequence of map, reduce, filter etc operations. This allows the programmers to represent their domain problem compactly, and offers them a greater control to manage their data and computation. For example, they can implement local caching schemes, use local threads for multi-core processing and so on.

Fault-tolerance: the framework provides a consistent fault-tolerance model for any kind of application developed with our programming model. Section 4.3 describes our approaches to ensure fault tolerance for simple applications, applications with long chain of dependent tasks, as well as iterative applications.

Efficient: the programming model and framework design does not constrain efficient implementation of typical bottleneck tasks, such as sorting data on disk, exploiting data locality and rack awareness to minimize network traffic, redundancy of straggler tasks, etc. In addition, the frameworks offers a novel strategy for caching data in memory. In [1], it was shown that computing from cached data, rather than reading from disk, can significantly improve the throughput and latency of applications. Section 4.2 describes our approaches for explicit caching of data by applications, as well as opportunistic caching by the framework. In particular, we note that because of this optimizations, the framework can very efficiently handle iterative algorithms.

Simple implementation: we developed a concise set of simple rules to implement the framework and the scheduler.

We make the following three assumptions for an application that can be represented in our framework:

1. Assumption of decomposability: the application can be decomposed into functions of form \( f(\mathcal{I}) \rightarrow \mathcal{O} \), which processes input data and generates output. The output of one function serves as input to another, thus producing a dependency graph. Note that the framework allows the graph to be cyclical.

2. Assumption of determinism: for the same input, a function always produces the same output.

3. Assumption of parallelism: the functions that are to be parallelized must be inherently parallelizable. We state this formally as:

\[
\text{if } \mathcal{I}_1 \subset \mathcal{I}, \mathcal{I}_2 \subset \mathcal{I}, \text{ such that } \mathcal{I}_1 \cap \mathcal{I}_2 = \phi, \text{ and } f(\mathcal{I}_1) \rightarrow \mathcal{O}_1, f(\mathcal{I}_2) \rightarrow \mathcal{O}_2, \text{ then there exists operator } \sqcup, \text{ s.t. } f(\mathcal{I}_1 \sqcup \mathcal{I}_2) \iff \mathcal{O}_1 \sqcup \mathcal{O}_2.
\]

This formalism covers all problems that be represented by the MapReduce paradigm, and its various extensions.

Limitations of Blue Framework: the framework is targeted for data-intensive computational problems, and is not best suited for task parallelism. It is also not suited for query processing with bounded response delays, and message queue based architectures. For algorithms with cyclic dependencies, the processes cannot persist across iterations; although this is a conscious design decision as discussed in Section 4.3. The processes cannot reopen or rewind their input queues.

3. BLUE PROGRAMMING MODEL

This section describes the concepts of the Blue framework from the point of view of the cluster program developers. We discuss our programming model that represents a cluster program as a collection of interconnected “tasks” and illustrate its expressive power by means of several examples. Later, in Section 4, we shall describe the implementation view of the framework. By the end of this section, we hope to illustrate that Blue is a concise and consistent programming model that can represent a wide class of cluster programs.
We will incrementally develop the concepts of the Blue framework. At the simplest level, a cluster program is a collection of interconnected tasks. For example, the Map Reduce program comprises of four tasks: file reader, mapper, reducer, and file writer. The tasks communicate by sending data over unidirectional links. The program, therefore, can be viewed as a directed graph.

In the next section we describe simple acyclic graphs with “unicast” link; in the subsequent two sections, we develop the concepts of links further; and finally, we consider the cyclical graphs.

3.1 Acyclic Graphs with Simple Links

Whereas the tasks and links are the static representations of a program, the runtime counterpart are called processes and queues. Figure 2 illustrate the distinction: at runtime, the task are launched as processes and the links are manifested as Output Queue and Input Queue at the source and destination processes, respectively. The assignment of data from output queues (at source) to the input queues (at destination) is opaque to the program developers, although they can control some high-level properties, as we shall discuss shortly. The parallelism is achieved by launching multiple processes (on different machines) for a given task.

Rule 1. Processes read data from input queues and write data to output queues as discrete records.

