DISTRIBUTED GRAPH PROCESSING: GIRAPH vs GELLY

Georgios Kosmas, National Technical University of Athens

I. Introduction

In our times, an ever-increasing number of practical computer applications are related to graphs, since innumerable real-life situations can be modeled in the form of vertices and edges. Subsequently, crucial insights are hidden inside massive graphs such as social networks, transportation networks or knowledge graphs. Analyzing these graphs and extracting information from them allows us to gain valuable understanding regarding the behaviour of social, economical or computer systems.

The great demand for efficient processing of huge graph datasets led to the development of software projects that specialize in graph processing. Both single-machine and distributed computing approaches have been successfully tried. In this work, we will focus on the comparison of two distributed graph processing systems, namely Apache Giraph and Apache Flink (Gelly).

The primary goal of this assignment is to compare the performance of the aforementioned systems. Our objective to understand how Giraph and Gelly scale in regard to dataset size and cluster size. In order to do so, we will execute graph algorithms on inputs of varying size using distributed clusters of different sizes. During the project we aim to get familiar with the distributed graph processing ecosystem, to learn how to implement graph algorithms using the vertex-centring approach and to gain experience in deploying them in a distributed environment.

II. Code Repository

The code that was written for this project can be found at the following Github repository:

http://github.com/gkosm314/giraph-vs-gelly

III. Setup Guide

A. Cluster Description

A cluster with four VMs will be used. One of them will act as the master, while the other three will act as slaves. The four VMs are connected to the same local network, but among them only the master is connected to the Internet. As hostnames, we will use 'master' for the master and 'one','two','three' for the slaves. Regarding the technical specifications of the machines, each node has 4 GB of RAM memory and 30 GB of disk storage. Furthermore, the three slave nodes have 4 CPUs each, while the master node has 2 CPUs. All the VMs run Ubuntu 16.04.3 LTS.

B. Initial Setup

Steps 1-3 should be executed at every node of the cluster:

1) Install Java 1.8:
   sudo apt-get install openjdk-8-jdk-headless

2) Change hostname (master/one/two/three respectively) of each machine with:
   sudo hostname new-hostname-here
   sudo vi /etc/hostname new-hostname-here

3) Update /etc/hosts with *sudo vi /etc/hosts* and add the following ip-hostname mappings:

   192.168.0.1 master
   192.168.0.2 one
   192.168.0.3 two
   192.168.0.4 three

   If a line of the form *127.0.1.1 snf-32538* exists, comment it out: *#127.0.1.1 snf-32538*

   Steps 4-9 should be executed at the master node only:

4) Setup passworldless SSH between the nodes:
ssh-keygen -t rsa -P ” -f /home/user/.ssh/id_rsa
cat /home/user/.ssh/id_rsa.pub >> /home/user/.ssh/authorized_keys
scp -r .ssh/ user@one:/home/user/
scp -r .ssh/ user@two:/home/user/
scp -r .ssh/ user@three:/home/user/

5) Download zstd to unzip datasets:
sudo apt-get install zstd

6) Create a directory for the datasets and then download and unzip a dataset to experiment on:
mkdir /home/user/datasets_local
mkdir /home/user/datasets_local/datagen-7_6
cd /home/user/datasets_local/datagen-7_6
wget -O datagen-7_6-fb.tar.zst https://surfdrive.surf.nl/files/index.php/s/pxl7rDvzDQJFhfc/download
tar –use-compress-program=unzstd -xvf datagen-7_6-fb.tar.zst
cd /home/user

7) Download the project's code:
git clone https://github.com/gkosm314/giraph-vs-gelly

8) Download maven to build Apache Giraph and Apache Flink:
sudo apt-get install maven

9) Create directories to store measurements:
mkdir measurements
mkdir measurements/giraph_measurements
mkdir measurements/gelly_measurements

C. Apache Giraph Setup

Step 1 and 2 should be executed at every node of the cluster:

1) Create a new file for enviroment variables: *cp /home/user/.bashrc /home/user/.bashrc_giraph*,
then append the contents of /home/user/giraph-vs-gelly/conf/envvars/giraph_env_vars at the end of /home/user/.bashrc_giraph
Finally, run *source /home/user/.bashrc_giraph*.

2) Download and unzip Apache Hadoop 1.2.1 binary:
wget https://archive.apache.org/dist/hadoop/core/hadoop-1.2.1/hadoop-1.2.1-bin.tar.gz
tar -xvf hadoop-1.2.1-bin.tar.gz

Steps 3-12 should be executed at the master node only:

3) Configure Hadoop by replacing the following files with the equivelant files from the directory /home/user/giraph-vs-gelly/conf/hadoop_giraph_conf_files/
   a) *hadoop-1.2.1/conf/core-site.xml*
   b) *hadoop-1.2.1/conf/hdfs-site.xml*
   c) *hadoop-1.2.1/conf/mapred-site.xml*
   d) *hadoop-1.2.1/conf/master*
   e) *hadoop-1.2.1/conf/slaves*
   f) *hadoop-1.2.1/conf/hadoop-env.sh*

4) Copy hadoop-1.2.1/conf to all the slaves:
scp -r /home/user/hadoop-1.2.1/conf/ user@one:/home/user/hadoop-1.2.1/
scp -r /home/user/hadoop-1.2.1/conf/ user@two:/home/user/hadoop-1.2.1/
scp -r /home/user/hadoop-1.2.1/conf/ user@three:/home/user/hadoop-1.2.1/

