An Experimental Evaluation of Giraph and Graphchi

by

Junnan Lu
B.Sc., University of Victoria, 2012

A Master's Project Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

An Experimental Evaluation of Giraph and Graphchi

by

Junnan Lu
B.Sc., University of Victoria, 2012

Supervisory Committee

Dr. Alex Thomo, Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Alex Thomo, Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Departmental Member
(Department of Computer Science)

ABSTRACT

Graphs are the ultimate data structure to capture and represent the data of different connected entities. Graphs have become a very practical tool to model complicated relationships in various application domains, such as social media, protein, transportation, bibliographical, or knowledge networks. With the growth of popularity of cloud computing, graphs with millions of nodes and billion edges are becoming more common. Graph analytics is a critical component of big data discovery. The major problem in processing large graph data is the size and the irregular structure of the graph. In this report, we evaluate a Pregel implementation, Apache Giraph, on several algorithms. Also, we compare our results with a disk-based (centralized) system, GraphChi. We observe that for a moderate number of very simple machines, Giraph outperforms GraphChi for all the algorithms and datasets tested. This is in contrast to the claim of the GaphChi authors that one needs a cluster of more than 1,000 computers to perform comparably to GraphChi.

Table of Contents

Contents

List of Figures

Chapter 1

1.0 Introduction

Graphs are very popular in analyzing data. Many interesting graph applications such as transportation routes, newspaper article similarity, disease outbreak and scientific work citations have been processed for decades [1]. Graph algorithms that have been applied frequently include shortest paths computations, weakly-connected-components, connection-counting, and Pagerank. The main problem when applying graph analytics is the typically very big size of the available graphs. Taking the web graph, for example, the estimation from Google shows that the population of web pages is now exceeding 1 trillion. The graph of the World Wide Web contains more than 20 billion nodes (web pages) and over 160 billion edges (hyperlinks). Social networks are other examples of very big graphs. In 2012, the number of users (as nodes) of Facebook exceeded over a billion with over 190 billion of friendship relationships, which can be viewed as links. Another example, LinkedIn, has more than 8 million users with more than 60 million of relationships [2].

Due to the irregular internal structure and the large scale of the data, processing such graphs is considered computationally hard in the traditional iterative way. The size of the graph based data is ever-growing. Determining the best paradigm for computing systems in order to handle and process such data is a hot research area in the data analytics communities. There are three challenges caused by the big size of data are facing the data scientists in this area [3]. First off, the data is dynamic. The analysis results are under frequent updates and retrieval. Secondly, due to the nature of the

quantity of the data, the data storage often is managed in distributed fashion. A petabyte of data is too large to be stored in a single computer. And finally, the computing environment should be fault tolerant.

The Bulk Synchronous Parallel (BSP) model, proposed by Valiant in 1989 provides a bridge to program and model graph data. The BSP model consists of a computer architecture, a class of algorithms, and a function for analyzing the costs to algorithms [4]. In the BSP model, a collection of computers or computer processors, connected with communication network channels, can send requests and messages to other processors for updating the processor state.

Google's Pregel is a simple yet a popular graph processing tool which is inspired by the BSP model. Similar to BSP, Pregel consists of a sequence of iterations, called supersteps [5]. In each superstep in Pregel, a user-defined-function, *compute,* will be invoked for each single vertex of the graph in order to conduct computation in parallel. The user defined function, such as reading a message from the previous superstep or sending a message to be read in the next superstep, defines the logic and behavior of each single vertex in each of the iterations or supersteps. Such a model is also labeled as *vertex centric* where the computation task is running independently and locally.

Apache Giraph is an open source implementation of proprietary Pregel. All the essential functionality for processing a graph in parallel like in Pregel is implemented in Apache Giraph. Systems like Apache Giraph and Pregel require distributed computing cluster to process large scale graph data quickly and effectively. Although the distributed computing facilities such as cloud computing clusters are becoming more common and accessible, the design and implementation of distributed algorithms are still challenging.

How to process large scale graph data effectively without distributed commodity computing clusters is an interesting question for a data analyst who may need to analyze a large graph dataset but is unable to access a distributed computing clusters. GraphChi proposed by Aapo Kyrola is a disk based system which segments a large graph into different partitions. Then, a novel parallel sliding window algorithm is implemented in GraphChi to reduce random access to the data graph. Graphchi can process hundreds to thousands of graph nodes per second. Thus, the focus of this paper is to compare the experimented performance results between a distributed cluster-based Apache Giraph implementation and a single computer, disk based, GraphChi by using four different algorithms and moderate sized datasets. The original GraphChi paper observed that GraphChi is so efficient that one would need more than 1,000 cluster machines in order to achieve the same performance that GraphChi has on just a single PC. However, since the time the GraphChi paper was published (2012), there were many versions of Apache Giraph released which offered many improvements. Hence, in this report we would like to verify anew the claims made in the GraphChi paper.

The metric of the performance will be the task total execution time. The remainder of this report is organized as follows. Section 2 provides an overview of the background on BSP. Section 3 provides a brief introduction on Pregel. A brief introduction to Apache Giraph will be in Section 4. An overview of Graphchi will be provided in Section 5. The overview of the four algorithms for evaluation in Section 6. Section 7 will briefly explain how to setup the distributed computing cluster by using the Amazon EC2. The detail of evaluation results will be provided in Section 8. And the conclusion will be in Section 9.

Chapter 2

2.1 BSP Overview

Bulk Synchronous Parallel (BSP) is a parallel programming model that uses a message passing interface (MPI) to address the scalability challenge of parallelizing jobs across multiple nodes [6]. A BSP system consists of a collection of processors which are connected by a communication network, as showing in Figure 1. In BSP model, each processor has the

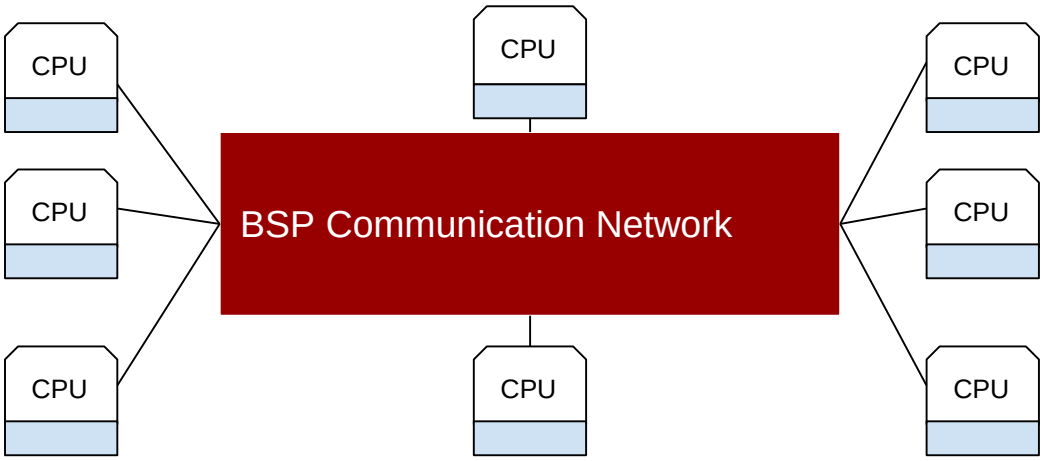

Figure 1

Figure 1. ***Architecture of BSP computing system. CPU is the processor and M is the local memory of the CPU.***

privileges to access any of the memory cells in the system. The speed of the memory access depends on the distance between the processor and the location of the data. If the data is located in the local memory of the processor, a memory access operation such as read or write is relatively fast. The memory access operation is relatively slow if the data is located in other processors' memory across the network channel. The access time for non-locally stored data for all processors is similar. The BSP communication network is

viewed as a black box. Users of the BSP system are not required to know the details of how the communication network will work. *Superstep* is an important building block of the BSP algorithm. A BSP algorithm consists of a sequence of supersteps [7]. A BSP superstep consists of a sequence of communication, computation steps, and a global barrier synchronization. The BSP processors perform a sequence of operations on local data in each of the computation steps. The BSP processors perform communication operations such as reading and sending messages in each of the communication steps. At the end of each communication superstep, the BSP processor will check whether or not it has finished sending or receiving all the messages. Similarly, at the end of each computation superstep, a processor will check whether or not it has finished all the computation tasks. The processor will wait for all other processor in the system to finish, in order to proceed to the next superstep. This form of synchronization is called bulk synchronization. Figure 2 shows an example of BSP algorithm.

