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: Section**

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

TODOs

- Appendix für Setup Guides - md zu pdf

I. Introduction

//TODO: Section

This paper makes the following contributions:

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

II. Related Work

//TODO: Section

III. Preliminaries

//TODO: Section beschreiben

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 is defined as

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

and in the *undirected* case, the direction is no longer relevant. Thus, in an undirected graph for each $(x, y) \in E$, it holds $(x, y) = (y, x)$. The size of a graph is defined as the number of edges $|E|$ [**?**]. Independently of the graph being directed or not, a graph can be *weighted*. In this case a function $w : E \to \mathbb{R}$ is introduced, that maps an edge to a numerical value, further describing the relation.

A *Path* from starting vertex s to target vertex t is a sequence of vertices

$$P = (x_1, x_2, \ldots, x_n) \in V^n$$

with the condition $(x_i, x_{i+1}) \in E$ for each $i \in \{1, \ldots, n-1\}$ and $x_1 = s, x_n = t$. Thus we call a target t *reachable* 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 the shortest path from a starting vertex to every other vertex in the input graph. Input to the problem is a weighted graph $G = (V, E)$ and a start vertex $s \in V$. Output is the shortest possible distance from s to each vertex 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 sequential implementations are Dijkstra's algorithm or BellmanFord [1]–[3].

C. Breadth-First Search

Breadth-first search (BFS) is a search problem on a graph. It requires an unweighted graph and a start vertex as input. The output is a set of vertices that are reachable from the start vertex. In some special cases, a target vertex is also given. 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 one i.e. all neighbors of the start vertex are checked before checking paths of length two and so on. The search algorithm, where the paths of maximum length are checked first is called Depth-First search.

D. PageRank

//TODO: Aufbau unschön, Formel fehlt *PageRank* (PR) is a link analysis algorithm that weighs the vertices of a graph, measuring the vertices relative importance within the graph [4]. The algorithm was invented by Sergey Brin and Larry Page, the founders of Google. To this date, Google Search uses PageRank to rank web pages in their search engine results.

This represents a centrality metric of the vertices. 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. 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.

An optimization to the traditional PageRank implementation is called *Delta-PageRank*. The PageRank score of a vertex is only updated if the relative change of the PageRank is larger than some user-defined delta. This effectively reduces the amount of vertices for which the PageRank has to be recalculated in following iterations.

E. Push and Pull Variants

Many parallel graph algorithms, as well as the three we consider here, are implemented by iterating over vertices, or edges [10]. The respective vertex or edge, which is considered in an iteration, is also called *active* vertex or edge. Applying an operator to this *active* vertex or edge is called an *activity*. This operator considers always only a local so called *neighborhood* to the *active* vertex or edge. To this *neighborhood* belong only vertices and edges, which are in the direct surrounding of the *active* vertex or edge, which allows to parallelise the iteration relatively easy.

In general such an operator can change the whole structure of the *neighbourhood*. But in the following we assume, that an operator changes only *labels* of vertices or edges in the *neighbourhood*. *Labels* of edges also called *weights*. Often this operator can be implemented in two different ways, called *push style* or *pull style*. A *push-style* operator reads the *label* of the *active* vertex or edge and updates the *label* of its *neighborhood*. In contrast the *pull-style* operator reads all values of its *neighborhood* and updates the value of the *active* vertex or edge.

Pull-style operators need less synchronization in parallel implementations, because unlike *pull style* there is only one write and many read operations. Thus locks can be avoided. So *push-style* operators are more efficient, if there are only a few *active* vertices or edges at the same time, or the *neighborhoods* do not overlap, which can not be avoided in general. On the other hand *pull-style* operators are more efficient, if there are many *active* vertices or edges at the same time.

F. Bulk-Synchronous Parallel Model

The Bulk-Synchronous Parallel (BSP) model is a computation model developed by Leslie Valiant [5]. It is commonly used in computation environments with large amounts of synchronous computation.

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. The Messages have a user-defined form and should be as small as possible to keep the network traffic low. **//TODO: Congest Model, log limit in was?** 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 graph processing systems, Pregel [3] is based on the BSP computation model. We include Giraph, an open-source variant of Pregel in our evaluation. Pregel, Giraph and many frameworks similar to those were built to process large graphs reliably (offering fault tolerance) on large MapReduce infrastructures [6]–[8].

G. MapReduce

The MapReduce model is a computation infrastructure developed by Google to reliably handle large data sets on distributed clusters [9].

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.

Giraph [6] is an example of a system using this framework.

H. NUMA-awareness

//TODO: Section

IV. Framework Overview

We start with a short overview describing functionality and characteristics of several state-of-the-art graph processing frameworks. All of the following frameworks are part of our testing. **//TODO:**

A. Galois and Gluon

Galois [**?**] is a general purpose library designed for parallel programming. The system reduces the complexity of writing parallel applications by providing implicitly parallel (unordered or partially ordered) set iterators. These iterators perform operations optimistically, detect arising conflicts and resolve these by invoking inverse methods accordingly. The tasks in the sets can be ordered, the tasks may be executed out of order, but the conflict resolution will ensure semantic equivalence to sequential strictly ordered execution.