5) Format hdfs namenode:
hadoop namenode -format

6) Download Giraph 1.3.0 for Hadoop 1 source and then unzip it with:
wget https://dlcdn.apache.org/giraph/giraph-1.3.0/giraph-dist-1.3.0-hadoop1-src.tar.gz –no-check-certificate
tar -xvf giraph-dist-1.3.0-hadoop1-src.tar.gz

7) Copy our algorithm implementations inside the source with:
*cp -r /home/user/giraph-vs-gelly/giraph_code/ *
/home/user/giraph-1.3.0-hadoop1/giraph-examples/src/main/java/org/apache/giraph/examples/

8) Build Giraph and copy the binary version at home:
cd /home/user/giraph-1.3.0-hadoop1/
mvn -Phadoop_1 -DskipTests -Dcheckstyle.skip clean install
cd /home/user
cp -r /home/user/giraph-1.3.0-hadoop1/giraph-dist/target/giraph-1.3.0-hadoop1-for-hadoop-1.2.1-bin/giraph-1.3.0-hadoop1-for-hadoop-1.2.1 /home/user/giraph-1.3.0-hadoop1-for-hadoop-1.2.1

Important note: use Java 1.8, otherwise the build will fail

9) Start hdfs and map-reduce:
./hadoop-1.2.1/bin/start-dfs.sh
./hadoop-1.2.1/bin/start-mapred.sh

10) Upload the dataset on HDFS:
hadoop fs -copyFromLocal /home/user/datasets_local/datagen-7_6/datagen-7_6-fb.v hdfs://master:9000/user/datasets/small.v
hadoop fs -copyFromLocal /home/user/datasets_local/datagen-7_6/datagen-7_6-fb.e hdfs://master:9000/user/datasets/small.e

11) Run tests from giraph-vs-gelly/scripts/run_all_giraph.sh

12) Stop hdfs and map-reduce with:
./hadoop-1.2.1/bin/stop-dfs.sh
./hadoop-1.2.1/bin/stop-mapred.sh

D. Apache Flink (Gelly) Setup

Steps 1 and 2 should be executed at every node of the cluster:

1) Create a new file for enviroment variables: *cp /home/user/.bashrc /home/user/.bashrc_flink*,
then append the contents of /home/user/giraph-vs-gelly/conf/envvars/flink_env_vars at the end of /home/user/.bashrc_flink
Finally, run *source /home/user/.bashrc_flink*.

2) Download and unzip Apache Hadoop 2.8.3 binary:
wget https://archive.apache.org/dist/hadoop/common/hadoop-2.8.3/hadoop-2.8.3.tar.gz
tar -xvf hadoop-2.8.3.tar.gz

Steps 3-12 should be executed at the master node only:

3) Configure Hadoop by replacing the following files with the equivelant files from the directory /home/user/giraph-vs-gelly/conf/hadoop_flink_conf_files/
   a) *hadoop-2.8.3/etc/hadoop/core-site.xml*
   b) *hadoop-2.8.3/etc/hadoop/hdfs-site.xml*
   c) *hadoop-2.8.3/etc/hadoop/slaves*
   d) *hadoop-2.8.3/etc/hadoop/hadoop-env.sh*

4) Copy hadoop-2.8.3/etc/hadoop to all the slaves:
scp -r /home/user/hadoop-2.8.3/etc/hadoop user@one:/home/user/hadoop-2.8.3/etc

scp -r /home/user/hadoop-2.8.3/etc/hadoop user@two:/home/user/hadoop-2.8.3/etc
scp -r /home/user/hadoop-2.8.3/etc/hadoop user@three:/home/user/hadoop-2.8.3/etc

5) Format hdfs namenode:
hdfs namenode -format

6) Download Apache Flink 1.14.4 binary and then unzip it:
wget https://archive.apache.org/dist/flink/flink-1.14.4/flink-1.14.4-bin-scala_2.11.tgz
tar -xvf flink-1.14.4-bin-scala_2.11.tgz

7) Configure Apache Flink by applying the following changes:
   a) Replace the content of the /home/user/flink-1.14.4/conf/masters with the following:
   master
   b) Replace the content of the /home/user/flink-1.14.4/conf/slaves with the following:
   one
   two
   three
   c) In /home/user/flink-1.14.4/conf/flink-conf.yaml, change the value of *jobmanager.rpc.address* to *master*.

8) Download pre-bundled Hadoop 2.8.3 library for Apache Flink and move it to flink-1.14.4/lib/ :
wget https://repo.maven.apache.org/maven2/org/apache/flink/flink-shaded-hadoop-2-uber/2.8.3-10.0/flink-shaded-hadoop-2-uber-2.8.3-10.0.jar
mv flink-shaded-hadoop-2-uber-2.8.3-10.0.jar flink-1.14.4/lib/

9) Copy flink-1.14.4/ to all the slaves :
scp -r /home/user/flink-1.14.4/ user@one:/home/user/
scp -r /home/user/flink-1.14.4/ user@two:/home/user/
scp -r /home/user/flink-1.14.4/ user@three:/home/user/

10) Create a new Apache Flink java project structure.
When asked, type: group-id=gelly-algorithms, artifact-id=gelly-algorithms, package=gellyalgos.

```
mvn archetype:generate \
-DarchetypeGroupId=org.apache.flink \
-DarchetypeArtifactId=flink-quickstart-java \
-DarchetypeVersion=1.14.4
```

Note: it is very important to name the package 'gellyalgos' in order for our implementations to compile.

