Algorithms on Massive Datasets: A Comparison of Pregel and the Original MapReduce

Yosef Goren, Karen Nativ

October 2021

1 Abstract

The popularization of the internet, social media, and more recently - the ever-growing AI training data-sets have made the handling of vast data see massive growth. As a result, data-sets seem to have outgrown CPU processing speeds. Cases where linear (undistributed) processing is simply not a feasible option have become common-place. The need to avoid repeating implementations has risen, and many avoid the writing process which is far more complicated due to the lack of powerful abstractions. In present day, many computational models exist in order to mitigate these issues, and in this essay we will be examining two such models: MapReduce and Pregel.
We will give a review of each of these models, compare their use cases and capabilities, and run a few basic benchmarks of actual performance. Both models have open source implementations by Apache; MapReduce is implemented in Apache Hadoop, and Pregel in Apache Giraph. As a part of the comparison, we have implemented solutions for two 'big-data' problems in each framework (Hadoop and Giraph), Graph triangle counting, and F1 frequency moments.

2 MapReduce

2.1 Introduction

MapReduce is a computation model proposed in a Google article from 2004 [4]. The computation model is designed to enable writing distributed cluster algorithms quickly and expressively by providing the developer a functional API.
An algorithm implemented using MapReduce is parallel by nature, hence the main use case of MapReduce is for big data processing in a distributed file system.

2.2 Interface

The essential part of the API consists of two user-provided functions:

- Map: The map task is performed using a map() function that basically performs filtering and sorting. This part is responsible for processing one or more chunks of data and producing the output which is generally referred to as intermediate results. These intermediate results must be key-value pairs. As shown in figure 1, the map task is generally processed in parallel provided the mapping operations are independent of each other.
- Reduce: The reduce task is performed by a reduce() function and performs a summary operation. It is responsible for consolidating all intermediate results with the same intermediate key which were produced by the worker nodes running the map() function.

2.3 Structure

MapReduce defines a system for distributing the data; it divides the cluster into three types of nodes:

- The master node; this node is in charge of partitioning and distributing the algorithm and the data to the worker nodes and handling failures in the system.
- Mapper worker nodes: These nodes each take a slice of data directly from the file system as directed by the master node, and invoke the map() function, over each line in the input data.
  In order to minimize the network load, all results with matching keys could be aggregated (using another user-provided function), before sending them to a reducer node.
- Reducer worker nodes: These nodes each take all records with a certain intermediate key and process them all together using the reduce() function. The result of the reduce function for that key is then returned to the file system.

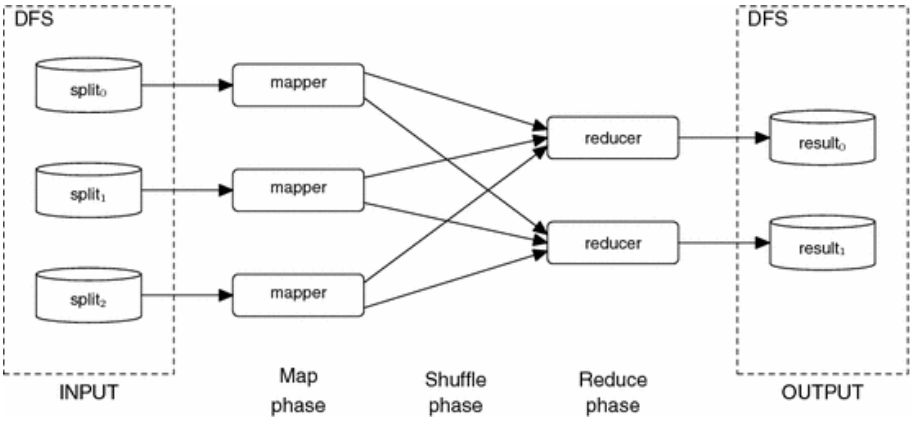

Figure 1: Data splits are taken from the file system, sent to mapper nodes, and then to the appropriate reducer nodes, before being returned to the file system.