The graph analysis subsystem of Galois [10] provides a library of scalable data structures and a topology aware priority scheduler, including optimizations for distributed execution. The scheduler splits the tasks into bags according to a specified partial order, which in turn provide the cores with chunks

of tasks. A global map manages the various bags. Every thread keeps a lazy cache of a portion of the global map, in order to reduce the strain on the global map. Galois includes applications for many graph analytics problems, among these are SSSP, BFS and pagerank. For most of these applications Galois offers several different algorithms to perform these analytics problems and many options e.g. 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 [11] is a framework written for Galois as a middleware for distributed graph analysis applications. It reduces the communication overhead needed in distributed environments by exploiting structural and temporal invariants. Depending on the used graph partitioning policy only a subset of the messages of a naive approach must be sent (structural invariant). Gluon establishes a mapping of local vertex ID's to the order in which the values will be sent/received between the owner and each mirror. A message includes a bit vector where a one in the i-th position means that the according vertex of the established order has been updated. Thus a message only has to include updated values without the need to state the vertex ID (temporal invariants). Gluon is embedded in Galois, but can be integrated in other graph analysis frameworks as well [11].

B. Gemini

Gemini is a framework for distributed graph processing [12]. 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.

The real bottleneck of distributed systems is not the communication itself, but the extra instructions, as well as memory references and a lower usage of multiple cores compared to the shared memory counterparts. The main reasons for this are (1) the use of hash maps to convert vertex IDs between local and global states, (2) the maintenance of vertex replicas, (3) the communication-bound apply phase in the GAS abstraction, and (4) the lack of dynamic scheduling [12]. Gemini tries to work around all the problems mentioned above, by implemening a message-based system from scratch and getting rid of the extra mapping layer between shared-memory computation and communication. Therefore Ligra's push-pull computation model was adopted and applied to the distributed computation. Furthermore a chunk-based partitioning scheme was implemented, which allows to partition a graph without a large overhead. Gemini also implements a co-scheduling mechanism to connect the computation and inter-node communication.

Gemini is fairly lightweight and has 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 (PR), 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-F. This means that Giraph is based on computation units that communicate using messages and are synchonized with barriers [6].

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.

The 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 direct access to values of other vertices. The only visible information is the set of attached edges and their weights. 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. A vote-to-halt method is applied, i.e. if all vertices are inactive or if a user defined superstep number 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 [6]. 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 computations are performed. Giraph is thus, even when only using a single node, running in a distributed manner. Hence, 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. While writing

this paper, several new updates were pushed to Giraph's source repository[1].

D. Ligra

//TODO: Rückmeldung 2 umsetzen

Ligra is a lightweight graph processing framework for shared memory machines [2]. It offers a vertex-centric programming interface which can be used to apply a function to each vertex or outgoing edge of a set of vertices in parallel. While doing so the framework can generate a set of active vertices for the next iteration. This abstraction makes it well suited for writing graph traversal algorithms.

When mapping over edges Ligra optimizes algorithms by switching between a push-based and a pull-based approach based on the size of the set. When the size of the set is above a threshold a pull approach is used. If the threshold is not reached a push approach is used.

E. Polymer

Polymer is a NUMA-aware graph-analytics system that inherits the scatter- gather programming interface from Ligra [2]. Key differences are the data layout and access strategies, Polymer implements. The goal is to minimize random and remote memory accesses to improve performance.

The first optimization Polymer applies is data locality and access methods across NUMA nodes [1]. A general design principle for NUMA machines is to partition the input data so that computation can be grouped with the corresponding data on one node. Polymer adopts this and allocates graph data according to the access patterns. It treats a NUMA machine like a distributed cluster and splits work and graph data accordingly between the nodes. Vertices and Edges are partitioned and then allocated across the corresponding memory nodes of the threads, eliminating most remote memory accesses. However, some computation requires vertices to perform computations on edges that are not in the local NUMA-node. For this case, Polymer introduces lightweight vertex replicas that are used to initiate computation on remote edges.

Polymer furthermore has custom storage principles for application-defined data. For such data, the memory locations are static but the data undergo frequent dynamic updates. Due to frequent exchanges of application-defined data between the nodes, remote memory accesses are inevitable. Hence, Polymer allocates application-defined data with virtual adresses, while distributing the actual memory locations across the nodes. Thus, all updates are applied on a single copy of application-defined data.

Runtime states that are dynamically allocated in each iteration would however create overhead due to repeated construction of a virtual address space. These states are thus stored in a custom lock-less (i.e. avoiding contention) lookup table.

