Chapter 1

Apache Giraph

Contributed by Neil Butcher

1.1 Introduction

Giraph is a project developed by Apache to utilize their Hadoop MapReduce software to perform graph processing on large graphs. [4] Giraph is currently being used by Facebook to maintain users and their connections. Facebook has reported being able to process 1 trillion edges in 4 minutes, while using 200 machines [2]. Giraph is an adapted and adjusted version of Googles graph processing system, Pregel. [5] Giraph, unlike Pregel is open source and is build off of Hadoop and so it can be easily installed into Hadoop clusters. Giraph programs are written from the perspective of a single vertex and can only pass messages to other vertices, because of this the model of programming is referred to as 'think-like-a-vertex'. Giraph is a framework to process graphs with specific goals in mind. [7]

The main goal of Giraph is to produce a robust graph processing software that scales from a single hardware node to thousands of nodes. In order to do this properly Giraph adapts what is commonly referred to as a Bulk synchronous parallel (BSP) model. [3] This concept characterizes program execution as a series of supersteps, supersteps occur sequentially. Each superstep can be defined in three steps: concurrent computation, communication, barrier synchronization. Concurrent computation allows each processor to perform some local operations, however there can be no communication required to complete the computation in this superstep. Next is communication, any information that is required by other threads in the next superstep is transmitted here. Finally there is barrier synchronization, this happens after all of the communication and computation has been completed by the threads, at this point the messages are transferred between threads and the next superstep begins. Using this BSP model often results in applications that scale efficiently [3]. Often graph problems consist of enormous data sets that can not even fit on a single node or even a cluster of nodes. Giraph fixs this problem by minimizing the memory needed for a each node to perform the computation. This makes reducing the amount of storage used by each node essential to scaling up to large problems [4].

An important goal of Giraph is to ensure even on large clusters that there will be no single point of failure. In other words, if any node breaks and can not complete its task, the system will be able to restore and redistribute that nodes work to complete properly. This problem is solved through a technique often referred to as checkpointing. Checkpointing is essentially saving the state of the computation at some point onto multiple disks that the program can revert back to in the case of a failure. This will clearly solve the problem but also can be incredibly time consuming to store

all of that information onto disk. The larger the system the more likely the usage of checkpointing will become required as there is more likely to be a failure.

There is also a goal of Giraph in which it greatly simplifies writing graph programs that can take advantages of the Hadoop MapReduce framework. It is common for developers to want to develop codes for MapReduce because it provides the advantages stated earlier, however often representing graphs in Hadoop is both difficult and inefficient. This has raised heavy demand for a graph based version of Hadoop and thus Giraph was made. [4]

1.2 Computations in Giraph

Giraph closely follows the BSP model discussed in the introduction section. [4] [3] This means computation occurs as a series of asynchronous supersteps. Giraph forces the programmer to write computations from the perspective of a vertex. Each vertex is only aware of their own neighbors (as opposed to being aware of the entire graph), this reduces the amount of information to be stored on each processor. For each vertex a user defined 'compute' portion is executed, each vertex can send any number of messages to neighboring vertices throughout computation. The messages sent however will not be received until the following superstep begins, this keeps the compute stages independent within a superstep. The next superstep can only being if all the computation in the previous superstep has been completed. [3]

From a user perspective graphs are magically distributed by the inner workings of Giraph and how a input graph can be read can be defined in a user function, [1] or there are default functions that can read standard graph input formats. This is a complex system that is built upon the Hadoop framework but takes ideas to improve performance from Pregel [5]. This makes expressing graphs simple and adaptable to different problems. Giraph can be easily configured to work on different types of graphs, Graphs can also be expressed with edge weights and allows multiple types of weights. This adaptability of the graph model is often desirable when working on real applications. Often real applications will not perfectly fit into the typical graph model so having an adaptable graph format is desirable.

Giraph is written entirely in Java, the high level language gives adaptability and consistency to the language. There is a user defined 'start' function this function has the goal of 'starting' the program its much like a 'main' function in a typical program. Compute is a user defined function that is the most essential component of any Giraph program. The compute function happens on every vertex in each superstep. It takes as parameter a list of messages, these messages were sent in the previous superstep to the vertex. There are two key functions in Giraph that are needed in order to properly write a compute function: voteToHalt and sendMessage. The function sendMessage takes as a parameter the message and the vertex number of the vertex the message to be delivered to. The purpose of sendMessage is to allow vertices to communicate between supersteps. The function 'voteToHalt' takes no parameters and exists as a mechanism for a vertex to signal that it has finished computing. Given these components almost any graph problem can be solved using Giraph. There are more complex functions that exist for performance reasons.

