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**

I. Introduction

1. Einführung, warum ist Graph analysis wichtig
- Graph processing is gaining increasing attentions in both academic and industrial communities. (Gemini)
- Graphen werden immer wichtiger in vielen Domänen **//TODO: welche**
-Many machine learning, data mining and scientific computation can be modeled as graph-structured computation, resulting in a new application domain called graph analytics. (Polymer)
- In recent years graph sizes have increased significantly. Thus performance and memory efficiency of the graph analysis applications is now more important than ever. **//TODO: Quelle**
- Many applications like, Wachsende graphen, benötigt für folgende Applikationen, verwenden graph algorithmen, diese müssen schnell und verteilt laufen
- Graphen wachsen schneller als CPUs **//TODO: Quelle**

2. Ausgangssituation und Problem
- dafür wurden Zahlreiche Framweworks entwickelt (Paper: Galois)
- Domänenspezifische Sprachen DSL (Paper: Galois)
- Es wurden auch Librarys und Technologien entwickelt (boost, hugepages, numa, ...)
- Um es zu erleichtern Graphprogramme zu schreiben (Paper: Galois)
- Größe der Graphen fordert Parallelität/Verteiltheit =¿ Probleme (Paper:Galois)
- Prolem der Vergleichbarkeit der Systeme
- Systeme entwickeln sich weiter und wurden nur zu deren Start verglichen
- Vergleiche wurden von Machern der Systeme erledigt

3. Was wir zur Lösung des Problems beitragen
- ersten, die unabhängig alle Systeme Vergleichen 2020
- Wir vergleichen die Frameworks nach folgenden Gesichtspunkten:
(Benutzerfreundlichkeit, Performanz/Varianz, Weiterentwicklung) - We compare several non-uniform memory access (NUMA) aware systems in terms of their performance on three graph algorithms (PageRank, SSSP, BFS).
- The comparison is performed on both real world data sets and synthetic graphs. To provide a comparison to a non-NUMA aware system, Giraph[1] is included in the testing. Giraph itself has often been compared to other state of the art systems like Pregel or GraphX. One of the most famous is the (closed source) distributed graph processing system Pregel. Pregel and several open source versions of . This results in suboptimal performance, due to missing cache locality, because the Pregel like systems are written in an object oriented manner relying on pointer chasing mechanisms. To this end several non-uniform memory access (NUMA) aware systems were proposed like Polymer [2], Galois [3] or Ligra [4].
- Comparison of several state-of-the-art graph processing frameworks

This paper makes the following contributions:

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

4. Wie ist das Paper aufgebaut
- Section 2: Relatet Work (Wie haben die Macher der Systeme die Systeme verglichen)
- Section 3: Preliminaries (Was muss man an technischen Details wissen)
- Section 4: Framework Overview (Welche Frameworks und was macht diese aus)
- Section 5: Evaluaten (Was wurde gemessen?, Wie wurde gemessen?, Was sind die Ergebnisse?)
- Section 6: Discussion (Warum sehen die Ergebnisse so aus, wie sie aussehen?)
- Section 7: Concluseion (Was lernen wir daraus?)

II. Preliminaries

This section briefly explains the concepts and applications necessary, but not directly related to our work. Initialy graphs and paths are defined, followed by explanations for various graph analysis applications. Afterwards a few models for compution on graphs or large data sets in general are explained as well as hugepages.

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 $|E|$ [5]. 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 [2], [4], [6].

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 and otherwise false. 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 [7]. 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, including the three we consider here, are implemented by iterating over vertices [8]. The vertex, to which the operator is applied, is called *active*. It's possible to iterate over edges and consider them as *active*, but in the following, whithout loss of generality, we will use the term *active vertex*. This operator considers only a so called *neighborhood* to the *active* vertex. To this *neighborhood* belong only vertices, which are in the direct surrounding of the *active* vertex. This allows to parallelise the iteration relatively easy, caused by the locality of the operator.

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 in the *neighbourhood*. *Labels* of edges are 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 and updates the *label* of its *neighborhood*. These operators are more efficient, if there are only a few *active* vertices at the same time, or the *neighborhoods* do not overlap, which can not be avoided in general. In contrast the *pull-style* operator reads all values of its *neighborhood* and updates the value of the *active* vertex. *Pull-style* operators need less synchronization in parallel implementations, because unlike *push style* there is only one write and many read operations. Thus locks can be avoided. So these operators are more efficient, if there are many *active* vertices at the same time.

F. Bulk-Synchronous Parallel Model

The Bulk-Synchronous Parallel (BSP) model is a computation model developed by Leslie Valiant [9]. 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. The *Congest model* is a closely related model and furthermore describes the messages. There, the message length has to be logarithmic in the graph size [10].

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, beginning 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 [6] 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 [1], [11], [12].

G. MapReduce

The MapReduce model is a computation infrastructure developed by Google to reliably handle large data sets on distributed clusters [13].

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

H. Hugepages

Most systems in use today use a so-called virtual memory management [14]. It is implemented in the kernel and represents an abstraction between the memory devices of a machine and the individual programs. Each program gets its own virtual leading address space. This simplifies the implementation of applications considerably and also increases security, since each program can only access its own virtual address space. The address areas are organized in so called pages, which in most cases are 4 KiB in size. In the Translation lookaside buffer (TBL) the most recent translations of virtual memory to physical memory are cached.