11) Edit gelly-algorithms/pom.xml by:
   a) add the following dependecy before the '</dependencies>' tag:
   <dependency>
   <groupId>org.apache.flink</groupId>
   <artifactId>flink-gelly_2.11</artifactId>
   <version>1.14.4</version>
   </dependency>

   b) change main class from
   <mainClass>gelly-algorithms.StreamingJob</mainClass>
   to
   <mainClass>gelly-algorithms.BatchJob</mainClass>

12) Copy our algorithm implementations (and remove unnecessary files) inside the newly created project:
cp /home/user/giraph-vs-gelly/gelly_code/ /home/user/gelly-algorithms/src/main/java/gellyalgos/*
rm /home/user/gelly-algorithms/src/main/java/gellyalgos/StreamingJob.java

13) Compile our implementations:

cd /home/user/gelly-algorithms
mvn clean package
cd /home/user

14) Start hdfs:
./hadoop-2.8.3/sbin/start-dfs.sh

15) Upload the dataset on HDFS:
hdfs dfs -mkdir hdfs://master:9100/user/
hdfs dfs -mkdir hdfs://master:9100/user/datasets
hdfs dfs -mkdir hdfs://master:9100/user/outputs
hdfs dfs -copyFromLocal /home/user/datasets_local/datagen-7_6/datagen-7_6-fb.v hdfs://master:9100/user/datasets/small.v
hdfs dfs -copyFromLocal /home/user/datasets_local/datagen-7_6/datagen-7_6-fb.e hdfs://master:9100/user/datasets/small.e

16) Start Apache Flink standalone cluster:
./flink-1.14.4/bin/start-cluster.sh

17) Run tests from giraph-vs-gelly/scripts/run_all_gelly.sh

18) Stop hdfs and Flink standalone cluster:
./flink-1.14.4/bin/stop-cluster.sh
./hadoop-1.2.1/bin/stop-dfs.sh

IV. Software

A. Apache Giraph

1) Introduction: Apache Giraph is an open-source graph processing framework. Giraph began as an open-source implementation of Pregel [1], which is a graph-processing framework that Google proposed in a research paper published in 2010. Giraph extends Pregel by implementing extra functionalities such as master computation, additional I/O formats and out-of-core processing. Nowadays, Giraph is being utilized by many tech giants. As an example, a team of Facebook engineers published a paper in which they described how they managed to achieve trillion-edge-scale computations with Apache Giraph [3].

Giraph is directed towards offline(batch) computations designed to run periodically for long periods of time, as opposed to online computations, which have the form of quick interactive queries. As a rule of thumb, if your computation explores most of the graph's vertices, Giraph may be a good option. Otherwise, if your computation only touches a small portion of the graph, a graph database like Neo4j may be a better option [2].

Apache Giraph runs on top of Apache Hadoop, submitting its computations as Map-Reduce jobs. However, since graph algorithms are usually iterative algorithms, they do not fully take advantage of the Map-Reduce framework. Instead, Giraph submit map-only jobs that are equivalent to the . In other words, Giraph mainly uses Hadoop Map-Reduce for the coordination the distributed computation. [2]

2) Programming Paradigm - Vertex Centric: Giraph essentially implements the vertex-centric paradigm proposed by Pregel [2]. According to this approach, the algorithm is executed in iterative rounds that are called "supersteps". At the start of each superstep, every vertex receives messages from the vertices with which it is connected. Next, every vertex performs a computation based on the messages it received at the beginning of the computation. After its computation is done, each vertex can send messages to other vertices that are connected with it. Finally, every vertex decides if it wants to "sleep". Vertices that sleep are woken up only if they receive a message. The algorithm terminates when all vertices are asleep.

When using Giraph the the user has to "think like a vertex" and define a function that will define how each vertex will behave during each superstep. To put it in another way, the developer writes a user-defined function that will read the messages a vertex received, perform a computation and then send some messages to some of the vertices that belong neighborhood of the vertex. The vertex-centric model may seem limiting at first. However, it comes with many advantages, since it allows the programmer to develop algorithms that are guaranteed to be decentralized and scalable. In more detail, each vertex performs independent computations based on its own data. As a result the algorithm can exploit the benefits of parallelization. At the same time, the algorithms developed under the vertex-centric approach are simple and intuitive. [2]

3) Combiners: Messages are the core of Giraph's algorithms, because they allow the communication between the different vertices. At the same time, they are one of the most crucial performance bottlenecks, due to the fact that they often have to travel between machines through the network, in case the two vertices do not reside in the same node of the cluster.

In order to reduce the overhead that comes with the transmission of messages, Giraph allows the user to implement combiners. Combiners are user-defined functions that combine two messages into one. In this way, the volume of the messages that will be transmitted through the network decreases substantially. The utilization of combiners visibly improves an algorithm's performance. This can be observed in the results of our experiments as well. As is the case with the classical Map-Reduce combiners, a user-defined function that implements a combiner must be commutative and associative, because the framework does not provide any guarantees in regard to the order that the combiner will be applied to the messages.

