Distributed Graph Processing: Giraph vs Gelly

Analysis and Design of Information Systems 2021-2022

Giorgos Kosmas - 03118071

Introduction

Why graph processing?

- innumerable real-life situations can be modeled in the form of vertices and edges
- crucial insights are hidden inside massive graphs such as social networks, transportation networks or knowledge graphs
- extracting information from graphs allows us to gain valuable understanding regarding the behaviour of social, economical or computer systems

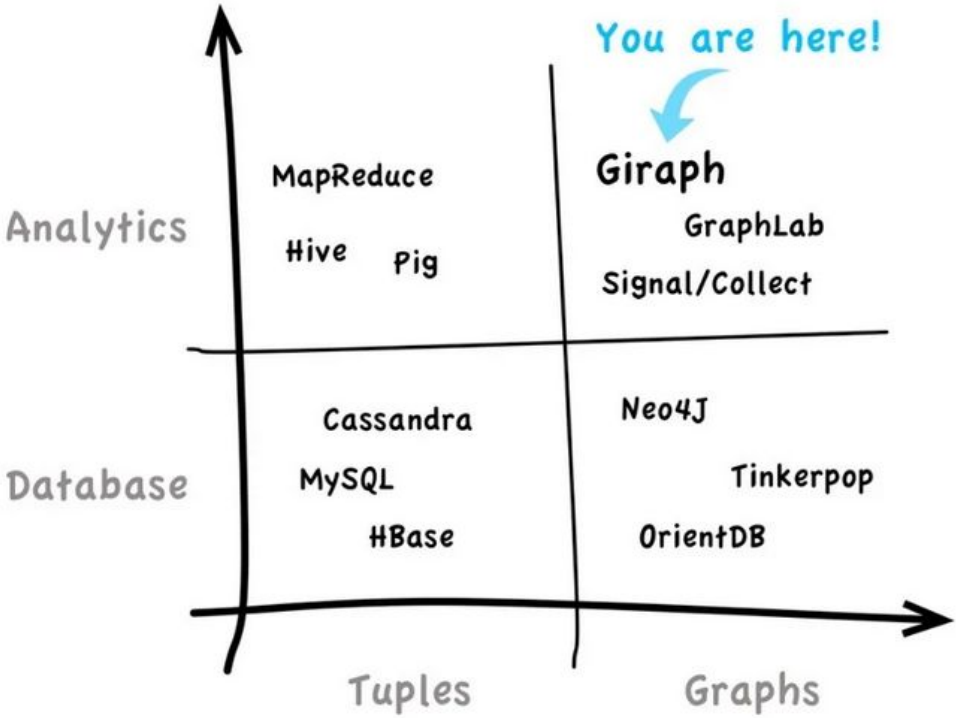
Why **distributed** graph processing?

Why don't we just use single-machine solutions? Most datasets fit in my RAM memory anyway!

- graphs are stored in distributed storage, so bringing them to a single node, performing the computation and sending them back would take more time than running a distributed graph processing algorithm
- the intermediate results of graph algorithms can get huge