3 Apache Hadoop

3.1 Introduction

Apache Hadoop is a collection of open-source software utilities that facilitates using a network of many computers to efficiently solve problems involving massive amounts of data and computation. It is said to be the most popular open source implementation of MapReduce [2].
Using Hadoop, users can implement their own MapReduce algorithms in Java by sub-classing generic MapReduce classes provided in the Hadoop source code.

3.2 Design Philosophy

Hadoop is designed with an assumption that all hardware fails sooner or later and the system should be robust and able to handle the hardware failures automatically.

The beauty of Hadoop MapReduce is that users usually only have to define the map and reduce functions. The framework takes care of everything else, such as parallelisation and failover. The Hadoop MapReduce framework utilises a distributed file system to read and write its data. Typically, Hadoop MapReduce uses the Hadoop Distributed File System (HDFS), which is the open source counterpart of the Google File System. We will use this file system in our project.

3.3 Strengths & Weaknesses

One of the major advantages of Hadoop MapReduce is that it allows non-expert users to easily run analytical tasks over big data. Users code their queries using Java rather than SQL. This makes Hadoop MapReduce easy to use for a larger number of developers; no background in databases is required; only a basic knowledge in Java is required. Furthermore, Hadoop is truly accessible since the framework only requires standard commodity hardware.

However, these features come at a price - the performance of Hadoop MapReduce is usually far from the performance of a well-tuned parallel database. Scaling out to very large computing clusters results in high costs in terms of hardware and power consumption. In addition, Hadoop MapReduce has performance problems due to its physical data organization.

Hadoop MapReduce jobs often suffer from a row-oriented layout. In a distributed system, a pure column store has severe drawbacks as the data for different columns may reside on different nodes leading to high network costs. Thus, whenever a query references more than one attribute, columns have to be sent through the network in order to merge different attribute values into a row. This can significantly decrease the performance of Hadoop MapReduce jobs.

4 Pregel

4.1 Motivation

The year is 2009 and social networks are booming. With them awakens the need to represent social circles on a global scale, in a graph. Google is relentlessly attempting to enter the social networks market [3], but the existing infrastructures for processing very large graphs are limited:

- Any frameworks for graphing algorithms that do not allow the distribution of the computation are too slow on this scale.
- The existing infrastructures designed exactly for the purpose of distributed graph algorithms are found to be insufficient in terms of fault tolerance, and often still require repeating further implementation for each algorithm created.
- Existing models for distributed computation that are more general, like MapReduce, often limit the ability to express the algorithm in terms of the computation model. This could often cause complexity overhead both in resources and implementation.

4.2 Introduction

Pregel is a distributed computation model for graph algorithms first published by Google in 2010 [1]. Pregel is intended to solve the problem described in the

section above. Its message-passing model, message combiners and structure as a series of 'supersteps' (described in 'Structure') - are meant to enable effective parallelization and fault tolerance, and allow algorithms to naturally avoid common issues in distributed systems, such as synchronization and IO bottlenecks.

Similar to MapReduce algorithms, a Pregel algorithm is defined by supplying a number of functions to be executed at different points of the algorithm. The centerpiece among these functions is called compute() and it is run on some of the graph's vertices in each superstep.

The functional interface and local iterative processing are hardly the only similarities between Pregel and MapReduce; the former is often implemented as a series of iterations of the latter.

4.3 Structure

As mentioned, each run of Pregel consists of a series of supersteps: In the first superstep, all of the vertices are considered active. In each superstep, every active vertex in the graph will execute the user-defined compute() function.
compute() can 'vote to halt', which will turn the vertex inactive. An inactive vertex will be reactivated if it receives a message from another vertex. To send a message to another vertex, a vertex must use the user defined function: sendMessageTo() within the implementation of compute().
The purpose of the messages does not end with reactivating vertices. They are used to pass information. compute() is therefore also responsible for handling incoming messages and their information. For each superstep S, the messages available to a vertex are only the messages sent to it in superstep S-1.
Note that compute() can also be defined to mutate the data stored within the vertex, or even the topology of the graph.

5 Apache Giraph

5.1 Introduction

Apache Giraph [5] is an open-source Apache project, that essentially extends Hadoop to allow implementing graph algorithms with Pregel-like API. With Giraph, the developer implements Pregel algorithms in Java by sub-classing classes from the Giraph source code (much like with Hadoop).