In this way, the Blue links are unlike traditional data “streams”, such as TCP sockets, which treat data as bytes sequence without any record boundaries. In contrast, the Blue framework must know the size of each data record.

In this section, we shall develop several cluster program and express them as Java code fragments. To facilitate understanding of those programs, we provide in Listing 1 the API interface for the InputQueue, OutputQueue and Task.

```
interface InputQueue extends Iterable<Data> {
    boolean isEnded();
    Data read();
    List<Data> readAll();
}

interface OutputQueue {
    void write(Data data);
    void write(Data data, Label lab);
    void close();
}
```

Figure 2: Distinction between static representation of program as a graph (above), and runtime representation as processes and queues.

3.2 Labeled Links

A link may be assigned the “sorted” property, as illustrated in Figure 3(b).

3.3 Fixed Processes and Cross-generational Link

A link may be assigned the “unicast” link; in the subsequent two sections, we develop the concepts of links further; and finally, we consider the cyclical graphs.

Figure 3: Illustration of different graph structures: (a) unicast link, (b) sorted link, (c) fixed processes for a task (Sec 3.1), (d) labeled link (Sec 3.2), (e) broadcast link (Sec 3.3), and (f) Cross-generational cyclic link (Sec 3.4)

```
class Task {
    void execute(List<InputQueue> inQueues,
                  List<OutputQueue> outQueues) {}
    void finish() {}  
}
```

Listing 1: Interfaces for Input, Output Queue and Task

Processes can read data records from InputQueue until the isEnded method returns true. For code brevity, we sometimes read data from the queue as an Iterable, or all at once with the readAll method. Processes write data records to OutputQueue. The overloaded method assigns a label to the data records; we will discuss this in Section 3.2. Note that the input and output queues are blocking in nature, that is, a process can potentially block when reading from or writing to queues.

Finally, the tasks in the cluster program inherit from Task class and provide implementation to the execute function, where they process data records from the input queues and write to output queues. The finish method (needed for cyclical graphs) is called by some task to indicate the program should finish. In cyclical graphs, one task may identify that the program has finished (that is, the desired output has been computed), but due to the nature of cyclical dependencies, the scheduler may never be able to infer that the program has terminated. This method is a directive to the scheduler to avoid this problem.

Figure 3(a) illustrates a simple graph with two tasks and the default “unicast” link between them.

Rule 2. On a unicast link, each data record in the output queue of the source process is assigned to input queue of exactly one destination process.

It is at the discretion of the Blue implementation, to which input queue a particular data record will be assigned.

A link may be assigned the “sorted” property, as illustrated in Figure 3(b).
Rule 3. On a sorted link, the data records in the input queue of a process are sorted.

Although this property is not strictly necessary for the completeness of our programming model (since the tasks can load all data and sort them locally), it is actually efficient if the framework does the sorting, which is why we include this feature as part of the programming model.

The link can be assigned the merge sort or full sort property. Merge sort assumes that data is generated (that is, written to the output queue by the source processes of the link) in sorted order. To produce sorted data at the input queues, the data records from different output queues have to be merged. Full sort assumes that data is arbitrarily generated. In this case, the system would have to wait until all data is generated, sort it and then assign to the input queues.

The final structural property of the graph is controlling the number of processes for a task. By default, the framework would determine the number of processes depending on resource availability and other considerations. However, a cluster program can explicitly indicate the exact count of processes that must be launched for a task. This configuration is illustrated in Figure 3(c).

3.1.1 Example: File Copy

To illustrate the preceding concepts in action we consider a simple cluster program to copy a file. The program graph is identical to the Figure 3(a), where the task A can be viewed as FileReader, and the task B as FileWriter. We omit configuration details that tie the input file to the input queue of the reader process, and written file to the output queue of the writer process. The pseudo-code for these two tasks is as follows:

```java
class FileReader extends Task {
    @Override
    void execute(InputQueue inQ, OutputQueue outQ) {
        for (Data data : inQ)
            outQ.write(data);
    }
}

class FileWriter extends Task {
    @Override
    void execute(InputQueue inQ, OutputQueue outQ) {
        while (Data data : inQ)
            outQ.write(data);
    }
}
```

Listing 2: File Reader and Writer tasks

Both tasks simply reads data from input queue and write to their output queue. If we constrain the number of instances of FileWriter (the graph in Figure 3(c)) to, say, 2, the program would split the input data into two files. Note that the input data is treated as unordered collection of data records.