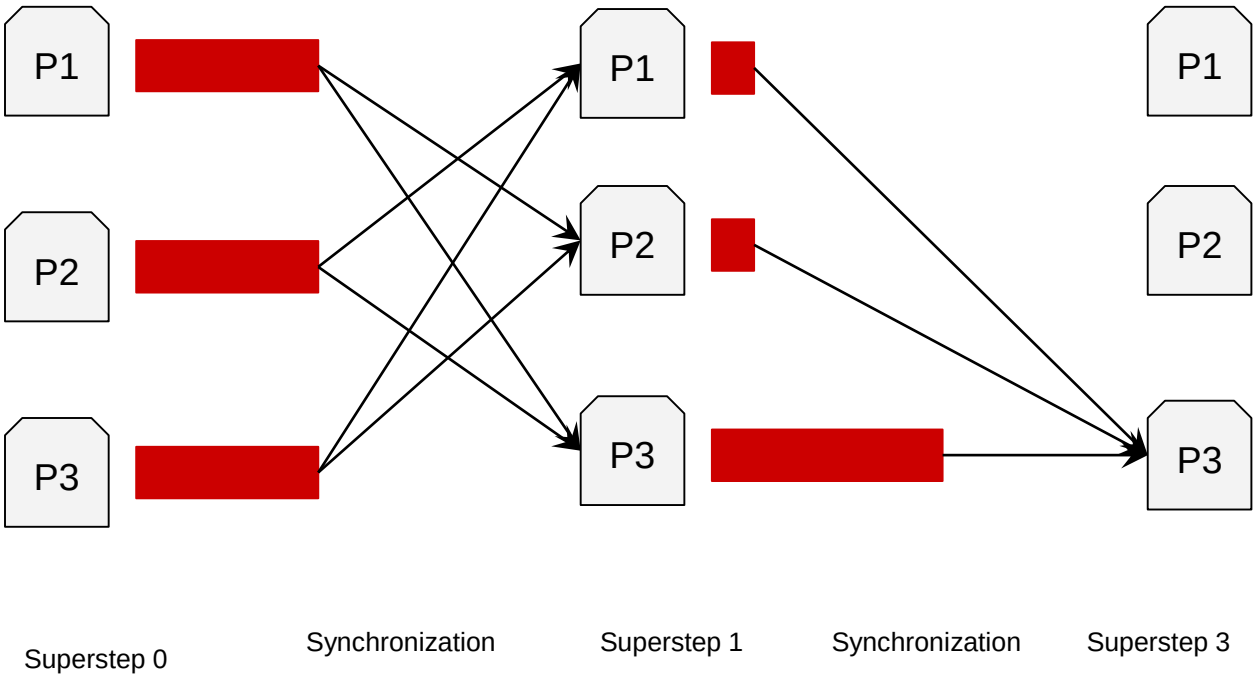

Figure 2, ***there are three supersteps are executed with three BSP processors which are labeled at P. The red bar represents the local computation within each of the processors. The arrow linking processors between two supersteps are the communications from one processor to others. At superstep 0, processor 1, labeled as p1, sends data to processor 2 and 3. And in superstep 1, processor 1, 2 and 3 all sends data to processor 3. After finishing all the computation, all three processors are terminated by the global synchronization.***

2.2 Pregel Overview

Pregel is the heavily inspired by the BSP model and is the first BSP implementation to provide application interface for user to define the algorithmic logic. Similar to BSP, Pregel is also consisted a sequence of supersteps, also called iterations. The user defined function for a vertex that is invoked in each superstep. The function defines the logic and behaviour for each vertex to perform tasks like reading messages sent from the vertex neighbors in previous superstep, sending messages to the neighbors of the vertex and updating the state of the vertex. The user-designed algorithms focus on the local action. Each vertex is updated independently. All these features are going to help process large datasets. The Pregel takes a directed graph as the input. Each vertex of the input graph is labeled with a vertex ID or user defined value. A directed edge is associated with its source vertex ID [8].

In Pregel, all vertices are active in superstep 0. And all the active vertices will participate in the computation in each of the superstep. The vertices move to inactive state by voting to halt. If all the vertices are in inactive state, then, there are is no more computation in the subsequent superstep.

The vertices can update their state from inactive to active state by receiving messages. The Pregel computation is terminated when all the vertices vote to halt and

there is no message in transit in the communication channel. Pregel loads the input graph data at once.

2.2.1 Pregel Architecture

Pregel takes a directed graph as input file for computation. Each of the vertices of the input graph is uniquely identified. Each of the vertices of the input graph is also associated with a user defined value. Each of the directed edges of the input graph is associated with two vertices. And each edge can also be associated with user defined values as the edge value. There are three steps for a typical Pregel computation which are input loading, iterating over sequences of supersteps with global synchronization points until termination and finally producing output. Vertices compute in parallel in each superstep [9]. A vertex in the Pregel computation model can modify its own state and mutate the topology of the graph.

In superstep 0, all vertices of the graph are in active state. Once a vertex is in active state, the vertex will participate in computation not only in superstep 0 but also in any subsequences of supersteps. If a vertex finishes all the computing tasks and has nothing to do, then it will transfer itself to de-active state by voting to halt. Such a vertex will come back to active state if it receives messages in the upcoming superstep. The Pregel vertex state transformation is illustrated in Figure 3.

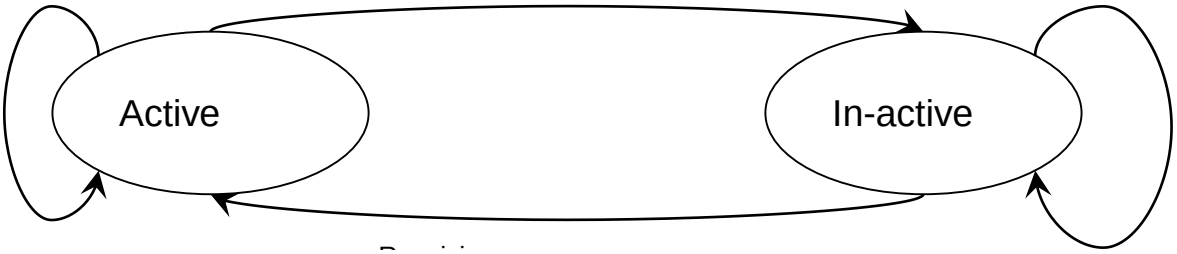

Figure 3, ***Vertex state transform.***

2.2.2 Model of Computation

Pregel is a master/worker model system. Users must choose one of the commodity computers as the master node if the computing platform is in fully distributed mode. There is one master node, and the rest of the computers are running as workers. The master's role is to coordinate the synchronization at the superstep barriers and to control workers to update their states. Each of the workers independently executes the *compute* function. The *compute* function is the main Pregel API for the programmer to define the computation logic for each vertex.

Workers also maintain a message queue to receive messages from the vertices assigned to other workers. In addition, user defined combiners can be used to combine messages for improving performance. Each vertex can contribute a value to an aggregator in one superstep and obtain a globally reduced value in the next superstep [10].