4) Aggregators: Giraph also includes aggregators, which are a tool that enables the user to perform aggregations over global data. Aggregators are user-defined functions to which vertices can send values at the end of each superstep. The vertices can access the results of the aggregator at the start of the next superstep. Like combiners, aggregators must be commutative and associative. [2]

5) Architecture: A node may provide one or more of the following services to a Giraph computation:

- Master: They participate in the computation by coordinating it. Their primary responsibility is handling the graph partitioning and the coordination of the supersteps.
- Worker: They participate in the coordination by executing the computations . Most of the nodes fall into this category.
- Coordinator: They do not participate in the computation, but they take over the coordination of the distributed cluster. They usually are implemented with Apache Zookeper.

6) Synchronous Computation: A superstep is over only when all vertices have finished their computations and have sent their messages. In other words, the computation is synchronous, meaning that all "awake" vertices wait until every other vertex is done with superstep N before they start with superstep $N + 1$.

The synchronization model of Apache Giraph is very similar to the classical Batch Synchronous Parallel (BSP) model [4]. Despite that, it should be noted that the model Apache Giraph follows is not identical to BSP. This is due to the fact that Giraph vertices begin sending messages to the other vertices as soon as they are produced. In contrast to that, the BSP's computation units start sending messages once their computation is finished. This slight difference allows Giraph to spread the network workload of a superstep over a longer period of time. [2]

Yet, graphs obtained from real life applications tend to present skew when it comes to degree distribution. For example, in a road network most towns would have few edges, but a small number of large cities would have noticeably higher number of edges connected to them. As we can see, the low-degree vertices, which are the vast majority of the vertices, would be enforced to wait for the few high-degree vertices to finish their heavy computations. [5] Consequently, according to a published comparison between graph processing frameworks [5], Giraph's synchronization phase is characterized by low CPU activity.

GiraphUC [6] is a research attempt to address these particular problems. Specifically, GiraphUC is an asynchronous version of Giraph, whose creators propose a new model, the barrierless asynchronous parallel (BAP) model, in contrast to the bulk synchronous parallel (BSP) model.

7) Out-Of-Core: In the beginning, Giraph was designed to run the whole computation in-memory. The disk storage was only used for I/O and checkpointing. However, use cases exists where the graph's scale is disproportionately larger than the available memory. In addition to this, some algorithms produce messages of substantial size. With intention to solve these issues, Giraph introduces out-of-core processing. Out-of-core processing allows Giraph to use the hard disk as swap space to temporarily store graph partitions or messages. The user does not have to worry about changing the algorithms he developed, since swapping is being handled by the system alone. Unfortunately, as expected, disk swaps leads to significantly worse I/O performance.

For algorithms that do not change the graph structure, such as SSSP and PageRank, we do not have to rewrite the vertices and the edges to the disk. Thus, after every superstep, only the updated vertex values are being re-written. This trick gives Giraph the opportunity to limit the impact that the out-of-core computation has on I/O performance, when an algorithm that does not apply mutations is being executed. [2]

8) Scaling: As we previously described, messages are the limiting factor when it comes to Apache Giraph programs, due to the increased network I/O. Due to this, Apache Giraph applications perform better when the cluster consists of few big machines, as opposed to a lot of smaller machines, since the larger number of nodes in the cluster comes with higher network I/O. Therefore scaling up(vertical scaling) is preferred to scaling out(horizontal scaling) when the main purpose of a cluster is to run Apache Giraph applications. Furthermore, homogeneous clusters are preferable to heterogeneous clusters, because of the synchronized nature of Giraph, which forces it to wait for the slowest machine to finish. [2]

9) Giraph++: Giraph++ [7] is an interesting extension of Apache Giraph that proposes a new programming paradigm called "think-like-a-graph". The creators of Giraph++ claim that, for a number of graph algorithms, the vertex-centric approach under-performs because it does not take into account information related to the way the graph is partitioned. Giraph++ enables

the developers to exploit graph partitioning to their advantage by giving vertices access to all the data(value,messages,edges) of other vertices in the same partition. Essentially, in each superstep, the local computation is performed on the whole sub-graph and the messages between vertices are replaced by messages between sub-graphs.

B. Apache Flink (Gelly)

1) Introduction: Apache Flink is an open-source framework that can manage both streaming and batch workloads. In Flink, any kind of data is produced as a stream of events, which can be either bounded or unbounded. Flink deals with batch compilations as a special case of streaming computations. [12] Apache Flink can run on a standalone cluster, but it can also leverage well-known cluster resource managers such as Hadoop YARN or Kubernetes. [9] Flink applications execute dataflow programs, which describe how data flows between applications.

2) PACT: Apache Flink is an implementation of PACT [11], which is a generalized version of Map-Reduce. When working with PACT, the user writes a function which he/she passes to a PACT operator, together with an input dataset. Then, the PACT operator (e.g. map,reduce,...) applies the user-defined function to the data and returns the results. [11] By passing the output of one operator to the other, a dataflow is being created. PACT does not have rules regarding the order of the operations, in contrast to Map-Reduce. When writing Apache Flink applications, users actually write programs according to the PACT paradigm. Apache Flink analyzes the dataflow graph that represents the dataflow program the user wrote, in order to apply optimizations that will result in improved performance. [4]

3) Gelly: Gelly is a Graph API for Apache Flink. It is a library that includes functions and methods to ease graph processing with Flink. At the moment, Gelly implements three graph processing programming paradigms: vertex-centric, scatter-gather and gather-sum-apply. The philosophy behind these models is very much alike. Since the vertex-centric paradigm was described above, we will only describe the latter two in this section.

