Pregel: A System for Large-Scale Graph Processing

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn,
Naty Leiser, and Grzegorz Czajkowski
Google, Inc.
{malewicz,austern,ajcbik,dehnert,ilan,naty,gczaj}@google.com

ABSTRACT

Many practical computing problems concern large graphs. Standard examples include the Web graph and various social networks. The scale of these graphs—in some cases billions of vertices, trillions of edges—poses challenges to their efficient processing. In this paper we present a computational model suitable for this task. Programs are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology. This vertex-centric approach is flexible enough to express a broad set of algorithms. The model has been designed for efficient, scalable and fault-tolerant implementation on clusters of thousands of commodity computers, and its implied synchronicity makes reasoning about programs easier. Distribution-related details are hidden behind an abstract API. The result is a framework for processing large graphs that is expressive and easy to program.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*

General Terms

Design, Algorithms

Keywords

Distributed computing, graph algorithms

1. INTRODUCTION

The Internet made the Web graph a popular object of analysis and research. Web 2.0 fueled interest in social networks. Other large graphs—for example induced by transportation routes, similarity of newspaper articles, paths of disease outbreaks, or citation relationships among published scientific work—have been processed for decades. Frequently applied algorithms include shortest paths computations, different flavors of clustering, and variations on the page rank theme. There are many other graph computing problems of practical value, *e.g.*, minimum cut and connected components.

Efficient processing of large graphs is challenging. Graph algorithms often exhibit poor locality of memory access, very little work per vertex, and a changing degree of parallelism over the course of execution [31, 39]. Distribution over many machines exacerbates the locality issue, and increases the probability that a machine will fail during computation. Despite the ubiquity of large graphs and their commercial importance, we know of no scalable general-purpose system for implementing arbitrary graph algorithms over arbitrary graph representations in a large-scale distributed environment.

Implementing an algorithm to process a large graph typically means choosing among the following options:

1. Crafting a custom distributed infrastructure, typically requiring a substantial implementation effort that must be repeated for each new algorithm or graph representation.

2. Relying on an existing distributed computing platform, often ill-suited for graph processing. MapReduce [14], for example, is a very good fit for a wide array of large-scale computing problems. It is sometimes used to mine large graphs [11, 30], but this can lead to sub-optimal performance and usability issues. The basic models for processing data have been extended to facilitate aggregation [41] and SQL-like queries [40, 47], but these extensions are usually not ideal for graph algorithms that often better fit a message passing model.

3. Using a single-computer graph algorithm library, such as BGL [43], LEDA [35], NetworkX [25], JDSL [20], Stanford GraphBase [29], or FGL [16], limiting the scale of problems that can be addressed.

4. Using an existing parallel graph system. The Parallel BGL [22] and CGMgraph [8] libraries address parallel graph algorithms, but do not address fault tolerance or other issues that are important for very large scale distributed systems.

None of these alternatives fit our purposes. To address distributed processing of large scale graphs, we built a scalable

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0032-2/10/06 ...$10.00.

and fault-tolerant platform with an API that is sufficiently flexible to express arbitrary graph algorithms. This paper describes the resulting system, called Pregel[1], and reports our experience with it.

The high-level organization of Pregel programs is inspired by Valiant's Bulk Synchronous Parallel model [45]. Pregel computations consist of a sequence of iterations, called *supersteps*. During a superstep the framework invokes a user-defined function for each vertex, conceptually in parallel. The function specifies behavior at a single vertex V and a single superstep S. It can read messages sent to V in superstep $S-1$, send messages to other vertices that will be received at superstep $S+1$, and modify the state of V and its outgoing edges. Messages are typically sent along outgoing edges, but a message may be sent to any vertex whose identifier is known.

The vertex-centric approach is reminiscent of MapReduce in that users focus on a local action, processing each item independently, and the system composes these actions to lift computation to a large dataset. By design the model is well suited for distributed implementations: it doesn't expose any mechanism for detecting order of execution within a superstep, and all communication is from superstep S to superstep $S+1$.

The synchronicity of this model makes it easier to reason about program semantics when implementing algorithms, and ensures that Pregel programs are inherently free of deadlocks and data races common in asynchronous systems. In principle the performance of Pregel programs should be competitive with that of asynchronous systems given enough parallel slack [28, 34]. Because typical graph computations have many more vertices than machines, one should be able to balance the machine loads so that the synchronization between supersteps does not add excessive latency.

The rest of the paper is structured as follows. Section 2 describes the model. Section 3 describes its expression as a C++ API. Section 4 discusses implementation issues, including performance and fault tolerance. In Section 5 we present several applications of this model to graph algorithm problems, and in Section 6 we present performance results. Finally, we discuss related work and future directions.

2. MODEL OF COMPUTATION

The input to a Pregel computation is a directed graph in which each vertex is uniquely identified by a string *vertex identifier*. Each vertex is associated with a modifiable, user defined value. The directed edges are associated with their source vertices, and each edge consists of a modifiable, user defined value and a target vertex identifier.

A typical Pregel computation consists of input, when the graph is initialized, followed by a sequence of *supersteps* separated by global synchronization points until the algorithm terminates, and finishing with output.

Within each superstep the vertices compute in parallel, each executing the same user-defined function that expresses the logic of a given algorithm. A vertex can modify its state or that of its outgoing edges, receive messages sent to it in the previous superstep, send messages to other vertices (to be received in the next superstep), or even mutate the topology of the graph. Edges are not first-class citizens in this model, having no associated computation.

[1]The name honors Leonhard Euler. The Bridges of Königsberg, which inspired his famous theorem, spanned the Pregel river.

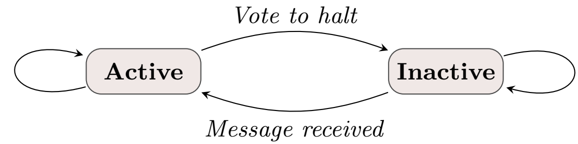

Figure 1: Vertex State Machine

Algorithm termination is based on every vertex *voting to halt*. In superstep 0, every vertex is in the *active* state; all active vertices participate in the computation of any given superstep. A vertex deactivates itself by voting to halt. This means that the vertex has no further work to do unless triggered externally, and the Pregel framework will not execute that vertex in subsequent supersteps unless it receives a message. If reactivated by a message, a vertex must explicitly deactivate itself again. The algorithm as a whole terminates when all vertices are simultaneously inactive and there are no messages in transit. This simple state machine is illustrated in Figure 1.

The output of a Pregel program is the set of values explicitly output by the vertices. It is often a directed graph isomorphic to the input, but this is not a necessary property of the system because vertices and edges can be added and removed during computation. A clustering algorithm, for example, might generate a small set of disconnected vertices selected from a large graph. A graph mining algorithm might simply output aggregated statistics mined from the graph.