Polymer does not only optimize data locality but also its scheduling is custom. Time to synchronize threads on different cores increases dramatically with the growing number of involved sockets. Inter-node synchronization takes one order of magnitude longer time than intra-node synchronization [1]. Thus, Polymer implements a topology-aware hierarchical synchronization barrier. A group of threads on the same NUMA-node shares a partition of data. This allows them to first only synchronize with threads on the same NUMA-node. Only the last thread of each group sychronizes across groups (i.e. nodes). This behaviour decreases the amount of needed cache coherence broadcasts across the nodes.

Furthermore, Polymer switches between different data structures representing the runtime state, similiar to Ligra's behaviour. The main deciding factor to switch is the amount of active vertices relative to an application-defined threshold. Polymer uses a lock-less tree structure representing the active vertices. The leaves use bitmaps, which are efficient for a large proportion of active vertices. When only a small amount of vertices is active, the drawbacks of traversing through sparse bitmaps can be avoided by switching data structures.

Polymer inherits the programming interfaces `EdgeMap` and `VertexMap` from Ligra as its main interface.

V. Graph Formats

Since every frameworks uses different graph input formats, we supply a conversion tool capable of translating from EdgeList to the required formats.

The two most popular graph databases are those associated with the Koblenz Network Collection (KONECT) [13] and Stanford Network Analysis Project (SNAP) [14]. Data Sets retreived from one of them can be directly read and translated.

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

A. AdjacencyGraph

The AdjacencyGraph and WeightedAdjacencyGraph formats used by Ligra and Polymer are similar to the more popular *compressed sparse rows* format. 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 [15].

The file looks as follows

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

where commas are newlines. The files always start with the name of the format i.e. AdjacencyGraph or WeightedAdjacencyGraph in the first line, here shown as x. Followed by n, 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 vertices of a directed edge. The index o_k in the list of target vertices 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 vertices.

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

B. EdgeList

The EdgeList format is 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 edges $(s_1, t_1), (s_2, t_2), \ldots$ 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 edge per line and the vertex IDs s_i, t_i are separated with an arbitray amount of whitespace characters.

For a WeightedEdgeList, the edge weights are appended to each line, again separated by any number of whitespace characters.

C. 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 derives the number of edges from the file size, so there is no file header or anything similar allowed.

D. 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 and `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}.

VI. Evaluation

//TODO: Einleitung

A. Environment

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. Furthermore, Galois is a framework capable of utilizing hugepages. We include an evaluation using those on the single node as well. Unless mentioned otherwise, we always show results of each framework utilizing 96 threads (i.e. the maximum on our machines) for the single-node evaluation.

The complete benchmark log files and extracted raw results are available in our repository[3].

[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

B. 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[4] associated with the Koblenz Network Collection (KONECT)[13]. 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.

C. 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. Furthermore, the *overhead* is the time difference between execution time and calculation time. For measuring the calculation time, came up with the following:

- For Galois, we extract console log time stamps. Galois outputs "`Reading graph complete.`". Calculation time is the time from this output to the end of execution. 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.
  However, this method is the only way of retreiving any measurements without introducing custom modifications to the Galois source code.

[3]https://github.com/SerenGTI/Forschungsprojekt // das ist hier unschön.
[4]http://konect.uni-koblenz.de/

- 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.

Each evaluation 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.

D. 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 both in our evaluation.

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

E. Comparison of setup

VII. Results

//TODO: Rename Section???

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[5].
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 the frameworks Polymer and Ligra was straight forward and did not require any special treatment.

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

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

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.

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 VII-E.

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

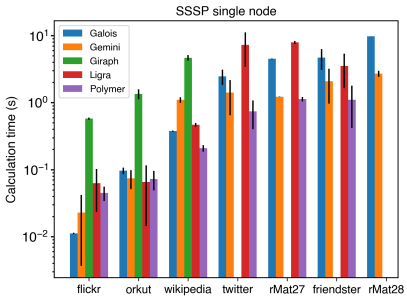
2) Distributed: Moving on to the distributed cluster, Figure 2 shows the benchmark results as calculation and execution times.

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 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.

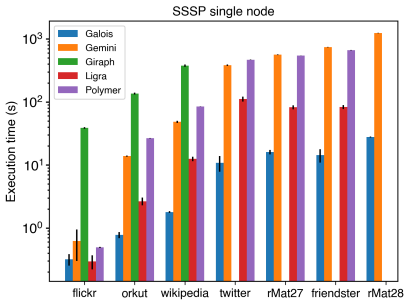
Both Galois implementations have significantly smaller execution times compared to Gemini or Giraph on all graphs (cf. Table II). You can see Gemini being worse by at least a factor of 4 compared to Galois Push on all graphs except flickr. Giraph's execution times in comparison to this are even worse,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