4) Programming Paradigm - Scatter-Gather: In the scatter-gather model, similarly to the vertex-centric model, the computation is expressed from the perspective of a vertex and the computation is executed in synchronized supersteps. Each superstep is divided into two phases:

- Scatter: the messages that the vertex will send to other vertices are produced
- Gather: the value of the vertex is updated according to the received messages

In Gelly, the user has to implement two user-defined functions, one for each superstep phase.

5) Programming Paradigm - Gather-Sum-Apply: Like the vertex-centring and the scatter-gather model, the gather-sum-apply model also performs computations in a synchronized matter, using supersteps. Each superstep is divided into three phases:

- Gather: for each edge of the vertex, a function that takes the edge value and the neighbour's value as parameter is called (in parallel for each edge)
- Sum: the values produced in the "Gather" are aggregated and a single value is produced
- Apply: the value of the vertex is updated by applying a function on the current value and the value produced by the "Sum" phase

In Gelly, the user has to implement three user-defined functions, one for each superstep phase.

6) Comparison of the three programming paradigms:

- The vertex-centric model is the most general model.
- The scatter-gather model separates messaging from local computation, which leads to programs that are intuitively easier to understand. The separation of the algorithm in two phases comes with an advantage and a disadvantage. It may result in better memory performance, because a vertex does not have access to its incoming and to its outcoming messages at the same time. However, many algorithms are much easier to implement when a vertex has concurrent access to both the incoming and the outcoming messages. In other words, the separation of messaging and computation has negative impact on the expressiveness. [10]
- The gather-sum-apply model should be preferred to the scatter-gather model if the "gather" phase involves large workload, because the gather-sum-apply model parallelizes the "gather" phase over the edges, while the scatter-gather model parallelizes the "gather" phase over the vertices. Consequently, the scatter-gather phase may lead to unbalanced workload for some nodes, since the vertices with larger degrees will be obligated to perform considerably more computations. [10]

7) Gelly Streaming: Gelly Streaming [8] is an experimental try based on Apache Flink that addresses single-pass graph streaming analytics.

V. Experimental Results

A. Experiments

In order to decide which algorithms should be chosen for this project, previous related work [4],[19] as well as several surveys [4],[14],[15],[16],[17],[18] were taken into account. After careful examination of the bibliography, I reached the conclusion that the LDBC Graphalytics benchmark was the most suitable, so I used it as a prototype for the structure of this project.

The following algorithms were selected as the basis of the comparsion between the two systems, so that they cover a broad and representative range of possible kinds of graph processing:

- In-Degree
- Out-Degree
- Single-Source Shortest Path (SSSP)
- PageRank
- Weakly Connected Components (WCC)
- Community Detection with Label Propagation (CDLP)
- Local Clustering Coefficient (LCC)

The datasets we used are part of the LDBC Graphalytics benchmark [20]. We worked with synthetically generated datasets that immitate the structure of a social netowrk.

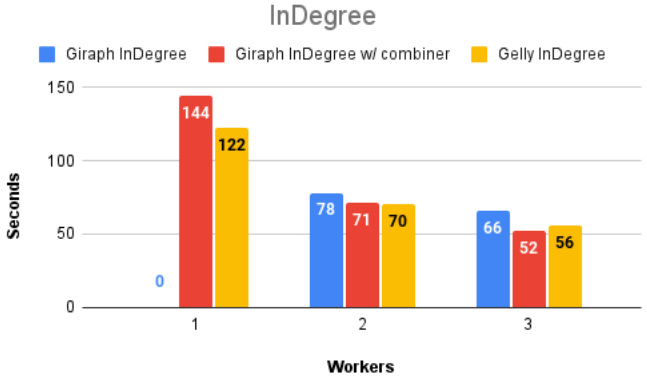
B. Implementations

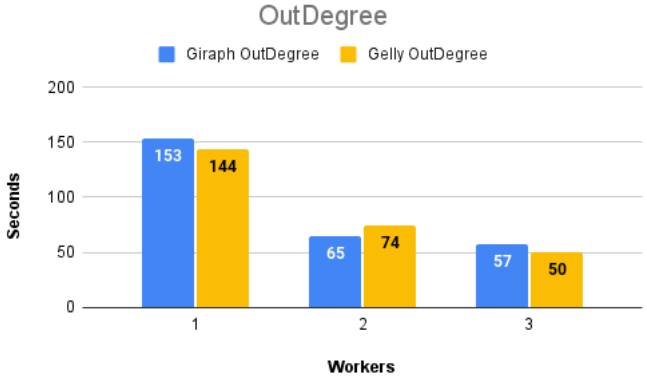
- For Giraph, I developed my own implementations for all the algorithms, the combiners and the input parsing. A large portion of my time was invested in this part of the assignment. My Giraph implementations can be found at the following link: https://github.com/gkosm314/giraph-vs-gelly/tree/main/giraph_code
- For Gelly, I used my own code for the I/O. Additionally, I wrote some mapping functions to initialize the vertex labels. For the algorithms, I used the built-in library methods. For more information regarding these built-in methods, refer to relevant documentation [13].
- Both systems read their input and write their output to a HDFS cluster. However, due to Hadoop version incompatibilities, in order to be able to run both systems simultaneously, I was forced to set-up two different HDFS clusters on the same computer cluster.

