A Comparison of Graph Processing Systems

Simon König (3344789) st156571@stud.uni-stuttgart.de
Leon Matzner (3315161) st155698@stud.uni-stuttgart.de
Felix Rollbühler (3310069) st154960@stud.uni-stuttgart.de
Jakob Schmid (3341630) st157100@stud.uni-stuttgart.de

***Abstract*—TODO**

***Index Terms*—graphs, distributed computing, Galois, Ligra, Polymer, Giraph, Gluon, Gemini**

TODOs

- Appendix für Setup Guides - md zu pdf

I. Introduction

This paper makes the following contributions:

- Comparison of several state-of-the-art graph processing frameworks
-
-
-

II. Related Work

III. Preliminaries

TODO //

A. Graphs and Paths

An *unweighted graph* is the pair $G = (V, E)$ where the *vertex set* is $V \subseteq \mathbb{N}$ and the E is the *edge set*. The edge set describes a number of connections or relations between two vertices. Depending on these relations, a graph can be directed or undirected. For a *directed graph* the edge set becomes

$$E \subseteq \{(x, y) \,|\, x, y \in V, x \neq y\}$$

and in the *undirected* case E is a set of two-sets

$$E \subseteq \{\{x, y\} \,|\, x, y \in V, x \neq y\}.$$

The main difference is thus, that in the case of a directed graph a connection between s and t is not the same as a connection between t and s – the direction matters.

Independently of the graph being directed or not, a graph can be *weighted*. In this case the edge set is expanded by a numerical value $E_{\text{weighted}} = E_{\text{unweighted}} \times \mathbb{R}$ further describing the relation.

The size of a graph is often described in the number of edges $|E|$ because this number is typically much larger than the number of vertices $|V|$.

A *Path* from start s to target t is a set of edges $P \subseteq E^n$ with

$$P = ((x_0, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n))$$

where all $(x_i, x_{i+1}) \in E$ and $x_0 = s, x_n = t$.

Thus we call a target t *accessible* from s if a Path from s to t exists.

B. Single-Source Shortest-Paths

Single-Source Shortest-Paths (SSSP) describes the problem of finding a path along some edges of the input graph from a given start node to a target.

Input to the problem is a weighted graph $G = (V, E)$ and a start node $s \in V$. Output is the shortest possible distance from s to each node in V. The distance is defined as the sum of edge weights w_i on a path from s to the target. In the case of a unweighted graph, the distance is often described in *hops*, i.e. the number of edges on a path.

The most common implementations are Dijkstra's algorithm or BellmanFord.

C. Breadth-First Search

Breadth-first search (BFS) is a search problem on a graph. It usually requires an unweighted graph and a start vertex as input. In some cases a target vertex is also given.

The output is usually a set of vertices that are accessible from the start vertex. In the case of a target being given, the output is true if a path from start to target exists or false otherwise.

It is called breadth-first search because the algorithm searches in a path length-based way. First all paths of length 1 i.e. all neighbors of the start node are checked before checking paths of length 2 and so on.

The search algorithm, where the paths of maximum length are checked first is called Depth-First Search.

D. PageRank

PageRank (PR) is a link analysis algorithm that weighs the vertices of a graph, measuring the vertices relative importance within the graph.

The analogy is that the graph represents website pages of the Word Wide Web, that are hyperlinked between one another. A website that is more important is likely to receive more links from other webiste. PageRank counts the number and quality of links to a page to estimate the importance of a page.

For example, Google Search uses PageRank to rank web pages in their search engine results.

The output of PageRank is a percentage for each vertex. This percentage, called the PageRank of the vertex, is the probability with which a web surfer starting at a random web page reaches this webpage (vertex). With a high probability the web surfer uses a random link from the web page they are currently on and with a smaller probability (called damping factor) they jump to a completely random web page.

a) Delta-PageRank: ?

E. Bulk-Synchronous Parallel Model

The Bulk-Synchronous Parallel (BSP) model is a computation model developed by Leslie Valiant [1]. It is commonly used in computation environment with large amounts of synchronous computation, for example Giraph is based on this model.

This model describes components, a communication network between those components and a method of synchronization. The components are capable of performing computations and transactions on *local* memory. Pairs of components can only communicate using messages, thus remote memory access is also only possible in this way. Synchronization is realized through barriers for some or all processes.

BSP algorithms are performed in a series of global supersteps. These consist of three steps, beginng with the processors performing local computations concurrently. This step can overlap with the second, the communication between components. Processes can exchange information to access remote data. Lastly, processes reaching a barrier wait until all other processes have reached the same barrier.

One of the most famous state-of-the-art graph processing systems is Pregel [2]. It is based on the BSP computation model. Pregel and many open-source versions similar to it were built to process large graphs reliably (offering fault tolerance) on large MapReduce infrastructures.

F. MapReduce

The MapReduce model is a computation infrastructure developed by Google to reliably handle large data sets on distributed clusters [3].

A user specifies just the two functions Map and Reduce. The system hides the details of parallelization, fault-tolerance, data distribution and load balancing away from the application logic. All of these features are automatically provided.

Execution is performed in three phases:

1) Map phase: The input data is distributed between a set of Map processes, the Map functionality is specified by the user. Ideally all Map processes run in parallel so the map processes need to be independent. Results from this phase are written into (multiple) intermediate storage points.
2) Shuffle phase: The results are grouped according to a key provided by the Map algorithm. Each set of results is then handed to one system for the next phase.
3) Reduce phase: Every set of intermediate results is input to exactly one reduce process. The Reduce functionality is again specified by the user and ideally runs in parallel.

We show Giraph [4] as an example of a system using this framework.

G. Congest Model

H. Push/Pull Differences

IV. Overview

We start with a short overview describing functionality and characteristics of each framework, followed by their required input data formats.

