Abstract

Implementing machine learning algorithms for large data, such as the Web graph and social networks, is challenging. Even though much research has focused on making sequential algorithms more scalable, their running times continue to be prohibitively long. Meanwhile, parallelization remains a formidable challenge for this class of problems, despite frameworks like MapReduce which hide much of the associated complexity. We present a framework for implementing parallel and distributed machine learning algorithms on large graphs, flexibly, through the use of functional programming abstractions. Our aim is a system that allows researchers and practitioners to quickly and easily implement (and experiment with) their algorithms in a parallel or distributed setting. We introduce functional combinators for the flexible composition of parallel, aggregation, and sequential steps. To the best of our knowledge, our system is the first to avoid inversion of control in a (bulk) synchronous parallel model.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming—Distributed and parallel programming: D.2.13 [Software Engineering]: Reusable Software—Reusable libraries

Keywords Parallel programming, distributed programming, machine learning, graph processing

1. Introduction

There is a growing need to facilitate the analysis of large data. Fields like bioinformatics, speech processing and medical imaging are being faced with data sets that are growing in both size and complexity, while the world of social media and other web services are struggling to deal with the terabytes of data they collect daily.

Machine learning (ML) has provided elegant and sophisticated solutions to many complex problems on a small scale, which if ported to large scale problems, could open up new applications and avenues of research for these and related fields. Unfortunately, ML research efforts are routinely limited by the complexity and running time of sequential algorithms. Meanwhile, while other areas of software development have been readily facing an ongoing shift to parallel and distributed hardware, the ML community, a community full of entrenched procedural programmers, has instead largely focused their efforts on the optimization specific algorithms, when faced with scaling problems. Overall, scant effort has been made to generalize and facilitate learning on large data, which currently limits many efforts and applications of ML.

The popularity of MapReduce [9], a functional abstraction designed to simplify the implementation of distributed algorithms, has inspired a number of efforts to utilize MapReduce/Hadoop for large scale learning tasks [1, 8, 21]. However, other efforts which have chosen not to base their system on MapReduce, like GraphLab [17] and Google’s Pregel [18], have noted several limitations of MapReduce in the context of ML. In particular, they point out that in many situations it is not straightforward (nor appropriate) to chain instances of MapReduce in order to achieve iteration, and that doing so introduces considerable communication and serialization overhead.

Although based on graphs and graph processing, GraphLab’s programming model is asynchronous, requiring so-called consistency models to prevent data-races. We argue that this programming model, while often faster than its synchronous counterpart, can lead to non-deterministic behavior (by allowing for inexact results through relaxation of the consistency model), and as a result is arguably more difficult to use and troubleshoot. Instead, we advocate a programming model more akin to Google’s Pregel, which capitalizes on the organization and determinism that results from a computational model inspired by the Bulk Synchronous Parallel (BSP) model [25].

While also based on graphs, Pregel is a closed system that was designed to solve large-scale “graph processing” problems, which are usually simpler in nature than typical real-world ML problems. In an effort to capitalize on Pregel’s strengths while focusing on a framework more aptly-suited to ML problems, we introduce a more flexible programming model, based on high-level functional abstractions.

Our goal in designing this framework is to enable researchers and practitioners to quickly implement and experiment with their algorithms in a parallel or distributed setting. We believe that a synchronous model which transparently distributes functional computations across cores (and eventually, machines in a cluster) is a first step toward this goal, by simplifying reasoning about program semantics.

OptiML, a Matlab-like domain-specific language (DSL) shares this goal. It is embedded in Scala via lightweight modular staging [24]. In contrast to OptiML, our programming makes some parallelism explicit in the form of (bulk) synchronous parallel supersteps. Like in FlumeJava [6], chains of supersteps can be optimized through delayed computations. Language virtualization [4] coupled with the Delite DSL framework [5] enables more extensive, expression-level optimizations in OptiML. In the future, we would like to use this approach to optimize the supersteps of our
model. Additionally, extending Delite with support for graphs and distribution would benefit both the present framework and OptiML.

Our paper makes the following contributions:

- A general framework for parallel graph processing, suitable for parallelizing ML algorithms.
- The integration of a functional aggregation mechanism with a simple bulk synchronous parallel model.
- The introduction of functional combinators that avoid an inversion of control present in other graph-processing frameworks based on the BSP model.
- A Scala implementation and preliminary experimental evaluation of our framework using PageRank and hierarchical clustering on real-world data1.