Similarly, if the link is assigned the “sorted” property (Figure 3(b)), the program effectively sorts the contents of the input files.

While this was a very simple program, we will now extend our programming model further and illustrate it for much more complex cluster programs.

3.2 Labeled Links

Labeled link is a variant of unicast link. When a process writes data records to the output queue, it can assign a “label” to each record. Each record has one label, and multiple records can have same label. The labeled link is denoted by dashed arrow as shown in Figure 3(d).

Rule 4. On a labeled link, each data record written to output queue must be assigned a label. All data items with the same label are guaranteed to be present in the output queue of one process only.

A labelled link thus “gathers” the data based on the labels. Note that the “sorted” property can be applied to the labeled links as well.

3.2.1 Example: Map Reduce

The graph for the Map-Reduce program is illustrated in Figure 4. Notice the link between the Mapper and Reducer tasks is labeled and sorted.

```
File Reader -> Mapper
    |
    v
Reducer
    |
    v
File Writer
```

Figure 4: Map Reduce Program Graph

The FileReader and FileWriter tasks are similar as before. The following code listing describes the implementation of Mapper class; in this particular implementation the Mapper maintains the mapped data in memory. Alternate implementations such as flushing data upon some threshold, etc. can be easily designed.

```java
class Mapper<K1, V1, K2, V2> extends Task {
    private Map<K2, List<V2>> store;

    @Override
    void execute(InputQueue inQ, OutputQueue outQ) {
        for (Map.Entry<K1, V1> data : inQ)
            map(data.getKey(), data.getValue());

        for (Map.Entry<K2, List<V2>> e : store) {
            K2 key = e.getKey();
            List<V2> values = combine(e.getValue());
            KeyValue<K2, List<V2>> data = new KeyValue<K2, List<V2>>(key, values);
            outQ.write(data, key); // 'key' is the label
        }
    }

    protected abstract void map(K1 key, V1 value);

    protected final void emit(K2 key, V2 value) {
        List<V2> values = store.get(key);
        if (values == null) {
            values = new ArrayList<V2>();
            store.put(key, values);
        }
        values.append(value);
    }
```

The MapReduce program graph is illustrated in Figure 4. Notice the link between the Mapper and Reducer tasks is labeled and sorted.
protected void combine(K2 key, List<V2> values) {
    // default implementation — do nothing
    return values;
}

Listing 3: Map Task

The Mapper class maintains an in-memory storage for mapped data as a map of key to list of values. The Mapper process would read data records from the input queue and calls the user-provided map function. The emitted key-value pairs are stored in the local storage, and finally written to the output queue. Since the output queue belong to a labeled link, the data record must be assigned a label; in this case, the label is same as the key portion of the data. For brevity, we included the combine method in this class. The Reducer class is similarly implemented.

3.2.2 Example: Joins and Map-Reduce-Merge

Another example of labeled links is joining multiple data sets. The left graph in Figure 5 illustrates how two datasets can be joined together. The key insight here is that the links between the sources and the task that would join the data are labeled and sorted. The property of labeled links (Rule 4) ensures that all data items with the same key are assigned to same process. Note here that the rule applies to data records labels from all links. The sorted property ensures that the data records on the two input queues can be matched against each other.

A similar example is the Map-Reduce-Merge [10] programming model, illustrated in the right side of Figure 5. In this model, two datasets are processed separately in a Map-Reduce pipeline, and the results of each pipeline are merged together on the key emitted by the reduce phase.

3.3 Broadcast Links

The third kind of link in a graph is a broadcast link, denoted by a double-lined arrow as shown in Figure 3(e). Similar to labeled link, the broadcast link may be assigned “sorted” property as well.

Rule 5. On a broadcast link, each data record in output queues is assigned to each input queue.