//

A. Galois and Gluon

Galois[5] is a general purpose library designed for parallel programming. The Galois system supports fine grain tasks, allows for autonomous, speculative execution of these tasks and grants control over the task scheduling policies to the application. It also simplifies the implementation of parallel applications by providing an implicitly parallel unordered-set iterators.

For graph analytics purposes a topology aware work stealing scheduler, a priority scheduler and a library of scalable data structures have been implemented. Galois includes applications for many graph analytics problems, among these are single-source shorthest-paths (sssp), breath-first-search (bfs) and pagerank. For most of these applications Galois offers several different algorithms to perform these analytics problems and many setting options like the amount of threads used or policies for splitting the graph. All of these applications can be executed in shared memory systems and, due to the Gluon integration, with a few modifications in a distributed environment.

Gluon[6] is a framework written by the Galois team as a middleware to write graph analysis applications for distributed systems. It reduces the communication overhead needed in distributed environments by exploiting structural and temporal invariants.

The code of Gluon is embedded in Galois. It is possible to integrate Gluon in other frameworks too, which the Galois team showed in their paper[6].

B. Gemini

Gemini is a framework for parallel graph processing [7]. It was developed with the goal to deliver a generally better performance through efficient communication. While most other distributed graph processing systems achieve very good results in the shared-memory area, they often deliver unsatisfactory results in distributed computing. Furthermore, a well optimized single-threaded implementation often outperforms a distributed system. Therefore it is necessary to not only focus on the performance of the computation but also of the performance of the communication. Gemini tries to bridge the gap between efficient shared-memory and scalable distributed systems. To achieve this goal, Gemini, in contrast to the other NUMA-aware frameworks discussed here, does not support shared-memory calculation, but chooses the distributed message-based approach from scratch.

Gemini is fairly lightweight and seems well structured with a clearly defined API between the core framework and the implementations of the individual algorithms. The five already

implemented algorithms are single source shortest path (sssp), breath first search (bfs), pagerank, connected components (cc) and biconnected components (bc).

C. Giraph

Apache Giraph is an example for an open-source system similar to Pregel. Thus, Giraph's computation model is closely related to the BSP model discussed in subsection III-E. This means that Giraph is based on computation units that communicate using messages and are synchonized with barriers [4].

The input to a Giraph computation is always a directed graph. Not only the edges but also the vertices have a value attached to them. The graph topology is thus not only defined by the vertices and edges but also their initial values. Furthermore, one can mutate the graph by adding or removing vertices and edges during computation.

Computation is vertex oriented and iterative. For each iteration step called superstep, the Compute method implementing the algorithm is invoked on each active vertex, with every vertex being active in the beginning. This method receives messages sent in the previous superstep as well as its vertex value and the values of outgoing edges. With this data the values are modified and messages to other vertices are sent. Communication between vertices is only performed via messages, so a vertex has no access to values of other vertices or edges other than its own outgoing ones. Supersteps are synchronized with barriers, meaning that all messages only get delivered in the following superstep and computation for the next superstep can only begin after every vertex has finished computing the current superstep. Edge and vertex values are retained across supersteps. Any vertex can stop computing (i.e. setting its state to inactive) at any time but incoming messages will reactivate the vertex again. A vote-to-halt method is applied, i.e. if all vertices are inactive or if a user defined superstep is reached the computation ends. Once calculation is finished, each vertex outputs some local information (e.g. the final vertex value) as result.

In order for Giraph a achieve scalability and parallelization, it is built on top of Apache Hadoop [4]. Hadoop is a MapReduce infrastructure providing a fault tolerant basis for large scale graph processing. Hadoop supplies a distributed file system (HDFS), on which all computation is performed. Giraph is thus, even when only using a single node, running in a distributed manner. But expanding single-node processing to a multi-node cluster is seamless. Giraph uses the Map functionality of Hadoop to run the algorithms, Reduce is only used as the identity function.

Giraph being an Apache project makes it the most actively maintained and tested project in our comparison. Over the course of this project, several new updates were pushed to Giraph's source repository[1].

D. Ligra

Ligra[8] is a lightweight parallel graph processing framework for shared memory machines. It offers a programming inteface that allows expressing graph traversal algorithms in a simple way.

Algorithms can use the `EdgeMap` to make computations based on edges or the `VertexMap` to make computations based on vertices. Those mappings can be applied only to a subset of vertices. Based on the size of the vertex subset the framework automatically switches between a sparse and a dense representation to optimize speed and memory.

E. Polymer

Polymer is very similar to Ligra, in fact Polymer inherits the programming interfaces `EdgeMap` and `VertexMap` from Ligra as its main interface.[9]

Polymer is a vertex-centric framework, that tries to circumvent some of the random memory access drawbacks of such a design. It treats a NUMA machine as a distributed cluster and splits work and graph data accordingly between the nodes. Application-defined data is not distributed. Other runtime state data is allocated in a distributed way but only accessed through a global lookup table.

F. Important Graph Formats

Since every frameworks uses different graph input formats, we supply a conversion tool capable of translating from EdgeList to the required formats. Data Sets retreived from KONECT can be directly read and translated.

The following sections explain the output formats of our conversion tool.

1) AdjacencyGraph: The AdjacencyGraph and WeightedAdjacencyGraph formats used by Ligra and Polymer are similar to the more popular *compressed sparse rows* format.

The The format was initially specified for the Problem Based Benchmark Suite, an open source repository to compare different parallel programming methodologies in terms of performance and code quality [10].

The file looks as follows

$$n, m, o_1, \ldots, o_n, t_1, \ldots, t_m$$

where commas are $\backslash\mathrm{n}$. First, n is the number of vertices and m the number of edges in the graph.