The rest of this paper is structured as follows. Section 2 gives a high-level overview of our BSP-like computational model. In Section 3 we introduce our graph-based programming interface providing high-level control structures to compose supersteps. Section 4 discusses our actor-based implementation. In Section 5 we show examples of ML applications, parallelized using our framework. Section 6 discusses other related work, and Section 7 concludes with an outlook of future work.

2. Model of Computation

In our framework, all data to be processed is represented as a data graph. Vertices in such a graph typically represent atomic data items, while edges represent relationships between these atoms. For example, in the graph used to compute the page rank of a collection of URLs, each vertex would represent a URL and each edge would represent a link from one URL to another. For other data types not typically represented by graphs, like image or video data, for example, each pixel can be represented as a vertex with edges connecting neighboring pixels (either spatially or temporally).

In addition to the data item that it represents, a vertex stores a single value that is iteratively updated during processing—thus, an algorithm is implemented by defining how the value of each vertex changes over time. Updates to the vertices’ values proceed in synchronous supersteps that are reminiscent of Valiant’s Bulk Synchronous Parallel model [25]. A superstep consists of executing an update step on all vertices in the graph; update steps are defined locally for each vertex. The result of an update step is a list of outgoing messages to other vertices. These messages are made available to the target vertices as incoming messages at the beginning of the next superstep. Thus, the update of a vertex’s value is based on its current value, its list of incoming messages, as well as its local state.

Besides regular update steps (in which all vertices run in parallel), our programming model also provides a mechanism for aggregation using so-called crunch steps. A crunch step basically corresponds to an invocation of the standard reduce combinator in functional programming. It is used to compute a single aggregate value over all vertices in the graph. The main differences between crunch and the standard reduce combinator are that: (i) the result is made available to all vertices via an incoming message that is available at the beginning of the next superstep, and (ii) its invocation is vertex-local.

3. Programming Interface

The programming interface of our framework is centered around graphs and vertices. Implementing an algorithm consists of two steps. The first step involves the creation of a graph which holds the data to be processed in its vertices and edges. In the second step, we define how the value of each vertex in the graph is iteratively updated. The computation is then kicked off, providing a termination condition.

3.1 Graphs and vertices

Graphs are created by instantiating the Graph class; the class has a single type parameter Data which is the type of the data item that each vertex in the graph manages. The graph is built by adding instances of type Vertex[Data].

Figure 1 shows the interface of the Vertex class. First, there are (standard) neighbors and graph methods to access the neighbors of a vertex, as well as the Graph instance that it belongs to. The data item of a vertex is maintained in its value field. Finally, the update method defines how the value of the vertex is updated in each superstep of the computation. It is the only abstract method of Vertex. Implementing a concrete algorithm, such as page rank, consists mostly of providing an implementation of update in a subclass. The superstep member can be used to adapt the behavior of update according to the current phase (given by the superstep number) of the algorithm (we show an example below). The incoming member provides access to messages that other vertices have sent to the current vertex during the previous superstep (i.e., superstep - 1). Messages are instances of the following case class:

```scala
abstract class Vertex[Data](initialValue: Data) {
  def neighbors: List[Vertex[Data]]
  def graph: Graph[Data]
  var value: Data = initialValue
  def update(): Substep[Data]
  def superstep: Int
  def incoming: List[Message[Data]]
}
```

Figure 1. The public interface of the Vertex class.

In addition to carrying a value of the generic Data type, a message identifies both the source and destination vertices.

Invoking update returns an instance of Substep[Data] which represents a delayed update step to a Data value. Typically, update simply returns a list of outgoing messages to other vertices. In this case, the body of update is converted to an instance of Substep[Data] using an implicit conversion. Below we explain how substeps can be used to conveniently structure iterations while allowing our run-time system to optimize synchronization by merging substeps (Section 4.1).

3.2 Example: page rank

Figure 2 shows how to create a simple graph and populate it with a few vertices where each vertex manages a value of type Double. In this example, each vertex represents a web page, and the data value of each vertex is the page rank of its web page. The vertices that we add to the graph are instances of the PageRankVertex class which implements the page rank algorithm. As mentioned before, in our framework algorithms are expressed in terms of updates applied to (the value of) each individual vertex. Concretely, this is done by subclassing the generic Vertex class and implementing its update method.