The vertices of the Pregel program will produce a set of values as the format of the output file. The output format is also a graph which is isomorphic to the input graph. However, during Pregel computation, the edges or vertices can be modified either by removing or adding, thus, the output format is not always isomorphic to the input graph. Figure 4 illustrates the Pregel computation concepts. The example in Figure 4 shows a strongly connected graph. Suppose for an example that we want to compute the max value of the graph vertices. Each of the vertices of the graph is assigned a value. The model, then, propagates the

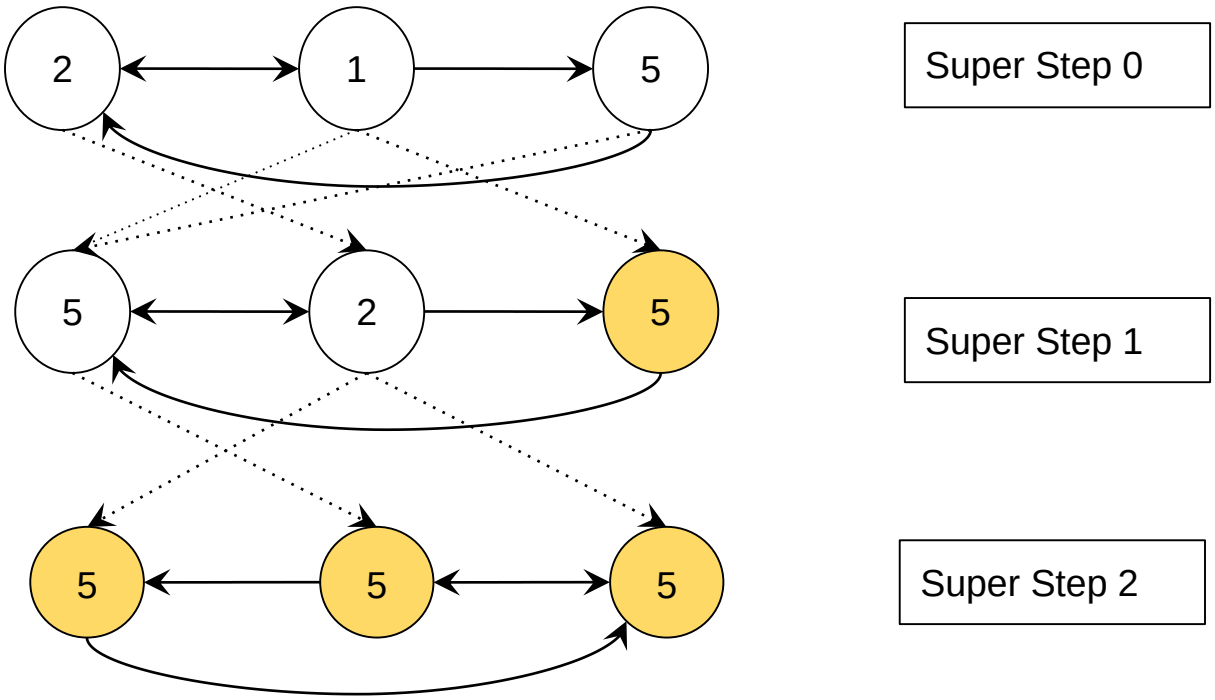

Figure 4. ***Finding the max value. The dashed lines are messages sending from the vertex. The colored vertex is de-active by the voted to halt.***

largest value to all other vertices in the graph, in sequence of supersteps. In each of the supersteps, if any of the vertices finding a larger value than its original, then this vertex will update its value and send the value to all other vertices which are connected with it. This example will be terminated when there is no changing value vertices existing in a superstep [11]. Vertices send messages to each other. The message consists of the value and the destination vertex ID represented as number. Users can define the message format by specifying the template parameters in the vertex class which is provided by the Pregel application interface. A vertex, in Pregel, can send any number of messages. The vertex sending message through the iterator of outgoing edges. Each of the vertex can read all the messages it receiving in the S+1 superstep through the iterator. The message is guaranteed to be delivered without duplication. However, the order of the messages is not guaranteed [11].

2.3 Giraph Overview

Apache Giraph is an open source software system which implements the Pregel model. Giraph runs on top of Hadoop. Giraph uses the HDFS for managing the data input and output. Apache Zookeeper is used by Giraph for controlling the coordination, checkpointing and failure recovery schemes. Just like Pregel, Giraph exposes a vertex centric programming model. Users of Giraph do not need to worry about the details of complexity of large scale parallel and distributed systems. Instead, users of Giraph can focus on the task specific aspect of the algorithm. The algorithm implementation in Giraph is simple [12]. Normally, a Giraph algorithm consists of fewer than 50 lines of code.

Giraph is often used as offline data processing in the back office. Taking Facebook for example, Facebook provides user interface control tools such as “like”, friend request bottoms and chat box on the front end. The texts that user enter and bottom clicks will be collected and stored on the back end server as user activities. Such user activities will be transformed into graph data representation [13].

For example, if Mark accepts Sara’s friendship request, then Mark and Sara marked as nodes in the graph will be updated with one added link between them. Unlike twitter, the friendship relation in Facebook is viewed as undirected link where twitter relationship is directed link. Giraph residents in the back office will take the entire social network graph including Mark and Sara as nodes as inputs then executing algorithms such as friendship rank or shortest paths [14]. The reason for placing Giraph in the back office is that the entire graph processing progress will not be interrupted by the front end user activities.

Giraph can run on Hadoop as a Mapreduce program, read data from Hadoop Distributed File System (HDFS), Hadoop Database (HBase), Hadoop Hive, Cassandra, HCatalog, use Hadoop ZooKeeper to coordinate the computation and provide fault tolerance. Apache Hama is a BSP model system. Similar to Giraph, Hama also uses iteration to solve graph problem. However, Hama requires extra software to be installed on top of Hadoop in order to process graph. Giraph does not require any extra software installation.

Giraph is used to process large scaling graph data and is designed for non-interactive, offline, periodic data processing. Giraph is not suitable for interactive and fast information retrieval applications. Giraph is used to run algorithms which can scale.

Giraph adapts the vertex centric model which is a decentralized distributed model. There are five factors needed to be considered when designing algorithms running on Giraph. Firstly, each vertex should be able to make decisions such as send, read or vote to halt independently based on the messages it receives. Secondly, each vertex should be initialized with a value. Thirdly, the halting condition of each vertex should be carefully designed and well understood. And finally, tools such as combiner and aggregator should be used to simplify and improve the global computation if necessary.

2.3.1 Giraph API

Similar to Pregel, users of Giraph define the vertex computational logic by subclassing the BasicComputation class and overriding the compute() method. Giraph invokes the compute() method on all the active vertices in each superstep to process messages either by sending or receiving. The vertex and edge format are identical to the Pregel API, where each of the Giraph vertex is again associated with a user defined value and the edge which is shared between two vertices is associated with an edge value. However, users of Giraph need to specify the message type. The message value could just be an integer or a composite type value such as a list of integers. The graph data input format for Giraph may pose challenges for new users. Two main Giraph input formats are vertex based input format and edge based input format. Giraph provides API for both in abstract classes such as VertexInputFormat and EdgeInputFormat. The user needs to extend the abstract classes in order to implement the input reader for their computational algorithms. Similar to the input format, Giraph provides the user two API so that the user

can define his output format. The user can extend either the VertexOutputFormat or EdgeOutputFormat abstract classes to specify the data output format.

2.4 Graphchi Overview

GraphChi is a member of the GraphLab Family. GraphChi is implemented in C++ originally. The Java version of GraphChi implementation is made available recently. Unlike Pregel and Giraph, GraphChi is a centralized, disk based model which is able to process large scale graph data from a hard disk in a single computer. GraphChi processes graphs by using the Parallel Sliding Window mechanism [14]. Compared to distributed systems, such as Pregel and Giraph, GraphChi avoids communication overheads, distributed cluster management and node failures.

GraphChi is an open source project. Users can download and install GraphChi for free and process large scale graph data just on a single computer. Programming in GraphChi is similar to Pregel and Giraph where the user needs to implement or override the API such as *compute* method. GraphChi provides the *update* method for the user to define algorithmic logic for processing the graph. The update method plays the same role as the compute method in Pregel. The main difference between GraphChi and Pregel like systems is that Pregel uses a synchronization mechanism to update the vertex value and state, while GraphChi updates vertex value and state asynchronously [15].

2.4.1 Model of Computation

The input graph data of GraphChi consists of vertices and edges. Each vertex and edge is associated with a value. GraphChi updates vertex values by reading and computing edge

values. Because GraphChi is an asynchronous model, GraphChi always reads the most recent edge values to perform the computation.