The o_k are the so-called offsets. Each vertex k has an offset o_k, that describes an index in the following list of the t_i. The t_i are vertex IDs describing target nodes of a directed edge. The index o_k in the list of target nodes is the point where edges outgoing from vertex k begin to be declared. So vertex k has the outgoing edges

$$(k, t_{o_k}), (k, t_{o_k+1}), \ldots, (k, t_{o_{k+1}-1}).$$

For the WeightedAdjacencyGraph format, the weights are appended to the end of the file in an order corresponding to the target nodes.

The files always start with the name of the format i.e. AdjacencyGraph or WeightedAdjacencyGraph in the first line.

[1] https://gitbox.apache.org/repos/asf?p=giraph.git

2) EdgeList: The EdgeList format is the most intuitive and one of the most commonly used in online data set repositories. The KONECT database uses this format and thus it is the input format for our conversion tool.

An edge list is a set of directed eges $(s_1, t_1), (s_2, t_2), \ldots$ where s_i is a vertex ID representing the start vertex and t_i is a vertex ID representing the target vertex. In the format, there is one edges per line and the vertex IDs s_i, t_i are separated with any whitespace character.

For a WeightedEdgeList, the edge weights are appended to each line, again separated by a whitespace character.

3) Binary EdgeList: The binary EdgeList format is used by Gemini.

For s_i, t_i some vertex IDs and w_i the weight of a directed edge (s_i, t_i, w_i), Gemini requires the following input format

$$s_1 t_1 w_1 s_2 t_2 w_2 \ldots$$

where s_i, t_i have `uint32` data type and the optional weights are `float32`. Gemini will derive the number of edges from the file size, so there is no file header or anything similar allowed.

4) Giraph's I/O formats: Giraph is capable of parsing many different input and output formats. All of those are explained in Giraph's JavaDoc[2]. Both edge- and vertex-centric input formats are possible.

One can even define their own input graph representation or output format. For the purposes of this paper, we used an existing format similar to AdjacencyList but represented in a JSON-like manner.

In this format, the vertex IDs are specified as `long` with `double` vertex values, `float` out-edge weights. Each line in the graph file looks as follows

$$[s, v_s, [[t_1, w_{t_1}], [t_2, w_{t_2}]...]]$$

with s being a vertex ID, v_s the vertex value of vertex s. The values t_i are vertices for which an edge from s to t_i exists. The directed edge (s, t_i) has weight w_{t_i}.

There is no surrounding pair of brackets and no commas separating the lines as it would be expected in a JSON format.

V. Testing Methods

For testing the graph processing systems, we used 5 machines with two AMD EPYC 7401 (24-Cores) and 256 GB of RAM each. One of those machines was only used as part of the distributed cluster, since it only has 128 GB of RAM. All five machines were running Ubuntu 18.04.2 LTS.

Setup of each framework was performed according to our provided installation guides available in Appendix A. All benchmark cases were initiated by our benchmark script available in our repository. All five frameworks are tested on a single server. Galois, Gemini and Giraph were benchmarked in on the distributed 5-node cluster as well. Since Galois supports this parameter, we ran multiple tests comparing Galois' performance with different thread counts on a single machine.

The complete benchmark log files and extracted raw results are available in our repository.

[2]http://giraph.apache.org/apidocs/index.html

TABLE I: Size Comparison of the Used Graphs

Graph	# Vertices (M)	# Edges (M)
flickr	0.1	2
orkut	3	117
wikipedia	12	378
twitter	52	1963
rMat27	63	2147
friendster	68	2586
rMat28	121	4294

A. Data Sets

The graphs used in our testing can be seen in detail in Table I. We included a variaty of different graph sizes, from relatively small graphs like the flickr graph with 2 million edges up to an rMat28 with 4.2 billion edges. All graphs except the rMat27 and rMat28 are exemplary real-world graphs and were retrieved from the graph database[3] associated with the Koblenz Network Collection (KONECT)[11]. Both the rMat27 and rMat28 were created with a modified version (we changed the output format to EdgeList) of a graph generator provided by Ligra.

B. Measurements

For every framework, we measured the *execution time* as the time from start to finish of the console command.

For the *calculation time*, we tried to extract only the time the framework actually executed the algorithm. We came up with the following:

- For Galois, we resulted to extracting console log time stamps. Galois outputs `Reading graph complete.`. Calculation time is the time from this output to the end of execution.
  We know that this is not the most realiable way for measuring the calculation times. Not only due to unavoidable buffering in the console output we expect the measured time to be larger than the actual. First, it is not clear that all initialization is in fact complete after reading the graph. Second, we include time in the measurement that is used for cleanup after calculation.
- Polymer outputs the name of the algorithm followed by an internally measured time.
- Gemini outputs a line `exec_time=x`, which was used to measure the calculation time.
- Ligra outputs its time measurement with `Running time : x`.
- Giraph has built in timers for the iterations (supersteps), the sum of those is the computation time.

Furthermore, the *overhead* is the time difference between execution time and calculation time.

Each test case consisting of graph, framework and algorithm was run 10 times, allowing us to smooth slight variations in the measured times. Later on, we provide the mean values of the individual times as well as the standard deviation where meaningful.

[3]http://konect.uni-koblenz.de/

C. Algorithms

The three problems Breadth-first search (BFS), PageRank (PR) and Single-source shortest-path (SSSP) were used to benchmark framework with every graph. We always show the results of PageRank with a maximum of five iterations. For frameworks that support multiple implementations (i.e. PageRank in push and pull modes), we included the alternatives in our testing.

