Processing large-scale graph data: A guide to current technology

Learn about Apache Giraph, GraphLab, and other open source systems for processing graph-structured big data

Sherif Sakr (ssakr@cse.unsw.edu.au)
Senior Research Scientist
National ICT Australia

10 June 2013

With emphasis on Apache Giraph and the GraphLab framework, this article introduces and compares open source solutions for processing large volumes of graph data. The growth of graph-structured data in modern applications such as social networks and knowledge bases creates a crucial need for scalable platforms and parallel architectures that can process it in bulk. Despite its prominent role in big data analytics, MapReduce is not the optimal programming model for graph processing. This article explains why and then explores systems in development to tackle the graph-processing challenge.

In computer science and mathematics, *graphs* are abstract data structures that model structural relationships among objects. They are now widely used for data modeling in application domains for which identifying relationship patterns, rules, and anomalies is useful. These domains include the web graph, social networks, the Semantic Web, knowledge bases, protein-protein interaction networks, and bibliographical networks, among many others. The ever-increasing size of graph-structured data for these applications creates a critical need for scalable systems that can process large amounts of it efficiently.

Glossary

Adjacency list: In a graph data structure, the representation of a collection of unordered lists, one for each vertex in the graph. Each list describes the set of neighbors of its vertex.

Diameter: The largest number of vertices that must be traversed to travel from one vertex to another when paths that backtrack, detour, or loop are excluded from consideration.

Edge: The connection between any two nodes (vertices) in a graph.

Natural graph: A graph that has many nodes with few connections, and few nodes with many connections.

Node: One of a graph's objects. (Synonymous with vertex.)

PageRank: A link-analysis algorithm that is used by the Google web search engine. The algorithm assigns a numerical weight to each element of a hyperlinked set of documents of the web graph, with the purpose of measuring its relative importance within the set.

Shortest-path algorithm: The process of finding a path (connection) between two nodes (vertices) in a graph such that the number of its constituent edges is minimized.

Vertex (plural *vertices*): One of a graph's objects. (Synonymous with node.)

Vertex degree: The number of edges incident to a vertex.

Web graph: Graph that represents the pages of the World Wide Web and the direct links between them.

The *web graph* is a dramatic example of a large-scale graph. Google estimates that the total number of web pages exceeds 1 trillion; experimental graphs of the World Wide Web contain more than 20 billion nodes (pages) and 160 billion edges (hyperlinks). Graphs of social networks are another example. Facebook reportedly consists of more than a billion users (nodes) and more than 140 billion friendship relationships (edges) in 2012. The LinkedIn network contains almost 8 million nodes and 60 million edges. (Social network graphs are growing rapidly. Facebook went from roughly 1 million users in 2004 to 1 billion in 2012.) In the Semantic Web context, the ontology of DBpedia (derived from Wikipedia), currently contains 3.7 million objects (nodes) and 400 millions facts (edges).

Several *graph database* systems — most notably, Neo4j — support online transaction processing workloads on graph data (see Resources). But Neo4j relies on data access methods for graphs without considering data locality, and the processing of graphs entails mostly random data access. For large graphs that cannot be stored in memory, random disk access becomes a performance bottleneck. Furthermore, Neo4j is a centralized system that lacks the computational power of a distributed, parallel system. Large-scale graphs must be partitioned over multiple machines to achieve scalable processing.

With Google's MapReduce framework, commodity computer clusters can be programmed to perform large-scale data processing in a single pass. Unlike Neo4j, MapReduce is not designed to support online query processing. MapReduce is optimized for analytics on large data volumes partitioned over hundreds of machines. Apache Hadoop, an open source distributed-processing framework for large data sets that includes a MapReduce implementation, is popular in industry and academia by virtue of its simplicity and scalability.

However, Hadoop and its associated technologies (such as Pig and Hive) were not designed mainly to support scalable processing of graph-structured data. Some proposals to adapt the MapReduce framework (or Hadoop) for this purpose were made and this article starts by looking at two of them. The most robust available technologies for large-scale graph processing are based on programming models other than MapReduce. The remainder of the article describes and compares two such systems in depth:

- Giraph, a distributed and fault-tolerant system that adopts the Bulk Synchronous Parallel programming model to run parallel algorithms for processing large-scale graph data
- GraphLab, a graph-based, high-performance, distributed computation framework that is written in C++

At the conclusion of the article, I also briefly describe some other open source projects for graph data processing. I assume that readers of this article are familiar with graph concepts and terminology. For any who might not be, I include a glossary of terms.