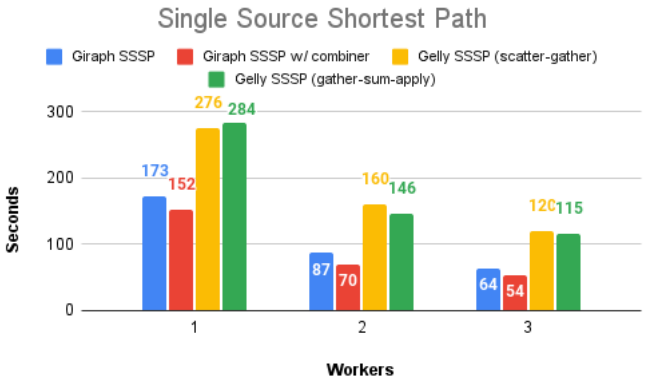
C. Plots

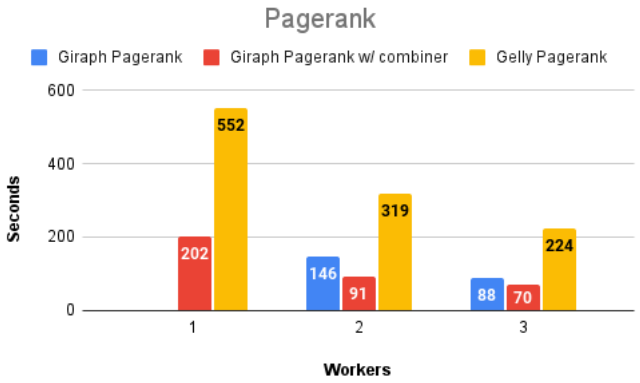
The results of the measurements for the dataset $datagen - 7\_6 - fb$, which has 754,147 vertices, 42,162,988 edges and a total uncompressed size of 1.1 GB, can be found below.

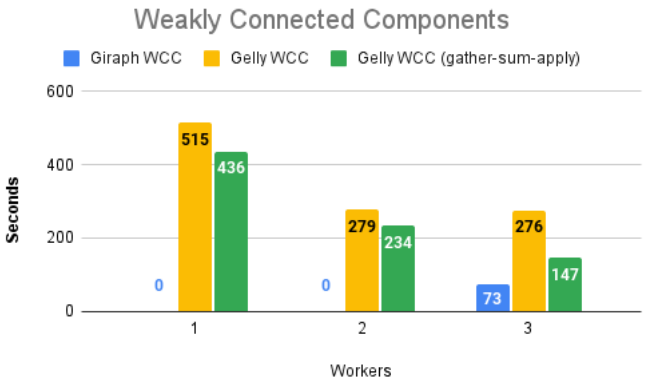
When the total number of seconds is zero, the computation was not completed because one of the nodes faced memory-related problem, usually related to the JVM heap running out. It should be noted that for Apache Giraph the out-of-core functionality was not utilized on purpose. In other words, Giraph was forced to run exclusively in-memory.

