International Journal of Engineering & Technology, 7 (2.12) (2018) 374-379

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research Paper

Designing and analyzing highly scalable and reliable of full fledged parallel algorithm for computing strongly connected com-ponents to analyze social graphs

Dr. Lokesh A [1] *, Mr. Maria Navin J R [1], Mr. Balaji K [2], Mr. Pradeep M4 [3]

[1] *Associate Professor, Dept. of ISE, Sri Venkateshwara College of Engineering, Bengaluru, Karnataka, India*
[2] *Assistant Professor, Dept. of CSE., Sri Venkateshwara College of Engineering, Bengaluru, Karnataka, India*
[3] *Assistant Professor, Dept. of CSE., M S Engineering College, Bengaluru, Karnataka, India*
**Corresponding author E-mail: lokeshyadav.ka@gmail.com*

Abstract

With the recent advent of Big Data, developing efficient distributed algorithms for computing Strongly Connected Components of a large dataset has received increasing interests. For example, social networks, information networks and communication networks such as the communities of people that have formed on those networks, what community a person belongs or finding cyclic de-pendencies in the graph. Apache Giraph is an open-source implementation of Google's Pregel. It is an iterative and real-time graph processing engine designed to be scalable, fault tolerant and highly efficient. This framework provides an accurate platform for the development of parallel algorithms in a distributed environ-ment. It adopts a vertex-centric programming model inspired by Bulk Synchronous Parallel model. A strongly connected component is a maximal sub graph in which all vertices are reachable from every other vertex. Maximal means that it is the largest possible sub graph. It is not possible to find another vertex anywhere in the graph such that it could be added to the sub graph and all the verti-ces in the sub graph would still be connected. In a directed graph G, a pair of vertices u and v are said to be strongly connected to each other if there is a path in each direction between them. Here, we have implemented a parallel algorithm which is based on the new paradigm of graph decomposi-tion for computing strongly connected components. The final outcome mainly focuses on the reduc-tion of total communication costs.

Keywords: *Apache Giraph; Graph; Strongly Connected; Cost; Parallel; Graph Processing.*

1. Introduction

1.1. Scope of work

A graph is a structure that represents abstract entities (or vertices) and their relationships (or edges). Graphs are used in computer science in a variety of domains to represent data and express problems. Many common data structures like trees and
linked lists are graphs; their nodes are connected with links according to the criteria that define each specific data structure. Graphs have been useful in representing domains such as social media, communication systems, computer network, web and many more. However, efficient processing of graphs poses challenges due to their scalability [1].
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 [2].
Apache Giraph provides a vertex-centric programming model that requires the developer to think like a vertex that can exchange messages with other vertices. The programming model hides the complexity of programming a parallel and distributed system.

1.2. Giraph as a hadoop tool

Hadoop is a popular platform for the management of Big Data. It has an active community and a high adoption rate with yearly growth of around 60%, and a number of companies make supporting these enterprises their mission. Hadoop started as an implementation of the Google distributed file system and the MapReduce framework in the Apache Nutch project. Over the years, it has turned into an independent Apache project. It has evolved into a full-blown ecosystem. Hadoop Distributed File System and the MapReduce framework lie under its functionalities. All these systems were developed to tackle different challenges related to managing large amounts of data, from storage to processing and more. Giraph is a relatively newcomer to the Hadoop ecosystem.

2. Literature review

Recently frameworks available for parallel processing for large graph applications on high performance computing clusters [3] [4], massively multi-threaded architectures [5] and GPUs [6]. MapReduce [7] [8] which is a tuple based method has been considered as a an appropriate solution for large graph processing and not suitable for many graph applications [9].