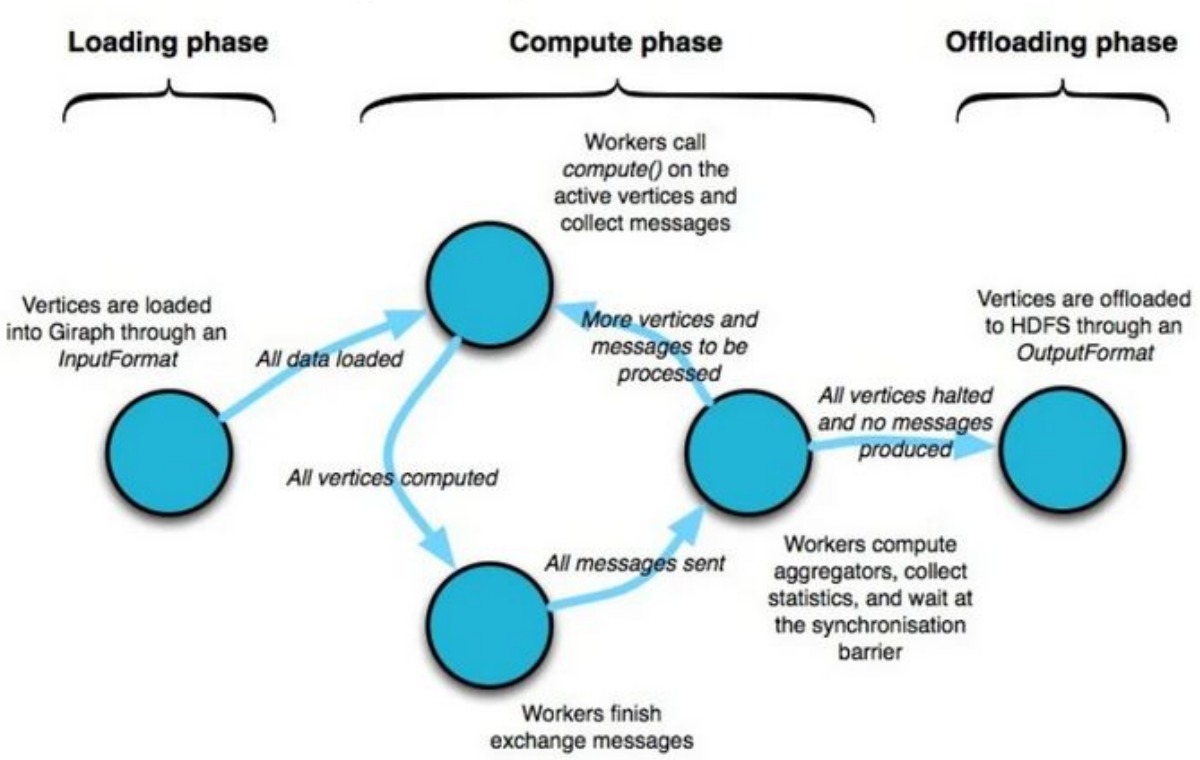
Apache Giraph - (1)

- graph processing framework suitable for batch (offline) computations
- Giraph applications are submitted as regular Apache Hadoop jobs
- open-source implementation of Google Pregel


You are here!
Giraph
GraphLab
Signal/Collect
MapReduce
Hive
Pig
Analytics
Cassandra
MySQL
HBase
Neo4J
Tinkerpop
OrientDB
Database
Tuples
Graphs

Apache Giraph - (2)

- vertex-centric model / “think-like-a-vertex”
- combiners and aggregators
- (almost) follows bulk synchronous paradigm (BSP)



A Giraph superstep

Apache Giraph - Extras

- Out-of-core functionality - both for the graph and the messages
- Scaling - vertical scaling prefered to horizontal scaling due to network I/o
- Giraph++ - “think-like-a-graph”
- GiraphUC - a research attempt for an asynchronous version of Giraph

Apache Flink

- open-source framework suitable for both streaming and batch applications
  - “batch is a special case of streaming” - Kostas Tzoumas, Apache Flink commiter
- implementation of PACT, which is a generalized version of Map-Reduce
  - does not force specific order of operators, in contrast to classical Map-Reduce
  - users write “dataflow” program by feeding operators with user-defined functwhich Apache Flink passes through an optimizer
- can run standalone/with Hadoop YARN/with Kubernetes

Gelly

- Gelly: Graph API for Apache Flink
- vertex-centric, scatter-gather and gather-sum-apply models
- Gelly Steaming - experimental try that addresses single-pass graph streaming analytics

Preparation

Readings

- Practical Graph Analytics with Apache Giraph - very useful
- Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications - nicely written, but not so relevant to this project

Readings

- A lot of published surveys and benchmarks to understand what I will try to do and to get a first glance of the distributed graph processing ecosystem
- **LDBC Graphalytics** benchmark seemed the most suitable, but most benchmarks had quite similar approaches anyway

Our algorithms

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

\+ Breadth First Search for Apache Giraph

Some more sophisticated algorithms…

- SV algorithm for the Weakly Connected Components problem
- Strongly Connected Components (SCC)
- GHS algorithm for the Minimum Spanning Tree problem
- Luby’s algorithm for the Maximal Independent Set problem
- Facility Location problem

Implementation

Implementations - (1)

- For Giraph: I invested a large part of time in writing my own implementations for the eight algorithms, as well as for the combiners I used .
- 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.

Implementations - (2)

Spending time implementing everything for Apache Giraph was time-consuming, but and it taught me:

- how to navigate on large codebases I did not wrote
- how to “think-like-a-vertex”!

Deployment

My cluster

4 VMs from okeanos-knossos, one master and three slaves:

- master - 2xCPU, 4 GB RAM, 30GB disk storage
- slaves - 4xCPU, 4 GB RAM, 30GB disk storage

Distributed storage = HDFS

I used HDFS as the distributed storage for both software systems. However,...