In the case of the unicast and labeled links, it is guaranteed that finite amount of data will traverse the link (since the source processes will close the output queues at some point, thus marking the end of data). Unfortunately, the same cannot be said for the broadcast links. In order for the Blue framework to determine how many instances of the destination task to launch, we require:

Rule 6. If a task is a destination to a broadcast link, it must either be configured to have a fixed number of process instances or must also be a destination to some other unicast or labeled link.

3.3.1 Example: Interactive Query Engine

Consider a log analysis program, where the raw data is first filtered, and an operator issues queries interactively and repeatedly. This problem was motivated in [1] as an example where loading data in memory can significantly improve the latency of query responses. Figure 6 illustrates an approach to achieve the same objective with the Blue programming model.

In this particular program, we assume that the sum of memory available to the processes in the cluster is sufficient to retain the filtered data in memory. The general case is discussed in Section 4.2. The Query Engine (Listing 4) first reads from the Filter task completely and thus retains all data in memory. The engine then waits for the queries from the operator. The operator issues queries on a broadcast link, therefore, the query is delivered to all processes of the Query Engine task. These processes execute the query on their local data and send the result to the Aggregator task. This Aggregator task, which is limited to launch one process only, aggregates all partial results to generate the complete response.

class QueryEngine extends Task {
    void execute(InputQueue filterInQ, InputQueue queryInQ, OutputQueue outQ) {
        // load all data in memory
        List<Data> allData = filterInQ.readAll();
        // wait for query from operator
        for (Data query : queryInQ) {
            Data response = process(allData, query);
            outQ.write(response);
        }
        outQ.close();
    }
}

Listing 4: Query Engine Task

3.4 Cyclical Graphs

Cyclical graphs are needed for representing iterative algorithms. Unlike acyclic graphs, the cyclical graphs pose some unique challenges, including:

- Whereas the acyclic graphs have only one interpretation, the cyclic graphs may have multiple interpret-
Cyclical graphs are handled by our programming model, where:

Rule 7. For every cycle in a cyclic graph, at least one link in the cycle must be labeled as cross-generational.

With this background, we can now define how the cyclical graph is unfolded:

Rule 8. Generation 0 sub-graph is identical to the original cyclical graph except that the cross-generational links are removed. Generation $i + 1$ sub-graph is identical to the original cyclical graph, except that the cross-generational links connect the source in generation $i$ with the destination in generation $i + 1$.

The Figure 7 shows three possible unfoldings of a 2-node cyclical graph. In the first two cases, only one link is marked as cross-generational, while in the last case, both links are marked such. Since the structure of the program graph, and consequently the behavior of the program, is determined by which links are marked as XG, we consider this a user-defined property.

Referring to the Figure 7 again, when one task in the original cyclical graph (say, $B$) is unfolded to the counterpart sequence of tasks ($B_1, B_2, B_3,...$). Although these tasks in the unfolded graphs will all share the same implementation code, we still consider them as different tasks. It implies that the processes for tasks in different generations will be launched separately and independent of each other. Another way of saying this is that the processes do not persist across generations. Therefore, if a process has to maintain state across generations, it must encapsulate the state in the messages it communicates with its neighbors. That is, when a task at generation $i$ is completed, it sends its local state to its downstream neighbor. The neighbor would copy this state in the messages it sends, and so on until the message reaches back to the originating task at generation $i + 1$. If the serialized local state size is non-trivial or the cycle length is long, we can add a self-loop to the task in the graph. This self-loop will carry the local state (thus avoiding long cycle traversal), and the scheduler can launch next generation processes on same machines, or even same JVMs (thus avoiding the need to transfer large data).

3.4.1 Example: Kmeans Clustering

Kmeans [14] is an iterative algorithm to cluster a dataset. In each iteration, the algorithm begins with an initial estimate of cluster centroids, and processes the entire dataset to improve upon the centroids estimates. Figure 8 illustrates the program graph for the kmeans algorithm, as well as the first three generations of the unfolded graph. Each generation of the unfolded graph is equivalent to an iteration of the algorithm. In each generation, the KMeans task reads the dataset from the FileReader, and the initial estimate of the cluster centroids from the Looper task. Each process for the KMeans task receives these centroid estimates (since Looper sends them over a broadcast link); and each process then executes the kmeans algorithm on its portion of the data set and finally sends the locally computed result to the next generation of the Looper. The next generation of Looper would merge the partial results, and decide if the program termination conditions are met. If so, the Looper process invokes the finish() method (Listing 1), otherwise the next generation unfolds.