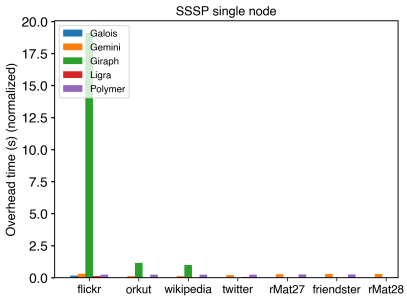
When taking a closer look at Giraph, it seems to not cope well with synthetic data sets. Analyzing the computation times in Figure 2a, we see that it is the fastest framework on our real-world graphs. And that with a considerable margin of other frameworks always taking at least 50% longer (lower



(a) Calculation times for SSSP on a single node



(b) Execution times for SSSP on a single node



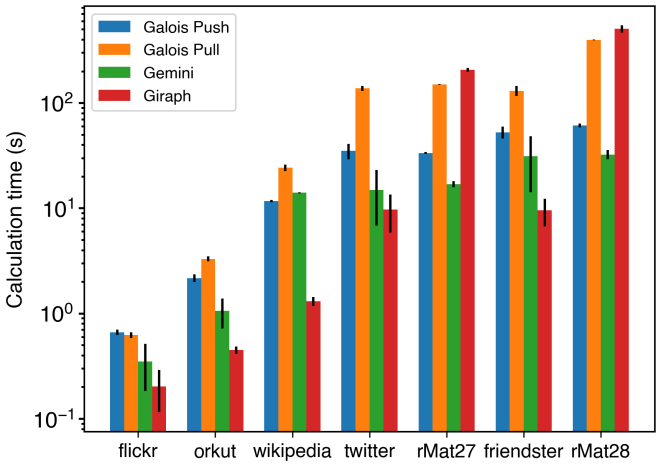
(c) Overhead time normalized by the graph size in million edges

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.

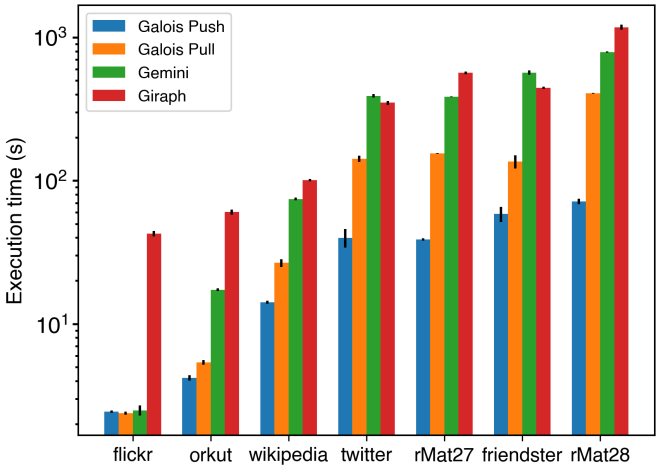
bound here is Gemini on flickr) up to Galois Pull needing 18× more time on wikipedia. On both synthetic graphs however, Giraph is actually the slowest to compute. Giraph requires 12× or even 15× the computation time of Gemini on rMat27 or rMat28 respectively.

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



(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.

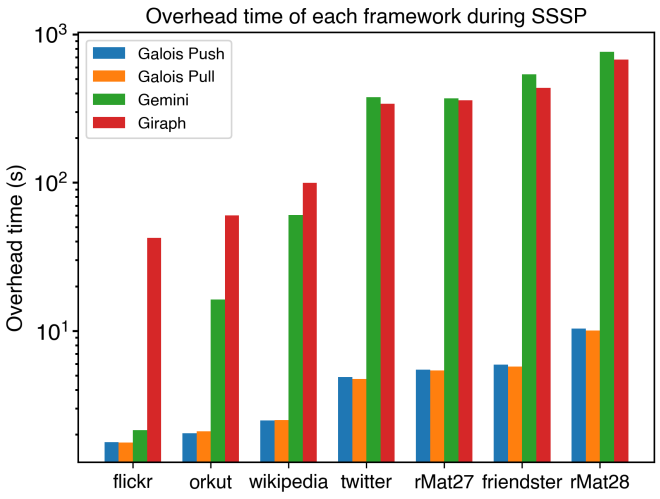


Fig. 3: Distributed Single-Source Shortest-Paths overhead times of each framework

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: Vergleich**

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

C. Breadth-first search

1) Single-Node: **//TODO: Hier fehlt noch Giraph, daher noch keine vollständige Auswertung.**

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

2) Distributed: For both the calculation and the execution times, Breadth-First Search shows similar behaviour as the distributed SSSP test case. This is expected since both are graph traversal algorithms starting in one source vertex. Hence, calculation complexity for each vertex and communication overhead is similar. 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) Single-Node: Ligra and Polymer support both regular and Delta-PageRank variants. Ligra's regular PR implementation is faster on 4 of 7 graphs. If the regular version is slower than delta, that is only by a small difference. Explicitly, regular is slower than delta by 19% on twitter, 17% on rMat27 or 6% for friendster. For the other graphs, the delta version is slower by a far greater margin, 50% on flickr, 45% on orkut, 13% on wikipedia and 68% on rMat28. Hence, we only show the results of Ligra's regular PageRank implementation in our evaluation. For Polymer we found the delta version to be faster on all graphs except rMat28. Delta-PR is on average 15% faster on the first six graphs, while only being 0.3% slower on rMat28. Thus, the following only shows Polymer's faster Delta-PR implementation.