Version incompatibilities!

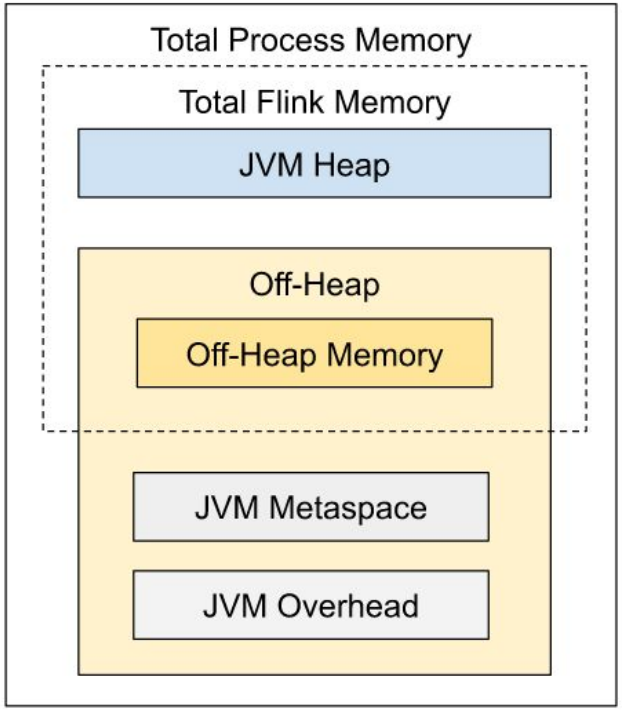
I was forced to setup two different HDFS clusters running on the same computer cluster.

I spent a lot of of time configuring Hadoop to be able to achieve that.

Memory configuration

- mapred.child.java.opts
- taskmanager.memory.flink.size
- ….

With 4GB RAM, JVM heap size was a problem.



Quiz: How to upload large input directly to HDFS?

Solution: use streaming!

Stream the data using linux pipes. One process (scp) receives the data from the remote server and immediately dumps them into the pipe for the other process (hdfs -put) to upload them to the distributed file system.

Results

Performance Comparison

- Giraph outperforms Gelly for the vast majority of the experiments
- Out-of-core functionality of Apache Giraph was not used ⇒
  - when the graph and the messages fitted into main memory, Apache Giraph performed an in-memory execution
  - when the graph and/or the messages did not fit into main memory, Apache Giraph could not complete the execution
- This results aligns with what we expected, according to the bibliography.

Combiners impact on the performance

- Combiners, as expected, improve the performance of Apache Giraph
- Combiners restrict the volume of the messages :
    - without combiner: Giraph failed because messages could not fit into memory
    - with combiner: Giraph did not fail because the messages were far less

This is a practical demonstration of how combiners drive superior memory management by drastically restricting the volume of the messages.

In-memory execution

- 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.
- 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.
- When Giraph had more workers available, it automatically had more RAM storage available, that is why in some experiments the in-memory execution failed when one worker was available but succeeded when more workers became involved in the computation.

Scatter-Gather vs Gather-Sum-Apply

- Our experiments suggest that the gather-sum-apply implementations slightly out-performs the scatter-gather implementations
- But remember! Our dataset mimics a social network graph ⇒ its degree distribution is skewed!
- GAS performs better than the other models in such cases because it parallelizes the computation over the edges and the many low-degree vertices are not forced to wait for the few low-degree vertices to finish their computations!

Intermediate results

- Intermediate results of the algorithms usually are much bigger than the graph itself.
  - Apache Flink monitoring user interface allowed us to inspect the dataflow graph, some of the operators produced data that had considerably larger size than the size of the graph
  - Local Clustering Coefficient is a perfect example

Learning Outcomes

What did I learn?

- Vertex-centric model: implement graph algorithms from scratch
- Hadoop!

How have my software engineering skills improved?

- How to navigate large codebases
- Version incompatibilities actually exist!

Lessons hard learned…

- Debugging in distributed systems != Debugging in single-machine systems
- Debugging in distributed systems == logs, logs, logs, logs…
- Memory configuration tuning == key factor

Overall, this project was:

- A gentle introduction to the world of distributed graph processing
- A practical way to improve as a software engineer
- A humbling experience:

  I struggled a lot at the start, but learned a ton in the process

Code

You can find the project's code here:

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