In each generation, the KMeans task reads the same dataset
from the FileReader. It was demonstrated in [1] that up to 20x improvements can be expected if the data is retained and read from memory, instead of fetching from disk in each iteration. The Blue framework supports this performance optimization by opportunistic caching of data in cyclical graphs, as discussed in Section 4.2.

3.4.2 Example: Iterative Graph Processing

Pregel [7] introduced a programming model inspired by Bulk Synchronous Protocol (BSP) for large-scale graph processing. In each iteration of the program, each graph vertex would receive messages from its neighbors, and at the end of the iteration each vertex would possibly update its state and send further messages to its neighbors. There is a “barrier synchronization” between each iteration: a vertex would not execute the next iteration, until all vertices have executed the current iteration.

Figure 9 illustrates the Blue program graph to emulate the Pregel programming model. There are two self-links at the Pregel task: S-Link (to transfer State) and M-link (to transfer Messages). Each of these links is labeled, sorted and cross-generational (all self-loops must be cross-generational, because of Rule 7). In the first iteration, the Pregel task reads the structure of the graph and the initial state of the vertices from a file, and each vertex generates some outgoing messages. The messages are send on M-link, where the label of the message is equal to the destination vertex ID. The modified state of the vertex is sent on the S-link, and the label of data record is equal of the vertex ID. These messages are delivered to the next generation of the Pregel task. At this point, the property of the labeled links ensure that data record containing the state of a vertex, and all messages that are destined to it are delivered to same process. The sorted property of the link ensures that the process can easily “match” the records from two input queues. This behavior is exactly similar to the example of joining multiple datasets we discussed earlier.

As a side note, if the complete state of the vertices is sent on the S-link, there is no need to actually read from the FileReader from second iteration onwards. The implementation can provide shortcuts such as ‘seek to the end of stream’ to avoid the unneeded effort of reading data.

4. IMPLEMENTATION STRATEGY

4.1 Scheduling

The scheduler is the heart of the framework implementation. The centralized scheduler maintains, in memory, the entire program graph and assigns to each task and link one of the following states:

- WAITING: initially all tasks and links are in state, and are waiting to be scheduled.
- ACTIVATED: when the scheduling rules (described below) have determined that a task or link is now considered active, and must be monitored by the scheduler.
- RUNNING: a task in this state implies that the corresponding processes have been spawned and are running; in case of links, it implies that there is some data in the output queues.
- DEACTIVATED: when the scheduling rules have determined that the task or link will not change state from this point on.

In the following, we state the scheduling rules first following by an intuitive understanding. In Appendix A we prove the correctness of these rules by showing that the scheduler will always make progress when there is outstanding work, and will never fail to infer when the program terminates.

Rule 9. A task \( T \) is activated, if

\[
\text{input\_links}(T) = \emptyset \text{ OR } (\forall L \in \text{input\_links}(T) : \text{state}(L) = \text{DEACTIVATED} \text{ OR } (\text{NOT is\_sorted}(L) \text{ AND state}(L) = \text{RUNNING}))
\]

The scheduler would launch the data sources in the beginning. All other tasks will be activated when any input link is deactivated. However, if the link is not sorted, the task can be activated earlier when the link is in running state.

Rule 10. A task \( T \) is running, if state(T) = ACTIVATED and at scheduler’s discretion.

Although the rules determine when a task is activated, it is at the scheduler’s discretion when it is actually launched. The scheduler, thus, has flexibility to launch processes by taking into account the resource availability and other considerations.

Rule 11. A task \( T \) is deactivated, if

\[\forall L \in \text{output\_links}(T), \text{state}(L) = \text{DEACTIVATED}\]

A task is considered deactivated when all output links are deactivated. Clearly, the task will not generate any further output, and therefore, can be considered inactive.

Rule 12. A link \( L \) is activated, when the task \( \text{source}(L) \) is activated.