5.2 Strengths & Weaknesses

From a developer's perspective, due to Giraph being an extension of Hadoop, its environment suffers from most complications present in the Hadoop environment (and often more); this is exacerbated by the very poor documentation.

On the other hand, as we will see later in this essay, writing a graphing algorithm in terms of MapReduce can often result with far more complicated algorithms when compared to writing it as Pregel algorithms, and Giraph enables that. Also, as mentioned before - testing has shown performance improvements with Giraph [6].

In a paper published by Cornell University [8], performance comparison is made between eight parallel systems for graph processing with Hadoop and Giraph among them. The paper finds Giraph to be on-par with other similar frameworks, but in section 5.5 (of that paper) it is described as falling behind when it comes to especially large scales due the resource management system it shares with Hadoop - which was designed for the MapReduce model.

6 Testing setup

To enable us to later test each framework with the programs we have created pseudo-distributed clusters with ubuntu as the host OS. These ubuntu systems run as virtual machines emulated by VMware Workstation - one VM for the Hadoop setup and one for Giraph. To allow us a better comparison of the results, the running environments for the tests were nearly identical. For every test done, the VM of the appropriate framework runs by itself, and with no other user processes running on that VM.

7 Triangle Enumeration

7.1 Overview

Triangle Enumeration is a well-known graphing problem:
let $G = \{V, E\}$ be a graph, with V as the set of vertices and E as the set of edges,
Calculate the number of triads: $\{u, v, w\} \subseteq V$, s.t. $\{(u, v), (u, w), (v, w)\} \subseteq E$.

Since this problem is described in terms of graphs, we expect that a graph (or vertex) oriented computing model would be better suited for solving this problem. Hence, we expect that at the very least the Giraph implementation triangle enumeration will be simpler than the Hadoop implementation. Prior trials over common graphing algorithms [6] claim a 25% reduction of run-time with Giraph when compared with Hadoop.

7.2 MapReduce Implementation

For the Hadoop implementation, we have used the algorithm described in [7]. The algorithm's core idea is as follows: We use one MapReduce iteration to take the input set of edges, and return a set of open triangles such as $\{(A-B), (A-C)\}$, with the key being the edge that would be needed to complete the open triangle; (B, C) in the case above. We later combine those results with the set

of edges to see which triangles can be completed.
We will describe MapReduce computation as a pair of lambda expressions. $Key : Value$ describes a record, $A - B$ describes an edge.

The first MapReduce can be described as:

- Map 1:
$$A - B : Null \longrightarrow \{A : A - B, B : A - B\}$$
- Reduce 1:
$$A : \{A - V_i | 1 \leq i \leq n\} \longrightarrow \{V_i - V_j : \{A - V_i, A - V_j\} | 1 \leq i < j \leq n\}$$

For the second part, the input is both the original set of edges, as well as the records resulting from the prior MapReduce.

- Map 2: Identity.
- Reduce 2:
$$B - C : \{(V_{i_k} - B), (V_{j_k} - C) | 1 \leq k \leq n\} \cup \{Null\} \longrightarrow B - C : n$$

When this stage is over, we can simply sum the results from all edges. This sum is the number of edges that complete a triangle within the graph, so if we want the number of triangles we must divide that sum by three.

Given a strong ordering of the vertices, we can avoid counting every triangle three times; each triangle has exactly one minimal vertex, so the number of edges opposing a minimal vertex in a triangle is the same as the number of triangles.
We can take advantage of this by changing 'Map 1' to:

$$A - B : Null \longrightarrow min(A, B) : A - B$$

In order to analyze the modified algorithm, we will use to following terminology: Some vertex A 'proposes' a triangle Δ if 'Reduce 1' receives A as a key, and one of the records resulting from 'Reduce 1' instance has all of Δ's edges as either a key or a value.
To prove correctness, first we notice that since the proposal of a real triangle yields 1 to the sum, and the proposal of a non-existing triangle yields nothing, we need to prove that each real triangle is proposed exactly once - and in our case, the vertex proposing will be the minimal vertex in that triangle.