Translating from the virtual to the physical address space significantly reduces the performance of today's data processing systems, since the size of the RAM grows much faster than the size of the TBL [15], [16]. To counteract this problem, hugepages were developed, which are usually several mibibibytes in size. This minimizes the CPU time needed for table lookups, because there are much less TBL misses. This speedup is especially noticeable in applications that are very memory intensive, like graph processing systems.

III. Framework Overview

The following paragraphs are a short overview describing functionality and characteristics of several state-of-the-art graph processing frameworks. All of the described frameworks are part of our testing.

A. Ligra

Ligra is a lightweight graph processing framework for shared memory machines [4]. It offers a vertex-centric programming interface which consists of two routines. The `VertexMap` applies a user defined function to each vertex and `EdgeMap` applies a user defined function to each outgoing edge of a set of vertices. Active vertices can be represented by a `VertexSubset`. This data structure can be passed to both of the routines and is maintained by them. That makes Ligra well suited for expressing graph traversal algorithms.

When `EdgeMap` is used Ligra decides between a push and a pull style execution. The decisive factor is the number of active vertices and their outdegree. When this number is small a push style execution is used, otherwise a pull style execution is used. The default threshold is set to one twentieth of all edges. This hybrid approach can increase performance significantly [**?**]. When push and pull mode changes also the representation of the `VertexSubset` is changed. A push style execution corresponds to a sparse representation as array of the active vertex IDs, while a pull style execution corresponds to a dense representation as bitmap.

B. Polymer

Polymer is a Non Uniform Memory Access (NUMA) aware graph-analytics system that inherits the scatter-gather programming interface from Ligra [4]. 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 [2]. 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. Data such as the active vertices in each iteration are runtime states, that are dynamically allocated in each iteration. This allocation would however create overhead due to repeated construction of a virtual address space. These states are thus stored in a custom lockless (i.e. avoiding contention) lookup table.

Polymer does not only optimize data locality but also its scheduling is custom. The 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 [2]. 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. 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.

C. Gemini

Gemini is a framework for distributed graph processing [17]. It was developed with the goal to deliver a generally better performance through efficient communication. While most other 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 [18]. 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 [17]. To achieve this goal, Gemini, in contrast to the other 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. There are four main reasons for this. The first reason is the use of hash maps to convert the vertex IDs between the global and the local state. The second reason is the maintenance of vertex replicas on the different systems. Another reason is the communication-bound apply phase in GAS abstraction. And the last reason is the lack of dynamic scheduling.

Gemini tries to work around all the problems, 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 Betweenness-Centrality (BC).

D. Galois and Gluon

Galois [19] 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 them by invoking inverse methods accordingly. The tasks can be ordered, the ordering ensures a sequential strictly ordered semantic. Ordered tasks may still be executed out of order without affecting the ordered semantic due to the conflict resolution.

The graph analysis subsystem of Galois [3] 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 [20].

Gluon [20] 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 [20].

E. 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 II-F. This means that Giraph is based on computation units that communicate using messages and are synchonized with barriers [1].

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 vertex 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 using 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 [1]. Hadoop is a MapReduce infrastructure providing a fault tolerant basis for large scale data 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].

IV. 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) [21] and Stanford Network Analysis Project (SNAP) [22]. 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 [23].

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.

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.

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

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

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

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}.

V. Related Work

//TODO: Der erste Abschnitt hier kann eigentlich so wie er ist auch in die Introduction.

Almost every framework, that gets published in a paper compares itself to other, similar frameworks. Hence there already exists some information on the relative performance between frameworks. However, many of the most common frameworks today have been released many years ago. Because of that, two problems arise. First, the comparisons are based on very old versions of the frameworks, with no information on the current behaviour available. And second, many of the other frameworks are no longer in use because they were replaced by something else, making the comparisons difficult to interpret. We are thus providing an objective benchmark on what changed since the initial release, with a comparison between the most commonly used frameworks today.

It quickly becomes apparent that many of the comparisons provided

The comparisons often include PowerGraph or Apache's Spark GraphX, two frameworks that we did not include in our testing.

The most recent publication on Galois was in 2013 [3], along with a comparison to GraphLab, PowerGraph and Ligra, which is in our testing lineup itself. The results of their comparison is mainly, that Galois outperforms Ligra on BFS and SSSP on most graphs. Meanwhile, performance of Ligra and Galois on PR is similar but very dependent on the graph. In some cases, Ligra outperforms Galois, on other graphs Galois is faster than Ligra.

Gemini puts itself against Ligra, Galois, PowerGraph, PowerLyra and GraphX [17]. In a single-threaded scenario, Ligra was shown to be faster than both Galois and Gemini in SSSP and BFS applications. On the other hand, Gemini beat the other two in PR applications. Furthermore, Gemini is presented to be one order of magnitude faster than PowerGraph, PowerLyra and GraphX on a multi-node computation unit.

The publication on Ligra is mainly about the multicore behaviour of Ligra rather than a comparison to other frameworks [4].

Polymer provides a large set of comparisons to Galois, Ligra and X-Stream, all on a single computation node. In nearly all of their test cases, Polymer outperformed the other three frameworks [2].