Figure 2 illustrates these concepts using a simple example: given a strongly connected graph where each vertex contains a value, it propagates the largest value to every vertex. In each superstep, any vertex that has learned a larger value from its messages sends it to all its neighbors. When no further vertices change in a superstep, the algorithm terminates.

We chose a pure message passing model, omitting remote reads and other ways of emulating shared memory, for two reasons. First, message passing is sufficiently expressive that there is no need for remote reads. We have not found any graph algorithms for which message passing is insufficient. Second, this choice is better for performance. In a cluster environment, reading a value from a remote machine incurs high latency that can't easily be hidden. Our message passing model allows us to amortize latency by delivering messages asynchronously in batches.

Graph algorithms can be written as a series of chained MapReduce invocations [11, 30]. We chose a different model for reasons of usability and performance. Pregel keeps vertices and edges on the machine that performs computation, and uses network transfers only for messages. MapReduce, however, is essentially functional, so expressing a graph algorithm as a chained MapReduce requires passing the entire state of the graph from one stage to the next—in general requiring much more communication and associated serialization overhead. In addition, the need to coordinate the steps of a chained MapReduce adds programming complexity that is avoided by Pregel's iteration over supersteps.

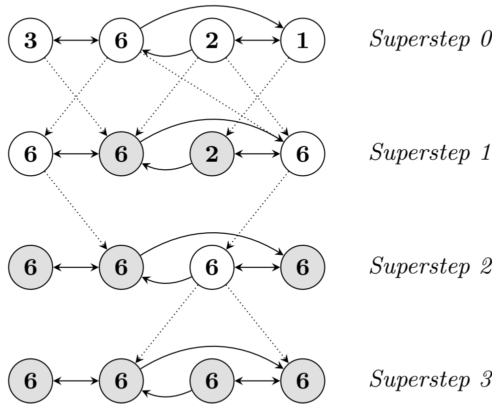

Figure 2: Maximum Value Example. Dotted lines are messages. Shaded vertices have voted to halt.

```
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
 public:
  virtual void Compute(MessageIterator* msgs) = 0;

  const string& vertex_id() const;
  int64 superstep() const;

  const VertexValue& GetValue();
  VertexValue* MutableValue();
  OutEdgeIterator GetOutEdgeIterator();

  void SendMessageTo(const string& dest_vertex,
                     const MessageValue& message);
  void VoteToHalt();
};
```

Figure 3: The Vertex API foundations.

3. THE C++ API

This section discusses the most important aspects of Pregel's C++ API, omitting relatively mechanical issues.

Writing a Pregel program involves subclassing the predefined `Vertex` class (see Figure 3). Its template arguments define three value types, associated with vertices, edges, and messages. Each vertex has an associated value of the specified type. This uniformity may seem restrictive, but users can manage it by using flexible types like protocol buffers [42]. The edge and message types behave similarly.

The user overrides the virtual `Compute()` method, which will be executed at each active vertex in every superstep. Predefined `Vertex` methods allow `Compute()` to query information about the current vertex and its edges, and to send messages to other vertices. `Compute()` can inspect the value associated with its vertex via `GetValue()` or modify it via `MutableValue()`. It can inspect and modify the values of out-edges using methods supplied by the out-edge iterator. These state updates are visible immediately. Since their visibility is confined to the modified vertex, there are no data races on concurrent value access from different vertices.

The values associated with the vertex and its edges are the only per-vertex state that persists across supersteps. Limiting the graph state managed by the framework to a single value per vertex or edge simplifies the main computation cycle, graph distribution, and failure recovery.

3.1 Message Passing

Vertices communicate directly with one another by sending messages, each of which consists of a message value and the name of the destination vertex. The type of the message value is specified by the user as a template parameter of the `Vertex` class.

A vertex can send any number of messages in a superstep. All messages sent to vertex V in superstep S are available, via an iterator, when V's `Compute()` method is called in superstep $S+1$. There is no guaranteed order of messages in the iterator, but it is guaranteed that messages will be delivered and that they will not be duplicated.

A common usage pattern is for a vertex V to iterate over its outgoing edges, sending a message to the destination vertex of each edge, as shown in the PageRank algorithm in Figure 4 (Section 5.1 below). However, `dest_vertex` need not be a neighbor of V. A vertex could learn the identifier of a non-neighbor from a message received earlier, or vertex identifiers could be known implicitly. For example, the graph could be a clique, with well-known vertex identifiers V_1 through V_n, in which case there may be no need to even keep explicit edges in the graph.

When the destination vertex of any message does not exist, we execute user-defined handlers. A handler could, for example, create the missing vertex or remove the dangling edge from its source vertex.

3.2 Combiners

Sending a message, especially to a vertex on another machine, incurs some overhead. This can be reduced in some cases with help from the user. For example, suppose that `Compute()` receives integer messages and that only the sum matters, as opposed to the individual values. In that case the system can combine several messages intended for a vertex V into a single message containing their sum, reducing the number of messages that must be transmitted and buffered.

Combiners are not enabled by default, because there is no mechanical way to find a useful combining function that is consistent with the semantics of the user's `Compute()` method. To enable this optimization the user subclasses the `Combiner` class, overriding a virtual `Combine()` method. There are no guarantees about which (if any) messages are combined, the groupings presented to the combiner, or the order of combining, so combiners should only be enabled for commutative and associative operations.

For some algorithms, such as single-source shortest paths (Section 5.2), we have observed more than a fourfold reduction in message traffic by using combiners.

3.3 Aggregators

Pregel *aggregators* are a mechanism for global communication, monitoring, and data. Each vertex can provide a value to an aggregator in superstep S, the system combines those values using a reduction operator, and the resulting value is made available to all vertices in superstep $S+1$. Pregel includes a number of predefined aggregators, such as `min`, `max`, or `sum` operations on various integer or string types.

Aggregators can be used for statistics. For instance, a `sum` aggregator applied to the out-degree of each vertex yields the

total number of edges in the graph. More complex reduction operators can generate histograms of a statistic.

Aggregators can also be used for global coordination. For instance, one branch of `Compute()` can be executed for the supersteps until an `and` aggregator determines that all vertices satisfy some condition, and then another branch can be executed until termination. A `min` or `max` aggregator, applied to the vertex ID, can be used to select a vertex to play a distinguished role in an algorithm.

To define a new aggregator, a user subclasses the predefined `Aggregator` class, and specifies how the aggregated value is initialized from the first input value and how multiple partially aggregated values are reduced to one. Aggregation operators should be commutative and associative.