In detail, the algorithms for each framework are:

- Galois: CPU, Push, Pull für PR und SSSP..
- Gemini:
- Giraph: Giraph does not natively supply a BFS algorithm, so in our comparisons a custom implementation is used.
- Ligra:
- Polymer: Polymer supports two implementations of Pagerank

D. Comparison of setup

VI. Results

A. The frameworks during setup and benchmark

We would like to raise some issues we encountered first while installing and configuring and second while running the different frameworks.

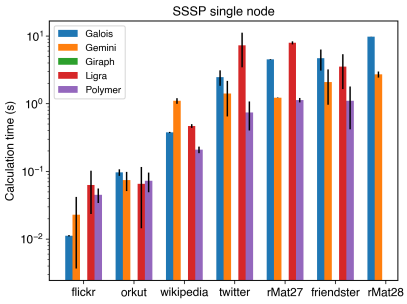
1) During setup and benchmark of Gemini, we encountered several bugs in the cloned repository. These include non zero-terminated strings or even missing return statements.
   The errors rendered the code as-is unable to perform calculations, forcing us to fork the repository and modify the source code. Our changes can be found in one of our repositories[4].
2) Furthermore, we would like to address the setup of Hadoop for Giraph. It requires multiple edits in `xml` files that aren't easily automized. This makes the setup rather time consuming, especially if reconfiguration is needed later on.
3) In order for Giraph to run, several Java tasks (the Hadoop infrastructure) have to be constantly running in the background. While we don't expect this to have a significant performance impact on other tasks, it is still suboptimal.
4)

On a plus side, setup of frameworks like Polymer or Ligra was straight forward and did not require any special treatment.

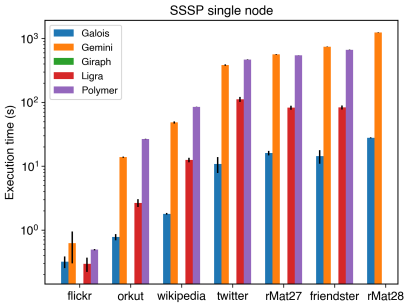
B. Single-source Shortest-paths

For the results of our SSSP runs, we first analyze the single-node and distributed performances separately before comparing the two.

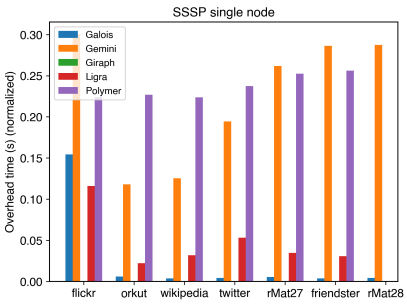
[4]https://github.com/jasc7636/GeminiGraph



(a) Calculation times for SSSP on a single node



(b) Execution times for SSSP on a single node



(c) Overhead time normalized by the graph size in million edges

Fig. 1: Average times on a single computation node, black bars represent one standard deviation in our testing. The runs on rMat28 for Ligra and Polymer failed and the frameworks were unable to complete the task.

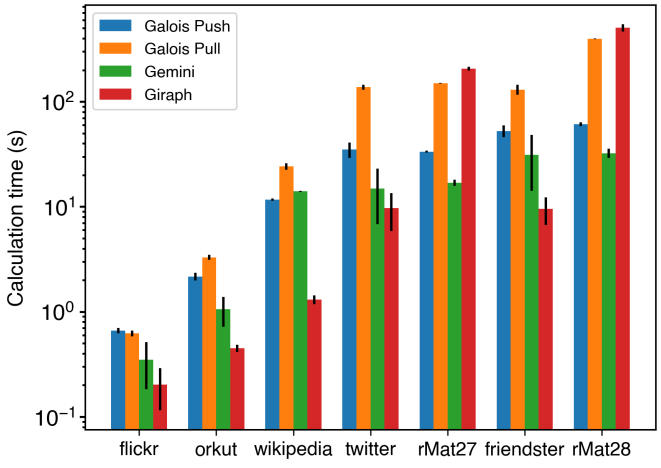
1) Single-node: Beginning with the single-node performance, Figure 1 shows the average calculation and execution times for SSSP on the different frameworks.

Because we did not see the perfomance going down when increasing thread counts on Galois, we always refer to Galois with 96 threads in this section (including the figures). For a more detailed analysis of Galois' behaviour with different thread counts, see subsection VI-E.

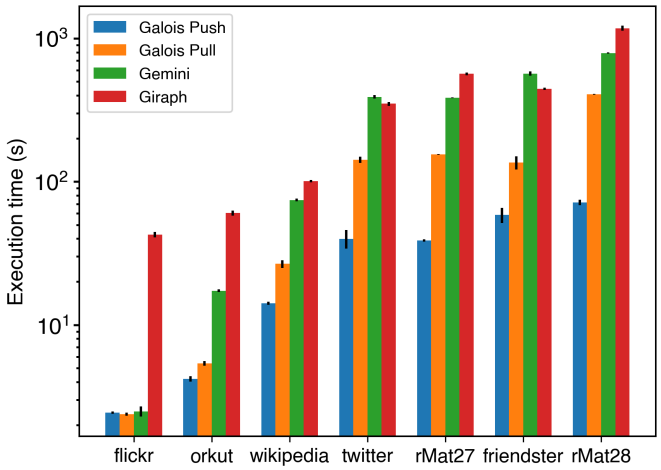
TODO // Hier fehlt noch Giraph, daher noch keine vollständige Auswertung.

2) Distributed: Moving on to the distributed cluster, Figure 2 shows the benchmark results as calculation and execution times.

Comparing the two Galois implementations, we find the calculation and execution times to be similar on smaller graphs and Push being the superior implementation for SSSP on larger



(a) Calculation times for distributed SSSP



(b) Execution times for distributed SSSP

Fig. 2: Average times on the distributed cluster, black bars represent one standard deviation in our testing.