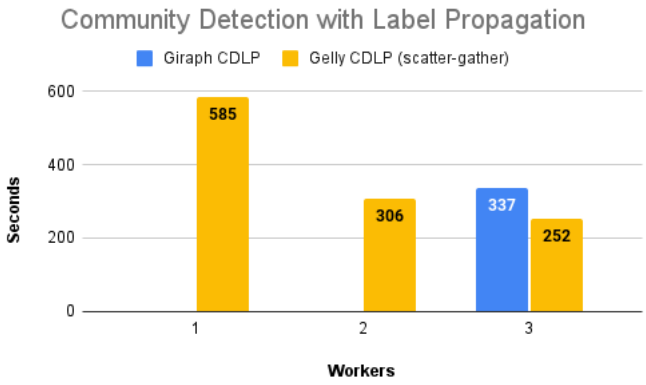


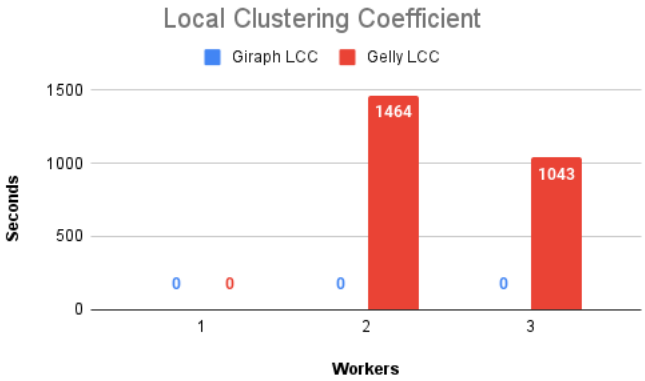


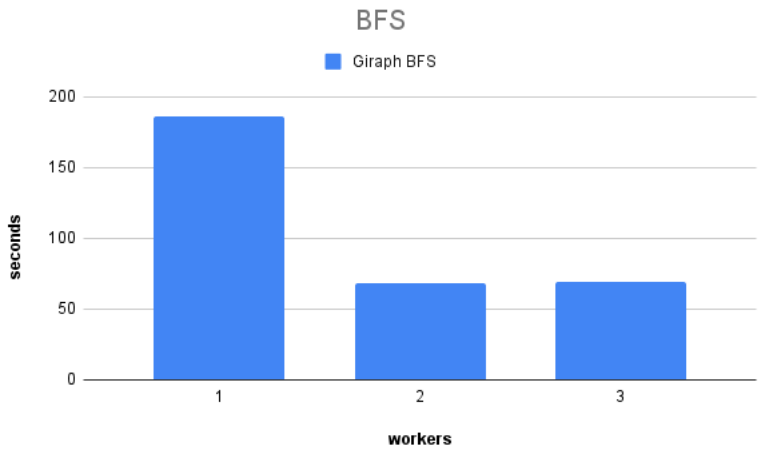




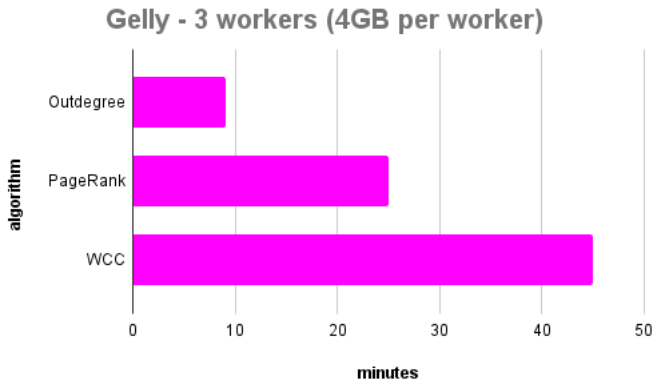








Tte results of some the measurements for the dataset $datagen - 8\_5 - fb$, which has 4,599,739 vertices, 332,026,902 edges and a total uncompressed size of approximately 10.3 GB, can be found below.



D. Notes

- The PageRank algorithm we implemented differs slightly from the PageRank variation that the LDBC Graphalytics specification suggests, since we do not redistribute the value of the vertices without outcoming edges.

  For all the tests that involved Apache Giraph, we always parsed the input and returned a directed and weighted graph. We made this strategic decision in order to stay as close as possible to the LDBC Graphalytics prototype and to the built-in Gelly implementations of the algorithms. If, however, we allowed other forms of input scanning, we could deliver better results. To be more precise:

- Since we parse the input as a directed graph, our WCC and CDLP implementations use their first two supersteps to convert the directed graph to an undirected graph. If we have parsed the the input as an undirected graph in the first place, we would have transferred this workload to the I/O part of the execution. Since converting a directed graph to an undirected graph using Pregel requires each vertex to send a message for each edge, it is clear that parsing the input as an undirected graph would grant better performance.

- What is more, converting the directed graph to an undirected graph involves adding edges during the execution. This fact forbids the use of the $-isStaticGraph = true$ flag that we described in the out-of-core section. Of course, this has severe impact on the performance of the WCC and the CDLP algorithm when the out-of-core functionality is enabled.

VI. Discussion

1) Performance Comparison: It is clear that Apache Giraph outperforms Gelly for the majority of the experiments. This can be explained by the fact that Apache Giraph worked exclusively in-memory, since the out-of-core functionality was not utilized during the experiments. This means that, whenever Apache Giraph could fit the graph and the messages inside the workers' memory, it had a distinct advantage over Apache Flink. This result aligns perfectly with the conclusion to which all previous works have reached.

2) Combiners impact on the performance: As expected, the positive impact of the combiners to the performance of Apache Giraph was confirmed. Moreover, another noteworthy advantage of the combiners became noticeable through the analysis of the experimental results. As we can see, when combiners were not used, Giraph failed to execute a number of tests where only one worker was available. However, when it was possible to utilize a combiner, Giraph managed to execute every test, even if only one worker was available. This is a practical demonstration of how combiners drive superior memory management by drastically restricting the volume of the messages.

3) In-memory execution: The aforementioned fact leads us to another observation. We can safely reach the conclusion that Giraph sometimes failed to finish the in-memory execution due to the volume of the messages and not due to the size of the graph. This can be understood if we notice that between the two executions (the one with a combiner and the one without a combiner) the only factor that changed was the memory needed to handle the messages, since the size of the graph was fixed.

We should also remark that when Giraph had more workers available, it automatically had more RAM storage available. This explains why in some experiments the in-memory execution failed when one worker was available but succeeded when more workers became involved in the computation.

4) Importance of memory configuration: After many trials, it was evident that one the key performance factors was the choices the user makes regarding the memory configuration of each framework, and specifically the configuration of JVM. The limited RAM storage (4GB) of the machines of the cluster made this observation even more evident, since slight changes would critical affect the algorithms flawless completion, let alone its performance.

5) Scatter-Gather vs Gather-Sum-Apply: Our experiments suggest that the gather-sum-apply implementations slightly outperforms the scatter-gather implementations. Nonetheless, we should not forget that the particular dataset we used for our experiments mimics a social-network graph. As a consequence, the degree distribution of the dataset will be heavily skewed. As we described in the subsection "Comparison of the three programming paradigms", in such cases the gather-sum-apply model will often outperform scatter-gather, thanks to the fact that it parallelizes the computation over the edges and thus it does not assign an unbalanced workload to the machines of the cluster.

6) Intermediate results: Through the execution of the experiments it was noticeable that it was the intermediate results of the algorithms and the size of the graph itself that made distributed processing solutions necessary. The user interface monitoring tool that Apache Flink provides enabled me to inspect the dataflow graph, where I could see that some of the

operators produced data that had considerably larger size than the size of the graph. An experiment that illustrates this fact perfectly is the Gelly execution of the Local Clustering Coefficient algorithm with one worker.