MapReduce and large-scale graph processing

Surfer and GBASE

Surfer is an experimental large-scale graph-processing engine that provides two primitives for programmers: MapReduce and *propagation*. MapReduce processes multiple key/value pairs in parallel. Propagation is an iterative computational pattern that transfers information along the edges from a vertex to its neighbours in the graph. MapReduce is suitable for processing flat data structures (such as vertex-oriented tasks), while propagation is optimized for edge-oriented tasks on partitioned graphs. Surfer resolves network-traffic bottlenecks with graph partitioning adapted to the characteristics of the Hadoop distributed environment. It applies a bandwidth-aware mechanism that adapts the graph-partitioning process's varying bandwidth requirements to uneven network bandwidth.

Another proposed MapReduce extension, GBASE, uses a graph storage method that is called *block compression* to store homogeneous regions of graphs efficiently. Then, it compresses all nonempty blocks through a standard compression mechanism such as GZip. Finally, it stores the compressed blocks together with some meta information into a graph database.

The key feature of GBASE is that it unifies node-based and edge-based queries as query vectors and unifies different operations types on the graph through matrix-vector multiplication on the adjacency and incidence matrices. This feature enables GBASE to support multiple types of graph queries. The queries are classified into *global* queries that require traversal of the whole graph and *targeted* queries that usually must access only parts of the graph. Before GBASE runs the matrix-vector multiplication, it selects the grids that contain the blocks that are relevant to the input queries. Therefore, only the files corresponding to the grids are fed into Hadoop jobs that GBASE runs.

In the MapReduce programming model, the Map function takes key/value pairs as input and produces a set of intermediate key/value pairs. The framework groups all intermediate values that are associated with the same intermediate key and passes them to the Reduce function. The Reduce function receives an intermediate key with its set of values and merges them together.

On the implementation level, the intermediate key/value pairs are buffered in memory. Periodically, the buffered pairs are written to local disk and partitioned into regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the designated *master* program instance, which is responsible for forwarding the locations to the reduce workers. When a reduce worker is notified of the locations, it reads the buffered data from the local disks of the map workers. The buffered data is then sorted by the intermediate keys so that all occurrences of the same key are grouped. The reduce worker passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to a final output file for this reduce partition.

MapReduce isolates the application developer from the details of running a distributed program, such as issues of data distribution, scheduling, and fault tolerance. From the graph-processing point of view, the basic MapReduce programming model is inadequate because most graph algorithms are iterative and traverse the graph in some way. Hence, the efficiency of graph computations depends heavily on interprocessor bandwidth as graph structures are sent over the network iteration after iteration. For example, the basic MapReduce programming model does not directly support iterative data-analysis applications. To implement iterative programs, programmers might manually issue multiple MapReduce jobs and orchestrate their execution with a driver program. In practice, the manual orchestration of an iterative program in MapReduce has two key problems:

- Even though much data might be unchanged from iteration to iteration, the data must be reloaded and reprocessed at each iteration, wasting I/O, network bandwidth, and processor resources.
- The termination condition might involve the detection of when a fix point is reached. The condition itself might require an extra MapReduce job on each iteration, again increasing resource use in terms of scheduling extra tasks, reading extra data from disk, and moving data across the network.

Surfer and GBASE are examples of extensions for MapReduce that are proposed to help it process graphs more efficiently. (For a technical summary, see Surfer and GBASE. In Resources, find links to the full papers that propose these extensions.) These two proposals promise only limited success:

- Compared with Hadoop's user-defined functions, Surfer's propagation-based implementation is more programmable and more efficient when the access pattern of the target application matches that of propagation — mainly, in edge-oriented tasks. When the access pattern of the tasks does not match propagation — in vertex-oriented tasks, for example — implementation of the target application with propagation is tricky.
- GBASE is unlikely to be intuitive for most developers, who might find it challenging to think of graph processing in terms of matrices. Also, each iteration is scheduled as a separate Hadoop job with increased workload: When the graph structure is read from disk, the map output is spilled to disk and the intermediary result is written to the Hadoop Distributed File System (HDFS).