GraphChi adapts the Parallel Sliding Window (PSW) mechanism to efficiently retrieve the graph data which is stored in local hard disk with small number of disk read. Under the PSW method, the vertices of the input graph data are split into K intervals. Each of the K intervals is associated with one or more shards. A shard is a unit for storing partial graph data, namely all the edges with destination vertex in an interval, e.g. shard 1 stores all the edges with destination vertex in interval 1. Each shard can be loaded into memory completely (Figure 5). PSW loads the

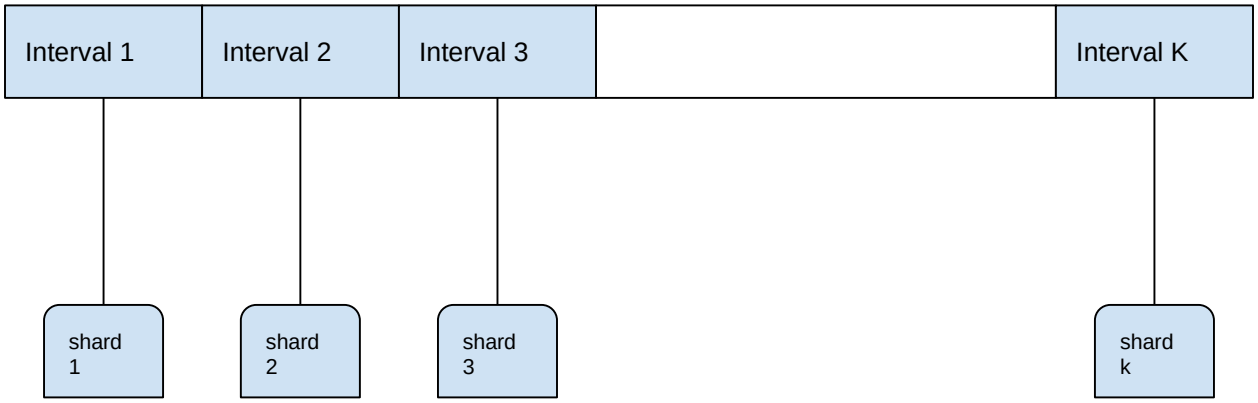

Figure 5, ***the graph is splitted into K intervals. And each interval has one shard.***

vertices and associated edges from the local hard disk, one interval at a time. Vertices and associated edges in Shard (1) associated with interval (1) are loaded into main memory first. After a shard is loaded into main memory, PSW will slide over the shard through a window [14]. Depending on the graph distribution, the window size of each shard could be different.

After finishing the computation for an interval, PSW writes the updated block back to the hard disk, replacing the old data. The number of hard disk write back is the same to the number of disk read.

Chapter 3

3.1 Introduction

Four algorithms are evaluated in this experiment. These four algorithms are Pagerank, Single Source Shortest Paths (SSSP), Weakly Connected Components (WCC) and Vertex Connection Count (VCC).

3.2 PageRank

PageRank is a popular random walk algorithm addressing the ranking of linked object problem. The main goal for PageRank is to assign a numeric ranking value to each of the web pages by exploring the structure of the web. Web page receives more link from other pages will be ranked higher by PageRank. PageRank uses damping factor, a probability value, to determine the likelihood that a user will jump to a random page by clicking a outgoing link from the current page when the user is browsing. In this evaluation, the damping factor is set to 0.85. A user may jump to a random page with 85% likelihood. And 15% likelihood for a user to jump to a random page with in the entire web.

At the superstep(0), each vertex is assigned with value 1.0 as the vertex value. At each subsequent supersteps, each vertex will sum all the value from its in-edges and store the summation in x. The vertex value will be updated by combine the dumping factor and the in-edge summation value x, then store the value in p. The p equation is as p=0.15+0.85*x. After updating the vertex value, the vertex will calculate the expectation value e. The expectation value formula is e=p/number of out-edges. After calculating the expectation value, the vertex sends the value through its out edges [16].

3.3 SSSP

SSSP here stands for single source shortest path. SSSP is a sequential traversal algorithm which finds the shortest paths between a random vertex and all other connected vertices. A path in an undirected graph is a sequence of adjacent vertices, between p and q, and is shortest if the sum of the constituent edge weights is minimized. The Giraph implementation of SSSP is a parallel variant of Bellman-Ford algorithm. At the first superstep, the distance of the source vertex is set to 0 and the value of all other vertices are set to infinite. Also, the source vertex p is the only active vertex in superstep(0). The source vertex calculates the sum of the current minimum distance value and the edge weight then sends the summation to all its neighbours. In the subsequent superstep(1), all the neighbours of vertex p receive the message from p, pick the minimum distance value and update their state into active. The number of supersteps taken by SSSP is limited by the graph's longest shortest path. SSSP is a simple algorithm to test the system ability for handling changing communications and bottle neck.

3.4 WCC

Weakly Connected Components (WCC) is a parallel traversal algorithm which is used to find the number of weakly connected components of a graph. A component is weakly connected if every pair of vertices is mutually reachable when ignoring edge directions. The idea for this algorithm is simple. Each vertex has its own id and each vertex propagates the smallest vertex id to all other vertices with in a connected components. After receiving the smaller id, a vertex update its own id with the smaller id and propagates the new id to all its neighbours. The number of supersteps is equal to the number of connected components in the graph. At superstep(0), all vertices are active

which is different from SSSP. Thus the communication overhead is heavier than SSSP.

At each of the supersteps, a vertex can vote to halt if there is nothing left to update.

Chapter 4

4.1 System Setup

The Giraph experimentation is conducted on 20,25,30,25, and 39 machines. All machines are T2.micro Amazon Elastic Computing instances which are located in us-west-2a. A single EC2 T2.micro instance is used for running GraphChi program. Each T2.micro instance has one 2.5 GHz, Intel Xeon Family CPU, 1 GB of RAM Memory. All instances run Amazon Linux AMI 2015.09.1 (HVM) Operating System. The default image includes AWS command line tools, Python, Ruby, Perl, and Java. The repositories include Docker, PHP, MySQL, PostgreSQL, and other packages. There is one worker for each machine. All machines are EBS backed by default. No additional EBS attachment is added for the simplicity of management. All the T2.micro machines, used in this experimentation, are in the AWS Free Tier for reducing the cost.

To log into the T2.micro machine, a pair of public and private keys is generated from the AWS EC2 console tool box when creating T2.micro instances. For EBS volume, the default type of EBS volume is used in both experimentations. The default type of EBS volume is 8.0 GB. The Amazon Linux AMI image will use 1.1 GB out of 8.0 GB of the EBS space. In this experimentation, Giraph version 1.1.0 and Hadoop version 1.2.1 are chosen for the installation and management simplicity. User needs to build the tar file after download the Giraph source package. The default Giraph tar file is for Hadoop version 1.2.1. For Hadoop version 2 and higher, user needs to build Giraph tar with -Phadoop_p option. And from our experience, the built Giraph tar for Hadoop version 2 does not work. The default Java runtime environment and JDK is 1.7. For the Giraph

experimenting cluster, there is one master and multiple workers. For the simplicity, there is one worker per machine. The Giraph, Hadoop and Java SDK and runtime environment are installed to both the master and workers. Thus we use the python fabric scripts to administrate the master and workers.

Python fabric provides a basic suite of operations for executing local or remote shell commands and uploading/downloading files, as well as auxiliary functionality such as prompting the running user for input, or aborting execution. By using python fabric, all required software is installed to all machines in parallel. Typical use involves creating a Python module containing one or more functions, then executing them via the fab command line. GraphChi installation is simpler comparing to Giraph. However, GraphChi requires Java 6 to build the package after download the source. After build the package, User need to switch to Java 7 for compiling and running the user updated graph algorithm.

The data set used in this experimentation is Social LiveJournal which is a free online community with almost 10 million members. Social LiveJournal allows members to maintain journals, individual and group blogs, and it allows people to declare which other members are their friends they belong. In this dataset, there are 4 million nodes and 60 million directed edges. The data set is stored in Hadoop Distributed File System (HDFS) as uncompressed ASCII text file.

Chapter 5

5.1 Results

The metric for this experiment is total execution time. The total execution time is the total sum of the graph reading, processing, computation, and the writing time. Figure 6, 7, 8, and 9 illustrate the computational comparison between the Giraph and the GraphChi system. Four algorithms are experimented for both of Giraph and GraphChi. These four algorithms are Single Source Shortest Path (SSSP), PageRank, Weakly Connected Components (WCC), and Vertex Degree Count (VCC). Single Source Shortest Path, Weakly Connected Components and PageRank algorithms are implemented in the Giraph example library. Weakly Connected Components and PageRank algorithms are also implemented in GraphChi. Thus, we added our own implementations for Single Source Shortest Path on GraphChi and Vertex Connection Count for both GraphChi and Giraph. The Giraph experimentation is conducted five times. The number T2.micro instances running Giraph are increased each time. The data set for this experimentation is the Social LiveJournal, available from Stanford Large Network Data Collection.