By default an aggregator only reduces input values from a single superstep, but it is also possible to define a *sticky* aggregator that uses input values from all supersteps. This is useful, for example, for maintaining a global edge count that is adjusted only when edges are added or removed.

More advanced uses are possible. For example, an aggregator can be used to implement a distributed priority queue for the Δ-stepping shortest paths algorithm [37]. Each vertex is assigned to a priority bucket based on its tentative distance. In one superstep, the vertices contribute their indices to a `min` aggregator. The minimum is broadcast to all workers in the next superstep, and the vertices in the lowest-index bucket relax edges.

3.4 Topology Mutations

Some graph algorithms need to change the graph's topology. A clustering algorithm, for example, might replace each cluster with a single vertex, and a minimum spanning tree algorithm might remove all but the tree edges. Just as a user's `Compute()` function can send messages, it can also issue requests to add or remove vertices or edges.

Multiple vertices may issue conflicting requests in the same superstep (*e.g.*, two requests to add a vertex V, with different initial values). We use two mechanisms to achieve determinism: partial ordering and handlers.

As with messages, mutations become effective in the superstep after the requests were issued. Within that superstep removals are performed first, with edge removal before vertex removal, since removing a vertex implicitly removes all of its out-edges. Additions follow removals, with vertex addition before edge addition, and all mutations precede calls to `Compute()`. This partial ordering yields deterministic results for most conflicts.

The remaining conflicts are resolved by user-defined handlers. If there are multiple requests to create the same vertex in the same superstep, then by default the system just picks one arbitrarily, but users with special needs may specify a better conflict resolution policy by defining an appropriate handler method in their `Vertex` subclass. The same handler mechanism is used to resolve conflicts caused by multiple vertex removal requests, or by multiple edge addition or removal requests. We delegate the resolution to handlers to keep the code of `Compute()` simple, which limits the interaction between a handler and `Compute()`, but has not been an issue in practice.

Our coordination mechanism is lazy: global mutations do not require coordination until the point when they are applied. This design choice facilitates stream processing. The intuition is that conflicts involving modification of a vertex V are handled by V itself.

Pregel also supports purely local mutations, *i.e.*, a vertex adding or removing its own outgoing edges or removing itself. Local mutations cannot introduce conflicts and making them immediately effective simplifies distributed programming by using an easier sequential programming semantics.

3.5 Input and output

There are many possible file formats for graphs, such as a text file, a set of vertices in a relational database, or rows in Bigtable [9]. To avoid imposing a specific choice of file format, Pregel decouples the task of interpreting an input file as a graph from the task of graph computation. Similarly, output can be generated in an arbitrary format and stored in the form most suitable for a given application. The Pregel library provides readers and writers for many common file formats, but users with unusual needs can write their own by subclassing the abstract base classes `Reader` and `Writer`.

4. IMPLEMENTATION

Pregel was designed for the Google cluster architecture, which is described in detail in [3]. Each cluster consists of thousands of commodity PCs organized into racks with high intra-rack bandwidth. Clusters are interconnected but distributed geographically.

Our applications typically execute on a cluster management system that schedules jobs to optimize resource allocation, sometimes killing instances or moving them to different machines. The system includes a name service, so that instances can be referred to by logical names independent of their current binding to a physical machine. Persistent data is stored as files on a distributed storage system, GFS [19], or in Bigtable [9], and temporary data such as buffered messages on local disk.

4.1 Basic architecture

The Pregel library divides a graph into partitions, each consisting of a set of vertices and all of those vertices' outgoing edges. Assignment of a vertex to a partition depends solely on the vertex ID, which implies it is possible to know which partition a given vertex belongs to even if the vertex is owned by a different machine, or even if the vertex does not yet exist. The default partitioning function is just $hash(ID)$ mod N, where N is the number of partitions, but users can replace it.

The assignment of vertices to worker machines is the main place where distribution is not transparent in Pregel. Some applications work well with the default assignment, but some benefit from defining custom assignment functions to better exploit locality inherent in the graph. For example, a typical heuristic employed for the Web graph is to colocate vertices representing pages of the same site.

In the absence of faults, the execution of a Pregel program consists of several stages:

1. Many copies of the user program begin executing on a cluster of machines. One of these copies acts as the master. It is not assigned any portion of the graph, but is responsible for coordinating worker activity. The workers use the cluster management system's name service to discover the master's location, and send registration messages to the master.

2. The master determines how many partitions the graph will have, and assigns one or more partitions to each worker machine. The number may be controlled by the user. Having more than one partition per worker allows parallelism among the partitions and better load balancing, and will usually improve performance. Each worker is responsible for maintaining the state of its section of the graph, executing the user's `Compute()` method on its vertices, and managing messages to and from other workers. Each worker is given the complete set of assignments for all workers.

3. The master assigns a portion of the user's input to each worker. The input is treated as a set of records, each of which contains an arbitrary number of vertices and edges. The division of inputs is orthogonal to the partitioning of the graph itself, and is typically based on file boundaries. If a worker loads a vertex that belongs to that worker's section of the graph, the appropriate data structures (Section 4.3) are immediately updated. Otherwise the worker enqueues a message to the remote peer that owns the vertex. After the input has finished loading, all vertices are marked as active.

4. The master instructs each worker to perform a superstep. The worker loops through its active vertices, using one thread for each partition. The worker calls `Compute()` for each active vertex, delivering messages that were sent in the previous superstep. Messages are sent asynchronously, to enable overlapping of computation and communication and batching, but are delivered before the end of the superstep. When the worker is finished it responds to the master, telling the master how many vertices will be active in the next superstep.

   This step is repeated as long as any vertices are active, or any messages are in transit.

5. After the computation halts, the master may instruct each worker to save its portion of the graph.

4.2 Fault tolerance

Fault tolerance is achieved through checkpointing. At the beginning of a superstep, the master instructs the workers to save the state of their partitions to persistent storage, including vertex values, edge values, and incoming messages; the master separately saves the aggregator values.

Worker failures are detected using regular "ping" messages that the master issues to workers. If a worker does not receive a ping message after a specified interval, the worker process terminates. If the master does not hear back from a worker, the master marks that worker process as failed.

When one or more workers fail, the current state of the partitions assigned to these workers is lost. The master reassigns graph partitions to the currently available set of workers, and they all reload their partition state from the most recent available checkpoint at the beginning of a superstep S. That checkpoint may be several supersteps earlier than the latest superstep S' completed by any partition before the failure, requiring that recovery repeat the missing supersteps. We select checkpoint frequency based on a mean time to failure model [13], balancing checkpoint cost against expected recovery cost.