Thus, a crucial need remains for distributed systems that can effectively support scalable processing of large-scale graph data on clusters of horizontally scalable commodity machines. The Giraph and GraphLab projects both propose to fill this gap.

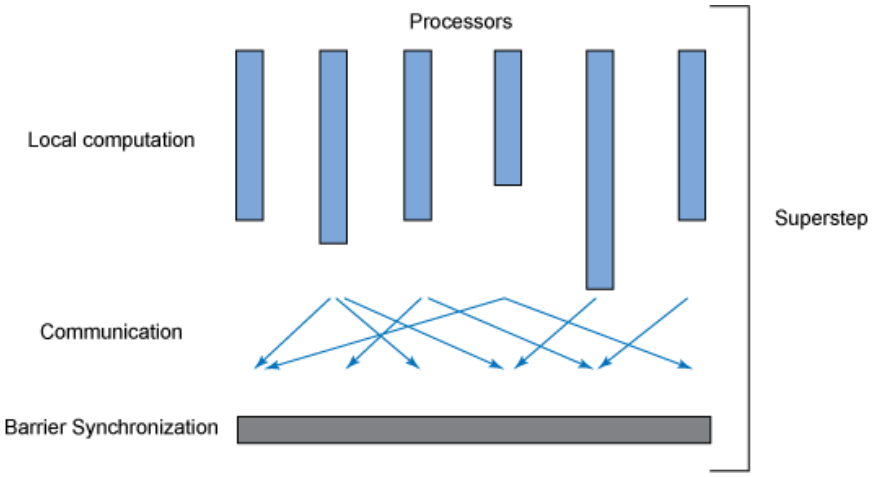
Giraph

In 2010, Google introduced the Pregel system as a scalable platform for implementing graph algorithms (see Resources). Pregel relies on a vertex-centric approach and is inspired by the Bulk Synchronous Parallel (BSP) model (see Resources). In 2012, Apache Giraph launched as an open source project that clones the concepts of Pregel. Giraph can run as a typical Hadoop job that uses the Hadoop cluster infrastructure.

How it works

In Giraph, graph-processing programs are expressed as a sequence of iterations called *supersteps*. During a superstep, the framework starts a user-defined function for each vertex, conceptually in parallel. The user-defined function specifies the behaviour at a single vertex *V* and a single superstep *S*. The function can read messages that are sent to *V* in superstep *S-1*, send messages to other vertices that are received at superstep *S+1*, and modify the state of *V* and its outgoing edges. Messages are typically sent along outgoing edges, but you can send a message to any vertex with a known identifier. Each superstep represents atomic units of parallel computation. Figure 1 illustrates the execution mechanism of the BSP programming model:

Figure 1. BSP programming model



In this programming model, all vertices are assigned an active status at superstep 1 of the executed program. All active vertices run the `compute()` user function at each superstep.

Each vertex can deactivate itself by voting to halt and turn to inactive state at any superstep if it does receive a message. A vertex can return to the active status if it receives a message in the execution of any subsequent superstep. This process, illustrated in Figure 2, continues until all vertices have no messages to send, and become inactive. Hence, program execution ends when at one stage all vertices are inactive.