Copyright © 2018 Dr. Lokesh A et. al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Vertex-centric and block-centric approaches are the two graph processing systems. In vertex-centric approaches, messages are exchanged directly between vertices and whereas in block centric approaches, messages are exchanged between connected subgraph [10].
Based on distributed in-memory big graph processing we can divide them as systems for offline graph analytics and systems for online graph querying[11].
The key to success is parallelization based on a "shared nothing" architecture and non-blocking networks ensuring a smooth communication among servers. I/O and data processing are distributed by moving computing tasks to the server nodes where the data resides. Distributed parallel processing provides several advantages. Execution of a query or similar data operation by many nodes at the same time increases performance and delivers fast results. In order to make the overall configuration fault-tolerant, data is automatically replicated to several nodes. If a server fails, the respective task can be continued on a server where a data replica resides, fully transparently for software development and operation. Data updates need to be considered for all data copies, otherwise the system would be inconsistent. On the software layer the server farm is coordinated as a distributed storage system which coincidentally runs as a parallel processing cluster. This is seen as one of the biggest enablers for Big Data analytics.

3. System architecture

3.1. Applications of strongly connected components

- Finding Strongly Connected Components in a social network graph can reveal information about the communities of people that have formed on those networks. Social Networks can study the evolution of those communities and teach what.
- Community a person belongs to and may help getting better contacts for him.
- Strongly Connected Component algorithms can help find the cyclic dependencies in a program. Given a dependency graph G (directed graph), the Strongly Connected Components which contains more than one node of G (if any), gives the cycles in the graph.
- The whole idea is that collapse the graph to the essentials, removing redundancy. If you can reach one vertex from another and vice versa, you can treat them as essentially.

The same vertex. This new graph has no cycles. Since you may be reducing out many nodes, you may be able to more efficiently find how one vertex is related to another.

4. Implementation

4.1. Computation

It is the most crucial stage in achieving a new successful system for efficient performance and this action requires preliminary thinking. Implementation of any software or system is always carried by different factors and platforms. Thus, implementation defines the final phase of model building.

4.2. Loading phase

- The input data contains a representation of the graph and metadata on the vertices or edges.
- The graph is loaded into Giraph through a vertex or edge input format.
- The output is observed in several output formats.
- The Input and Output formats are present in org. apache. giraph. io. formats.
- Apache Giraph consists of several built-in vertex input formats such as :
- Giraph File Input Format
- GiraphTextInputFormat
- IntIntNullTextInputFormat
- JsonBase64VertexFormat
- JsonBase64VertexInputFormat
- GeneratedVertexInputFormat
- LongLongNullTextInputFormat
- SequenceFileVertexInputFormat
- IntIntNullTextVertexInputFormat
- IntIntTextVertexValueInputFormat
- AdjacencyListTextVertexInputFormat
- PseudoRandomIntNullVertexInputFormat
- JsonLongDoubleFloatDoubleVertexInputFormat
- LongDoubleDoubleAdjacencyListVertexInputFormat
- TextDoubleDoubleAdjacencyListVertexInputFormat

Apache Giraph consists of several built-in edges input formats such as:

- TextEdgeInputFormat
- IntNullTextEdgeInputFormat
- PseudoRandomEdgeInputFormat
- PseudoRandomLocalEdgesHelper
- IntNullReverseTextEdgeInputFormat

Giraph's I/O builds on top of Hadoop's input/output format API. This allows us to easily incorporate existing Hadoop formats.

- The input graph may be laid out in two main ways:
- The directed edges may be grouped by source vertex i.e., represented as an adjacency list. Here, any metadata for the vertex can be read together with its outedges. This is achieved by implementing VertexInputFormat.
- They may appear in arbitrary order. Here, edges will be read by means of an EdgeInputFormat. If there is additional data for the vertices, it will be read separately by a VertexValueInputFormat.
- It is also possible to develop a customized vertex input format as required.
- In our implementation we use have developed a customized vertex input format named as

SccLongLongNullTextVertexInputFormat.

4.2.1. Input graph

The first step of any Giraph application is to determine the format for the graph input data.