Confined recovery is under development to improve the cost and latency of recovery. In addition to the basic checkpoints, the workers also log outgoing messages from their assigned partitions during graph loading and supersteps. Recovery is then confined to the lost partitions, which are recovered from checkpoints. The system recomputes the missing supersteps up to S' using logged messages from healthy partitions and recalculated ones from recovering partitions.

This approach saves compute resources during recovery by only recomputing lost partitions, and can improve the latency of recovery since each worker may be recovering fewer partitions. Saving the outgoing messages adds overhead, but a typical machine has adequate disk bandwidth to ensure that I/O does not become the bottleneck.

Confined recovery requires the user algorithm to be deterministic, to avoid inconsistencies due to mixing saved messages from the original execution with new messages from the recovery. Randomized algorithms can be made deterministic by seeding a pseudorandom number generator deterministically based on the superstep and the partition. Nondeterministic algorithms can disable confined recovery and fall back to the basic recovery mechanism.

4.3 Worker implementation

A worker machine maintains the state of its portion of the graph in memory. Conceptually this can be thought of as a map from vertex ID to the state of each vertex, where the state of each vertex consists of its current value, a list of its outgoing edges (the vertex ID for the edge's target, and the edge's current value), a queue containing incoming messages, and a flag specifying whether the vertex is active. When the worker performs a superstep it loops through all vertices and calls `Compute()`, passing it the current value, an iterator to the incoming messages, and an iterator to the outgoing edges. There is no access to incoming edges because each incoming edge is part of a list owned by the source vertex, in general on a different machine.

For performance reasons, the active vertex flags are stored separately from the incoming message queues. Furthermore, while only a single copy of the vertex and edge values exists, two copies of the active vertex flags and the incoming message queue exist: one for the current superstep and one for the next superstep. While a worker processes its vertices in superstep S it is simultaneously, in another thread, receiving messages from other workers executing the same superstep. Since vertices receive messages that were sent in the previous superstep (see Section 2), messages for supersteps S and $S+1$ must be kept separate. Similarly, arrival of a message for a vertex V means that V will be active in the next superstep, not necessarily the current one.

When `Compute()` requests sending a message to another vertex, the worker process first determines whether the destination vertex is owned by a remote worker machine, or by the same worker that owns the sender. In the remote case the message is buffered for delivery to the destination worker. When the buffer sizes reach a threshold, the largest buffers are asynchronously flushed, delivering each to its destination worker as a single network message. In the local case an optimization is possible: the message is placed directly in the destination vertex's incoming message queue.

If the user has provided a Combiner (Section 3.2), it is applied when messages are added to the outgoing message queue and when they are received at the incoming message queue. The latter does not reduce network usage, but does reduce the space needed to store messages.

4.4 Master implementation

The master is primarily responsible for coordinating the activities of workers. Each worker is assigned a unique identifier at the time of its registration. The master maintains a list of all workers currently known to be alive, including the worker's unique identifier, its addressing information, and which portion of the graph it has been assigned. The size of the master's data structures is proportional to the number of partitions, not the number of vertices or edges, so a single master can coordinate computation for even a very large graph.

Most master operations, including input, output, computation, and saving and resuming from checkpoints, are terminated at *barriers*: the master sends the same request to every worker that was known to be alive at the time the operation begins, and waits for a response from every worker. If any worker fails, the master enters recovery mode as described in section 4.2. If the barrier synchronization succeeds, the master proceeds to the next stage. In the case of a computation barrier, for example, the master increments the global superstep index and proceeds to the next superstep.

The master also maintains statistics about the progress of computation and the state of the graph, such as the total size of the graph, a histogram of its distribution of out-degrees, the number of active vertices, the timing and message traffic of recent supersteps, and the values of all user-defined aggregators. To enable user monitoring, the master runs an HTTP server that displays this information.

4.5 Aggregators

An aggregator (Section 3.3) computes a single global value by applying an aggregation function to a set of values that the user supplies. Each worker maintains a collection of aggregator instances, identified by a type name and instance name. When a worker executes a superstep for any partition of the graph, the worker combines all of the values supplied to an aggregator instance into a single local value: an aggregator that is partially reduced over all of the worker's vertices in the partition. At the end of the superstep workers form a tree to reduce partially reduced aggregators into global values and deliver them to the master. We use a tree-based reduction—rather than pipelining with a chain of workers—to parallelize the use of CPU during reduction. The master sends the global values to all workers at the beginning of the next superstep.

5. APPLICATIONS

This section presents four examples that are simplified versions of algorithms developed by Pregel users to solve real problems: Page Rank, Shortest Paths, Bipartite Matching, and a Semi-Clustering algorithm.

5.1 PageRank

A Pregel implementation of a PageRank algorithm [7] is shown in Figure 4. The `PageRankVertex` class inherits from `Vertex`. Its vertex value type is `double` to store a tentative PageRank, and its message type is `double` to carry PageRank fractions, while the edge value type is `void` because edges do not store information. We assume that the graph is initialized so that in superstep 0, the value of each vertex is 1 / `NumVertices()`. In each of the first 30 supersteps, each vertex sends along each outgoing edge its tentative

```
class PageRankVertex
    : public Vertex<double, void, double> {
 public:
  virtual void Compute(MessageIterator* msgs) {
    if (superstep() >= 1) {
      double sum = 0;
      for (; !msgs->Done(); msgs->Next())
        sum += msgs->Value();
      *MutableValue() =
          0.15 / NumVertices() + 0.85 * sum;
    }

    if (superstep() < 30) {
      const int64 n = GetOutEdgeIterator().size();
      SendMessageToAllNeighbors(GetValue() / n);
    } else {
      VoteToHalt();
    }
  }
};
```

Figure 4: PageRank implemented in Pregel.

PageRank divided by the number of outgoing edges. Starting from superstep 1, each vertex sums up the values arriving on messages into `sum` and sets its own tentative PageRank to $0.15/$`NumVertices()` $+ 0.85 \times$ `sum`. After reaching superstep 30, no further messages are sent and each vertex votes to halt. In practice, a PageRank algorithm would run until convergence was achieved, and aggregators would be useful for detecting the convergence condition.

5.2 Shortest Paths

Shortest paths problems are among the best known problems in graph theory and arise in a wide variety of applications [10, 24], with several important variants. The *single-source* shortest paths problem requires finding a shortest path between a single source vertex and every other vertex in the graph. The *s*-*t* shortest path problem requires finding a single shortest path between given vertices s and t; it has obvious practical applications like driving directions and has received a great deal of attention. It is also relatively easy—solutions in typical graphs like road networks visit a tiny fraction of vertices, with Lumsdaine *et al* [31] observing visits to 80,000 vertices out of 32 million in one example. A third variant, all-pairs shortest paths, is impractical for large graphs because of its $O(|V|^2)$ storage requirements.