Figure 2. Vertex voting

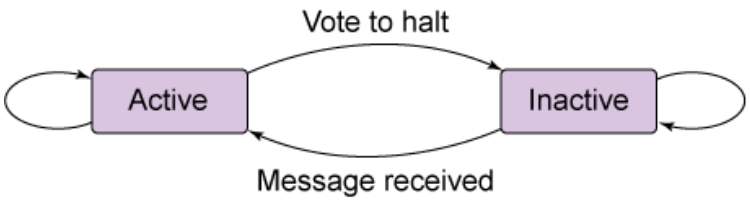
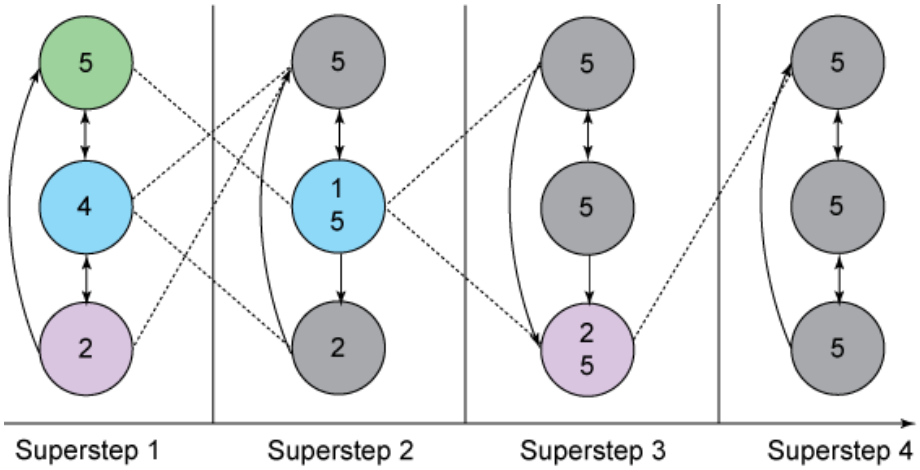


Figure 3 illustrates an example for the communicated messages between a set of graph vertices for computing the maximum vertex value:

Figure 3. BSP example of computing the maximum vertex value

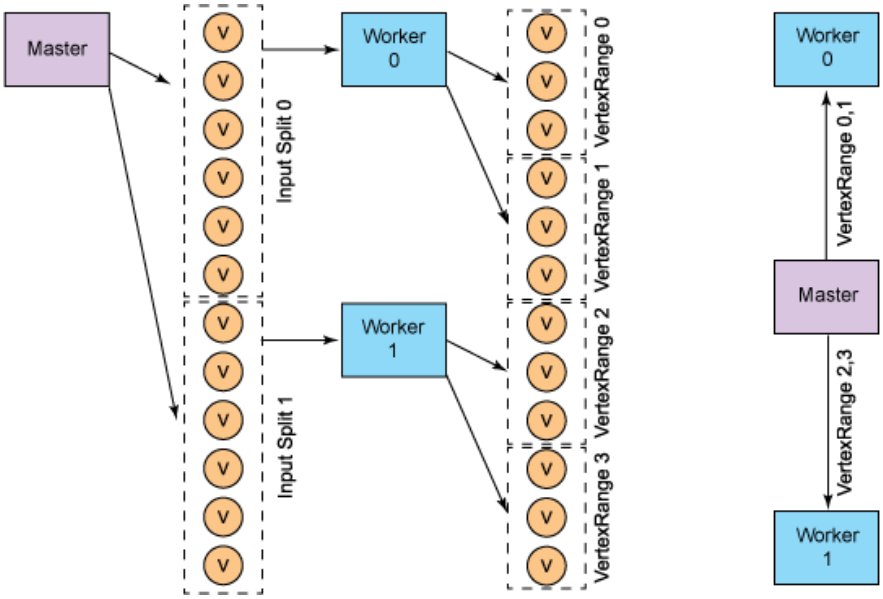


In Superstep 1 of Figure 3, each vertex sends its value to its neighbour vertex. In Superstep 2, each vertex compares its value with the received value from its neighbour vertex. If the received value is higher than the vertex value, then it updates its value with the higher value and sends the new value to its neighbour vertex. If the received value is lower than the vertex value, then the vertex keeps its current value and votes to halt. Hence, in Superstep 2, only the vertex with value 1 updates its value to higher received value (5) and sends its new value. This process happens again in Superstep 3 for the vertex with the value 2, while in Superstep 4 all vertices vote to halt and the program ends.

Like the Hadoop framework, Giraph is an efficient, scalable, and fault-tolerant implementation on clusters of thousands of commodity computers, with the distribution-related details hidden behind an abstraction. On a machine that performs computation, it keeps vertices and edges in memory and uses network transfers only for messages. The model is well-suited for distributed implementations because it doesn't show any mechanism for detecting the order of execution within a superstep, and all communication is from superstep *S* to superstep *S+1*. During program execution, graph vertices are partitioned and assigned to workers. The default partition mechanism is hash-partitioning, but custom partition is also supported.