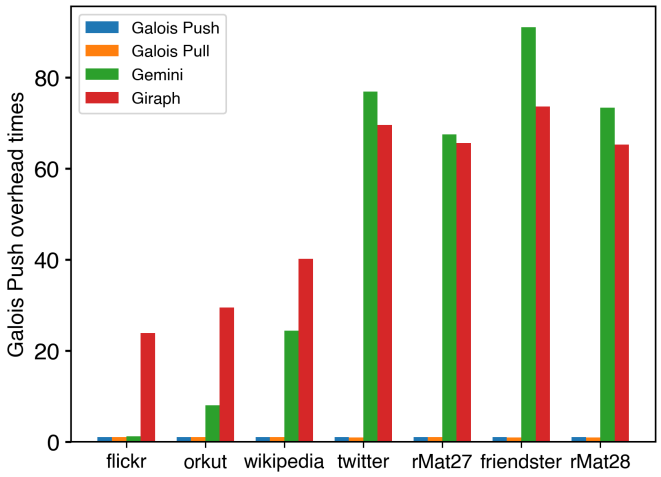


Fig. 3: Distributed Single-Source Shortest-Paths overhead times of each framework normalized by the overhead time of Galois Push

TABLE II: Distributed SSSP Execution Times Relative to Galois Push

Data Set	Galois Push	Galois Pull	Gemini	Giraph
flickr	1.0	0.97	1.02	17.46
orkut	1.0	1.28	4.12	14.38
wikipedia	1.0	1.88	5.26	7.11
twitter	1.0	3.56	9.79	8.76
rMat27	1.0	3.98	9.91	14.52
friendster	1.0	2.32	9.72	7.59
rMat28	1.0	5.70	11.07	16.5

data sets. Galois Pull being anywhere from just as fast to 3.5× slower on real-world data sets compared to the Push variant. The synthetic graphs are more extreme, execution times are close to 4× (rMat27) and 5× (rMat28) longer.

Both Galois implementations have significantly smaller execution times compared to Gemini or Giraph on all graphs, see Table II. You can see Gemini being worse by at least a factor of 4 compared to Galois Push on all graphs except flickr. Giraph's execution times in comparison to this are even worse, taking at least 7× longer than Galois Push on all graphs.

Evidently, Galois Push is the fastest algorithm in our lineup on 6 out of 7 graphs. With the exception being flickr, where Galois Push takes negligibly longer than the Pull counterpart.

When taking a closer look at Giraph, it seems to not cope well with synthetic data sets. Analyzing the computation times in Figure 2a, we see that it is the fastest framework on our real-world graphs. And that with a considerable margin of other frameworks always taking at least 50% longer (lower bound here is Gemini on flickr) up to Galois Pull needing 18× more time on wikipedia. On both synthetic graphs however, Giraph is actually the slowest to compute. Giraph requires 12× or even 15× the computation time of Gemini on rMat27 or rMat28 respectively.

While Giraph's computation times are very competitive, when comparing the execution times in Figure 2b we see that Giraph is actually the slowest framework on 5 out of 7 graphs. For the other two, namely twitter and friendster, Giraph is second slowest with only Gemini taking longer to complete.

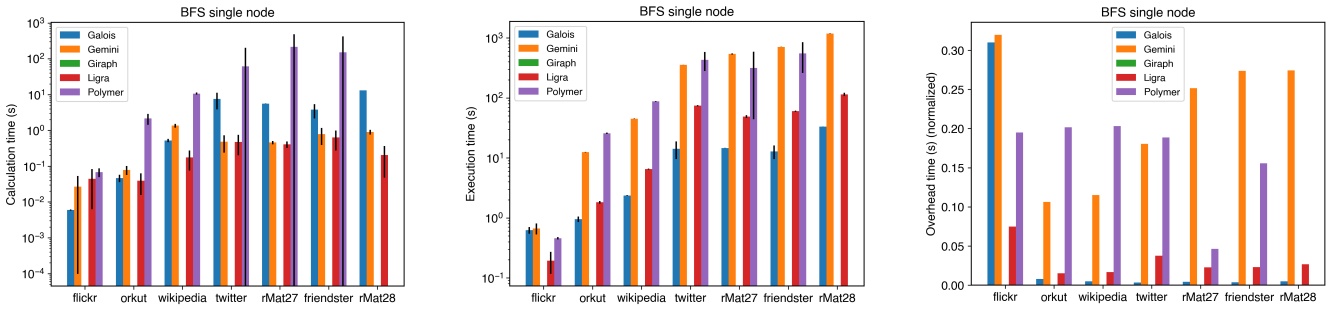
Giraph and Gemini's very long execution times are only due to their overhead being many orders of magnitude larger than Galois overhead (Figure 3). Overhead for Gemini is greater than that of Galois on every graph. From just a 20% increase on flickr up to friendster, where the overhead is 90× that of Galois Push. For Giraph the overhead times are not as extreme but still generally worse. Even on flickr, Giraph's overhead time is already 23× that of Galois. On friendster, where Gemini was worst, Giraph *only* requires 73× the overhead time of Galois.

3) Single-Node vs. Distributed: TODO

- GaloisPush vs Galois CPU // Galois Pull verliert bei Dist.
- Giraph vs. Giraph
- Gemini vs Gemini

C. Breadth-first search

In these figures, Galois with 96 threads is shown. Again, we show the impact of Galois' thread count in subsection VI-E.