VI. Evaluation

The main part of this paper is the evaluation, which consists of two parts. The first is a description and explanation of our testing environment and methodology. Followed by a presentation and discussion of the test results of the frameworks.

A. Testing Methods

The testing methods cover the test environment, including hardware and the setup of the frameworks, the graphs used, followed by the algorithms utilized and how the times are measured.

1) 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.

The 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) 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

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

[3]https://github.com/SerenGTI/Forschungsprojekt

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

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

Koblenz Network Collection (KONECT)[21]. Both the rMat27 and rMat28 were created with a modified version of a graph generator provided by Ligra (we changed the output format to EdgeList).

3) Algorithms: The three problems Breadth-first search (BFS), PageRank (PR) and Single-source shortest-path (SSSP) were used to benchmark each 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. We chose SSSP and BFS because they are iterative traversal algorithms. Active vertices typically are locally concentrated in the graph. The results of these algorithms can give some insight on the behaviour of the framework with other, similar behaving algorithms. PageRank on the other hand is an algorithm that is very different to SSSP or BFS for that matter. With PR, there are many active vertices spread across the entire graph, enforcing different data handling strategies from the framework.

In detail, the algorithms for each framework are:

- Ligra supports SSSP based on BellmanFord, BFS and two implementations of PageRank. The two implementations are a regular PR and a Delta Variant.
- Polymer supports the same algoritms as Ligra.
- Gemini supports all of our tested algorithms and there are no setting options or specifications which implementations for the algorithms are used.
- Galois supports all of our tested algorithms too, with both a Push and a Pull variant for PageRank available. In the distributed scenario, there are Push and Pull versions for SSSP and BFS available as well. It also supports multiple implementations of the shared-memory allgorithms. The default implementation of SSSP is deltaTile. A lot of setting options are avilable as well, but we're gone with the defaults.
- Giraph does not natively supply a BFS algorithm, so in our comparisons a custom implementation is used. For SSSP, slight variations had to be made to the default implementation, to allow us to use different start vertices. For PageRank the supplied implementation is used.

4) 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. This includes time to read the input graph, initialization and any other tasks other than the actual user-defined algorithm. Measuring the execution time is straight forward and was done using console time stamps. For measuring the calculation time, we 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.
- 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.

B. Results

Ahead of the discussion of the results a few of the issues we encountered while testing are mentioned. Afterwards the benchmark results for SSSP, BFS, PR and the speedup of Galois given a varying number of threads are discussed in that order.

1) Encountered Issues: 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

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

significant performance impact on other tasks, it is still suboptimal.

4) Giraph ran us into disk space problems on multiple occasions. First, deleting files on the Hadoop distributed file system (HDFS) does not immediately free up disk space because the files are moved to a *recycling bin*-like location. Second, some log files that can easily be multiple gigabytes in size are stored outside of the HDFS and are never mentioned in the Giraph documentation.

On a plus side, setup of the frameworks Polymer and Ligra was straight forward and did not require any special treatment. **//TODO: Gemini? Außerdem sollte hier noch ein bisschen was hin, die beiden Zeilen sehen ja nur traurig aus.**

2) Single-Source Shortest-Paths: In this section, we compare the different frameworks in their performance on the SSSP algorithm. We first analyze the frameworks on a single computation node and compare to the distributed setup after that.

a) Single-node: Beginning with the single-node performance, Figure 1 shows the average calculation and execution times for SSSP on the different frameworks. Note, that Giraph ran out of memory (>250 GB) for all graphs larger than wikipedia (twitter, rMat27, friendster and rMat28). Thus, the data points on the larger graphs for Giraph are missing in the figures and our evaluation. Also, both Ligra and Polymer failed on rMat28.

Upon analyzing the calculation time (cf. Figure 1a), most obvious is the fact that Giraph is at least one order of magnitude slower than any other framework. The only exception is Gemini, where Giraph is *only* 4× slower on wikipedia. In general however, Giraph is on average 24× slower than the other frameworks on flickr, 17× on orkut and 12× on wikipedia.

Comparing the other frameworks (i.e. excluding Giraph in the further comparisons), shows them to perform similar on the smaller graphs. There, the average calculation times of the four frameworks (Galois, Gemini, Ligra, Polymer) are 35ms on flickr, 77ms on orkut and 538ms on wikipedia. The four frameworks are close to this average for the smaller graphs. Galois and Gemini being 24ms and 12ms faster than the average. Ligra and Polymer are slower by 27ms and 9ms. Orkut is the graph where all the four frameworks are within 20ms of their average, Galois being the outlier at 19ms slower than average. For wikipedia, Galois and Ligra are close to each other, while Gemini is 560ms (104%) slower than the average of the four frameworks. And Polymer is the fastest framework on wikipedia, being 330ms (61%) faster than the others. For the larger graphs, Polymer has the shortest computation time as well. Gemini is second, taking about 1.9× times the computation time of Polymer on twitter and friendster while being equally fast (8% longer) on rMat27. Galois and Ligra however have much longer computation times compared to Polymer. Galois takes anywhere from 3.3× (twitter) to 4.2× (friendster) the computation time of Polymer. Ligra requires between 3.2× (friendster) and 9.8× (twitter) the computation time of Polymer on the larger graphs. This makes Ligra the slowest Framework on both twitter and rMat27. Galois is the slowest on friendster. However, while especially Galois is comparably slow to Polymer in the computation time, it is important to keep in mind that Polymer could not finish computation on rMat28. Meanwhile both Gemini and Galois managed just fine.