When a task is activated, it is in position to generate output data. Therefore, all the outgoing links are considered activated.

Rule 13. A link \( L \) is running, if there is data on some output queue.

Whereas the activated state of a link implies that it is possible that there will be data on the link; the running state confirms that there is actually some data on the link.

Rule 14. A link \( L \) is deactivated, if

\[(\text{input\_links}(	ext{source}(L)) = \emptyset \text{ OR } \forall E \in \text{input\_links}(	ext{source}(L)) : \text{state}(E) = \text{DEACTIVATED}) \text{ AND all output queues are closed}\]
Consider a link between tasks \( A \) and \( B \). This link will be deactivated when we can assert that there will not be any more data written to the output queues of this link. We can assert this in two steps: (a) all incoming links to \( A \) are deactivated, which affirms that the processes for \( A \) will not receive any new data, and (b) all processes for \( A \) have closed their output queues, thus affirming that will not write any more data.

**Rule 15.** InputQueue for a process ends when the corresponding link is deactivated, all data from all OutputQueues is assigned to the input queues, and this process has read all data in this input queue.

Consider again a link between \( A \) and \( B \). For a process, \( b_i \) belonging to task \( B \), we need to make an assertion when this process will not receive any more data in its input queue. This is asserted in three steps: (a) this link is deactivated, which affirms that there will not be new data generated by source task (see Rule 14), (b) all data from output queues have been assigned amongst the processes for \( B \), that is, there is no data ‘in transit’, and finally, (c) the process \( b_i \) has read all data in the input queue.

**Rule 16.** OutputQueue can be closed only after all InputQueues have ended.

This rule asserts that a process must read all input before it can signal the end of output.

**Rule 17.** The finish() method can be called only after all output queues are closed.

This rule asserts that a process can signal the end of program only after its has finished.

**Rule 18.** The program completes when all tasks are DEACTIVATED or all processes of some task have called the finish() method.

The first clause asserts that the state of the overall program will not change, therefore, the program can be considered as completed. The second clause is a direct notification for program termination.

### 4.2 In-memory Optimizations

It was shown in [1] that for the interactive and iterative algorithms, keeping data in memory can significantly improve the program latency and throughput. In the following we illustrate three unique characteristics of the Blue Framework, which can be exploited to develop highly efficient programs:

**Program-aware Caching:** The query engine program illustrated in section 3.3.1 was an example where the program explicitly stores data in memory. The strength of the Blue programming model lies in the fact that the programs can effectively determine the lifetime of tasks, by suitably constructing the program graph and by deciding when to close the output queues. With this control over the lifetime of tasks, the program can load data in memory (during any stage of the program) and keep it around for as long as needed.

**Opportunistic Caching in Cyclical Graphs:** The Blue framework handles the cyclical graphs by unfolding them as an unbounded sequences (generations) of acyclic graphs. The unfolding rule states that the acyclic links from the original graph are copied as-it-is in each generation. It implies that in each generation the “acyclic portion” of the graph must be computed again. However, by the assumption of determinism, we know that the data produced in each generation will be identical. Clearly, this data can be cached on disk, and the scheduler can ensure that the processes for the next generation are spawned on the machines where their cached data is located. The scheduler can further optimize by opportunistically caching the data in memory as well. As an example, the data is loaded in a JVM, and the process for a new generation is launched in the same JVM. When this process reads from input queues, it is actually reading data from process-local memory. This is opportunistic, since if the in-memory data is lost, it can always be loaded from the disk-backed cache.

**Static Graph Analysis:** The Blue framework has some knowledge of the program behavior even before it is launched, via the graph structure. The scheduler can statically analyze the graph and recommend optimizations. As an example, a unicast link can be scheduled by having destination process run on the same machine, or in fact the same process space, as the source process.

### 4.3 Fault Tolerance

The assumption of determinism as stated in Section 2 is critical for ensuring the fault-tolerance of the programs. The assumption states that if the same input is provided to a process, it will generate the same output. We also noted earlier that the scheduler is responsible for assigning data from output queue of a source process to input queues of destination processes. The scheduler, therefore, only needs to maintain a mapping of a data record in output queue to the identity of the input queue where it was assigned. If the destination process dies, the scheduler can recreate its input queues re-executing the previous processes, and extracting relevant data by consulting this mapping table. This process can be repeated if the previous processes are also dead.