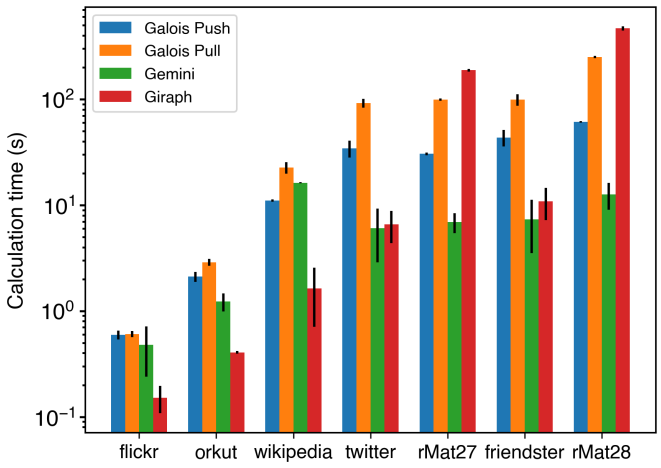


(a) Calculation times for BFS on a single node

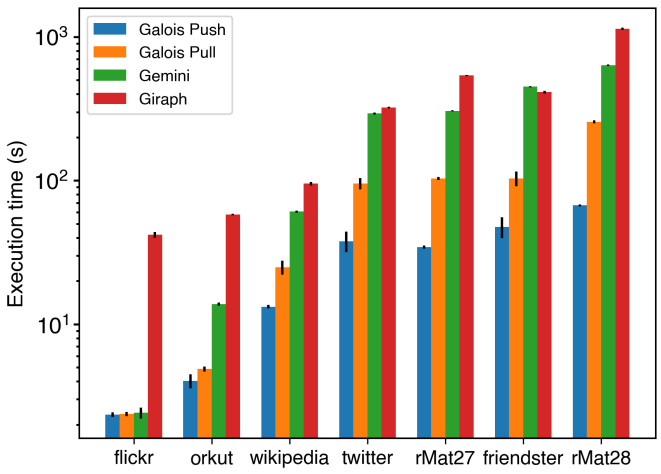
(b) Execution times for BFS on a single node

(c) Overhead time normalized by the graph size in million edges

Fig. 4: Average times on a single computation node, black bars represent one standard deviation in our testing



(a) Calculation times for distributed BFS



(b) Execution times for distributed BFS

Fig. 5: Average times on the distributed cluster, black bars represent one standard deviation in our testing

1) Distributed: For both the calculation and the execution times, Breadth-First Search shows similar behaviour as the distributed SSSP test case. All measurements can be seen in Figure 5.

First, the calculation times in Figure 5a. It shows Giraph having the shortest calculation times on the real-world graphs, while Giraph's calculatin times on both rMat27 and rMat28 are the worst of all frameworks.

Comparing the execution times in Figure 5b results in the same findings as with SSSP. While Gemini can compete with Galois on the small flickr graph, moving to larger data sets shows the worse performance of Gemini compared to Galois.

Much like when running SSSP, Giraph is slowest on all but one graph. Only on friendster is Gemini marginally slower, which was also the case for SSSP.

Both Galois implementations are again similar to the behaviour on SSSP. Galois Push is generally faster than the Pull alternative while both Push and Pull versions are faster than Gemini and Giraph across all graphs. This makes Galois Push the clear winner for distributed BFS.

D. PageRank

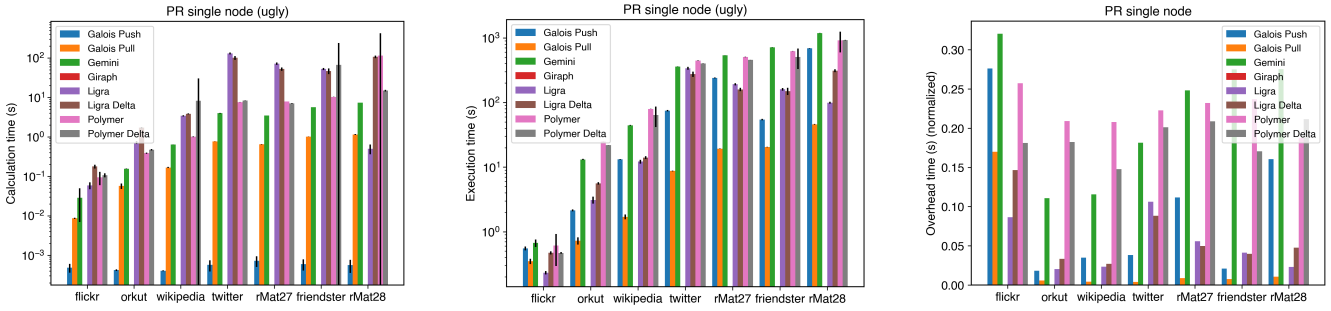
1) Distributed: The Figure 7 shows our results of PageRank on the distributed cluster.

First of all, Giraph was unable to complete the test because it required more than 250GB of RAM for rMat28. Thus this results is missing.

When comparing the calculation times in Figure 7a to the execution times in Figure 7b, we see similar behaviour of all frameworks. This means that unlike with SSSP or BFS, the calculation times and execution times are similar with respect to the relations of the frameworks to one another. More specifically, there are no overhead outliers like it was the case with Giraph on SSSP.

E. Galois speedup

Analyzing the calculation time speedups for Galois, we can compare how or if the different algorithms benefit from increasing thread numbers.