Giraph applies a master/worker architecture, illustrated in Figure 4:

Figure 4. Giraph's master/worker execution steps



The master node assigns partitions to workers, coordinates synchronization, requests checkpoints, and collects health statuses. Like Hadoop, Giraph uses Apache ZooKeeper for synchronization. Workers are responsible for vertices. A worker starts the `compute()` function for the active vertices. It also sends, receives, and assigns messages with other vertices. During execution, if a worker receives input that is not for its vertices, it passes it along.

Giraph in action

To implement a Giraph program, design your algorithm as a `Vertex`. Each `Vertex` can be an instance one of the many existing classes such as `BasicVertex`, `MutableVertex`, `EdgeListVertex`, `HashMapVertex`, and `LongDoubleFloatDoubleVertex`. You must define a `VertexInputFormat` for reading your graph. (For example, to read from a text file with adjacency lists, the format might look like `(vertex, neighbor1, neighbor2)`. You need also to define a `VertexOutputFormat` to write back the result (for example, `vertex, pageRank`).

The Java code in Listing 1 is an example of using the `compute()` function for implementing the PageRank algorithm:

Listing 1. PageRank algorithm in Giraph

```
public class SimplePageRankVertex extends Vertex<LongWritable, DoubleWritable,
                          FloatWritable, DoubleWritable> {
    public void compute(Iterator<DoubleWritable> msgIterator) {
        if (getSuperstep() >= 1) {
            double sum = 0;
            while (msgIterator.hasNext()) {
                sum += msgIterator.next().get();
            }
            setVertexValue(new DoubleWritable((0.15f / getNumVertices()) + 0.85f * sum);
        }
        if (getSuperstep() < 30) {

            long edges = getOutEdgeIterator().size();
            sentMsgToAllEdges(new DoubleWritable(getVertexValue().get() / edges));
        } else {
            voteToHalt();
        }
    }
```

Listing 2 shows an example of using the `compute()` function to implement the shortest-path algorithm:

Listing 2. Shortest path in Giraph

```
public static class ShortestPathVertex extends Vertex<Text, IntWritable, IntWritable> {
     public void compute(Iterator<IntWritable> messages) throws IOException {
      int minDist = isStartVertex() ? 0 : Integer.MAX_VALUE;
      while (messages.hasNext()) {
         IntWritable msg = messages.next();
         if (msg.get() < minDist) {
            minDist = msg.get();
         }
      }
      if (minDist < this.getValue().get()) {
         this.setValue(new IntWritable(minDist));
         for (Edge<Text, IntWritable> e : this.getEdges()) {
            sendMessage(e, new IntWritable(minDist + e.getValue().get()));
         }
      } else {
         voteToHalt();
      }
   }
}
```

GraphLab

GraphLab is a graph-based and distributed computation framework written in C++. The project started in 2009 at Carnegie Mellon University. GraphLab provides a parallel-programming abstraction that is targeted for sparse iterative graph algorithms through a high-level programming interface. Each GraphLab process is multithreaded to use fully the multicore resources available on modern cluster nodes. GraphLab supports reading from and writing to both Posix and HDFS file systems.

How it works

GraphLab is an asynchronous distributed shared-memory abstraction in which graph vertices share access to a distributed graph with data stored on every vertex and edge. In this

programming abstraction, each vertex can directly access information on the current vertex, adjacent edges, and adjacent vertices — irrespective of edge direction.

The GraphLab abstraction consists of three main parts: the *data graph*, the *update function*, and the *sync operation*. The data graph represents a user-modifiable program state that both stores the mutable user-defined data and encodes the sparse computational dependencies. The update function represents the user computation and operates on the data graph by transforming data in small overlapping contexts called *scopes*.