The above approach is, however, not efficient for very “long” acyclic graphs and cyclical graphs will large number of iterations. In this case the scheduler would cache the contents of output queues on disk, so that upon failure the data can be recreated from the cache rather than recomputing the previous tasks. The scheduler can determine which links must be cached by offline analysis (e.g., labeled links should have higher priority for caching than unicast links, since they tend to distribute data all over the cluster) or using hints supplied by users. Additionally, the scheduler can also determine at runtime, for example, it may cache results of a task if it took more time than some threshold to complete, or generated more data than some threshold.

The Blue framework is specifically designed for ensuring fault-tolerance of iterative algorithms with cyclical program graphs. We noted earlier that when a cyclic graph is unfolded, each task in the original graph is copied as a new task for each generation of the unfolded acyclic graphs. These tasks, in each generation, are considered independent and the processes spawned by these tasks bear no dependencies with each other. In other words, we enforce that the processes do not persist across generations. The only prescribed mechanism to maintain state across generation is to serialize it and send it over the communication links. We opted for this design, rather than alternative approaches based on process persistence across generations, because we believe the current design provides better guarantees for fault-tolerance.
and is easy to implement in context of distributed data-centric computations.

Let us consider the alternative design approach of persistent processes. These processes can be fault-tolerant only if the program developer has encoded “state checkpoint” and “checkpoint recovery” in the application. This calls for extraneous efforts from both developers and the framework: developers have to encode the logic of the application as well as failure recovery, and the framework has to support this new mode of fault recovery. Secondly, if the granularity of checkpointing is not sufficient, or if the developer has not written the checkpointing and recovery code, or if there is a bug in this code itself, then the framework has no option but to rollback to the first iteration, which can be very costly for long running iterative programs.

In our design, the task is implicitly checkpointed at the end of each iteration, when it writes its state to the communication link. Thus, if a task dies, its inputs and state can be recreated from the cached output queues of previous generation tasks. Recovery by recreating input queue from cached output queues was exactly the same mechanism we employed for fault-tolerance of acyclic graphs. This is the second, and perhaps a greater, benefit of our design: the fault-tolerance mechanism is easy to implement in Blue framework, since the same mechanism of fault-tolerance works for both acyclic and cyclic graphs.

5. RELATED WORK

Dryad [3] was an early attempt to develop a common framework to model different kinds of applications. Similar to Blue, the framework assumes that a program can be decomposed into a graph of tasks. The tasks are programmed as processes that read data from input links and write data to output links. However, Dryad is limited to acyclic graphs, and thus unsuitable for modeling iterative algorithms.

Resilient Distributed Datasets (RDD) [1] is a distributed memory abstraction and a programming model to develop iterative and interactive applications. While the architecture is generic to model different computation paradigms, the focus of the system is towards in-memory computation. The Blue framework shares a similar approach as RDD to opportunistically retain the dataset in memory, in order to improve the program latency and throughput. The Blue framework, however, is not limited to in-memory computation only; it is suitable for batch-oriented and other paradigms as well that would not necessarily benefit from caching data.

6. CONCLUSION

In this paper we have presented a approach towards a unified cloud computing framework for data-intensive applications. The key insight is a generic programming model that can represent a wide variety of computation paradigms. The proposed programming model provides a rich set of constructs to decompose a program into a graph, possibly cyclical, of inter-dependent tasks. We outlined the scheduling principles, stated as a set of concise and consistent rules, and an efficient and fault-tolerant implementation strategy for the framework.

We have presented several examples to illustrate the flexibility of our programming model in representing applications such as Map Reduce, interactive queries, Pregel, dataset joins, and iterative algorithms. We have also shown that the framework can improve the overall throughput and latency of the programs by caching repeatedly processed data in memory. We have further defined concise and consistent fault-tolerance semantics that applies to graphs with acyclic as well as cyclical dependencies.