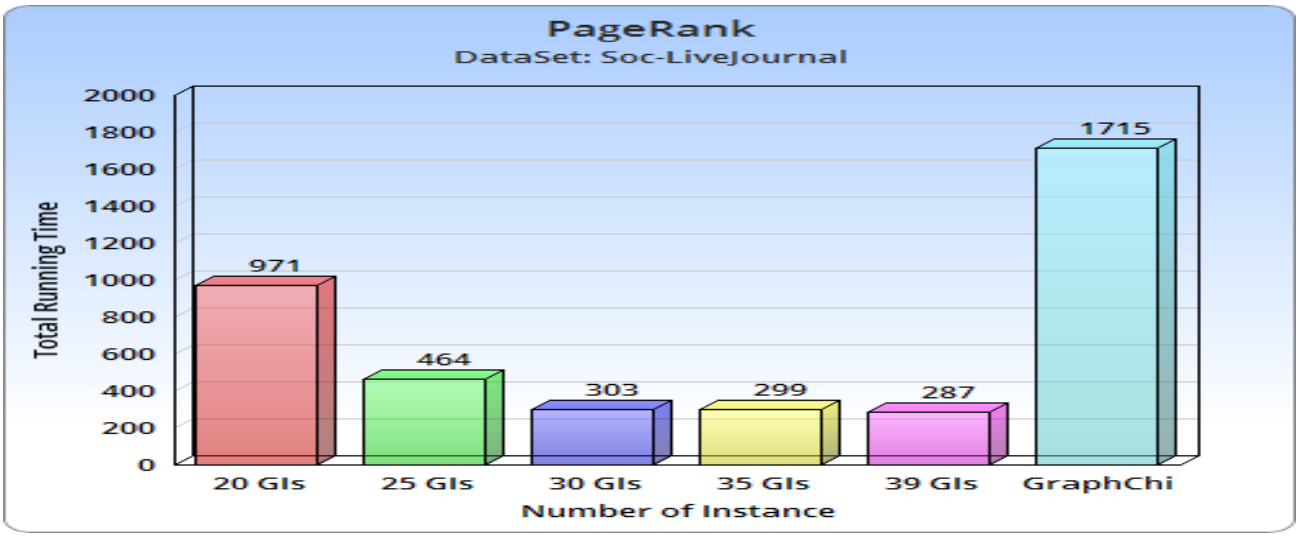

Figure 6, ***The total execution time for running PageRank on Giraph and GraphChi is measured in seconds. GIs stands for number of Giraph instances.***

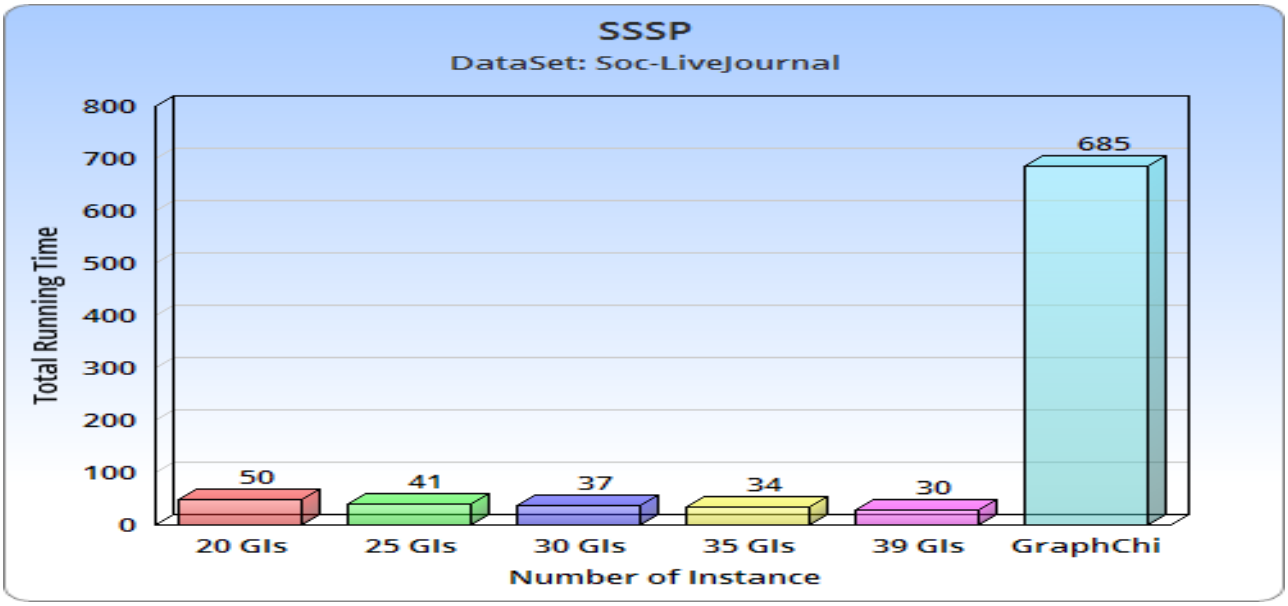

Figure 7. ***The total execution time for running Single Source Shortest Path on Giraph and GraphChi. GIs stands for number of Giraph instances.***

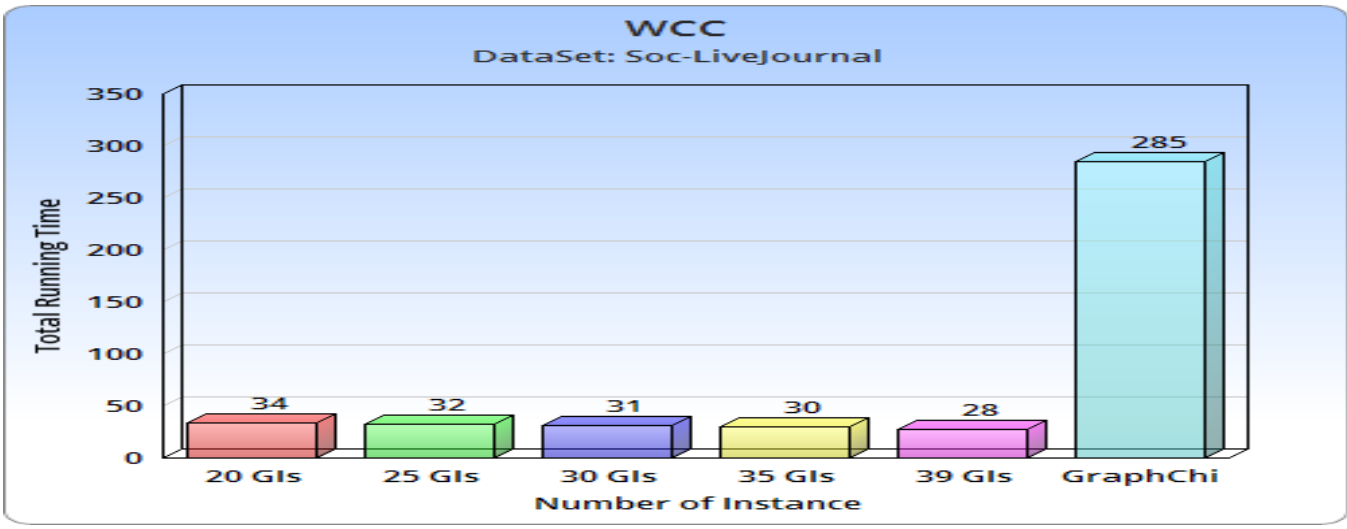

Figure 8. ***The total execution time for running Weakly Connected Components on Giraph and GraphChi. The GIs stands for number of Giraph instances.***