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

2) 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

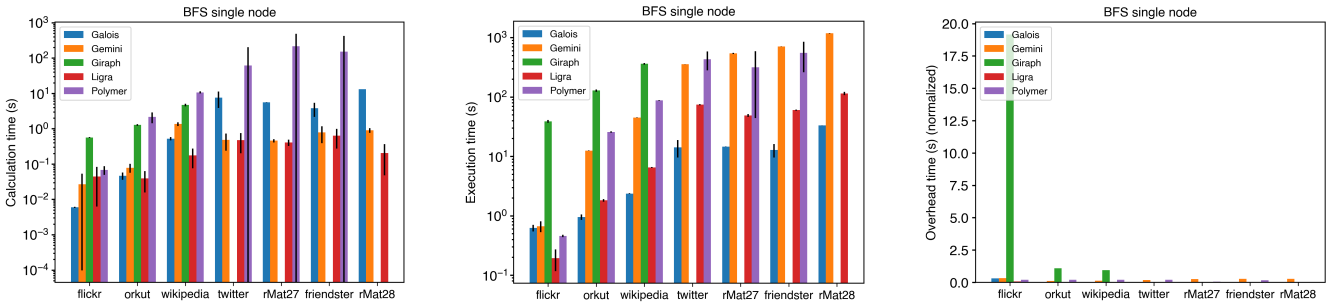
Analyzing the calculation time speedups for Galois, we can compare how or if the different algorithms benefit from increasing thread numbers.

1) Single-source Shortest-path: Starting with SSSP, we see an algorithm that benefits from many available 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 overall. It is 2.6× with 2 threads compared to one, 4× with 4, 7.7× with 8 and 9.7× using 16 threads. Behaviour on friendster is similarly good. Here speedup is 1.9× at 2 threads compared to one, 3.5× at 4, 6.1× at 8 threads and 9.7× at 16 threads. Anything above 16 threads however no longer helps decrease the computation time significantly on any graph. Speedup above 16 threads is always less than double the speedup of 16 threads. The maximum measured speedups are 10× (96 threads) for wikipedia, 17× (96 threads) for twitter, 11× (96 threads) for rMat27, 16× (48 threads) for friendster and 19× (40 threads) for rMat28. In some cases increasing thread counts even prolongues calculation time. For example calculation on rMat28 is actually slower with 48 or 96 threads compared to 40 threads. For 40 threads, the speedup is nearly 19×, on 48 threads 17× and with 96 threads only 15× compared to one thread.

Small graphs, i.e. flickr and orkut neither benefit from more threads nor is the performance significantly held up by synchronization overhead. Performance on flickr can not be sped up at all, with speedup on flickr being very close to 1 for 1 to 8 threads and between 0.7× to 0.9× from 16 to 96 threads. Orkut reaches maximum speedup of 1.6× at 16 threads. However on orkut, the speedup is always greater or equal to 1.

2) Breadth-first search: For our speedup results on BFS, Figure 9 shows the calculation time speedup of Galois' BFS. On all graphs, the speedup never exceeds 6× even when using 96 threads. For the smaller graphs (flickr, orkut), we have the same behaviour as on SSSP. Speedup is close to 1 in all cases, with orkut reaching a maximum speedup of 1.6× at 24 threads. On the larger graphs, speedup is possible but only to a very small degree. Wikipedia has a maximum speedup of 4.9× at 96 threads, for twitter it is 2.5× at 96 threads and 5.6× on friendster. The two synthetic data sets reach speedups of 4.2× and 4.7× for rMat27 and rMat28 on 96 threads.

3) PageRank: We want to first take a look at the results for PageRank in Pull mode, seen in Figure 10a. This is a