- Vertex ID - The Vertex ID is the identifier for a vertex in the graph. The framework defines it to be something no more complex than a label.
- Vertex Value -The Vertex Value optional, and is another place to store additional information associated with a vertex. Typically, this field is used to store values or objects that should be updated during graph processing.
- Edge Tuples - The final piece of input data is the collection of information necessary to define the set of out-edges associated with the source vertex ID. It is composed of tuples with two components per edge: the destination Vertex ID and the Edge Weight such as in JsonLongDoubleFloatDoubleVertexInputFormat. Any bi-directional edge must be defined as two separate out-edges with opposite directions.
- Input Format Selection - Once the data types for each of these pieces of information have been defined, a MapReduce Input Format must be selected. Giraph package holds the implementation of many different input formats so that common data types can be easily accommodated.

4.2.2 Example input graph

Consider a graph input represented in the JsonLongDoubleFloatDoubleVertexInputFormat.

- The first value represents the vertex ID
- The second represents the vertex value.
- The edge tuple is represented by the destination ID and the edge weight.

The input is defined in a text file as follows:
[1, 4.3, [[2, 2.1], [3, 0.7]]]
[2, 0.0, []]
[3, 1.8, [[1, 0.7]]

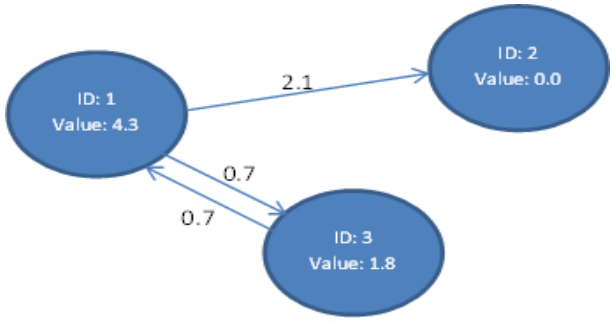


Fig. 4.1: Graphical Representation of Vertex Input Format in Giraph.

4.2.3 Strongly connected components vertex input format

- A customised vertex input has been defined for the computation of Strongly Connected Components.
- It is named as SccLongLongNullTextVertexInputFormat.
- Here, the values are separated by a space or tab as a delimiter.
- The first value represents the vertex ID.
- The vertex value is not specified as it is an optional field.
- The edge tuple consists of the outgoing edges or destination vertex ID.
- The edge weight is not mentioned as it is an un-weighted graph.

The input is defined in a text file as follows:
1 4
2 8
3 6
4 7
5 2
6 9
7 1
8 6 5
9 7 3

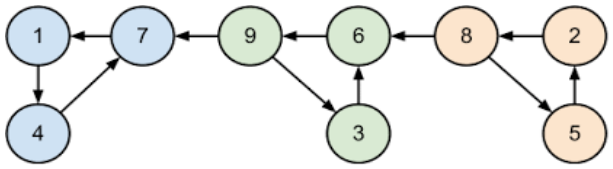


Fig. 4.2: Graphical Representation of the Input Graph.

4.3 Compute Phase

Our algorithm consists of multiple computational steps called "phases". For example, an algorithm might prune the graph in one phase and traverse it in another. A global object "phase"stores the current phase of the algorithm that is executing. The master.compute() function contains the logic of which phase should be executed in the next superstep, depending on the current phase and possibly other global objects.
The different phases to be performed by master compute () are as follows:

- Transpose Graph Formation
- Trimming
- Forward-Traversal
- Backward-Traversal

For each phase, the Vertex class contains one subroutine implementing the vertex-centric logic of the phase. According to the information found in phase global object vertex.compute() function calls the specific subroutine.

4.3.1. Generate transpose graph

The algorithm first constructs the transpose of the input graph G. This construction requires 2 supersteps where in each superstep is considered to be one iteration.

- In the first Superstep, each vertex creates an array of its parent vertices based on the traversal through the outgoing edges.
- In the second superstep, send the vertex ID and vertex value to its neighbours.