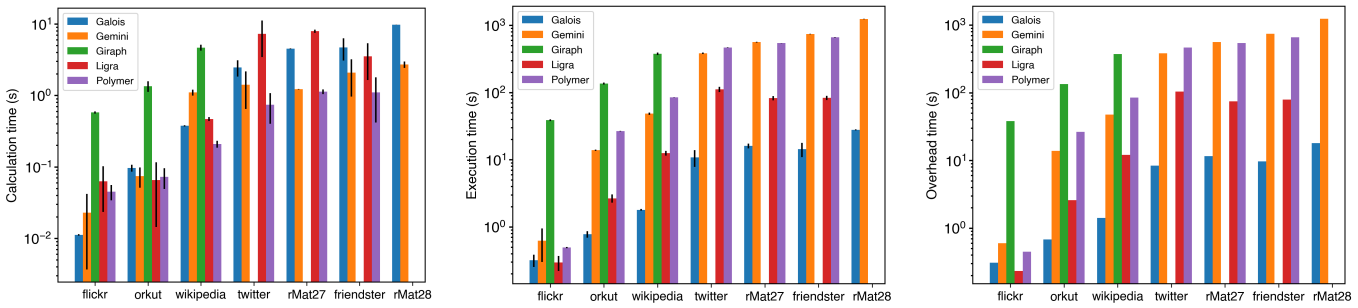
TABLE II: Distributed SSSP Execution Times and Their Realation to Galois Push

Data Set	Galois Push		Galois Pull		Gemini		Giraph	
	×G.Push	time (s)	×G.Push	time (s)	×G.Push	time (s)	×G.Push	time (s)
flickr	1.0	2.4	0.97	2.4	1.02	2.5	17.46	42.6
orkut	1.0	4.2	1.28	5.4	4.12	17.3	14.38	60.4
wikipedia	1.0	14.2	1.88	26.6	5.26	74.5	7.11	100.8
twitter	1.0	40.0	3.56	142.2	9.79	391.1	8.76	349.9
rMat27	1.0	39.0	3.98	154.9	9.91	386.0	14.52	565.8
friendster	1.0	58.5	2.32	136.0	9.72	568.9	7.59	444.0
rMat28	1.0	71.5	5.7	407.9	11.07	792.0	16.5	1180.2

The execution times show Galois to be the clear winner on most graphs (cf. Figure 3b). It has the smallest computation times on all graphs except flickr, where Galois is 25ms (8%) slower than Ligra. On the other 5 graphs however, Galois has the smaller execution time by one or sometimes even two orders of magnitude. For example, on orkut Galois requires 0.78s to execute, while the other frameworks require between 2.66s (Ligra) and 26.6s (Polymer). This goes on for the larger graphs, Ligra being second-fastest. Polymer and Gemini close together but both always at least one order of magnitude slower than Galois.

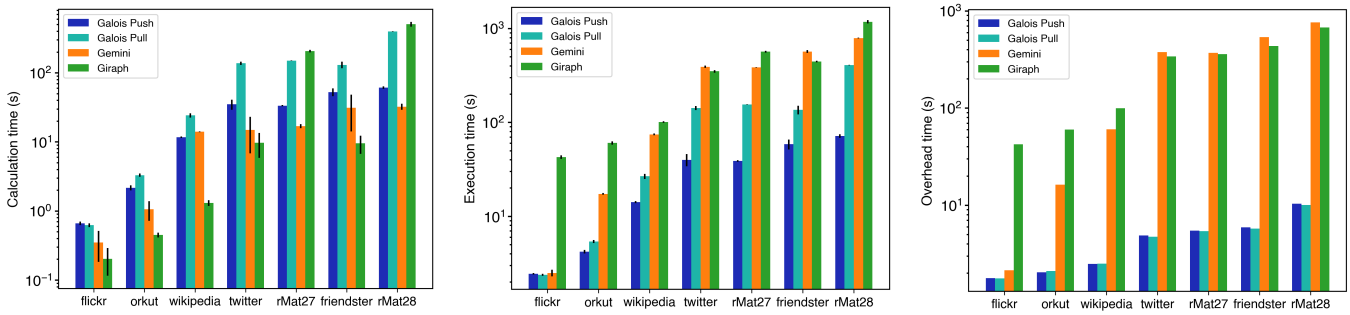
b) Distributed: On the distributed cluster, we find similar results as on the single node (cf. Figure 2). Both Galois implementations have significantly smaller execution times compared to Gemini or Giraph on all graphs (cf. Table II). You can see Gemini being worse by at least a factor of 4 compared 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. 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 is 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 on Pull. 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



(a) Calculation times (b) Execution times (c) Overhead time

Fig. 1: Average times for SSSP on a single computation node, black bars represent one standard deviation in our testing.



(a) Calculation times (b) Execution times (c) Overhead times

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

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 2c). 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) Breadth-first search: As with our results for SSSP, for BFS we will begin with our single-node results before looking at the distributed scenario.