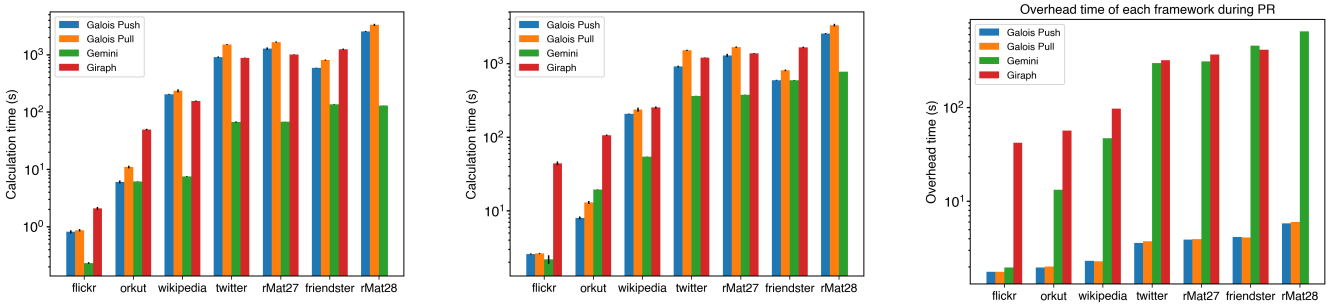


(a) Calculation times for PR on a single node

(b) Execution times for PR on a single node

(c) Overhead time normalized by the graph size in million edges

Fig. 6: Average times on a single computation node, black bars represent one standard deviation in our testing



(a) Calculation times for distributed PR

(b) Execution times for distributed PR

(c) Overhead time

Fig. 7: Average times on the distributed cluster, black bars represent one standard deviation in our testing

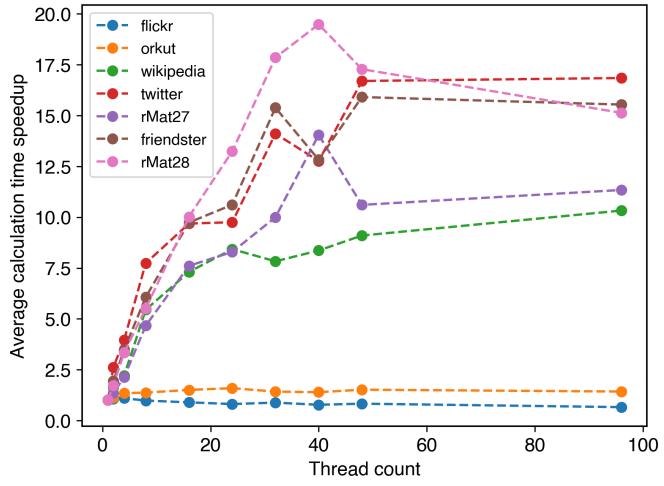


Fig. 8: Calculation time speedup with increasing thread count for Galois Single-source Shortest-paths

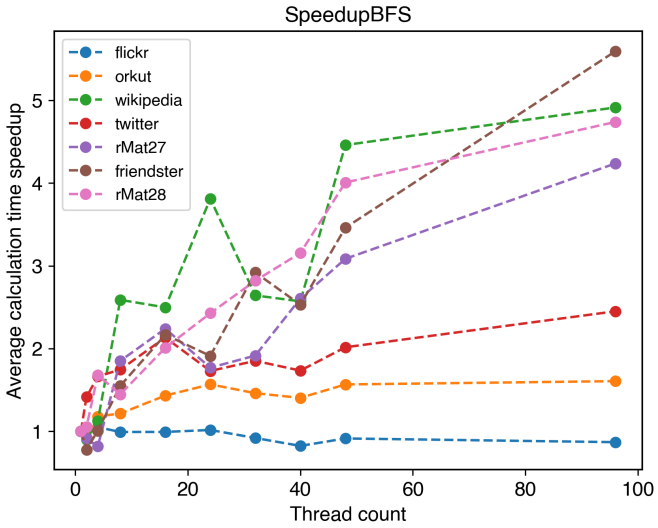


Fig. 9: Calculation time speedup with increasing thread count for Galois Breadth-first search

1) Single-source Shortest-path: Starting with SSSP which is the algorithm that really is at an advantage when using many threads in Figure 8.

For all larger graphs, speedup is in most cases very close to optimal up to about 8 threads. Twitter has the best speedup, requiring only 38% the calculation time with 2 threads compared to one, 25% using 4 and 12.9% using 8 threads. Behaviour on friendster is similar with 52% at 2 threads, 29% at 4 and 16% at 8 threads compared to one.

Anything above 32 threads however no longer helps decrease the computation time, in some cases even the opposite e.g. calculation on rMat28 is actually slower with 48 (13% slower) or 96 threads (28% slower) compared to 40 threads.

Small graphs like flickr or orkut, neither benefit from more threads nor is the performance held up by synchronization overhead.

2) Breadth-first search: For our speedup results on BFS, Figure 9 shows the calculation time speedup of Galois' BFS.

Flickr is sped up by about 5% with 4 threads, any more than that will actually decrease performance, making computation time up to 15% (96 threads) longer.

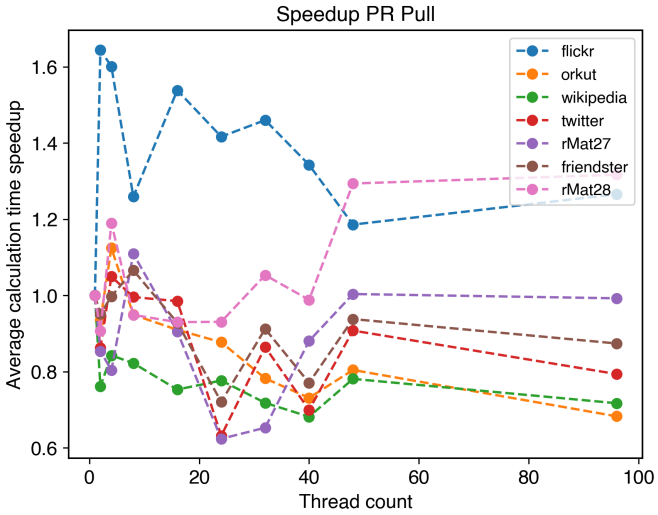
On all graphs, the speedup never exceeds 6× even when using 96 thread. The initial speedup when switching from one to two threads is actually smaller than one, thus decreasing performance, on 4 of 7 graphs. Only flickr, twitter and rMat28 can benefit slightly by a speedup of 2%, 41% and 5% respectively.