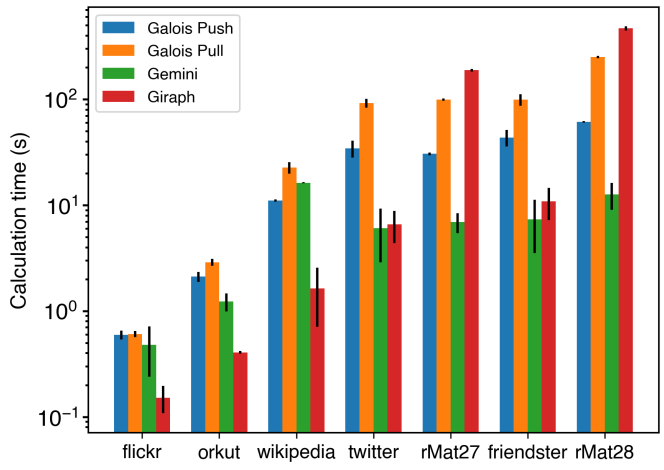


(a) Calculation times for BFS on a single node

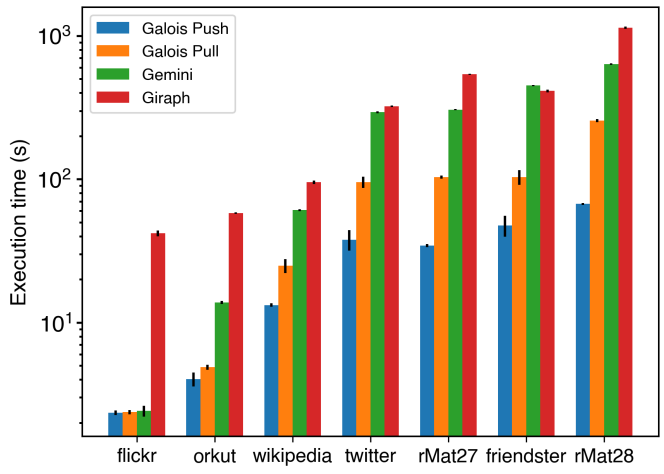
(b) Execution times for BFS on a single node

(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

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 at large thread counts. 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.

VIII. Hugepages

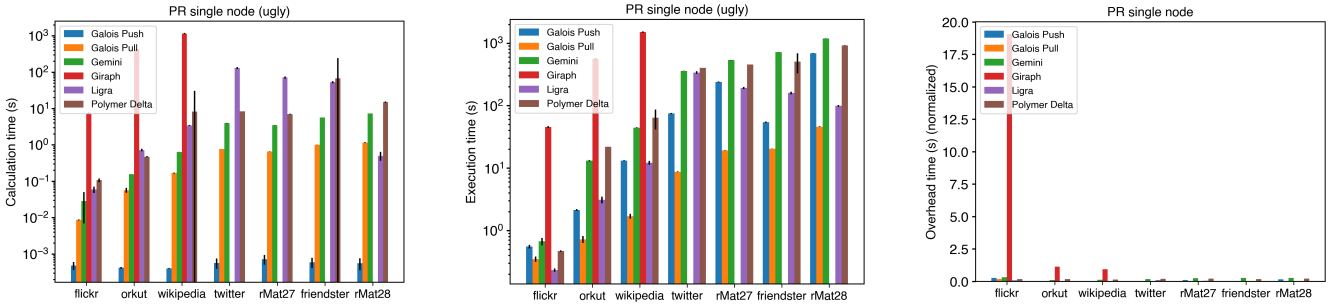
//TODO: Section

IX. Discussion

//TODO: Section

X. Conclusion

//TODO: Section

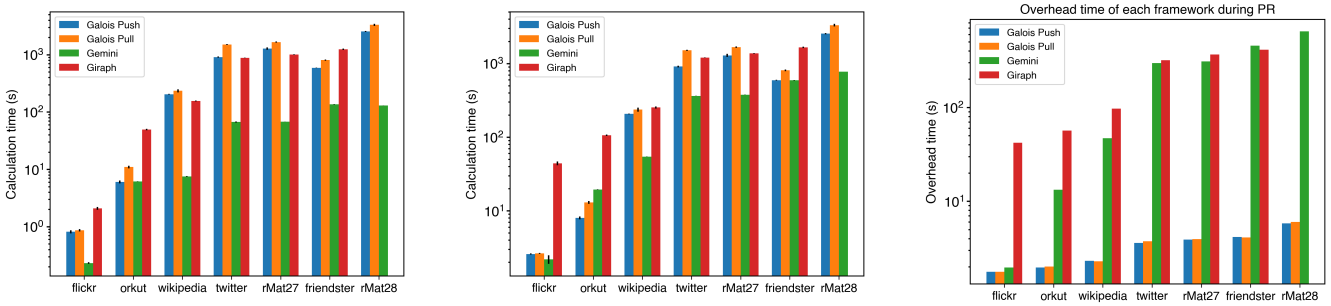


(a) Calculation times for PR on a single node

(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

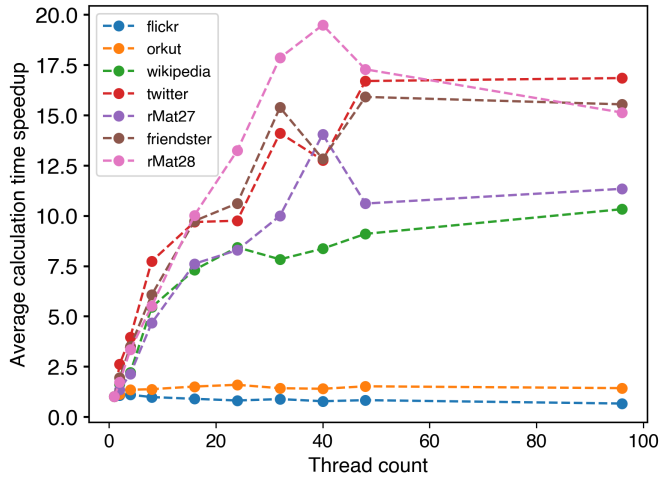


Fig. 8: Calculation time speedup with increasing thread count for Galois Single-source Shortest-paths