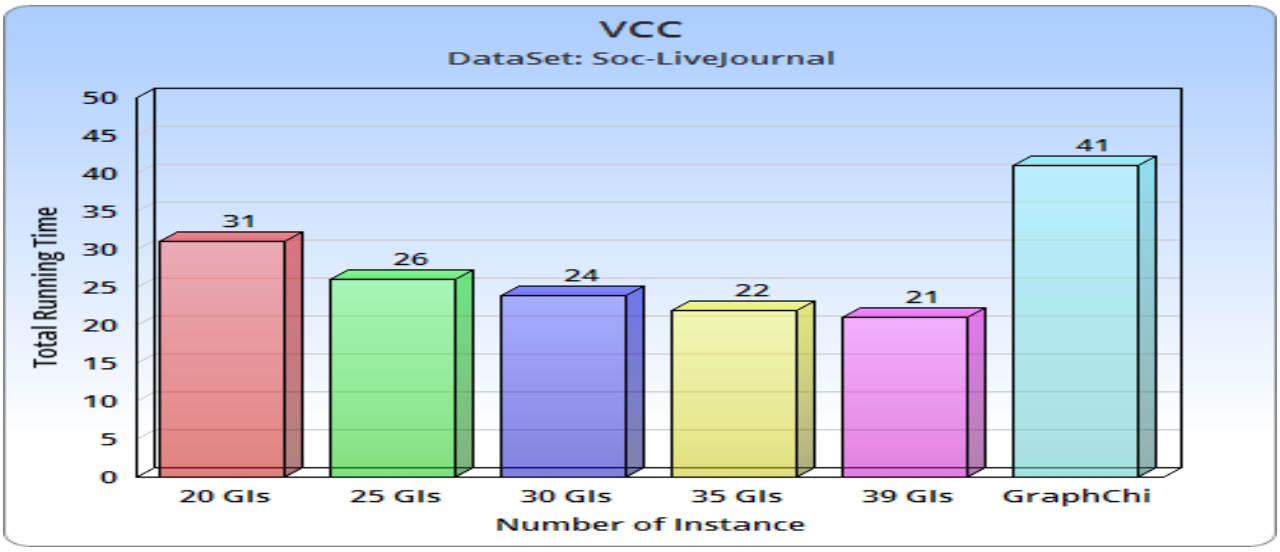

Figure 9. ***The total execution time for running Vertex Connection Count on Giraph and GraphChi. The GIs stands for number of Giraph instances.***

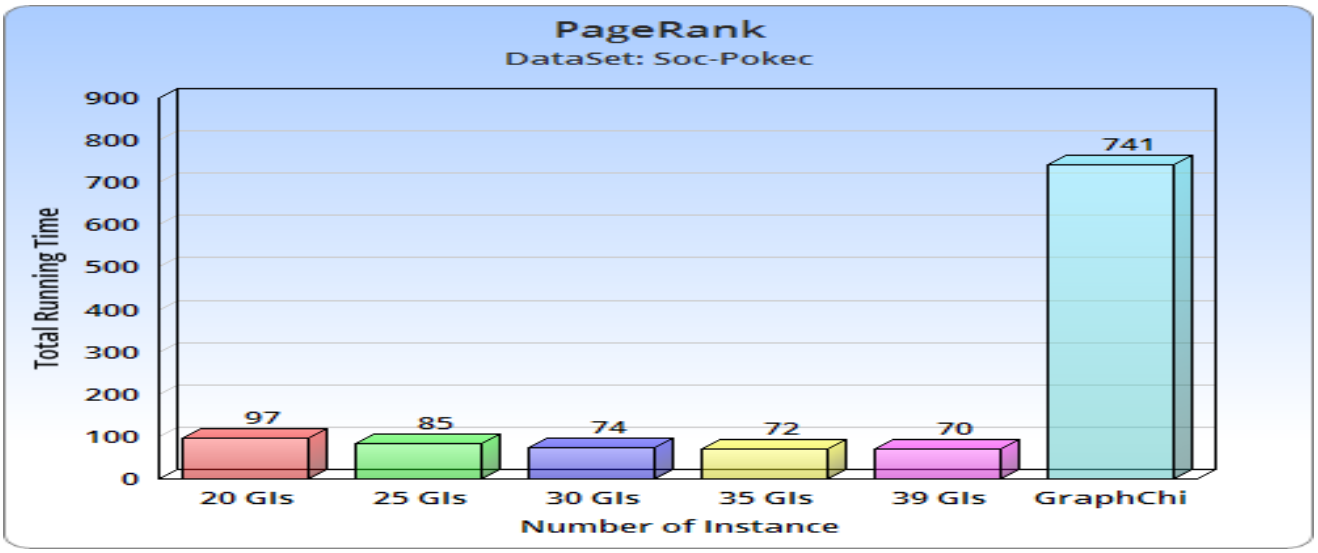

Figure 10, The total execution time for running PageRank on Giraph and GraphChi. ***The GIs stands for number of Giraph instances.***

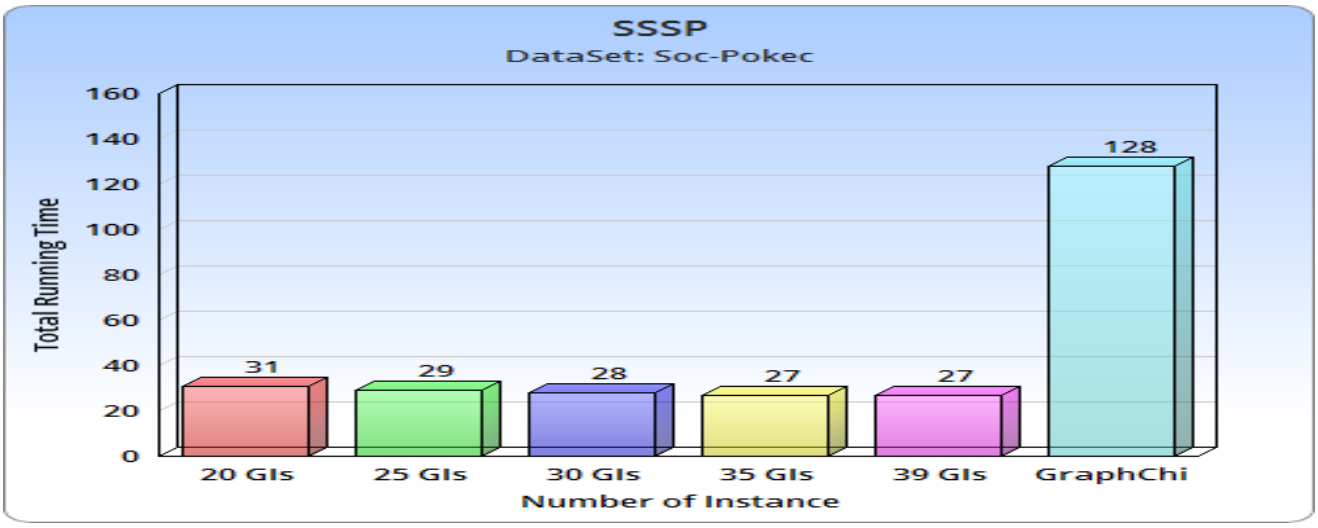

Figure 11, The total execution time for running Single Source Shortest Path (SSSP) on Giraph and GraphChi. ***The GIs stands for number of Giraph instances.***

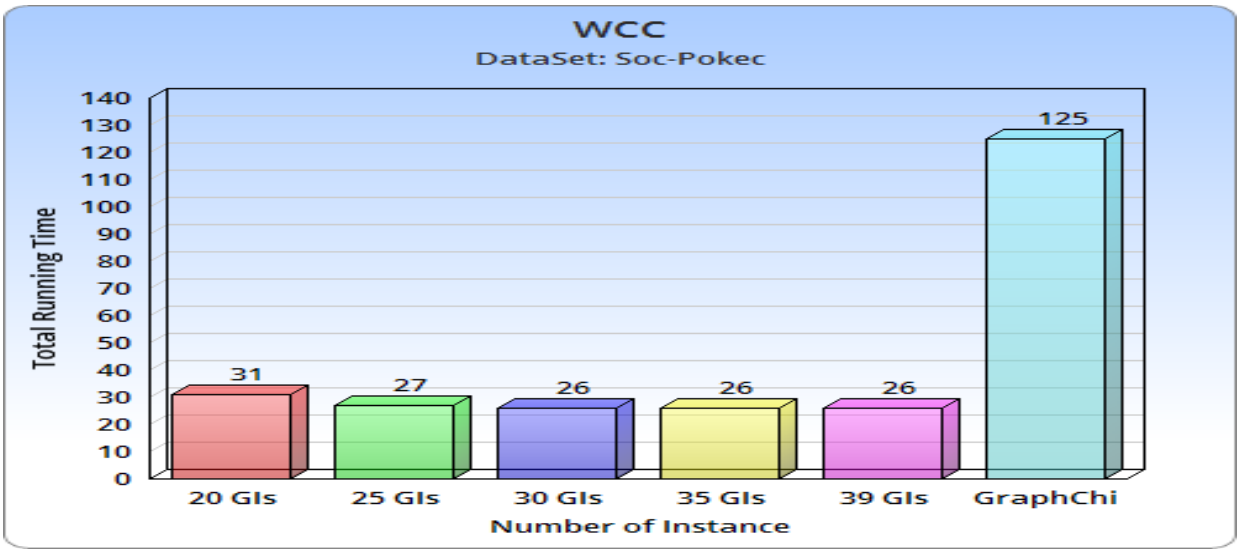

Figure 12, the total execution time for running Weakly Connected Components on Giraph and GraphChi. ***The GIs stands for number of Giraph instances.***