4.3.2. Trimming phase

In this phase the algorithm identifies the trivial SCCs i.e, vertices with only one incoming or onlyoutgoing edges or neither is regarded as disconnected component. This requires one superstep. Every vertex with only one incoming or only outgoing edges or neither sets its value to its own ID. Messages subsequently sent to the vertex are ignored. The remaining vertices propagate their values to their neighbours.

4.3.3. Forward-traversal phase

In the forward traversal phase, the algorithm traverses G in parallel from each vertex. During the traversals, each vertex v sets its own value as the maximum ID of the vertex that can reach v (possibly v itself). The Forward-Traversal phase has two properties:

- G is partitioned into disjoint sets of vertices according to their new vertex values.
- If the new maximum value is the value set for the partitioned vertices, then the vertex i belongs entirely to a particular SCCi.
- There are two subphases: Start and Rest.

4.3.3.1. Forward-traversal start phase

In the Start phase, each vertex sets its vertex value to its own ID and propagates its ID to its outgoing neighbors.

4.3.3.2. Forward-traversal rest phase

In the Rest phase, vertices update their own vertex values with the maximum value they have received, and propagate their values, if updated, until the maximum values converge.
The Master sets the phase global object to Backward-Traversal when the values converge.

4.3.4. Backward-traversal

In the Backward-Traversal phase, the algorithm detects one SCC for each maximum value set,by doing a traversal from vertex in the transpose of G and limiting the traversal to only the vertices in one SCC. The detected SCCs are then removed from the graph.
Here we again break the phase into Start and Rest.

4.3.4.1. Backward-traversal start phase

In Start, every vertex whose vertex ID equals its vertex value propagates its ID to the neighboring vertices.

4.3.4.2. backward-traversal rest phase

In each of the Rest phase super steps, each vertex receiving a message that matches its vertex value

- Propagates its vertex value in the transpose graph.
- Sets itself inactive.
- Sets the "converged vertex-exists" global object (false at the start of the superstep) to true. Messages subsequently sent to the vertex are ignored.

The Master sets the phase global object back to Trimming when "converged-vertex-exists" remains false at the end of a superstep.

4.4. Offloading phase

- Vertices are offloaded to HDFS through an OutputFormat.
- Output can be done both on a per-vertex and a per-edge basis: a VertexOutputFormat will specify what data to write for each vertex while EdgeOutputFormat will specify what
- data to write for each edge.
- In our implementation we make use of VertexOutputFormat.
- Giraph provides several built-in output formats such as:
- GraphvizOutputFormat
- GiraphTextOutputFormat
- TextEdgeOutputFormat
- IdWithValueTextOutputFormat
- InMemoryVertexOutputFormat
- JsonBase64VertexOutputFormat
- AdjacencyListTextVertexOutputFormat
- JsonLongDoubleFloatDoubleVertexOutputFormat
- LongDoubleDoubleAdjacencyListVertexOutputFormat

4.4.1. Strongly Connected components Output format

- The Output File should contain all vertices with the value of their vertex.
- Each line should have the VertexId and the maximum vertex value of the strongly connected component to which they belong.
- The VertexId and the vertex value should be of type int or double.
- In our algorithm we represent the output in the IdWithValueTextOutputFormat. This writes out vertices' IDs and values.

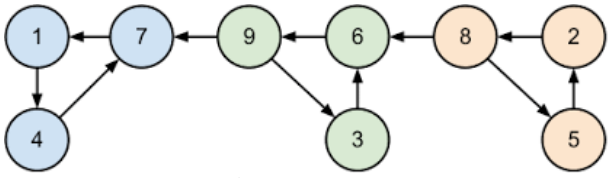


Fig. 4.3: Output of Strongly Connected Components.