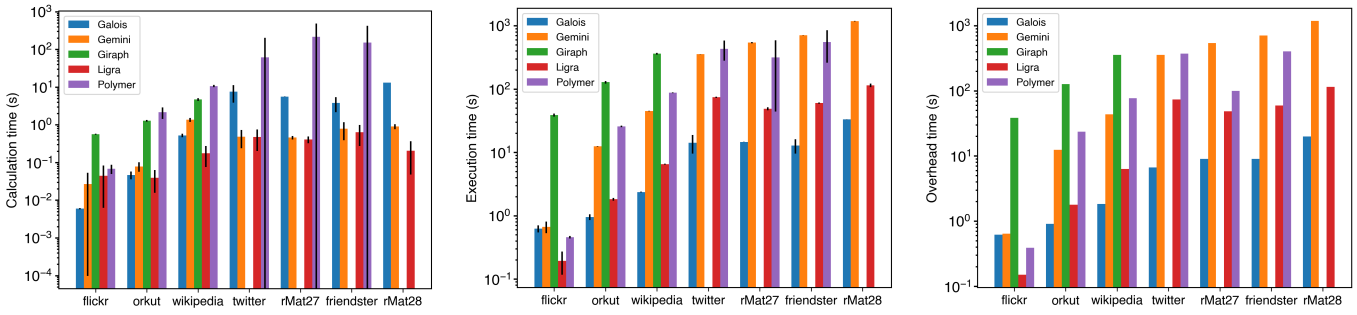
//TODO: Einleitung, analog SSSP

a) Single-Node: Just like with SSSP, Giraph ran out of memory (>250 GB) on any graph larger than wikipedia. Also, Polymer failed to complete on rMat28. Ligra, that failed during SSSP however completed our benchmark for BFS.

The calculation times provided in Figure 3a show Gemini and Ligra to be comparable in their performance. Their computation times deviate less than 151ms on all graphs except wikipedia and rMat28, with Ligra being the faster framework on most graphs. Ligra is between 2ms (twitter) and 151ms (friendster) faster than Gemini. In turn, Gemini is 17ms faster than Ligra on flickr. Only on wikipedia and rMat28, there is a noticeable difference between the two frameworks. Ligra is 1.2s (7.8×) faster than Gemini on wikipedia and 696ms (4.4×) faster on rMat28.

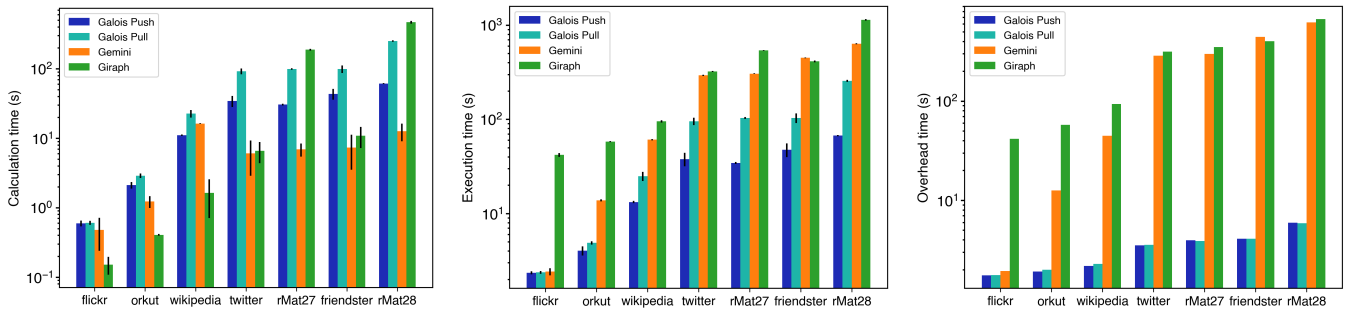
For the remaining frameworks, we compare to Ligra since it is generally slightly faster than Gemini. Giraph and Polymer are one to two orders of magnitude slower than Ligra. The only exception to this is Polymer on flickr, here Polymer is just 24ms (35%) slower than Ligra. Giraph takes between 13× and 32× longer than Ligra on flickr, orkut or wikipedia. As we said, Polymer is comparable to Ligra on flickr, but for the other graphs, its computation times are even longer than those of Giraph. The upside is though, that Polymer managed – contrary to Giraph – to finish computation on some larger graphs. Polymer takes between 55× and 530× longer than Ligra on the graphs larger than flickr. Especially interesting is here the difference between the times of rMat27 and friendster. There are 20% (439M) more edges in friendster, yet the computation for the synthetic rMat27 takes 42% longer. Galois calculation performance is comparable to Gemini or Ligra on the smaller three graphs (flickr, orkut and wikipedia). Only on the larger graphs is Galois slower than Gemini or Ligra, meanwhile Galois is always faster than Polymer by one order of magnitude.

The execution time results are very similiar to our findings of SSSP (cf. Figure 3b). Just like on SSSP, Galois is the fastest framework and Ligra is second fastest on all graphs except flickr. Again, Gemini and Polymer are comparable in their performance and at the same time the slowest frameworks on all graphs except flickr, orkut and wikipedia. On those three graphs, Giraph is the slowest, by a difference of one to two orders of magnitude compared to Galois. On the other graphs, Polymer and Gemini are one order of magnitude slower than Galois.