For simplicity and conciseness, we focus here on the single-source variant that fits Pregel's target of large-scale graphs very well, but offers more interesting scaling data than the *s*-*t* shortest path problem. An implementation is shown in Figure 5.

In this algorithm, we assume the value associated with each vertex is initialized to `INF` (a constant larger than any feasible distance in the graph from the source vertex). In each superstep, each vertex first receives, as messages from its neighbors, updated potential minimum distances from the source vertex. If the minimum of these updates is less than the value currently associated with the vertex, then this vertex updates its value and sends out potential updates to its neighbors, consisting of the weight of each outgoing edge added to the newly found minimum distance. In the first superstep, only the source vertex will update its value (from `INF` to zero) and send updates to its immediate neighbors. These neighbors in turn will update their values and send

```
class ShortestPathVertex
    : public Vertex<int, int, int> {
  void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    if (mindist < GetValue()) {
      *MutableValue() = mindist;
      OutEdgeIterator iter = GetOutEdgeIterator();
      for (; !iter.Done(); iter.Next())
        SendMessageTo(iter.Target(),
                      mindist + iter.GetValue());
    }
    VoteToHalt();
  }
};
```

Figure 5: Single-source shortest paths.

```
class MinIntCombiner : public Combiner<int> {
  virtual void Combine(MessageIterator* msgs) {
    int mindist = INF;
    for (; !msgs->Done(); msgs->Next())
      mindist = min(mindist, msgs->Value());
    Output("combined_source", mindist);
  }
};
```

Figure 6: Combiner that takes minimum of message values.

messages, resulting in a wavefront of updates through the graph. The algorithm terminates when no more updates occur, after which the value associated with each vertex denotes the minimum distance from the source vertex to that vertex. (The value `INF` denotes that the vertex cannot be reached at all.) Termination is guaranteed if all edge weights are non-negative.

Messages in this algorithm consist of potential shorter distances. Since the receiving vertex is ultimately only interested in the minimum, this algorithm is amenable to optimization using a combiner (Section 3.2). The combiner shown in Figure 6 greatly reduces the amount of data sent between workers, as well as the amount of data buffered prior to executing the next superstep. While the code in Figure 5 only computes distances, modifying it to compute the shortest paths tree as well is quite straightforward.

This algorithm may perform many more comparisons than sequential counterparts such as Dijkstra or Bellman-Ford [5, 15, 17, 24], but it is able to solve the shortest paths problem at a scale that is infeasible with any single-machine implementation. More advanced parallel algorithms exist, *e.g.*, Thorup [44] or the Δ-stepping method [37], and have been used as the basis for special-purpose parallel shortest paths implementations [12, 32]. Such advanced algorithms can also be expressed in the Pregel framework. The simplicity of the implementation in Figure 5, however, together with the already acceptable performance (see Section 6), may appeal to users who can't do extensive tuning or customization.

5.3 Bipartite Matching

The input to a bipartite matching algorithm consists of two distinct sets of vertices with edges only between the sets, and the output is a subset of edges with no common endpoints. A maximal matching is one to which no additional edge can be added without sharing an endpoint. We implemented a randomized maximal matching algorithm [1] and a maximum-weight bipartite matching algorithm [4]; we describe the former here.

In the Pregel implementation of this algorithm the vertex value is a tuple of two values: a flag indicating which set the vertex is in (L or R), and the name of its matched vertex once known. The edge value has type `void` (edges carry no information), and the messages are boolean. The algorithm proceeds in cycles of four phases, where the phase index is just the superstep index *mod* 4, using a three-way handshake.

In phase 0 of a cycle, each left vertex not yet matched sends a message to each of its neighbors to request a match, and then unconditionally votes to halt. If it sent no messages (because it is already matched, or has no outgoing edges), or if all the message recipients are already matched, it will never be reactivated. Otherwise, it will receive a response in two supersteps and reactivate.

In phase 1 of a cycle, each right vertex not yet matched randomly chooses one of the messages it receives, sends a message granting that request, and sends messages to other requestors denying it. Then it unconditionally votes to halt.

In phase 2 of a cycle, each left vertex not yet matched chooses one of the grants it receives and sends an acceptance message. Left vertices that are already matched will never execute this phase, since they will not have sent a message in phase 0.

Finally, in phase 3, an unmatched right vertex receives at most one acceptance message. It notes the matched node and unconditionally votes to halt—it has nothing further to do.

5.4 Semi-Clustering

Pregel has been used for several different versions of clustering. One version, *semi-clustering*, arises in social graphs.

Vertices in a social graph typically represent people, and edges represent connections between them. Edges may be based on explicit actions (*e.g.*, adding a friend in a social networking site), or may be inferred from people's behavior (*e.g.*, email conversations or co-publication). Edges may have weights, to represent the interactions' frequency or strength.

A semi-cluster in a social graph is a group of people who interact frequently with each other and less frequently with others. What distinguishes it from ordinary clustering is that a vertex may belong to more than one semi-cluster.

This section describes a parallel greedy semi-clustering algorithm. Its input is a weighted, undirected graph (represented in Pregel by constructing each edge twice, once in each direction) and its output is at most $C_{\mathtt{max}}$ semi-clusters, each containing at most $V_{\mathtt{max}}$ vertices, where $C_{\mathtt{max}}$ and $V_{\mathtt{max}}$ are user-specified parameters.

A semi-cluster c is assigned a score,

$$S_c = \frac{I_c - f_B B_c}{V_c(V_c - 1)/2}, \quad (1)$$

where I_c is the sum of the weights of all internal edges, B_c is the sum of the weights of all boundary edges (*i.e.*, edges connecting a vertex in the semi-cluster to one outside it), V_c is the number of vertices in the semi-cluster, and f_B, the boundary edge score factor, is a user-specified parameter, usually between 0 and 1. The score is normalized, *i.e.*, di-

vided by the number of edges in a clique of size V_c, so that large clusters do not receive artificially high scores.

Each vertex V maintains a list containing at most $C_{\mathtt{max}}$ semi-clusters, sorted by score. In superstep 0 V enters itself in that list as a semi-cluster of size 1 and score 1, and publishes itself to all of its neighbors. In subsequent supersteps:

- Vertex V iterates over the semi-clusters c_1,...,c_k sent to it on the previous superstep. If a semi-cluster c does not already contain V, and $V_c < M_{\mathtt{max}}$, then V is added to c to form c'.
- The semi-clusters $c_1, ..., c_k, c'_1, ..., c'_k$ are sorted by their scores, and the best ones are sent to V's neighbors.
- Vertex V updates its list of semi-clusters with the semi-clusters from $c_1, ..., c_k, c'_1, ..., c'_k$ that contain V.