The output would be present in the terminal as follows:
6 9
5 8
8 8
7 7
2 8
1 7
9 9
3 9
4 7

4.5. Execution in apache giraph framework

- In the hadoop directory run
- bin/start-dfs.sh
- bin/start-mapred.sh
- Create the input folder in the HDFS and move the input file there.
- hadoop dfs –put /input_file.txt /in/
- Run the command, in which you should include:
- The jar file generated when installing giraph,
- The path to the main code, The path to the code for reading the input file and the path to the input file,
- The path to the code for generating the output file and the path to the output file,
- The number of workers. Below I give all these parameters in the same order.

org.apache.giraph.GiraphRunner-Dgiraph.metrics.enable=true org.apache.giraph.examples.scc.SccComputation –vip slk-giraph/{input_file}.txt –vif org.apache.giraph.examples.scc.SccLongLongNullTextInput-Format -op slkgiraph/{ output_file} –vof org.apache.giraph.io.formats.IdWithValueTextOutputFormat -w 1 –ca giraph.zkList=orion-00:2181–ca giraph.checkpointFrequency=0 -yj giraphexamples-1.1.0-for-hadoop-2.6.0-jar-with-dependencies.jar-mc org.apache.giraph.examples.scc.SccPhaseMasterCompute

Run the command to check the output

hadoop dfs –cat /slk-giraph/{output_file}/part-m-00001

5. Results

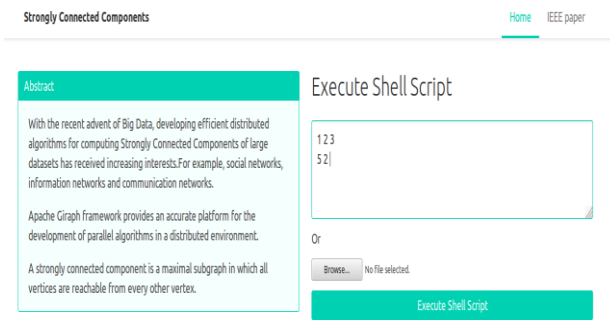


Fig. 5.1: Manual User Input.

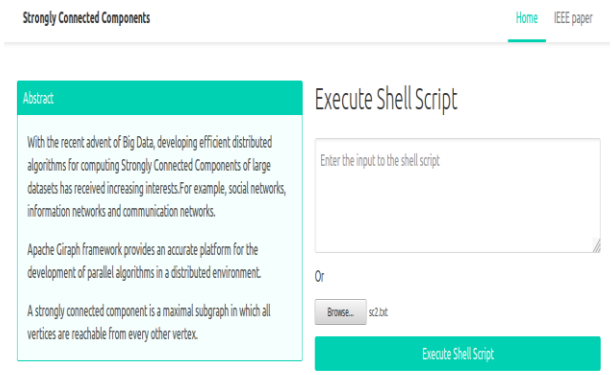


Fig. 5.2: Select File from Local System.



Fig. 5.3: Graph in Vertex Input Format.

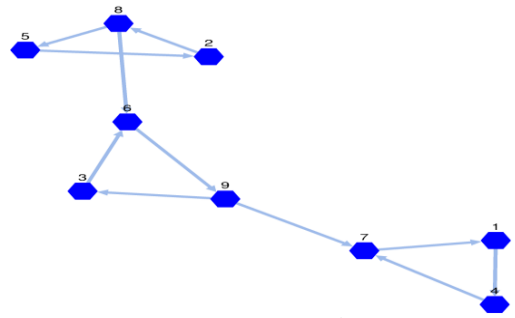


Fig. 5.4: Visual Representation of the Input Graph.

Output

DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.

6 9
5 8
8 8
7 7
2 8
1 7
9 9
3 9
4 7

Fig. 5.5: Output Representing Vertex ID and Value.

Strongly Connected Components Home IEEE paper

Input

1 2 3
2 4 5
3 4
4 1 7
5 6 7
6 7 8 9 10
7 5
8 10
9 10
10 9

Fig. 5.6: Graph in Vertex Input Format.