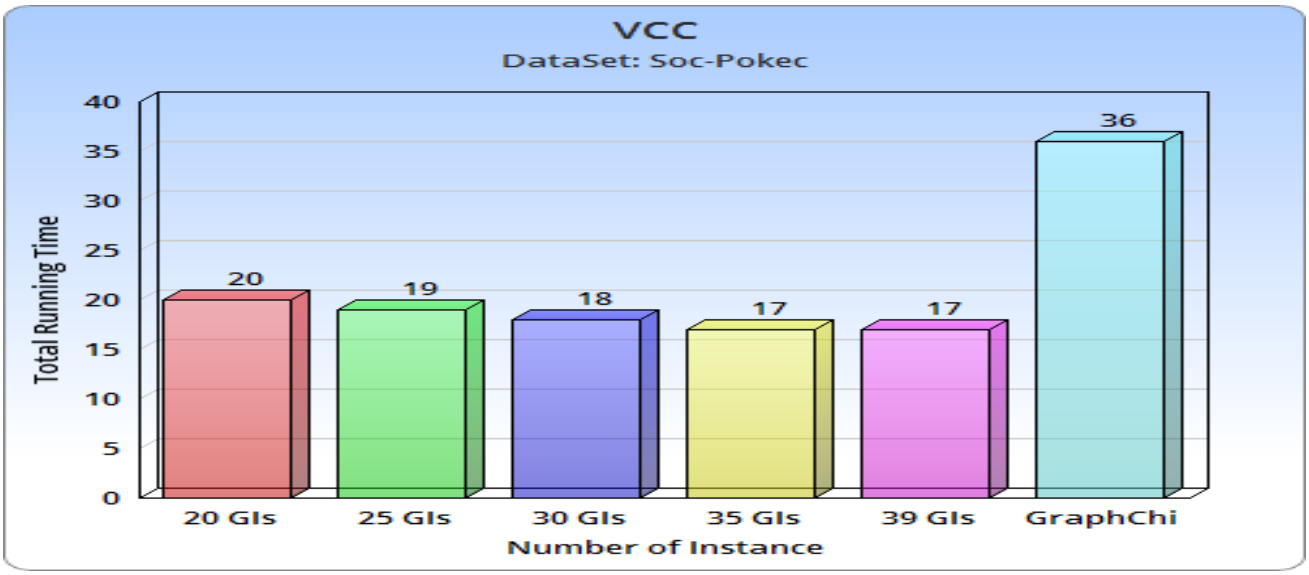

Figure 13, the total execution time for running Vertex connection count on Giraph and GraphChi. ***The GIs stands for number of Giraph instances.***

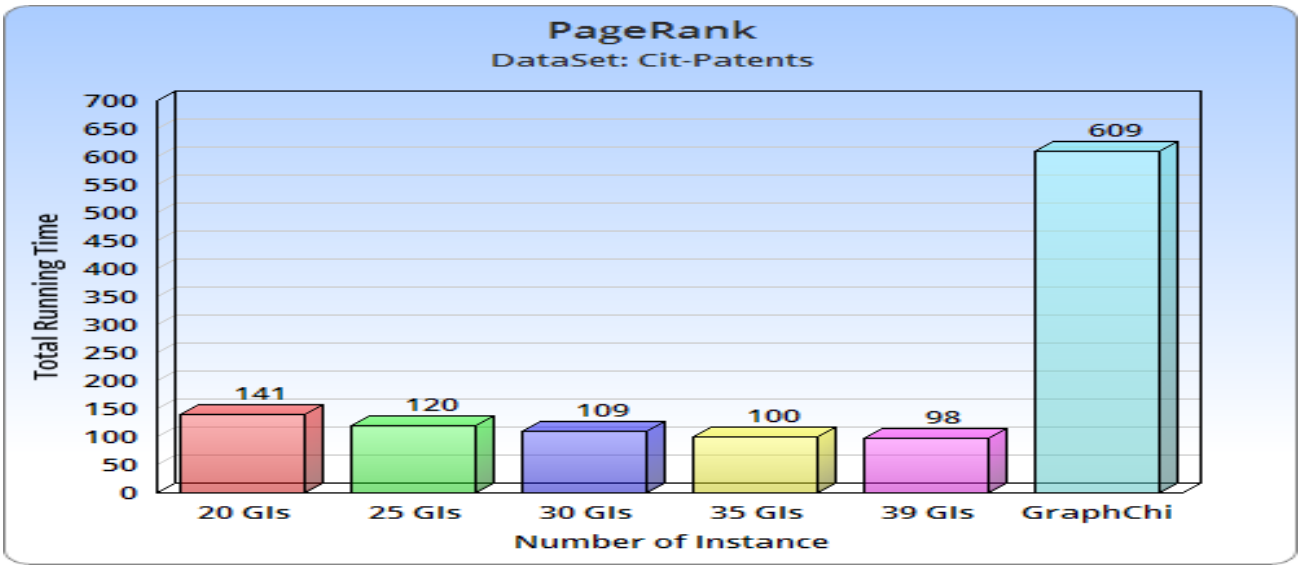

Figure 14, the total execution time for running PageRank on Giraph and GraphChi. ***The GIs stands for number of Giraph instances.***

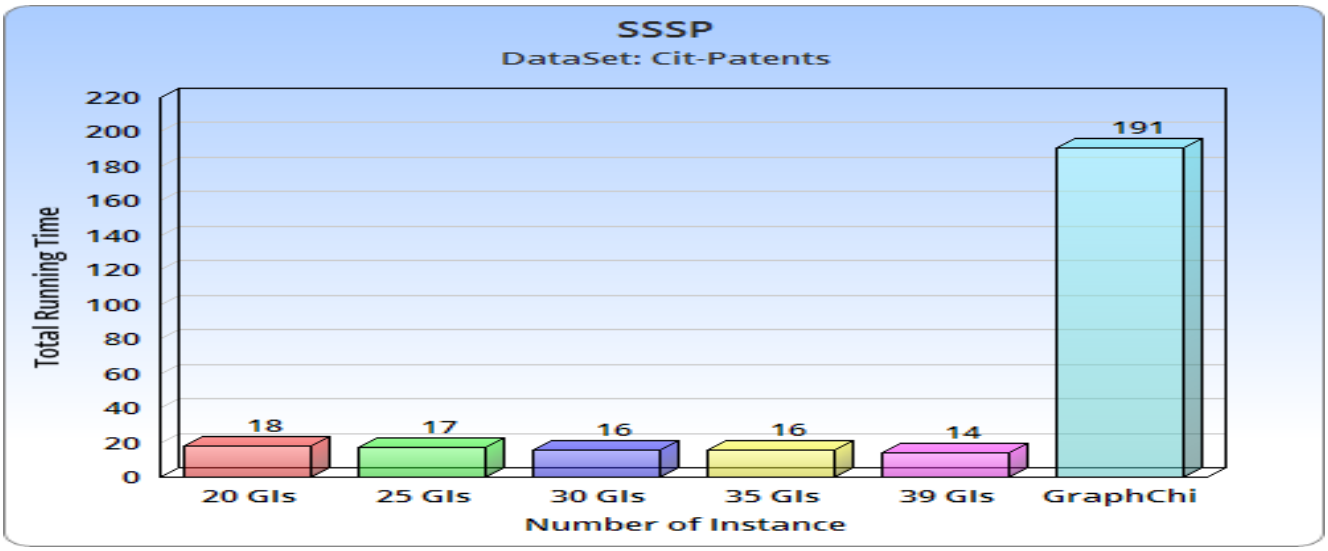

Figure 15, the total execution time for running Single Source Shortest Path on Giraph and GraphChi. ***The GIs stands for number of Giraph instances.***