Let $\Delta := \{(A, B), (B, C), (A, C)\} \subseteq E$ (a 'real triangle'), and assume lexicographic ordering. We will prove that only A proposes Δ.
Since A is smaller than B and C, 'Map 1' will yield both $A : A-B$ and $A : A-C$, therefore 'Reduce 1' will receive both records, and by definition it will create

$B - C : (A - B, A - C)$, meaning A proposes Δ.
On the other hand, B needs $B : A - B$ and $B : B - C$ to be passed to 'Reduce 1' in order to propose Δ; $B : A - B$ is not created with the modified 'Map 1' and so B does not propose Δ. Similarly, C requires $C : A - C$ and $C : B - C$ to propose, while neither of them are created.

The proof might also help us understand the last optimization, instead of having every single vertex of a triangle proposing it, now only the minimal vertex in each triangle proposes. Reducing the number of proposals made improves our algorithm in two ways: firstly each proposal requires further computations for 'checking' it in the next stage, secondly each extra proposal means more communication and network load required.

While we were able to minimize the number of proposals made by only counting distinct triangles, we can further improve the algorithm by trying to minimize the number of non-existing triangles proposed. This can be done by better selecting the ordering of 'min' (used in 'Map 1'); selecting a better ordering can vastly reduce the total number of triangles checked as can be seen in figure 2.

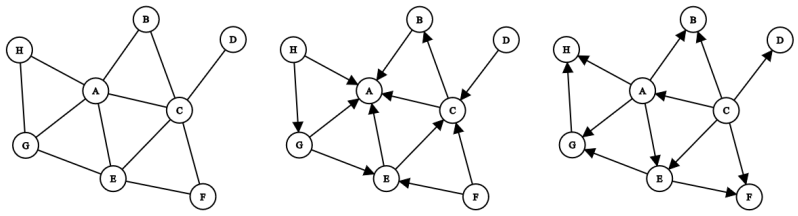

Figure 2: The number of proposed triangles as affected by the vertex ordering. In this example, the left graph is our input, and an ordering is represented by the arrow direction; for any $X, Y \in V$, $X \to Y$ means $X < Y$.
In the middle graph the ordering is reverse-lexicographic, and in the graph to the right the ordering is by vertex degree. While the middle graph results with 14 triangles proposed, only 5 are proposed in the right graph.

The example shown in the figure showcases the quadratic relation between the number of edges pointing at a vertex (or the number of edges connected to it - in which the vertex is the minimal one), and the resulting number of proposals made by that vertex.
From the definition of 'Reduce 1' (7.2) we can see that for the input $n = |\{A - V_i | 1 \leq i \leq n\}|$:

$$OutputSize(n) = |\{V_i - V_j : \{A - V_i, A - V_j\} | 1 \leq i < j \leq n\}| = \binom{n}{2} = \Theta(n^2)$$

Due to this relation, in order to minimize the number of triangles proposed in our algorithm we want to spread out the incoming edges so no particular vertex has significantly more incoming edges than any other vertex. If the edges are randomly directed - the expected in-degree of an edge should be linear (half) of degree of the vertex. On the other hand, if the edges are always directed towards the vertex with the lower degree, we can intuitively understand why this could equalize the in-degree:
A well connected vertex will have a lowered in-degree because its degree is high, and a vertex with few edges, will have relatively higher in-degree due to having more of the edges being incoming ones.

In order to compute the degree of each vertex, we can use a MapReduce algorithm described by the same paper [7]. Note that in the prior examples we did not distinguish the ordering of the vertices in the arch ($A - B = B - A$), but for the following, this distinction is made.