When comparing four threads to one, BFS on all but two graphs can be sped up by anywhere from 5% (flickr) to 68% (rMat28). The speedup is thus only possible to a very small degree. For the other two graphs, computation on friendster with 8 threads is just as fast as one thread and computation on rMat27 is actually 19% slower.

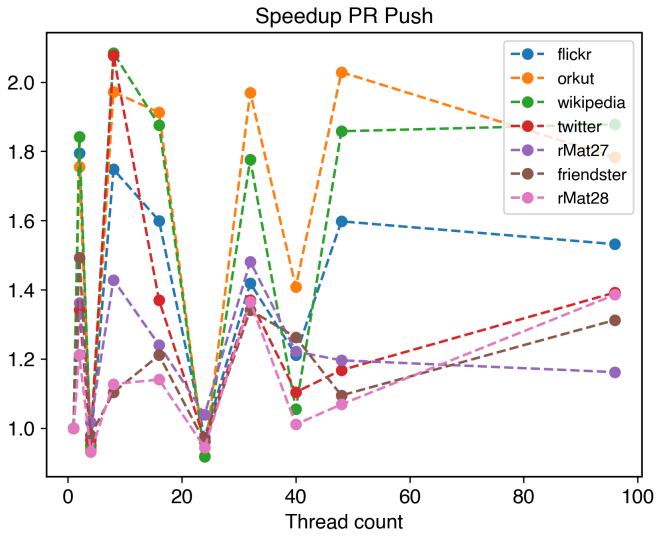
3) PageRank: We want to first take a look at the results for PageRank in Pull mode, seen in Figure 10a. This is a perfect example for an algorithm that does not benefit from multithreaded computation. Computation is hardly sped up on any graph other than flicker, where the reached maximum is 64%. This maximum is reached at two threads, with speedup steadily declining above that. The rMat28 is the only other graph of one could say computation was sped up. Here we reached a maximum speedup of 31% at 96 threads. All 5 other graphs only reach a speedup greater or equal to 1 in just one or two cases and if so only by a small margin. Computation on Orkut and Twitter reaches a speedup maximum of 12% and 5% at 4 threads, while being less or equal to 1 in all other cases. The wikipedia graph is never sped up. Friendster and rMat27 can be sped up by 6.5% or 10% respectively on 8 threads.

Speedup results on PageRank show odd behaviour in the Galois implementation. There is a significant performance loss on 4, 24 and 40 threads that is far from the expected behaviour. This is most visible for the Push variant seen in Figure 10b, we validated the shown results two times. Especially, the speedup for 24 threads is (by interpolating between 16 and 32 threads) expected to be anywhere between 25% and 94%. Actually however, the system does not reach a speedup of more than 4% on any graph, with only rMat27 actually reaching a value greater than 1. On all other graphs, using 24 threads is anywhere from 3% (flickr) to 9% (wikipedia) slower than using just one thread.

Similar yet less pronounced behaviour is oberservable for Pull in Figure 10a. Here especially the values for 24 and 40 threads show a loss in performance. It is most visible on the values for twitter and friendster, where both values drop significantly compared to the neighbouring 32 and 48 thread results.



(a) PageRank Pull



(b) PageRank Push

Fig. 10: Calculation time speedup with increasing thread count for Galois PageRank Push and Pull algorithms.

VII. Discussion

VIII. Conclusion

ACKNOWLEDGMENTS

We are using the graph frameworks Galois [5], Ligra [8], Polymer [9], Gemini [7] as well as Apache Giraph [4].

Also we use Gluon [6] for the distributed Galois setups.

Gemini [7]

APPENDIX A
INSTALLATION GUIDES

The complete installation guides with steps on how to run the same algorithms we used is available in our repository[5]

In case of frameworks like Giraph, where for example no BFS algorithm is provided by the framework itself, the algorithm and a guide on how to include it in the framework can be found in the repository as well.

APPENDIX B
CONVERSION TOOL

In order to

We have written a number of conversion tools and installation guides to help users or developers with the use of the tested frameworks.

Our GitHub repository: http://www.github.com/serengti/Forschungsprojekt.

REFERENCES

[1] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, p. 103–111, Aug. 1990. [Online]. Available: https://doi.org/10.1145/79173.79181

[2] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 135–146. [Online]. Available: https://doi.org/10.1145/1807167.1807184

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *In Proceedings of Operating Systems Design and Implementation (OSDI*, pp. 137–150.

[4] Apache Software Foundation. (2020, Jun.) Apache Giraph. [Online]. Available: https://giraph.apache.org

[5] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 456–471. [Online]. Available: https://doi.org/10.1145/2517349.2522739

[6] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 752–768. [Online]. Available: https://doi.org/10.1145/3192366.3192404

[7] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 301–316. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

[8] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 135–146. [Online]. Available: https://doi.org/10.1145/2442516.2442530

[9] K. Zhang, R. Chen, and H. Chen, "Numa-aware graph-structured analytics," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 183–193. [Online]. Available: https://doi.org/10.1145/2688500.2688507

[10] J. Shun, G. Blelloch, J. Fineman, P. Gibbons, A. Kyrola, K. Tangwonsan, and H. V. Simhadri. (2020, Aug.) Problem Based Benchmark Suite . [Online]. Available: http://www.cs.cmu.edu/~pbbs/

[11] J. Kunegis, "Konect: the koblenz network collection," 05 2013, pp. 1343–1350.

[5] https://github.com/SerenGTI/Forschungsprojekt/tree/master/documentation