Figure 3 shows how the page rank algorithm is implemented using a subclass of Vertex[Double]. Its update method defines

```scala
abstract class Vertex[Double](initialValue: Double) {
  def neighbors: List[Vertex[Double]]
  def graph: Graph[Double]
  var value: Double = initialValue
  def update(): Substep[Double]
  def superstep: Int
  def incoming: List[Message[Double]]
}
```

1http://lamp.epfl.ch/~phaller/menthor
// A tiny web graph:
// BBC -> MS, EPFL -> BBC, JohnDoe -> BBC,
// JohnDoe -> EPFL
val g = new Graph[Double]
val v1 = g.addVertex(new PageRankVertex("BBC"))
val v2 = g.addVertex(new PageRankVertex("MS"))
val v3 = g.addVertex(new PageRankVertex("EPFL"))
val v4 = g.addVertex(new PageRankVertex("JohnDoe"))
g.connect(v1, v2)
g.connect(v3, v1)
g.connect(v4, v1)
g.connect(v4, v3)

Figure 2. Creating a simple web graph.

class PageRankVertex(val label: String) extends Vertex[Double](0.0d) {
  def update() = {
    var sum = 0.0d
    for (msg <- incoming) {
      sum += msg.value
    }
    this.value = (0.15/graph.numVertices)+0.85*sum
    if (superstep < 30) {
      val n = neighbors.size
      for (neighbor <- neighbors) yield
        Message(this, neighbor, this.value/n)
    } else
      List()
  }
}

Figure 3. Implementing PageRank.

the update step that is executed in each superstep. By making
the current superstep accessible inside the body of update, it is
possible to customize the behavior according to a particular phase
of the algorithm. The example algorithm has two simple phases:
updating the value of a vertex according to the page rank algorithm
is only done until superstep 30. After that, the value doesn’t change
any more. The page rank algorithm itself is implemented using
messages that vertices exchange. At the beginning of a superstep, a
vertex may have access to incoming messages which other vertices
have produced during the previous superstep. These incoming
messages contain the per-edge page ranks of pages with incoming
links to the current page. The message values are summed up
and used to compute the updated page rank of the current page,
this.value. Subsequently, the vertex produces a list of outgoing
messages to its neighbors, containing its updated per-edge page
rank (Message(this, neighbor, this.value/n)).

3.3 Substeps
Many graph-processing algorithms require interleaving parallel up-
date steps, which are executed on all vertices concurrently, with
sequential code. Examples for such sequential blocks are aggregat-
ing the values of all vertices in the graph or (in a non-distributed,
multi-core setting) updating shared state.

Our framework supports these interleavings through substeps.
A substep is a closure implementing an update step. By default,
the update method of (a subclass of) Vertex creates a single
substep which defines the update executed on that vertex in each
superstep of the computation. A set of substep combinators allows
def update() = {
  // only the first vertex executes this substep
  if (this == graph.vertices(0)) {
    // find the minimal distance
    val closest = SharedData.distances(0)
    val newCluster = merge(closest)
    for (v <- graph.vertices)
      yield Message(this, v, newCluster)
  }
  else
    val mergedCluster = incoming.head.value
    // compute the pearson distance from
    // our cluster to the merged cluster
    if (value != mergedCluster) {
      val distance = pearson(value.vec, mergedCluster.vec)
      // put distance into SharedData.distances...
    }
    // no outgoing messages
    List()
  }
}

Figure 4. Using substeps to interleave sequential code with parallel
update steps in a clustering algorithm.

composing substeps to form chains of computations. Essentially, a
chain of substeps structures an iteration into several phases, where
each phase may be either parallel or sequential.

For example, consider the implementation of a clustering al-
gorithm in the multicore setting (where it is possible to leverage
shared memory.) On each iteration, the clustering algorithm must
look up the pair of clusters with the minimal distance between
them, stored in a table of distances in shared memory, and then
merge them. In an effort to simplify the implementation, we might
want to have only one vertex decide upon which vertices to merge.

In our framework it is easy to introduce this type of global
decision making step. Figure 4 shows how to divide an iteration
of the clustering algorithm into several substeps using the then
combinator. The first substep is executed only by a single vertex.

3.4 Avoiding inversion of control
Substeps introduced using then and other combinators (see below)
avoid an inversion of control of previous systems based on the
BSP model. Essentially, in a system based on inversion of control,
it is not possible to express computations involving (logically)
blocking operations in direct style. Instead, the control flow has
to be “inverted” by explicitly managing the execution state across
such operations.

In the BSP model and related models such as Pregel, blocking
operations are implicit due to bulk synchronization between super-
steps. Any messages sent during a superstep are made available
to all receivers only at the beginning of the next superstep. Previous
systems do not provide control structures for composing computa-
tions executed during a single superstep. Instead, the programmer
has to explicitly manage the execution state. The required program
rewrite usually obscures its control flow.