- Map 1:
$$A - B : Null \longrightarrow \{A : A - B, B : A - B\}$$
- Reduce 1:
$$A : \{A - V_i | 1 \leq i \leq n\} \cup \{V_i - A | 1 \leq i \leq m\} \longrightarrow$$
$$\{A - V_i : n + m - Null | 1 \leq i \leq n\} \cup \{V_i - A : Null - n + m | 1 \leq i \leq m\}$$
- Map 2: Identity.
- Reduce 2:
$$A - B : \{d_A - Null, Null - d_B\} \longrightarrow A - B : d_A - d_B$$

7.3 Pregel Implementation

The Pregel algorithm we have used utilizes the same main concept in order to avoid re-counting the same triangles: useing an ordering of the vertices allowing only the minimal vertex of each triangle do the counting.
The algorithm requires 4 super-steps in total, each is described from the perspective of a vertex that is running the compute function in that super-step:

1. Send a message with your id to every neighbor with a higher ordering than yourself.
2. For every message received, send it to every neighbor with a higher ordering than you.
3. Forward every message received to all of your neighbors.
4. Count the number of messages with your own id on them.

After running the last superstep, we simply need to sum up the counters of all vertices - and that sum is the number of triangles in the graph.

7.4 Testing input samples

For the performance testing of our two implementations we have utilized three different sets of data for which we would like to examine the results:

1. Social Network Graphs: as discussed earlier this is a real world example for the use of triangle enumeration. To keep this example case authentic we decided to pull the data-set for this type of graph from real social networks: In the data we took, a vertex is represented by some Facebook page, and an edge between two pages means that one of them liked the other. In the sample we have used [10], there are about 4000 pages, and 17000 likes.

2. Log-Random Graphs: These example graphs are created such that for each vertex in the graph, the expectation for the number of edges connected to it is the log of the number of vertices in the graph. To generate these graphs we have made a simple script of our own; given wanted size of the input graph it creates a set of vertices with randomly generated identifiers and randomly connects the vertices where each possible edge has a probability of $\frac{log(|V|)}{|V|}$ for existing, independent of any other edges in the graph.

3. Clicks: These are graphs with an edge between every possible pair of vertices, these serve as an extreme input case where the number of edges for each vertex is very large.

Note that there are two main varying metrics between the different graph types:

1. Homogeneity: the social network graph is the least homogeneous, with the Clicks still being more homogeneous than the Log-Random graphs. This metric might make a difference when it comes to MapReduce algorithm, which takes into account the degree of each vertex in order to minimize the triangles checked, while the equivalent Pregel algorithm does not.

2. Single-Vertex-Edge-Count: Since every vertex in a social network graph represents a single page - with its edges being other pages liked by it, we expect the number of edges on each vertex to be essentially constant ($O(1)$). When it comes to the Log-Random graphs, the edge count is $\Theta(log(|V|))$ by design, and similarly the Clicks have $\Theta(|V|)$ edges for each node.

7.5 Runtime Performance

In the real results we can see that other than the initial start-up runtime, the Giraph program is faster than the Hadoop version 4. This is despite the Hadoop algorithm being more optimized, and with no cost at abstraction - the Giraph algorithm is clearly simpler and more straight forward.
We expect that with some optimizations to the Giraph algorithm, there would be even more extreme differences.

8 F1 Frequency Moments

8.1 Overview

For stream or data-set $\{x_i\}_{i=1}^{n} \subseteq \Sigma$, the frequency of some $a \in \Sigma$ is defined as: $f_a = |\{i|x_i = a\}|$. For some natural number k, the F-k Frequency Moment is defined to be: $\sum_{a \in \Sigma} f_a^k$.
Despite their simplicity, Frequency Moments can often give useful first insights about a data-set. For example: for a given value of F1, a perfectly homogeneous set (all items have the same frequency) will have the minimal F2 value, while a data-set with a few items that account for most of the F1 sum, will have a much higher F2 value. This type of analysis is often useful for 'real-time' applications since these metrics are quick to enumerate and require little space; an instance of this might be algorithms designed to detect DDoS attacks on ISP networks which make use of the Frequency Moments for their evaluation [9].

In our essay we are going to look at the F1 Frequency Moment; it essentially outputs the total size of the input stream or data-set.
While this is quite a simple metric, it will still require putting to work the distributed aspects of the frameworks we are testing.

As this problem has no natural graph representation (without further assumptions over the input data), we think it is likely that a Pregel algorithm for this will not come naturally, while Mapreduce seems like a better model for this kind of simple enumeration.