The algorithm terminates either when the semi-clusters stop changing or (to improve performance) when the number of supersteps reaches a user-specified limit. At that point the list of best semi-cluster candidates for each vertex may be aggregated into a global list of best semi-clusters.

6. EXPERIMENTS

We conducted various experiments with the single-source shortest paths (SSSP) implementation of Section 5.2 on a cluster of 300 multicore commodity PCs. We report runtimes for binary trees (to study scaling properties) and log-normal random graphs (to study the performance in a more realistic setting) using various graph sizes with the weights of all edges implicitly set to 1.

The time for initializing the cluster, generating the test graphs in-memory, and verifying results is not included in the measurements. Since all experiments could run in a relatively short time, failure probability was low, and checkpointing was disabled.

As an indication of how Pregel scales with worker tasks, Figure 7 shows shortest paths runtimes for a binary tree with a billion vertices (and, thus, a billion minus one edges) when the number of Pregel workers varies from 50 to 800. The drop from 174 to 17.3 seconds using 16 times as many workers represents a speedup of about 10.

To show how Pregel scales with graph size, Figure 8 presents shortest paths runtimes for binary trees varying in size from a billion to 50 billion vertices, now using a fixed number

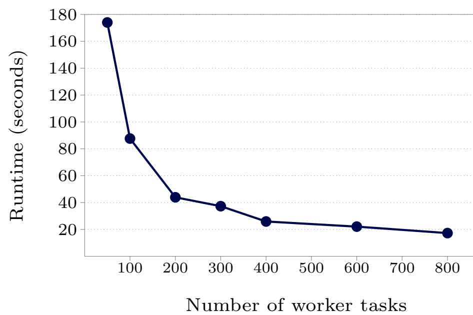

Figure 7: SSSP—1 billion vertex binary tree: varying number of worker tasks scheduled on 300 multicore machines

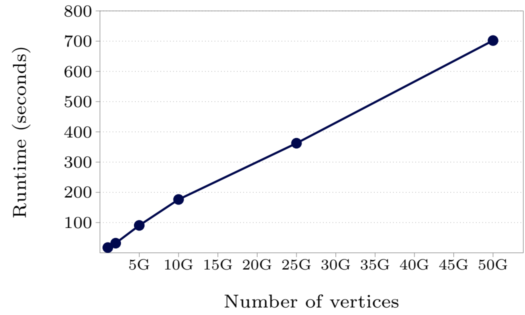

Figure 8: SSSP—binary trees: varying graph sizes on 800 worker tasks scheduled on 300 multicore machines

of 800 worker tasks scheduled on 300 multicore machines. Here the increase from 17.3 to 702 seconds demonstrates that for graphs with a low average outdegree the runtime increases linearly in the graph size.

Although the previous experiments give an indication of how Pregel scales in workers and graph size, binary trees are obviously not representative of graphs encountered in practice. Therefore, we also conducted experiments with random graphs that use a log-normal distribution of outdegrees,

$$p(d) = \frac{1}{\sqrt{2\pi}\,\sigma d}\, e^{-(\ln d - \mu)^2/2\sigma^2} \tag{2}$$

with $\mu = 4$ and $\sigma = 1.3$, for which the mean outdegree is 127.1. Such a distribution resembles many real-world large-scale graphs, such as the web graph or social networks, where most vertices have a relatively small degree but some outliers are much larger—a hundred thousand or more. Figure 9 shows shortest paths runtimes for such graphs varying in size from 10 million to a billion vertices (and thus over 127 billion edges), again with 800 worker tasks scheduled on 300 multicore machines. Running shortest paths for the largest graph took a little over 10 minutes.

In all experiments the graph was partitioned among workers using the default partitioning function based on a random hash; a topology-aware partitioning function would give better performance. Also, a naïve parallel shortest paths

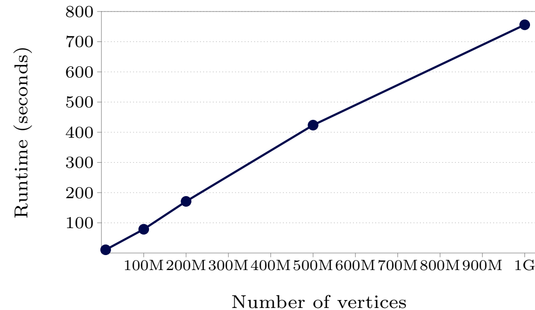

Figure 9: SSSP—log-normal random graphs, mean out-degree 127.1 (thus over 127 billion edges in the largest case): varying graph sizes on 800 worker tasks scheduled on 300 multicore machines

algorithm was used; here too a more advanced algorithm would perform better. Therefore, the results of the experiments in this section should not be interpreted as the best possible runtime of shortest paths using Pregel. Instead, the results are meant to show that satisfactory performance can be obtained with relatively little coding effort. In fact, our results for one billion vertices and edges are comparable to the Δ-stepping results from Parallel BGL [31] mentioned in the next section for a cluster of 112 processors on a graph of 256 million vertices and one billion edges, and Pregel scales better beyond that size.

7. RELATED WORK

Pregel is a distributed programming framework, focused on providing users with a natural API for programming graph algorithms while managing the details of distribution invisibly, including messaging and fault tolerance. It is similar in concept to MapReduce [14], but with a natural graph API and much more efficient support for iterative computations over the graph. This graph focus also distinguishes it from other frameworks that hide distribution details such as Sawzall [41], Pig Latin [40], and Dryad [27, 47]. Pregel is also different because it implements a stateful model where long-lived processes compute, communicate, and modify local state, rather than a dataflow model where any process computes solely on input data and produces output data input by other processes.

Pregel was inspired by the Bulk Synchronous Parallel model [45], which provides its synchronous superstep model of computation and communication. There have been a number of general BSP library implementations, for example the Oxford BSP Library [38], Green BSP library [21], BSPlib [26] and Paderborn University BSP library [6]. They vary in the set of communication primitives provided, and in how they deal with distribution issues such as reliability (machine failure), load balancing, and synchronization. To our knowledge, the scalability and fault-tolerance of BSP implementations has not been evaluated beyond several dozen machines, and none of them provides a graph-specific API.

The closest matches to Pregel are the Parallel Boost Graph Library and CGMgraph. The Parallel BGL [22, 23] specifies several key generic concepts for defining distributed graphs, provides implementations based on MPI [18], and implements a number of algorithms based on them. It attempts to maintain compatibility with the (sequential) BGL [43] to facilitate porting algorithms. It implements *property maps* to hold information associated with vertices and edges in the graph, using *ghost cells* to hold values associated with remote components. This can lead to scaling problems if reference to many remote components is required. Pregel uses an explicit message approach to acquiring remote information and does not replicate remote values locally. The most critical difference is that Pregel provides fault-tolerance to cope with failures during computation, allowing it to function in a huge cluster environment where failures are common, *e.g.*, due to hardware failures or preemption by higher-priority jobs.