As a simple example, consider implementing a sequence of
computation steps where each step is executed during a single
superstep. In a system like Pregel it is necessary to express this
sequencing based on the current superstep or some local state of
def update() = {
  if (superstep % 4 == 0) {
    ...
  } else if (superstep % 4 == 1) {
    val mergedCluster = incoming.head.value
    ...
  } else if (...) {
    ...
  }
}

Figure 5. Structuring supersteps with inversion of control

def update() = {
  var acc = 0
  thenUntil (acc == 10) {
    ...
    acc += 1
  } then {
    ...
  }
}

Figure 6. Using until for iterations that span multiple substeps.

the vertex. This is shown in Figure 5. While simple sequencing is not too difficult to express, it is very hard to express more complex control structures, such as loops.

Besides composing substeps using the then combinator, our framework provides other high-level control structures that take full advantage of closures. For example, Figure 6 shows how to express iteration spanning multiple substeps using the thenUntil combinator. Importantly, “loop iterations” are evaluated in subsequent supersteps. This means that inside the loop body, incoming holds the incoming messages received in the global superstep in which the previous loop iteration was run. This way it is possible to directly express iteration over incoming messages received in subsequent supersteps which is impossible to do in existing systems, such as Pregel.

3.5 Aggregation

Besides regular substeps (in which all vertices run in parallel), our programming model also supports aggregation steps that we call crunch steps. A crunch step basically corresponds to an invocation of the standard reduce combinator in functional programming. It is used to compute a single aggregate value over all vertices in the graph. The main differences between crunch and the standard reduce combinator are: (i) the aggregation is initiated local to a vertex, even though it results in a single global operation, and (ii) the result of the aggregation is made available to the set of vertices via an incoming message that is available at the beginning of the next superstep.

Figure 7 shows how to use the crunch combinator to compute the sum of all vertex values in the graph. Since crunch creates a Substep instance it can be composed with other substeps using then and other combinators. By default, the result of a crunch step is sent to all vertices. Thus, each vertex can access the aggregated value through its incoming message list in the subsequent superstep.

To allow application-specific optimizations it is also possible to send the result of an aggregation to only a subset of the vertices; this is supported through variants of crunch which additionally specify the set of messages to be sent. For instance, this set can be computed using an additional closure which takes the aggregation result as an argument:

```
crunchTo: ((Data, Data) => Data)
(Data => List[Message[Data]])
```

4. Implementation

The implementation of our framework is based on Scala Actors [13]. The central graph instance is an actor which manages a set of worker actors. Each worker actor is assigned a partition of the graph’s vertices. A worker is responsible for executing update steps on the vertices of its partition. The graph actor synchronizes all workers using supersteps (see Section 2). A superstep is initiated by sending a special “Next” message to all workers. Upon receiving this message, each worker

1. removes all incoming messages that were sent in the previous superstep from its mailbox;
2. selects and executes the next update step on each vertex in its partition;
3. delivers outgoing messages which were generated by its vertices in the current update step.

In addition to synchronizing workers, the graph actor manages iteration, supporting termination conditions, and aggregation.

Aggregation Aggregation steps are initiated as substeps of the update method of a vertex: the vertex creates an aggregating step passing a closure which defines the combination of the data items of a pair of vertices. Upon invoking an aggregating step, a worker reduces the values of all vertices in its partition using the provided closure. The result and the closure that was used to compute it is sent to the graph actor which computes the final aggregated value. The aggregation result is then passed to all workers which make it available to their vertices as incoming messages at the beginning of the next superstep.

To reduce the number of messages sent during an aggregation step, and to parallelize aggregating intermediate results from different workers, we organize a group of aggregating actors in a tree. Aggregation results are propagated from the leaves to the root which is the graph actor. Whenever an aggregator has received the results of all of its children, it reduces them and sends the result to its parent. The final reduced result is then propagated down the tree. When it reaches a worker at a leaf, this worker initiates a new superstep for the vertices in its partition and makes the reduction result available to its vertices as incoming messages.

4.1 Merging substeps

Our framework uses closures to compose substeps of the update step of a vertex. Each substep corresponds to a delayed computation that produces a list of outgoing messages when evaluated.
In addition to these application-level messages, executing substeps gives rise to internal control messages that are used to synchronize workers according to the supersteps of the computation.

Delaying the execution of substeps allows us to optimize the message flow induced by a chain of substeps prior to their execution. For instance, executing two crunch steps in sequence can be optimized by computing the reduction results simultaneously. Of course, this is only a valid optimization if the reduction functions passed to the crunch combinator are side-effect free, which is the case for typical aggregation functions, such as min, max, or sum. In future work we intend to evaluate the effect of message flow optimization through substep merging (the benchmarks we have considered so far would not benefit from substep merging.)