(a) Calculation time (b) Execution time (c) Overhead time

Fig. 3: Average times for BFS on a single computation node, black bars represent one standard deviation in our testing



(a) Calculation time (b) Execution time (c) Overhead

Fig. 4: Average times for BFS on the distributed cluster, black bars represent one standard deviation in our testing

b) 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 4.

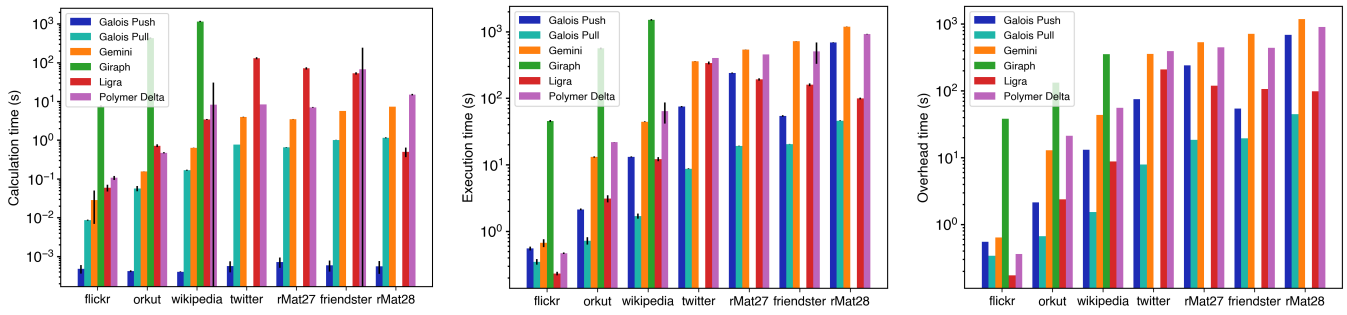
The results for calculation time (cf. Figure 4a) show Giraph to have very short calculation times on the real-world graphs, while Giraph's calculation times on both rMat27 and rMat28 are the worst of all frameworks. Thus, Giraph is fastest on the three smallest graphs and second fastest on twitter and friendster. On those two graphs, Gemini is fastest with only a small margin between the two frameworks. Just like with SSSP, the Galois implementations have the longest calculation time on the real-world graphs. And Galois is second slowest, on the synthetic graphs.

Comparing the execution times in Figure 4b results again in similar findings to 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. Similar to SSSP, Giraph is slowest on all but one graph. Only on friendster is Gemini marginally slower, this was also the case for 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.

4) PageRank: In this section we compare the PageRank performance of the various frameworks. As usual we begin with the single-node performance and finish by discussing the distributed variants.

a) 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 a range of 6% to 19% on twitter, rMat27 or friendster. For the other graphs, the delta version is slower by a far greater margin of 13% to 68%. 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. Giraph required more than the available 250 GB of RAM on any graph larger than wikipedia, hence all of Giraph's results for the larger graphs are missing here.

The calculation times show some odd behaviour of Galois Push. The required time is less than 1ms, regardless of the graph (cf. Figure 5a). Meanwhile there was no output produced, that would indicate any kind of error. These results would make the calculation times of Galois Push the smallest on all graphs, with a difference of at least one order of magnitude. However, we are very suspicious of these results and thus exclude the calculation time for Galois Push in further comparisons. Because the execution time of Galois Push is always considerably longer than the execution time of Galois Pull (cf. Figure 5a). This leads us to believe that the output that we used for our measurements contains an error.



(a) Calculation time (b) Execution time (c) Overhead

Fig. 5: Average times for PR on a single computation node, black bars represent one standard deviation in our testing

For the three graphs on which Giraph computed successfully, it is the slowest framework in both calculation and execution times. And that by a difference of three orders of magnitude in the calculation time and one to two orders of magnitude in execution time (cf Figure 5). On the larger graphs (i.e. those, where there is no data for Giraph), Gemini and Ligra are always slowest in execution time. Contrary to this, Galois Pull has the smallest execution times on all graphs except flickr (cf. Figure 5b). Ligra is fastest on flickr, while being second fastest on wikipedia, rMat27 and rMat28. Galois Push is second fastest on orkut and friendster. Interestingly, the execution time for Ligra is at a maximum for twitter. The required time is steadily decreasing with increasing graph size.

b) Distributed: Our benchmark results of PageRank on the distributed cluster can be seen in Figure 6. First of all, Giraph was unable to complete the test on rMat28 because it ran out of memory, thus this result is missing. When comparing the calculation times in Figure 6a to the execution times in Figure 6b, 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.

Gemini has the shortest calculation times on all graphs (cf. Figure 6a). The two Galois implementations are second on flickr, orkut, friendster and rMat28, with Giraph being second on the others. Generally, the calculation of Galois Push is anywhere from 6% (flickr) to 46% (orkut) faster than the Pull counterpart.

This applies to the execution times in almost the same way (cf. Figure 6b). Gemini is the fastest on all graphs except orkut, where both Galois implementations are faster. Galois Pull takes 13s, whereas Gemini requires 19.4s on orkut. Again, as for the calculation times, Galois is the second fastest framework on flickr, wikipedia, friendster and rMat28. And Galois Push has smaller execution times than the Pull version because the overhead times for both implementations are similar (cf. Figure 6c).