CGMgraph [8] is similar in concept, providing a number of parallel graph algorithms using the Coarse Grained Multicomputer (CGM) model based on MPI. Its underlying distribution mechanisms are much more exposed to the user, and the focus is on providing implementations of algorithms rather than an infrastructure to be used to implement them. CGMgraph uses an object-oriented programming style, in contrast to the generic programming style of Parallel BGL and Pregel, at some performance cost.

Other than Pregel and Parallel BGL, there have been few systems reporting experimental results for graphs at the scale of billions of vertices. The largest have reported results from custom implementations of *s-t* shortest path, rather than from general frameworks. Yoo *et al* [46] report on a BlueGene/L implementation of breadth-first search (*s-t* shortest path) on 32,768 PowerPC processors with a high-performance torus network, achieving 1.5 seconds for a Poisson distributed random graph with 3.2 billion vertices and 32 billion edges. Bader and Madduri [2] report on a Cray MTA-2 implementation of a similar problem on a 10 node, highly multithreaded system, achieving .43 seconds for a scale-free R-MAT random graph with 134 million vertices and 805 million edges. Lumsdaine *et al* [31] compare a Parallel BGL result on a x86-64 Opteron cluster of 200 processors to the BlueGene/L implementation, achieving .43 seconds for an Erdős-Renyi random graph of 4 billion vertices and 20 billion edges. They attribute the better performance to ghost cells, and observe that their implementation begins to get worse performance above 32 processors.

Results for the single-source shortest paths problem on an Erdős-Renyi random graph with 256 million vertices and uniform out-degree 4, using the Δ-stepping algorithm, are reported for the Cray MTA-2 (40 processors, 2.37 sec, [32]), and for Parallel BGL on Opterons (112 processors, 35 sec., [31]). The latter time is similar to our 400-worker result for a binary tree with 1 billion nodes and edges. We do not know of any reported SSSP results on the scale of our 1 billion vertex and 127.1 billion edge log-normal graph.

Another line of research has tackled use of external disk memory to handle huge problems with single machines, *e.g.*, [33, 36], but these implementations require hours for graphs of a billion vertices.

8. CONCLUSIONS AND FUTURE WORK

The contribution of this paper is a model suitable for large-scale graph computing and a description of its production quality, scalable, fault-tolerant implementation.

Based on the input from our users we think we have succeeded in making this model useful and usable. Dozens of Pregel applications have been deployed, and many more are being designed, implemented, and tuned. The users report that once they switch to the "think like a vertex" mode of programming, the API is intuitive, flexible, and easy to use. This is not surprising, since we have worked with early adopters who influenced the API from the outset. For example, aggregators were added to remove limitations users found in the early Pregel model. Other usability aspects of Pregel motivated by user experience include a set of status pages with detailed information about the progress of Pregel programs, a unittesting framework, and a single-machine mode which helps with rapid prototyping and debugging.

The performance, scalability, and fault-tolerance of Pregel are already satisfactory for graphs with billions of vertices. We are investigating techniques for scaling to even larger graphs, such as relaxing the synchronicity of the model to avoid the cost of faster workers having to wait frequently at inter-superstep barriers.

Currently the entire computation state resides in RAM. We already spill some data to local disk, and will continue in

this direction to enable computations on large graphs when terabytes of main memory are not available.

Assigning vertices to machines to minimize inter-machine communication is a challenge. Partitioning of the input graph based on topology may suffice if the topology corresponds to the message traffic, but it may not. We would like to devise dynamic re-partitioning mechanisms.

Pregel is designed for sparse graphs where communication occurs mainly over edges, and we do not expect that focus to change. Although care has been taken to support high fan-out and fan-in traffic, performance will suffer when most vertices continuously send messages to most other vertices. However, realistic dense graphs are rare, as are algorithms with dense communication over a sparse graph. Some such algorithms can be transformed into more Pregel-friendly variants, for example by using combiners, aggregators, or topology mutations, and of course such computations are difficult for any highly distributed system.

A practical concern is that Pregel is becoming a piece of production infrastructure for our user base. We are no longer at liberty to change the API without considering compatibility. However, we believe that the programming interface we have designed is sufficiently abstract and flexible to be resilient to the further evolution of the underlying system.

9. ACKNOWLEDGMENTS

We thank Pregel's very early users—Lorenz Huelsbergen, Galina Shubina, Zoltan Gyongyi—for their contributions to the model. Discussions with Adnan Aziz, Yossi Matias, and Steffen Meschkat helped refine several aspects of Pregel. Our interns, Punyashloka Biswal and Petar Maymounkov, provided initial evidence of Pregel's applicability to matchings and clustering, and Charles Reiss automated checkpointing decisions. The paper benefited from comments on its earlier drafts from Jeff Dean, Tushar Chandra, Luiz Barroso, Urs Hölzle, Robert Henry, Marián Dvorský, and the anonymous reviewers. Sierra Michels-Slettvet advertised Pregel to various teams within Google computing over interesting but less known graphs. Finally, we thank all the users of Pregel for feedback and many great ideas.

10. REFERENCES

[1] Thomas Anderson, Susan Owicki, James Saxe, and Charles Thacker, *High-Speed Switch Scheduling for Local-Area Networks*. ACM Trans. Comp. Syst. 11(4), 1993, 319–352.

[2] David A. Bader and Kamesh Madduri, *Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2*, in Proc. 35th Intl. Conf. on Parallel Processing (ICPP'06), Columbus, OH, August 2006, 523—530.

[3] Luiz Barroso, Jeffrey Dean, and Urs Hoelzle, *Web search for a planet: The Google Cluster Architecture*. IEEE Micro 23(2), 2003, 22–28.

[4] Mohsen Bayati, Devavrat Shah, and Mayank Sharma, *Maximum Weight Matching via Max-Product Belief Propagation*. in Proc. IEEE Intl. Symp. on Information Theory, 2005, 1763–1767.

[5] Richard Bellman, *On a routing problem*. Quarterly of Applied Mathematics 16(1), 1958, 87–90.

[6] Olaf Bonorden, Ben H.H. Juurlink, Ingo von Otte, and Ingo Rieping, *The Paderborn University BSP (PUB) Library*. Parallel Computing 29(2), 2003, 187–207.

[7] Sergey Brin and Lawrence Page, *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. in Proc. 7th Intl. Conf. on the World Wide Web, 1998, 107–117.