5. Applications and Preliminary Results

In this section, we introduce example ML applications which require extensive computation and do not easily scale to large data, but can be parallelized using our framework.

Page Rank We used the parallel page rank implementation of Section 3.2 to compute the page rank of large subsets of the (the English-language) Wikipedia. We ran the algorithm between 10 and 30 iterations, on subsets of 64,000 and 107,000 vertices. Figure 8 shows the results of running PageRank on an 2.1 GHz AMD Opteron(tm) 12-core processor using Oracle’s JDK 1.6.0 with a heap size of 8 GB. In each case we ran the entire benchmark, which includes reading the Wikipedia data from disk and building the graph in memory, and took the average of 5 runs. Despite significant time spent in sequential initialization code, the speed up when utilizing all 12 cores is around 8.7. Interestingly, these results show that despite an increase in input size, and number of iterations, the speed up nonetheless remains about the same.

Hierarchical Clustering We used our framework to implement a hierarchical clustering algorithm, a form of cluster analysis and a branch of unsupervised ML. The objective of a clustering algorithm is to group data elements into subsets, or clusters. Hierarchical clustering in particular seeks to iteratively choose pairs of data elements with the minimum distance (Euclidean, Manhattan, Pearson, etc.) between them, and to merge them together to form a cluster. The result is a hierarchy in the form of a dendrogram which groups member elements by their measure of similarity.

Due to our framework’s ability to seamlessly integrate sequential and parallel blocks of code, a sequential hierarchical clustering algorithm can quickly be implemented with parallel substeps. Using a partially sequential, partially parallel algorithm, we performed hierarchical clustering on a collection of blogs in an effort to cluster them by thematic category. Despite the fact that we only parallelized a single step of the algorithm, we were able to achieve a speed up of around 2 on an 8-core machine.

Belief Propagation We used our framework to implement Loopy Belief Propagation (BP), an algorithm for approximate inference on graphs. The goal of Loopy BP is to iteratively approximate the marginal distributions at unobserved nodes given observed nodes (through message passing), until some convergence condition is achieved. The standard formulation of BP is based on a message passing formulation which is well-supported in our framework.

6. Other Related Work

Our framework provides users with an API for parallel programming with graphs based on synchronous supersteps inspired by the BSP model. Details of parallelization, including load balancing and messaging are handled transparently by the implementation. There exist several general BSP library implementations [2, 11, 14, 19] which, unlike our framework, do not provide a graph-specific API. Furthermore, we use functional programming techniques to avoid an inversion of control in BSP-like systems with a stateful model of long-lived (vertex) processes.

Our approach is different from other systems that hide parallelization and distribution details such as Pig Latin [20], Dryad [15, 26], Sawzall [22], Piccolo [23], and Spark [27], since programs are expressed in terms of graphs and updates on vertices.

CGMGGraph [7] is an object-oriented framework providing MPI-based implementations of various parallel graph algorithms. Unlike our framework, its distribution mechanisms are exposed to the user. Moreover, our focus is on a re-usable infrastructure leveraging functional programming techniques. The Parallel Boost Graph Library [12] provides generic concepts for distributed graphs, with implementations based on MPI [10]. Distribution is supported using ghost cells that hold values associated with remote vertices and edges. The authors of Pregel note that ghost cells may not scale very well if many remote components are referenced. Like Pregel, our system uses explicit messaging to access remote information which does not have to be replicated. HIPG [16], is a framework for hierarchical parallel graph algorithms which was designed to automatically distribute computations. Our framework differs in that it is designed around an integration of a vertex-based BSP model and functional programming abstractions.

Orleans [3] is a high-level cloud platform for building client/cloud applications; it is related to our effort in that it uses an actor-based model to support communication and distribution.

7. Conclusions and Future Work

We have presented a new framework for parallel graph processing in Scala. Starting from a bulk-synchronous parallel model, we introduce high-level control structures to compose parallel, aggregating, and sequential supersteps. Practical experience suggests that our abstractions support real-world ML applications.

In future work we intend to experiment with other algorithms, such as Gibbs sampling and a possible extension to Deep Belief Networks. Supporting these and other classes of ML algorithms requires extending our framework with weighted graphs, and allowing dynamic changes in the graph topology. Extending our implementation for distribution is another area of future work. Some of the challenges that we foresee involve optimized aggregation and
routing based on the network topology, and programming support for fault handling.

References