8.2 MapReduce Implementation

Our method for calculating the number of items within the input is quite straight forward. For each item in the input, first create an intermediate item consisting of some common key (for example the string *total*), and the value 1. In the reduce phase, take all of the intermediate items created (they should all be with that same key and therefore be sent to the same reducer node) and sum up their values. At the end the reducer will output that aggregation sum.

This process can be optimized for lowering its load on the communications network by using an aggregating function: this function transforms the intermediate values which share a common key to a single output value (on each mapper node) which consists of the number of items sent by the mapping function and pair that aggregated value with the original key.

- Map:
$$S \longrightarrow total : 1$$
- Aggregate:
$$\{total : 1\}_{i=1}^{k} \longrightarrow (total, k)$$

- Reduce:

$$\{total : s_i\}_{i=1}^{k} \longrightarrow \sum_{i=1}^{k} s_i$$

8.3 Pregel Implementation

Our Pregel solution for F1 Frequency Moments takes the input graph as a linked list of nodes. In each step of the algorithm, each node in this list sends a message to the node after it, updating it with the highest sum it has seen so far (0 as an initial value). The actual sum we want to calculate is the sum that starts from the first node in our list, and makes its way from node to node, until it gets to the last one; at which point there are no more messages being sent within the graph, causing Pregel to halt. When the process is finished, we want to only pull the data from the last node. In order to distinguish the last node from the others we use the fact that it's the only one with no outgoing edges; when the program ends each node checks if it has no outgoing edges and if so - it outputs the sum it has stored.

Figure 3: Input graph for Giraph F1 enumeration algorithm.

Note that this algorithm causes higher network load and IO than needs be; this can be attributed to the message passing model discussed earlier. In the case of F1, there is no actual need for communication between the different items within the data-set, and we simply want to see how many of them there are. Therefore a model which requires the computation to be done as a set of messages being passed between nodes is simply not a descriptive model for our problem.

Simple analysis of the algorithm shows that $\Theta(n^2)$ messages are passed. On the bright side, no single node can expect more than one message at any given super-step, meaning the algorithm should do well in terms of how parallel the computation can be with constant time complexity over each node in a given super-step.

8.4 Testing sample

Since neither of our F1 algorithms are very dependent on the form of the input data (due to virtually no operations being done on the actual content of each item in the data-set) - we have only used one type of input data to go with the

F1 testing - A sample of randomly generated text: any actual meaning of the text is ignored: we are only interested in the word count in this case.

8.5 Runtime Performance

As we could expect, the Giraph implementation did much worse in this test: in some cases not even managing to finish after a 500 second time cap. This is likely because of the explosive square complexity of the number of messages being sent in relation to the number of items in the input data-set. On the other hand, in all scales of our test the Hadoop program did not even seem to run for more than its initial start-up overhead of about 5 seconds. These staggering differences can be clearly seen in Figure 5.

8.6 Memory complexity

Since none of the algorithms used make use of more than linear space to store the data being worked - all while being able to reach high runtimes with these samples, the memory cost of running this case does not seem to be a limiting factor in either.

9 Results

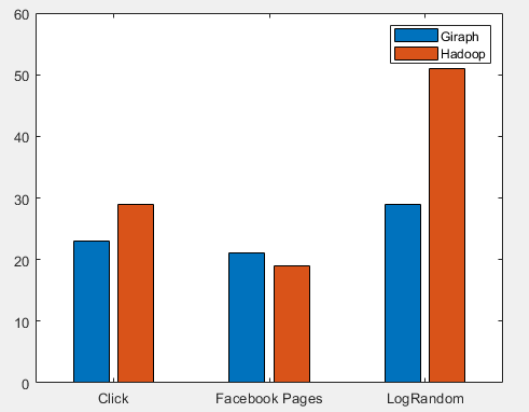

Figure 4: Hadoop vs Giraph Triangle Enumeration runtimes.
The Clicks were on the order of 50 vertices, and there are about 125000 edges. The set of Facebook pages with about 4000 vertices and 17000 edges and the LogRandom graphs were on the order of 10000 vertices meaning about 150000 edges.