5) Comparison of a Single-Node to the Distributed Cluster: Until now, we have only compared the frameworks with each other, under the same setup circumstances. Hence, a comparison of the same framework, in single-node versus distributed setup is the content of this section. Of course only frameworks that were tested in both setups are shown. We focus primarily on the results of the execution times, the data can be seen in Table III. We show the results of Galois Push for the 5-node cluster in the table, since it is faster than Pull.

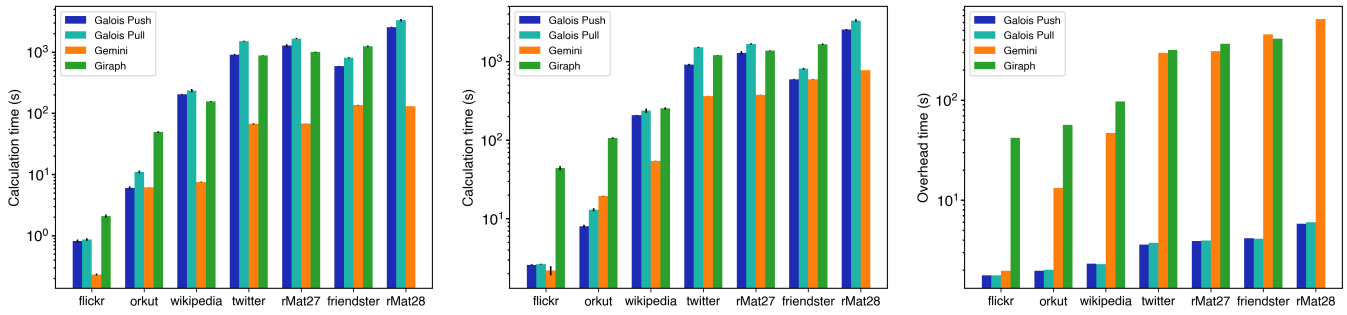
TABLE III: Execution Times in Seconds on a Single Node vs. 5-Node Distributed Cluster

		Galois		Gemini		Giraph	
	Graph	1N	5N	1N	5N	1N	5N
SSSP	flickr	**0.3**	2.4	**0.6**	2.5	**38.8**	42.6
	orkut	**0.8**	4.2	**13.9**	17.3	135.9	**60.4**
	wikipedia	**1.8**	14.2	**48.5**	74.5	377.2	**100.8**
	twitter	**10.8**	40.0	**383.0**	391.1	-	**349.9**
	rMat27	**16.0**	39.0	563.5	**386.0**	-	**565.8**
	friendster	**14.4**	58.5	742.6	**568.9**	-	**444.0**
	rMat28	**27.8**	71.5	1236.7	**792.0**	-	**1180.2**
BFS	flickr	**0.6**	2.3	**0.7**	2.4	**38.9**	41.9
	orkut	**0.9**	4.0	**12.5**	13.8	128.2	**58.1**
	wikipedia	**2.4**	13.2	**44.9**	60.9	360.7	**95.1**
	twitter	**14.2**	37.9	355.0	**293.2**	-	**322.3**
	rMat27	**14.6**	34.4	540.8	**305.3**	-	**539.2**
	friendster	**12.8**	47.6	708.4	**450.2**	-	**412.3**
	rMat28	**33.1**	67.1	1178.7	**634.3**	-	**1135.6**
PR	flickr	**0.3**†	2.6	**0.7**	2.2	45.6	**44.1**
	orkut	**0.7**†	8.0	**13.1**	19.4	561.3	**106.2**
	wikipedia	**1.7**†	206.5	**44.4**	54.4	1501.3	**252.6**
	twitter	**8.7**†	910.9	**359.9**	363.5	-	**1200.2**
	rMat27	**19.2**†	1287.4	536.3	**376.3**	-	**1369.5**
	friendster	**20.4**†	593.3	716.5	**591.1**	-	**1655.0**
	rMat28	**46.0**†	2540.2	1188.5	**774.2**	-	-

(†) Results of Galois Pull shown.

There are some, rather large differences in the execution times of each framework. Galois is consistently faster on a single computation node compared to the distributed setup. On both SSSP and BFS, the margin between the two setups is already noticeable. The distributed scenario requires from 2× (BFS, rMat28) to 7.8× (SSSP, wikipedia) the execution time of single-node Galois. For PR however, the difference is multiple orders of magnitude large. Distributed Galois requires from 8.6× to 121× more time than single-node Galois.

Gemini's distributed calculation only requires more time on the smaller graphs (i.e. flickr to wikipedia). There, the distributed scenario is anywhere from 1.1× (BFS, orkut) to



(a) Calculation time (b) Execution time (c) Overhead

Fig. 6: Average times for PR on the distributed cluster, black bars represent one standard deviation in our testing

4.2× (SSSP, flickr) slower. Twitter is the tipping point for Gemini's SSSP and PR algorithm. Here, execution times on the single-node and distributed cluster are within 2% of each other. For BFS, the tipping point is anywhere between twitter and rMat27. Above that, i.e. on the larger graphs, the added computation power can be leveraged to overcome the synchronization overhead of the distributed scenario. Hence, on those graphs single-node Gemini becomes up to 1.8× (BFS, rMat28) slower than the distributed version. This relation is very dependent on the graph size. The smaller the graph, the faster single-node Gemini is in comparison. Analogously, the larger the graph, the larger the gap becomes, in favour of the distributed version.