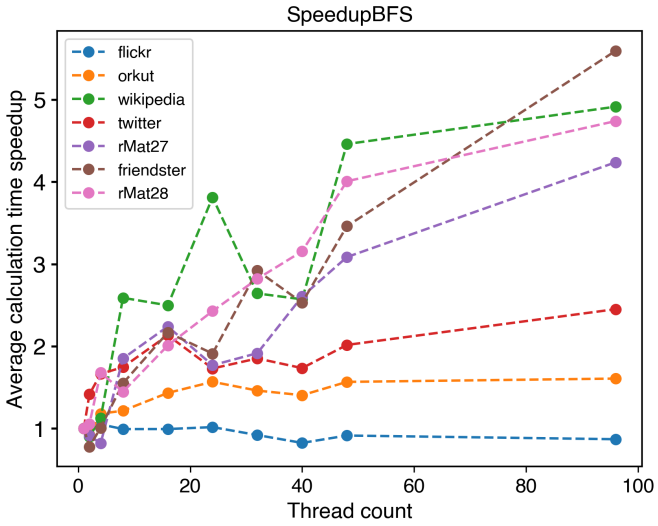
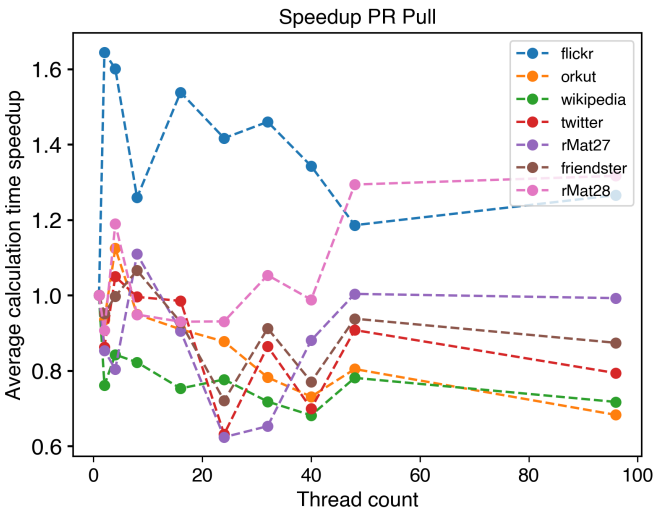
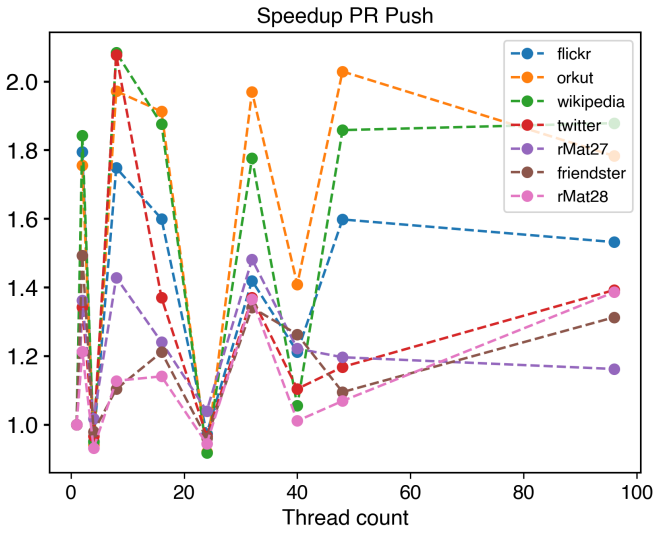


Fig. 9: Calculation time speedup with increasing thread count for Galois Breadth-first search



(a) PageRank Pull



(b) PageRank Push

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

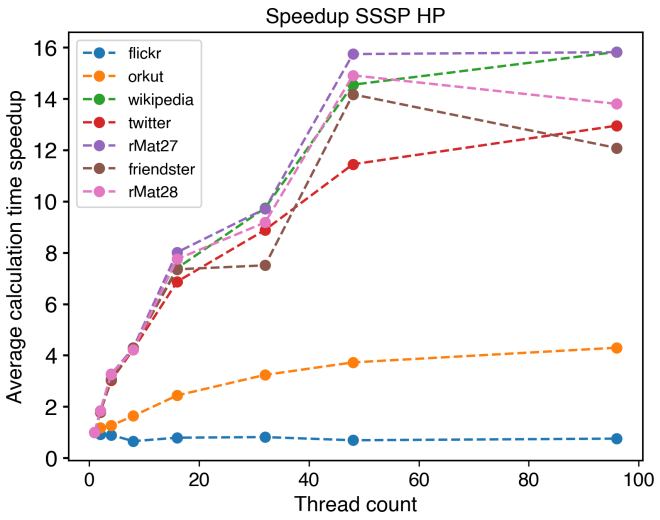


Fig. 11: Calculation time speedup with increasing thread count for Galois Single-source Shortest-paths with Hugepages