A simple algorithm for Single Source Shortest Path (SSSP) can be seen in figure 1.1. The code is taken from [1]. An illustration of the execution of this code can be found in 1.2 [1]. The definition of SSSP is given a graph with weighted edges and a source vertex, find the shortest path from the start vertex to all vertices in the graph. We assume there exists a start function that sends a '0' to the first vertex. This makes sense, because the shortest path from the start node to the start node is of course zero. The pseudo code of the compute function starts by checking if a message has been received, if one has not, then halt and wait for the next superstep. If messages have been

received, find the smallest value in all of the messages. Finally, go through all neighbors, send them a message that is the vertex shortest distance plus the distance to that neighbor. Now referring back to the figure 1.2 shows a typical execution of this code. The start node starts by setting its shortest path length to 0 and sending a message to its neighbor with the distance one since that is the edge weight. In the next superstep, the second node now has received the message from the first, and checks its neighbors and sends the edge weights plus one (since that vertex is one away from source). This is an example of a simple yet clever algorithm that only requires vertices computing independently and then passing messages. [1]

1.3 Conclusion

Apache Giraph is a robust highly scalable graph algorithm framework that uses the think like a vertex model of programming. Giraph is based off the Bulk Synchronous Parallel model [3] that ensures simple execution and efficient scaling to large numbers of processors. They also designed Giraph to stand through single point of failures through checkpointing. The programming of the language is done in Java which results in verbose but concise code. The base API of Giraph is simple and built off of a few elementary functions. This simplicity makes developing codes straightforward if the developer has a effecitive 'think-like-a-vertex' algorithm. Giraph can be adapted easily to a wide variety of graph based problems and I have shown a simple SSSP algorithm as well as explained the flow of program execution while it runs. There also exist algorithms for a variety of different kernels that use the 'think-like-a-vertex' model of programming i.e. Page Rank [5] Regular Path Queries [6] Neural Modeling [8]. Giraph is widely used and scalable framework that can be used to compute on large datasets.

```
public void compute(Iterable<DoubleWritable> messages) {
  double minDist = Double.MAX_VALUE;
  for (DoubleWritable message : messages) {
    minDist = Math.min(minDist, message.get());
  }
  if (minDist < getValue().get()) {
    setValue(new DoubleWritable(minDist));
    for (Edge<LongWritable, FloatWritable> edge : getEdges()) {
      double distance = minDist + edge.getValue().get();
      sendMessage(edge.getTargetVertexId(), new DoubleWritable(distance));
    }
  }
  voteToHalt();
}
```

Figure 1.1: Example Giraph Code to compute Single Source Shortest Path

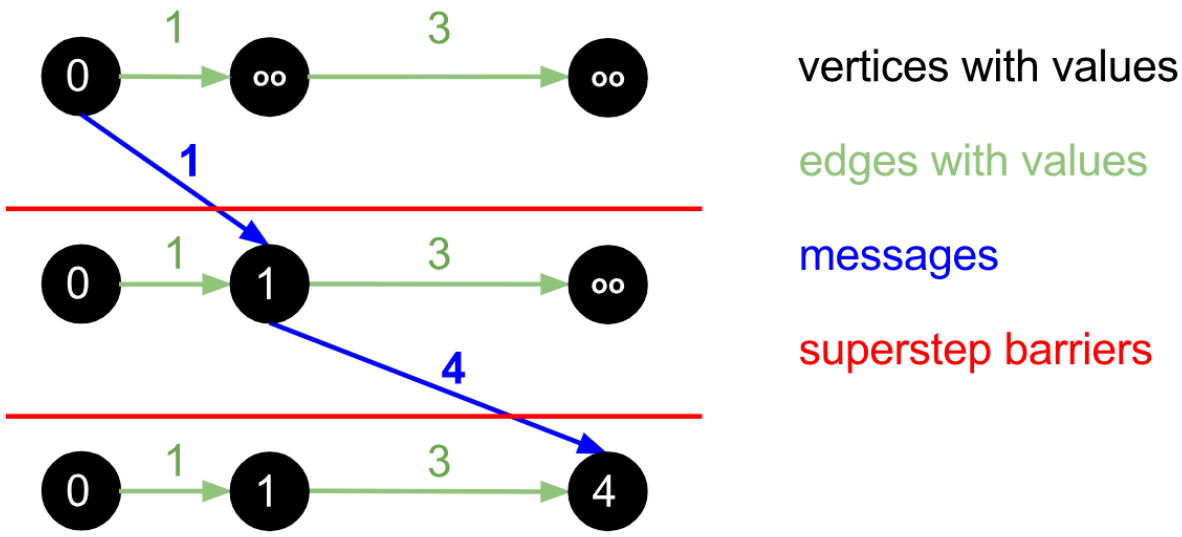

Figure 1.2: Example figure to compute Single Source Shortest Path

Bibliography

[1] Introduction to apache giraph. http://giraph.apache.org/intro.html. Accessed: 2018-11-20.

[2] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[3] Alexandros V Gerbessiotis and Leslie G Valiant. Direct bulk-synchronous parallel algorithms. *Journal of parallel and distributed computing*, 22(2):251–267, 1994.

[4] Minyang Han and Khuzaima Daudjee. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

[5] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[6] Maurizio Nolé and Carlo Sartiani. Processing regular path queries on giraph. In *EDBT/ICDT Workshops*, volume 14, page 3, 2014.

[7] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[8] Shuo Yang, Nicholas D Spielman, Jadin C Jackson, and Brad S Rubin. Large-scale neural modeling in mapreduce and giraph. In *Electro/Information Technology (EIT), 2014 IEEE International Conference on*, pages 556–561. IEEE, 2014.