VII. Possible extensions of this work

Below some possible extensions of this work are described:

- Implementation of more sophisticated algorithms such as:
  - the SV algorithm for the Weakly Connected Components problem [22]
  - Strongly Connected Components (SCC) [23]
  - GHS algorithm for the Minimum Spanning Tree problem [24]
  - Luby's algorithm for the Maximal Independent Set problem [25]
  - Facility Location problem [26]
- Comparison of a shared memory single-machine system such as NetworkX [27] with a distributed graph processing platform for graphs that fit into single memory.

References

[1] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10). Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[2] Claudio Martella, Roman Shaposhnik, and Dionysios Logothetis. 2015. Practical Graph Analytics with Apache Giraph. Apress Berkeley, CA. https://doi.org/10.1007/978-1-4842-1251-6

[3] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: graph processing at Facebook-scale. Proc. VLDB Endow. 8, 12 (August 2015), 1804–1815. https://doi.org/10.14778/2824032.2824077

[4] Leslie G. Valiant. 1990. A bridging model for parallel computation. Commun. ACM 33, 8 (Aug. 1990), 103–111. https://doi.org/10.1145/79173.79181

[5] Koch, J., Staudt, C.L., Vogel, M. et al. An empirical comparison of Big Graph frameworks in the context of network analysis. Soc. Netw. Anal. Min. 6, 84 (2016). https://doi.org/10.1007/s13278-016-0394-1

[6] Minyang Han and Khuzaima Daudjee. 2015. Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. Proc. VLDB Endow. 8, 9 (May 2015), 950–961. https://doi.org/10.14778/2777598.2777604

[7] Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J. From Think Like a Vertex to Think Like a Graph. Proceedings of the VLDB Endowment, 7(3). (2013)

[8] *https://github.com/vasia/gelly-streaming* , Retrieved September 21, 2022

[9] *https://flink.apache.org/flink-architecture.html* , Retrieved September 21, 2022

[10] *https://nightlies.apache.org/flink/flink-docs-master/docs/libs/gelly/iterative_graph_processing/* , Retrieved September 21, 2022

[11] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. 2010. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In Proceedings of the 1st ACM symposium on Cloud computing (SoCC '10). Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/1807128.1807148

[12] *https://www.ververica.com/blog/batch-is-a-special-case-of-streaming* , Retrieved September 21, 2022

[13] *https://nightlies.apache.org/flink/flink-docs-master/docs/libs/gelly/library_method/* , Retrieved September 21, 2022

[14] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. 2014. An experimental comparison of pregel-like graph processing systems. Proc. VLDB Endow. 7, 12 (August 2014), 1047–1058. https://doi.org/10.14778/2732977.2732980

[15] Coimbra, M. E., Francisco, A. P., and Veiga, L. (2021). An analysis of the graph processing landscape. Journal of big data, 8(1), 55. https://doi.org/10.1186/s40537-021-00443-9

[16] Khaled Ammar and M. Tamer Özsu. 2018. Experimental analysis of distributed graph systems. Proc. VLDB Endow. 11, 10 (June 2018), 1151–1164. https://doi.org/10.14778/3231751.3231764

[17] V. Kalavri, V. Vlassov and S. Haridi, "High-Level Programming Abstractions for Distributed Graph Processing" in IEEE Transactions on Knowledge and Data Engineering, vol. 30, no. 2, pp. 305-324, Feb. 1 2018.

[18] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale distributed graph computing systems: an experimental evaluation. Proc. VLDB Endow. 8, 3 (November 2014), 281–292. https://doi.org/10.14778/2735508.2735517

[19] The GAP Benchmark Suite, Scott Beamer, Krste Asanović, and David Patterson, arXiv:1508.03619 [cs.DC], 2015, http://arxiv.org/abs/1508.03619

[20] *https://ldbcouncil.org/benchmarks/graphalytics/*, Retrieved September 21, 2022

[21] LDBC Benchmark Technical Specification, *https://arxiv.org/pdf/2011.15028v4.pdf* , Retrieved September 21, 2022

[22] Y. Shiloach and U. Vishkin. An o(log n) parallel connectivity algorithm. J. Algorithms, 3(1):57–67, 1982.

[23] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation. In Proceedings of the 24th International Conference on World Wide Web (WWW '15). International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1307–1317. https://doi.org/10.1145/2736277.2741096

[24] M. Ahuja and Y. Zhu, "A distributed algorithm for minimum weight spanning trees based on echo algorithms," [1989] Proceedings. The 9th International Conference on Distributed Computing Systems, 1989, pp. 2-8, doi: 10.1109/ICDCS.1989.37923.

[25] M Luby. 1985. A simple parallel algorithm for the maximal independent set problem. In Proceedings of the seventeenth annual ACM symposium on Theory of computing (STOC '85). Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/22145.22146

[26] Kiran Garimella, Gianmarco De Francisci Morales, Aristides Gionis, and Mauro Sozio. 2015. Scalable Facility Location for Massive Graphs on Pregel-like Systems. In Proceedings of the 24th ACM International on Conference on Information and Knowledge Management (CIKM '15). Association for Computing Machinery, New York, NY, USA, 273–282. https://doi.org/10.1145/2806416.2806508

[27] *https://networkx.org/* , Retrieved September 21, 2022