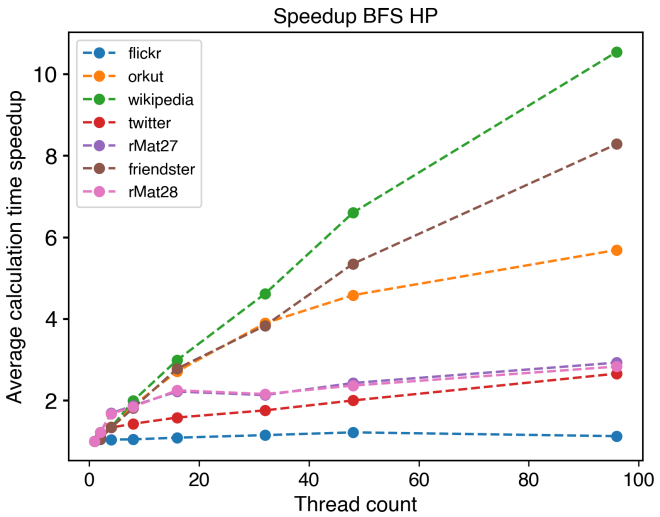
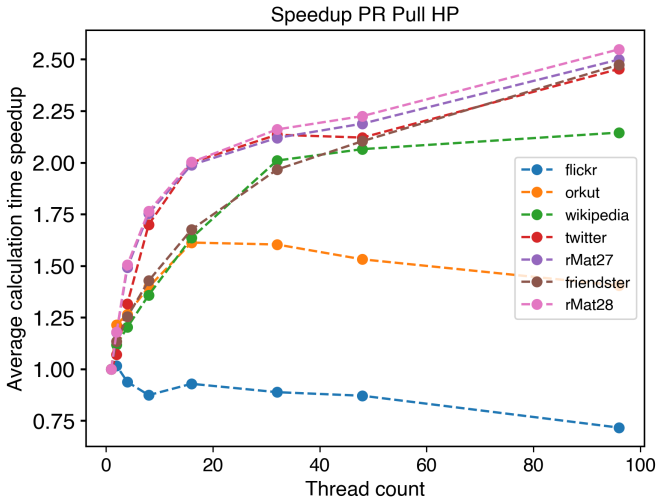
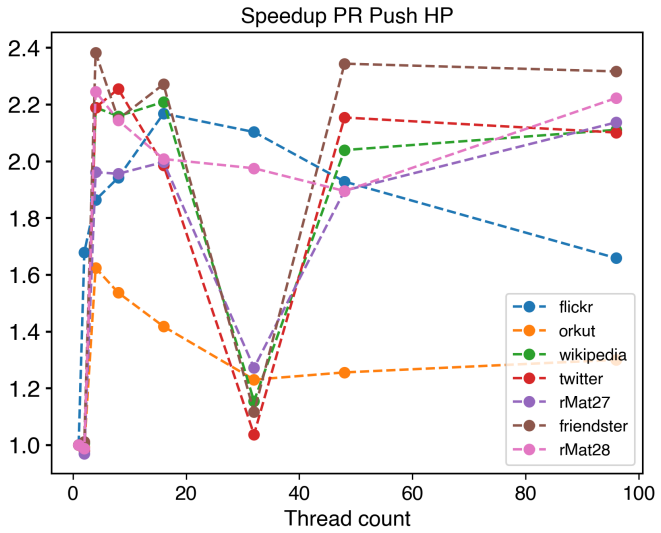


Fig. 12: Calculation time speedup with increasing thread count for Galois Breadth-first search with Hugepages



(a) PageRank Pull



(b) PageRank Push

Fig. 13: Calculation time speedup with increasing thread count for Galois PageRank Push and Pull algorithms using Hugepages.

Acknowledgments

//TODO: Beide zusammenfassen

Appendix A
Installation Guides

The complete installation guides with steps on how to run the same algorithms we used is available in our repository[6]

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] 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

[2] 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

[3] 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

[4] S. Brin and L. Page, "Reprint of: The anatomy of a large-scale hypertextual web search engine," *Computer Networks*, vol. 56, no. 18, pp. 3825 – 3833, 2012, the WEB we live in. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128612003611

[5] 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

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

[7] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 599–613.

[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 17–30. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[9] 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.

[10] 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

[11] 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

[12] 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

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

[14] Stanford University. (2020, Sep.) Stanford network analysis project. [Online]. Available: https://snap.stanford.edu

[15] 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/

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