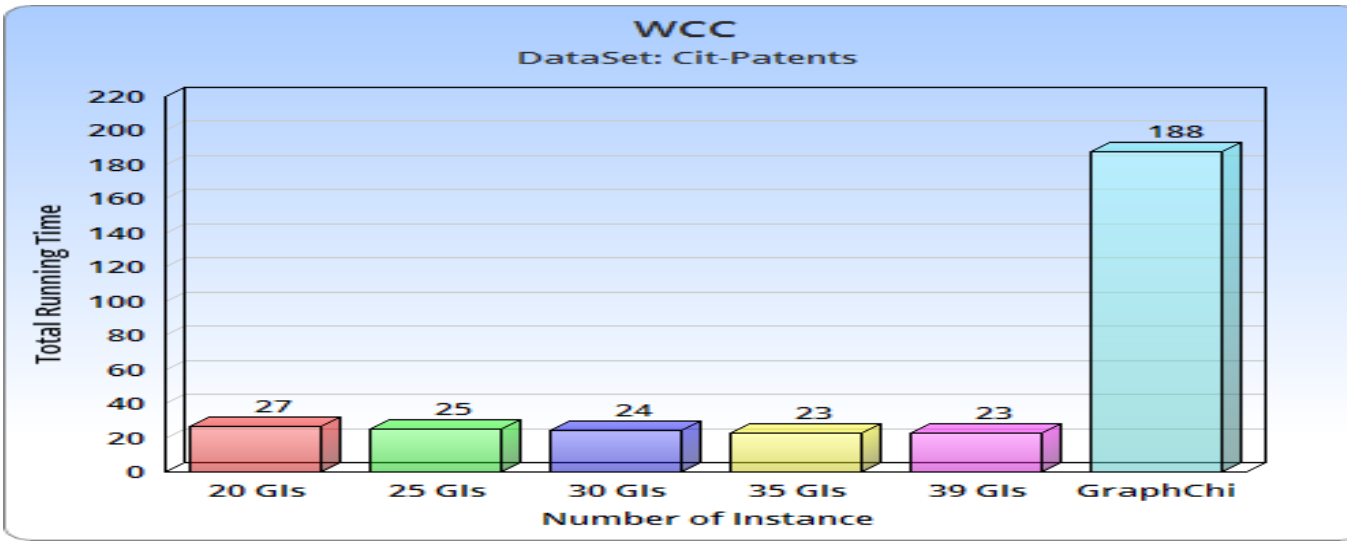

Figure 16, The total execution time for running Weakly Connected Components on Giraph and GraphChi. ***The GIs stands for number of Giraph instances.***

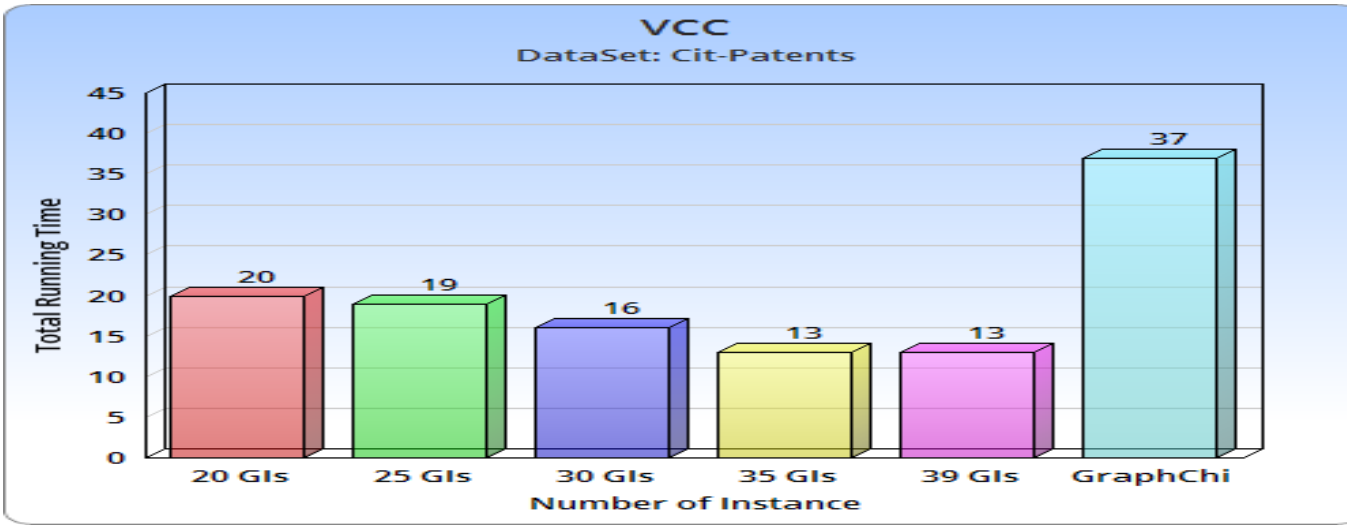

Figure 17, the total execution time for running Vertex Connection Count on Giraph and GraphChi. ***The GIs stands for number of Giraph instances.***

For all plotted figures, the bar represents the total running time on Giraph and GraphChi. There are five Giraph bars and one GraphChi bar. The Giraph bars represent the total running time when applying the number of Amazon EC2 T2.micro instances running

Giraph job. The GraphChi bar represents the total running time when GraphChi is running on a single T2.micro instance. The horizontal axis represents the number of Giraph instances launched from Amazon EC2 cluster plus one Amazon EC2 T2.micro instance for running GraphChi. The number of T2.micro instances are ordered incrementally. The vertical axis represents the total execution time measured in seconds. The first bar in Figure 6 shows that the total running time for executing Pagerank using Giraph across over 20 T2.micro instances is 971 seconds. The second bar represents that the total running time is decreased to 464 seconds when the number of T2.micro instances are increased to 25. The second last bar from Figure 11 represents that the total running time is decreased further to 288 seconds when Giraph is running across 39 T2.micro instances. The last bar from Figure 6 represents the total running time for executing Pagerank using GraphChi on one T2.micro instance. The total running time is 1715 seconds. For all tested algorithms, we find that Giraph outperforms GraphChi.

Chapter 6

6.1 Conclusion

Graphs play a vital role for modeling complicated relations in many application domains such as health, transportation, machine learning, social and knowledge based networking. With growing popularity of the cloud computing, graphs with millions of nodes and billions of edge are common. There are many tools for processing large scale graphs on the cloud computing platform, for example, Pregel, GraphX, Apache Giraph. And a new centralized software, GraphChi, is able to processing graph on a single computer. In this report, we have conducted evaluation experimentation between Apache Giraph and GraphChi through four different algorithms. We observed that Giraph outperformed GraphChi for a moderate number of simple machines. The evaluation is contrast to the claim that GraphChi outperformed a cluster of about 1000 computers.

Bibliography

[1] Apache Giraph. http://apache.giraph.org.

[2] Omar Batarfi, Radwa El Shawi, Ayman G. Fayoumi, Reza Nouri, Seyed-Mehdi-Reza Beheshti, Ahmed Barnawi and Sherif Sakr. Large Scale Graph Processing Systems: survey and an experimental evaluation.

[3] Luiz Barroso, Jeffrey Dean, and Urs Hoelzle, Web search for a planet: The Google Cluster Architecture.

[4] 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 .

[5] Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, and Thanasis Tsantilas, Portable and Efficient Parallel Computing Using the BSP Model.

[6] Olaf Bonorden, Ben H.H. Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) Library.

[7] Jonathan Hill, Bill McColl, Dan Stefanescu, Mark Goudreau, Kevin Lang, Satish Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling. BSPlib: The BSP Programming Library.

[8] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski . Pregel: A System for Large Graph Processing.

[9] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski . Pregel: A System for Large Graph Processing.

[10] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski . Pregel: A System for Large Graph Processing.

[11] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang and Tianqi Jin. An Experimental Comparison of Pregel-like Graph Processing System.

[12] Kajdanowicz T, Kazienko P and Indyk W. Parellel Processing Large Graph.

[13] Claudio Materella, Dionysios Logothetis and Roman Shaposhnik. Pratical Graph

Analytics with Apache Giraph.

[14] Aapo Kyrola, Guy Blelloch and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC.

[15] L. Page, S. Brin, R. Motwani, and T. Winogard. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, Novem- ber 1999.

[16] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2006.