On the runtime, the GraphLab execution model enables efficient distributed execution by relaxing the execution-ordering requirements of the shared memory and allowing the GraphLab runtime engine to determine the best order to run vertices in. For example, one function might choose to return vertices in an order that minimizes network communication or latency. The only requirement that is imposed by the GraphLab abstraction is that all vertices be run eventually. By eliminating messages, GraphLab isolates the user-defined algorithm from the movement of data, allowing the system to choose when and how to move program state. By allowing mutable data to be associated with both vertices and edges, GraphLab enables the algorithm designer to distinguish more precisely between data that is shared with all neighbors (vertex data) and data that is shared with a particular neighbor (edge data). The GraphLab abstraction also implicitly defines the communication aspects of the gather and scatter phases (see GraphLab in action later in the article) by ensuring that changes made to the vertex or edge data are automatically visible to adjacent vertices. It is also important to note that GraphLab does not differentiate between edge directions.

Generally, the behaviour of the asynchronous execution depends on the number of machines and availability of network resources, leading to nondeterminism that can complicate algorithm design and debugging. In practice, the sequential model of the GraphLab abstraction is translated automatically into parallel execution by allowing multiple processors to run the same loop on the same graph, removing and running different vertices simultaneously. To retain the sequential execution semantics, GraphLab must ensure that overlapping computation is not run simultaneously. To address this challenge, GraphLab automatically enforces serializability so that every parallel execution of vertex-oriented programs has a corresponding sequential execution. To achieve serializability, GraphLab prevents adjacent vertex programs from running concurrently by using a fine-grained locking protocol that requires sequentially grabbing locks on all neighbouring vertices. Furthermore, the locking scheme that is used by GraphLab is unfair to high-degree vertices.

GraphLab in action

Think of a GraphLab program as a small program that runs on a vertex in the graph and has three execution phases:

1. A *gather* phase in which the `gather()` function in the vertex class is called on each edge on the vertex's adjacent edges, returning a value with each gather.
2. An *apply* phase in which the values returned by the gathers are summed together and given to the `apply()` function in the vertex class.

3. A *scatter* phase in which the `scatter()` function in the vertex class is once again called on each edge on the vertex's adjacent edges.

The C++ code in Listing 3 is an example of implementing the PageRank algorithm with GraphLab:

Listing 3. PageRank algorithm in GraphLab

```
class pagerank_program :
            public graphlab::ivertex_program<graph_type, double>,
            public graphlab::IS_POD_TYPE {
public:
  // we are going to gather on all the in-edges
  edge_dir_type gather_edges(icontext_type& context,
                             const vertex_type& vertex) const {
    return graphlab::IN_EDGES;
  }
  // for each in-edge, gather the weighted sum of the edge.
  double gather(icontext_type& context, const vertex_type& vertex,
                edge_type& edge) const {
    return edge.source().data().pagerank / edge.source().num_out_edges();
  }

  // Use the total rank of adjacent pages to update this page
  void apply(icontext_type& context, vertex_type& vertex,
             const gather_type& total) {
    double newval = total * 0.85 + 0.15;
    vertex.data().pagerank = newval;
  }

  // No scatter needed. Return NO_EDGES
  edge_dir_type scatter_edges(icontext_type& context,
                              const vertex_type& vertex) const {
    return graphlab::NO_EDGES;
  }
};
```

Giraph and GraphLab compared

Both Pregel and GraphLab apply the GAS — gather, apply, scatter — model that represents three conceptual phases of a vertex-oriented program. However, they differ in how they collect and disseminate information. In particular, Pregel and GraphLab express GAS programs in different ways. In the Pregel abstraction, the gather phase is implemented by using message combiners, and the apply and scatter phases are expressed in the vertex class. Conversely, GraphLab exposes the entire neighborhood to the vertex-oriented program and allows users to define the gather and apply phases within their programs. The GraphLab abstraction implicitly defines the communication aspects of the gather and scatter phases by ensuring that changes made to the vertex or edge data are automatically visible to adjacent vertices.

Another difference between Pregel and GraphLab is in how dynamic computation is expressed. GraphLab decouples the scheduling of future computation from the movement of data. For example, GraphLab update functions have access to data on adjacent vertices even if the adjacent vertices did not schedule the current update. In contrast, Pregel update functions are initiated by messages and can only access the data in the message, limiting what can be expressed.