[8] Albert Chan and Frank Dehne, *CGMGRAPH/CGMLIB: Implementing and Testing CGM Graph Algorithms on PC Clusters and Shared Memory Machines*. Intl. J. of High Performance Computing Applications 19(1), 2005, 81–97.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*. ACM Trans. Comp. Syst. 26(2), Art. 4, 2008.

[10] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik, *Shortest paths algorithms: Theory and experimental evaluation*. Mathematical Programming 73, 1996, 129–174.

[11] Jonathan Cohen, *Graph Twiddling in a MapReduce World*. Comp. in Science & Engineering, July/August 2009, 29–41.

[12] Joseph R. Crobak, Jonathan W. Berry, Kamesh Madduri, and David A. Bader, *Advanced Shortest Paths Algorithms on a Massively-Multithreaded Architecture*. in Proc. First Workshop on Multithreaded Architectures and Applications, 2007, 1–8.

[13] John T. Daly, *A higher order estimate of the optimum checkpoint interval for restart dumps*. Future Generation Computer Systems 22, 2006, 303–312.

[14] Jeffrey Dean and Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*. in Proc. 6th USENIX Symp. on Operating Syst. Design and Impl., 2004, 137–150.

[15] Edsger W. Dijkstra, *A Note on Two Problems in Connexion with Graphs*. Numerische Mathematik 1, 1959, 269–271.

[16] Martin Erwig, *Inductive Graphs and Functional Graph Algorithms*. J. Functional Programming 1(5), 2001, 467–492.

[17] Lester R. Ford, L. R. and Delbert R. Fulkerson, *Flows in Networks*. Princeton University Press, 1962.

[18] Ian Foster and Carl Kesselman (Eds), *The Grid 2: Blueprint for a New Computing Infrastructure (2nd edition)*. Morgan Kaufmann, 2003.

[19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, *The Google File System*. in Proc. 19th ACM Symp. on Operating Syst. Principles, 2003, 29–43.

[20] Michael T. Goodrich and Roberto Tamassia, *Data Structures and Algorithms in JAVA. (second edition)*. John Wiley and Sons, Inc., 2001.

[21] Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, and Thanasis Tsantilas, *Portable and Efficient Parallel Computing Using the BSP Model*. IEEE Trans. Comp. 48(7), 1999, 670–689.

[22] Douglas Gregor and Andrew Lumsdaine, *The Parallel BGL: A Generic Library for Distributed Graph Computations*. Proc. of Parallel Object-Oriented Scientific Computing (POOSC), July 2005.

[23] Douglas Gregor and Andrew Lumsdaine, *Lifting Sequential Graph Algorithms for Distributed-Memory Parallel Computation*. in Proc. 2005 ACM SIGPLAN Conf. on Object-Oriented Prog., Syst., Lang., and Applications (OOPSLA'05), October 2005, 423–437.

[24] Jonathan L. Gross and Jay Yellen, *Graph Theory and Its Applications. (2nd Edition)*. Chapman and Hall/CRC, 2005.

[25] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart, *Exploring network structure, dynamics, and function using NetworkX*. in Proc. 7th Python in Science Conf., 2008, 11–15.

[26] Jonathan Hill, Bill McColl, Dan Stefanescu, Mark Goudreau, Kevin Lang, Satish Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling, *BSPlib: The BSP Programming Library*. Parallel Computing 24, 1998, 1947–1980.

[27] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly, *Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks*. in Proc. European Conf. on Computer Syst., 2007, 59–72.

[28] Paris C. Kanellakis and Alexander A. Shvartsman, *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers, 1997.

[29] Donald E. Knuth, *Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, 1994.

[30] U Kung, Charalampos E. Tsourakakis, and Christos Faloutsos, *Pegasus: A Peta-Scale Graph Mining System - Implementation and Observations*. Proc. Intl. Conf. Data Mining, 2009, 229-238.

[31] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry, *Challenges in Parallel Graph Processing*. Parallel Processing Letters 17, 2007, 5–20.

[32] Kamesh Madduri, David A. Bader, Jonathan W. Berry, and Joseph R. Crobak, *Parallel Shortest Path Algorithms for Solving Large-Scale Graph Instances*. DIMACS Implementation Challenge – The Shortest Path Problem, 2006.

[33] Kamesh Madduri, David Ediger, Karl Jiang, David A. Bader, and Daniel Chavarria-Miranda, *A Faster Parallel Algorithm and Efficient Multithreaded Implementation for Evaluating Betweenness Centrality on Massive Datasets*, in Proc. 3rd Workshop on Multithreaded Architectures and Applications (MTAAP'09), Rome, Italy, May 2009.

[34] Grzegorz Malewicz, *A Work-Optimal Deterministic Algorithm for the Certified Write-All Problem with a Nontrivial Number of Asynchronous Processors*. SIAM J. Comput. 34(4), 2005, 993–1024.

[35] Kurt Mehlhorn and Stefan Näher, *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[36] Ulrich Meyer and Vitaly Osipov, *Design and Implementation of a Practical I/O-efficient Shortest Paths Algorithm*. in Proc. 3rd Workshop on Multithreaded Architectures and Applications (MTAAP'09), Rome, Italy, May 2009.

[37] Ulrich Meyer and Peter Sanders, *Δ-stepping: A Parallelizable Shortest Path Algorithm*. J. Algorithms 49(1), 2003, 114–152.

[38] Richard Miller, *A Library for Bulk-Synchronous Parallel Programming*. in Proc. British Computer Society Parallel Processing Specialist Group Workshop on General Purpose Parallel Computing, 1993.

[39] Kameshwar Munagala and Abhiram Ranade, *I/O-complexity of graph algorithms*. in Proc. 10th Annual ACM-SIAM Symp. on Discrete Algorithms, 1999, 687–694.

[40] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins, *Pig Latin: A Not-So-Foreign Language for Data Processing*. in Proc. ACM SIGMOD Intl. Conf. on Management of Data, 2008, 1099–1110.

[41] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan, *Interpreting the Data: Parallel Analysis with Sawzall*. Scientific Programming Journal 13(4), Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure, 2005, 227–298.

[42] *Protocol Buffers—Google's data interchange format*. `http://code.google.com/p/protobuf/` 2009.

[43] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison Wesley, 2002.

[44] Mikkel Thorup, *Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time*. J. ACM 46(3), May 1999, 362–394.

[45] Leslie G. Valiant, *A Bridging Model for Parallel Computation*. Comm. ACM 33(8), 1990, 103–111.

[46] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and Umit Catalyurek, *A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L*, in Proc. 2005 ACM/IEEE Conf. on Supercomputing (SC'05), 2005, 25—43.

[47] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey, *DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language*. in Proc. 8th USENIX Symp. on Operating Syst. Design and Implementation, 2008, 10–14.