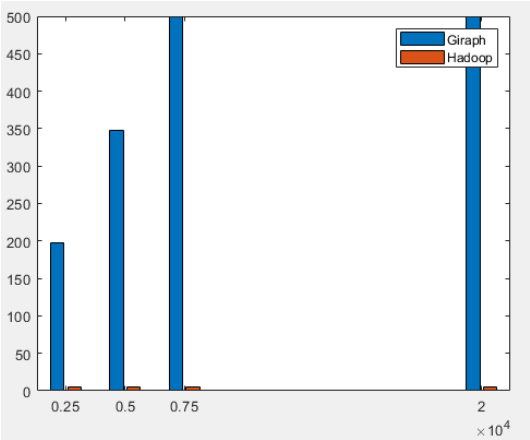

Figure 5: F1 Frequency Moments runtims with randomly generated text.

10 Conclusions

As we have seen in the tests, the comparison is not as simple as one of the models improving over the other, or allowing for better abstraction at no cost, rather each computational model has a use case in which it excels. In a way - MapReduce often offers less abstraction, but this is not always bad: as we have seen in the F1 example, the computational model of Pregel simply did not fit the problem and hence resulted with an inferior algorithm - as well as a worse runtime and more communication requirements.
On the other hand, if the problem we are facing is somehow inherent to a graph (or perhaps has a powerful graph representation) we would be far better off using Pregel for it, as can be seen in the MapReduce example; simply expressing our graphing algorithm in terms of MapReduce might not be easy, and it did not end with the implementation overhead. Pregel framworks are optimized to handle the operations done within the graph which translates to better overall performance.

With all that said, we did find Pregel to be generally more expressive, but coming with some overhead that results from abstractions that might not be needed, while MapReduce is in more 'bare metal' - with eveything that comes with it. Overall, we find that these considerations should only be second, while the most important factor is simply the problem itself and how good our model is at describing it.

To view the tests and code from our workspace, we have added a link to our Github repository [11].

References

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

[2] Effcient big data processing in Hadoop MapReduce:
Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz.
https://dl.acm.org/doi/abs/10.14778/2367502.2367562?casa_token=EfEz01htrf8AAAAA%3AkuXi4hBuh37StFC5O7UpDyeoAbl53YjYV2UD4Xo6oP6SHCjqg4ageeBP9aqOFO1I

[3] Google attempts at social media in the late 2000's:

> "Google+ was the company's fourth foray into social networking, following Google Buzz (introduced 2010, retired in 2011), Google Friend Connect (introduced 2008, retired by March 2012), and Orkut (introduced in 2004, as of 2013 operated entirely by subsidiary Google Brazil – retired in September 2014[7])."

Quoted from:
https://en.wikipedia.org/wiki/Google%2B

[4] MapReduce: Simplified Data Processing on Large Clusters:
Jeffrey Dean, Sanjay Ghemawat.
https://research.google/pubs/pub62.pdf

[5] Apache Giraph - home page: https://giraph.apache.org/

[6] Advantages of Giraph over Hadoop in Graph Processing:
C.L. Vidal-Silva, E. Madariaga, T. Pham, J.M. Rubio, L.A. Urzua, L. Carter, F. Johnson.
https://www.etasr.com/index.php/ETASR/article/view/2715

[7] Graph Twiddling in a MapReduce World:
Jonathan Cohen.
https://www.computer.org/csdl/magazine/cs/2009/04/mcs2009040029/13rRUxBJhis

[8] Experimental Analysis of Distributed Graph Systems:
Khaled Ammar, M. Tamer Ozsu.
https://arxiv.org/abs/1806.08082

[9] Streaming Algorithms for Robust, Real-Time Detection of DDoS Attacks:
Sumit Ganguly, Minos Garofalakis, Rajeev Rastogi, Krishan Sabnani.
https://www.researchgate.net/publication/4259752_Streaming_Algorithms_for_Robust_Real-Time_Detection_of_DDoS_Attacks

[10] The Triangle Counting Facebook pages graph taken from: https://github.com/benedekrozemberczki/datasets#facebook-page-page-networks

[11] The Github repository with our implementations and workspace:
`https://github.com/karen-nativ/big-data-algorithms`