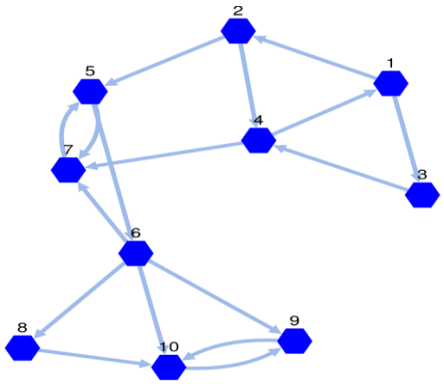


Fig. 5.7: Visual Representation of the Input Graph.

Output

DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.

6 7
5 7
8 8
7 7
2 4
9 10
10 10
1 4
3 4
4 4

Fig. 5.8: Output Representing Vertex ID and Value.

6. Conclusions

After several months of developing and testing our algorithm using a variety of approaches for distributed programming, we have finally developed a full-fledged parallel algorithm for computing strongly connected components. This algorithm is highly scalable and reliable due its development in Apache Giraph. The main target of our algorithm is to analyze the social graphs to identify the type of relationship among the entities such as weakly connected, strongly connected, interconnected or disconnected. Our algorithm has efficiently computed strongly connected components for large graphs up to 100 nodes and out-going edges of graphs. In the near future, our aim to enhance the scalability of our algorithm for large datasets such as amazon or orkut graphs with millions of vertices and edges. This will help us achieve greater efficiency in parallel programming and Big Data analysis. The algorithm must also be able to reduce the time computation cost by optimizing the algorithm based on Apache Giraph programming constructs. This method of distributed programming method can be extended to other algorithm development such as Bi-connected components and Graph coloring with appropriate analysis of the programming constructs and development tools.

References

[1] F. Pellegrini , “Current challenges in parallel graph partitioning”,
[2] Comptes Rendus Mécanique, vol. 339, no. 2-3, pp. 90-95, 2011.
[3] Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: ACM International Conference on the Management of Data (SIGMOD), ACM (2010) 135–146.
[4] Gregor, D., Lumsdaine, A.: The Parallel BGL: A Generic Library for Distributed Graph Computations. In: Parallel Object-Oriented Scientific Computing (POOSC). (2005).
[5] A. Lumsdaine, D. Gregor, B. Hendrickson and J. Berry, “Challenges in Parallel Graph Processing”, Parallel Processing Letters, vol. 17, no. 1, pp. 5-20, 2007.
[6] Ediger, D., Bader, D.: Investigating Graph Algorithms in the BSP Model on the Cray XMT. In: Workshop on Multithreaded Architectures and Applications (MTAAP). (2013).
[7] Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the gpu using cuda. In IEEE High performance computing (HiPC). (2007).
[8] Chen, R., Weng, X., He, B., Yang, M.: Large graph processing in the cloud. In: ACM International Conference on the Management of Data (SIGMOD), ACM (2010) 1123–1126.
[9] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, Proceedings of Sixth Symposium on Operating Systems Design and Implementation, San Francisco, California, USA, 2004.
[10] F. N. Afrati, A. Das Sarma, S. Salihoglu and J. D. Ullman, “Vision Paper: Towards an Understanding of the Limits of Map-Reduce Computation”, Proceedings of Cloud Futures 2012 Workshop, Berkeley, California, USA, 2012

[11] Sabeur Aridhia, Alberto Montresorb, Yannis Velegrakisb: BLADYG: A Graph Processing Framework for Large Dynamic Graphs, University of Lorraine, LORIA, Campus Scienti_que, BP 239, 54506 Vandoeuvre-l_es-Nancy, France University of Trento, Italy arXiv:1701.00546v1 [cs.DC] 2 Jan 2017.

[12] Arijit Khan, Sameh Elnikety: Systems for BigGraphs, 40th International Conference on Very Large Data Bases, September 1st 5th 2014, Hangzhou, China. Proceedings of the VLDB Endowment, Vol. 7, No. 13.