7. REFERENCES

APPENDIX

A. PROOF OF SCHEDULER CORRECTNESS

We show that the rules which define Blue’s programming model ensure correct and consistent behavior of its scheduler. In other words, Blue will always make progress on a program, eventually terminate and never deadlock. In the following, we prove these claims for programs with finite data sources.

Definition 1 (Pathological Process). A process is pathological if it doesn’t read input when input is available on an input queue, or it doesn’t close all its output queues, when it has read all input (and subject to Rule 16).

Definition 2 (Root Task). A root task is one with no dependencies on other tasks. This implies that a root task either has no incoming links, or all its incoming links originate from data sources.

Lemma 1. Each link will be deactivated, assuming there are no pathological processes.

Proof. We note that the program graphs are always acyclic, since the cyclic graphs are transformed to acyclic graphs by the unfolding technique (Rules 7 and 8). By the property of acyclic graphs, an ordering can be enforced for the tasks in a graph based on the longest path length required to reach each task from a root task. We prove the lemma by induction on the ordering of tasks.

(1) Base Case: The links ending at root task will be deactivated. When the root task admits incoming links from one or more data sources, by Rules 2, 3, 4 and 5, Blue will ensure that all records from each data source will be read and allocated. Once this has completed, Blue will also ensure that the corresponding OutputQueues will be closed. By rule 14, it follows that links ending at root tasks will be deactivated by Blue.

(2) Induction Step: For a task \(i\), let \(L_{in}\) be the set of all incoming links, and \(L_{out}\) be the set of all outgoing links. We show that if all links in \(L_{in}\) are deactivated, then all links in \(L_{out}\) will also be deactivated. By Rules 2, 3, 4 and 5, Blue will assign all data from the output queues for the links in \(L_{in}\) to the corresponding input queues. Finally, since \(i\) is not comprised of pathological processes, it will read all data on all its InputQueues. It follows that the 3 conditions required by Rule 15 have been satisfied, and therefore all input queues for \(i\) will end. Again, by Rule 16 and by assumption of non-pathological, the processes will close the output queues for all links in \(L_{out}\). It now follows from Rule 14 that all outgoing links \(L_{out}\) of \(i\) will be deactivated.

Lemma 2. Each task will be activated.

Proof. Since we proved in lemma 1 that all links will be deactivated, it follows from rule 9 that all tasks will be activated.

Lemma 3. Each link will be activated.

Proof. Since we proved in lemma 2 that all tasks will be activated, it follows from rule 12 that all links will be activated.

Theorem 1. Blue will always make progress until termination.

Proof. By lemmas 2 and 3, all links and tasks will be activated. Barring a call to finish(), it must ensure from rule 10, that all tasks will run. Consequently, by rules 13 and 14 all links will either run and deactivate or deactivate without running, subject to process behavior of preceding tasks. This is sufficient to shows that Blue will always make progress.

Progress will however cease either when all tasks have deactivated, or if a successful call to finish() is made. If no more tasks remain, Blue can trivially terminate the program. Alternatively, if a task successfully issues a call to finish() (pursuant to rule 17), this will prevent the following tasks in the task graph from progressing to the running state. However, this will also successfully initiate termination, as per rule 18. Hence, progress is ensured until termination.

Lemma 4. Each running task will be deactivated.

Proof. Since we proved in lemma 1 that all links will be deactivated, it follows from rule 11 that all tasks will be deactivated.

Theorem 2. Blue will never deadlock.

Proof. Rules 7 and 8 ensure that each cyclic graph is unfolded into a sequence of directed acyclic graphs. In such a graph, by definition, there are no cycles or loops. This prevents a circular wait from arising, thereby preventing a deadlock from occurring. This, of course, applies to acyclic task graphs as well.

Theorem 3. Assuming programs with cyclic graphs include at least one task that successfully calls finish(), Blue will terminate a program.

Proof. By lemmas 4, all tasks will be deactivated. Further, from theorem 2, Blue will never deadlock. Since Blue will always make progress (theorem 1), in the case of an acyclic graph Blue will run out of tasks that haven’t been deactivated, and will terminate the program. In the case of a cyclic graph that unfolds into an unbounded number of tasks, a call to finish() by a task is necessary to signal termination. Once called, by rule 18, Blue will terminate the program.