Both Pregel and GraphLab depend on graph partitioning to minimize communication and ensure work balance. However, in the case of natural graphs both are forced to resort to hash-based

(random) partitioning, which can have poor locality. While Pregel and GraphLab are considered among the main harbingers of the new wave of large-scale graph-processing systems, both systems leave room for performance improvements. Serious efforts to evaluate and compare their strengths and weaknesses in different application domains of large graph data sets have not started yet.

Conclusion

Giraph and GraphLab provide new models for implementing big data analytics over graph data. They are likely to continue to attract a considerable amount of interest in the ecosystem of big data processing. Meanwhile, Pregel concepts were cloned by several other open source projects that you also might want to explore (see Resources for links).

- Apache Hama — like Giraph, designed to run on top of the Hadoop infrastructure — focuses on general BSP computations, so it is not only for graphs. (For example, it includes algorithms for matrix inversion and linear algebra.) Hama released version 0.6.1 in April 2013.
- GoldenOrb, a younger project (at version 0.1.1 as of this writing), provides Pregel's API but requires the deployment of more software to your existing Hadoop infrastructure.
- Signal/Collect, applies a similar Pregel programming model approach but independently of the Hadoop infrastructure.

In the non-Pregel arena, PEGASUS is a large-scale graph-mining library that is implemented on top of Hadoop. PEGASUS supports typical graph-mining tasks such as computing the diameter of the graph, computing the radius of each node, and finding the connected components through a generalization of matrix-vector multiplication.

Resources

Learn

- Graph (mathematics): Read about graph data structures at Wikipedia.
- Apache Giraph: Visit the Giraph project site.
- GraphLab: Visit the GraphLab project site.
- Hadoop on developerWorks: Explore a wealth of articles and other resources on Apache Hadoop and its related technologies.
- "On the Efficiency and Programmability of Large Graph Processing in the Cloud" (Rishan Chen et al., 2010): Learn more about the Surfer system.
- "GBASE: A Scalable and General Graph Management System" (U Kang et al., 2012): Learn more about GBASE.
- "Pregel: A System for Large-Scale Graph Processing" (Grzegorz Malewicz et al., 2010): Read the seminal Google paper on Pregel.
- Bulk synchronous parallel: Check out Wikipedia's article on BSP.
- Neo4j: Learn more about this open source NoSQL graph database in developerWorks podcast interviews with project co-founder Emil Eifrem (April 2012) and agile architect Peter Bell (April 2012).
- Apache Hama: Visit the Hama project website.
- GoldenOrb: Visit the GoldenOrb project website.
- Signal/Collect: Visit the Signal/Collect project website.
- PEGASUS: Visit the PEGAGUS project website.
- The Open source area on developerWorks provides a wealth of information about open source tools and how to use open source technologies.
- The Open source technical library: Find more open source articles.
- developerWorks technical events and webcasts: Stay current with technology in these sessions.
- developerWorks on Twitter: Join today to follow developerWorks tweets.
- developerWorks podcasts: Listen to interesting interviews and discussions for software developers.
- developerWorks on-demand demos: Watch demos ranging from product installation and setup for beginners to advanced functionality for experienced developers.

Get products and technologies

- Giraph: Download Giraph from an Apache mirror.
- GraphLab: Download GraphLab.
- Access IBM trial software (available for download or on DVD) and innovate in your next open source development project with software especially for developers.
- Download IBM developer kits: Update your system and get the latest tools and technologies here.

Discuss

- The developerWorks community: Connect with other developerWorks users as you explore the developer-driven blogs, forums, groups, and wikis.

About the author

Sherif Sakr

Dr. Sherif Sakr is a senior research scientist in the Software Systems Group at National ICT Australia (NICTA), Sydney, Australia. He is also a conjoint senior lecturer in the School of Computer Science and Engineering at University of New South Wales. He received his doctorate in computer science from Konstanz University, Germany, in 2007. His bachelor's and master's degrees in computer science are from Cairo University, Egypt. In 2011, Dr. Sakr held a visiting research scientist position in the eXtreme Computing Group (XCG) at Microsoft Research, in Redmond, Wash. In 2012, he held a research MTS position in Alcatel-Lucent Bell Labs.