A very similiar behaviour can be observed for Giraph, but the relation tips in favour of the distributed version much sooner. Giraph already uses the distributed cluster efficiently on the smaller graphs as well. On flickr, the execution times of single-node vs distributed Giraph are already within 9% of each other. On the larger graphs, the differences quickly increase, again in favour of the distributed scenario. Giraph's single-node version runs up to 5.9× (PR, wikipedia) slower than the distributed version. Keep in mind, that only a comparison between the three smallest graphs can be made here.

6) Behaviour of Galois: This section is dedicated to analyzing the speedup behaviour of Galois under two parameters. First we compare the calculation and execution times with and without hugepages on the maximum thread count. Second we change the thread count Galois is using, we compare the *calculation speedups* of Galois on the different graphs. Thus, we show the calculation time on any thread count normalized by the calculation time in the single-threaded environment. This is done along with a comparison between Galois using hugepages and Galois without hugepages. We begin with the time comparisons, followed by the speedups. These begin with SSSP, second by BFS and finally the comparison for both PR Push and Pull.

a) Time Comparison for Hugepages: When comparing the performance with and without hugepages, the results really speak for themselves they can be seen in Table IV. It is obvious that enabling hugepages greatly improves performance in almost every case. This means that on every algorithm and almost all graphs, calculation time and execution tame can be reduced significantly with just the use of hugepages. The

TABLE IV: Times for 96 Threads With and Without Hugepages

		Calc Time (s)		Exec Time (s)	
	Graph	w/o	w/	w/o	w/
SSSP	flickr	0.01	**0.01**	0.3	**0.2**
	orkut	0.10	**0.02**	0.8	**0.5**
	wikipedia	0.38	**0.11**	1.8	**1.1**
	twitter	2.47	**0.94**	10.8	**5.1**
	rMat27	4.50	**1.39**	16.0	**6.4**
	friendster	4.70	**1.78**	14.4	**7.5**
	rMat28	9.77	**3.34**	27.8	**13.1**
BFS	flickr	0.01	**0.00**	0.6	**0.3**
	orkut	0.05	**0.01**	0.9	**0.6**
	wikipedia	0.52	**0.12**	2.4	**1.4**
	twitter	7.52	**3.08**	14.2	**6.9**
	rMat27	5.62	**4.79**	14.6	**9.2**
	friendster	3.78	**1.67**	12.8	**7.2**
	rMat28	13.14	**10.25**	33.1	**19.1**
PR Push	flickr	-	-	0.6	**0.3**
	orkut	-	-	2.1	**1.2**
	wikipedia	-	-	13.1	**11.5**
	twitter	-	-	**74.9**	201.3
	rMat27	-	-	239.7	**178.1**
	friendster	-	-	54.2	**44.4**
	rMat28	-	-	689.3	**340.1**
PR Pull	flickr	0.01	**0.01**	0.3	**0.2**
	orkut	0.06	**0.02**	0.7	**0.6**
	wikipedia	0.17	**0.03**	1.7	**1.4**
	twitter	0.77	**0.11**	**8.7**	9.3
	rMat27	0.65	**0.13**	19.2	**8.1**
	friendster	1.01	**0.14**	20.4	**13.1**
	rMat28	1.15	**0.24**	46.0	**16.4**

(-) Result is less or equal to 1ms

only exception is twitter on both PageRank implementations, where the execution time is smaller without hugepages (74.9s vs 201.3s for Push and 8.7s vs 9.3s for Pull with and without hugepages respectively). In all other cases (that's 26 of 28 test cases), the hugepage-version is significantly faster. The execution time is approximately halved on SSSP and BFS. And on PageRank, the execution time is reduced to 35% to 87% of the time without hugepages.

TABLE V: Mean Speedups and Variances for SSSP With and Without HugePages

	μ		σ^2	
#Threads	w/o	w/	w/o	w/
1	1.0	1.0	0.0	0.0
2	1.6	1.6	0.2	**0.1**
4	2.5	2.5	1.1	**0.9**
8	**4.5**	3.4	5.3	**2.1**
16	**6.7**	5.8	13.0	**7.3**
32	**9.6**	7.0	38.3	**10.8**
48	10.3	**10.7**	41.4	**31.4**
96	10.2	**10.8**	38.1	**29.9**

TABLE VI: Maxium Reached Speedups With And Without Hugepages

	Max Speedup			
	SSSP		**BFS**	
Graph	w/o	w/	w/o	w/
flickr	**1.1**	1.0	1.1	**1.2**
orkut	1.6	**4.3**	1.6	**5.7**
wikipedia	10.3	**15.8**	4.9	**10.5**
twitter	**16.8**	13.0	2.5	**2.7**
rMat27	14.0	**15.8**	**4.2**	2.9
friendster	**15.9**	14.2	5.6	**8.3**
rMat28	**19.5**	14.9	**4.7**	2.8

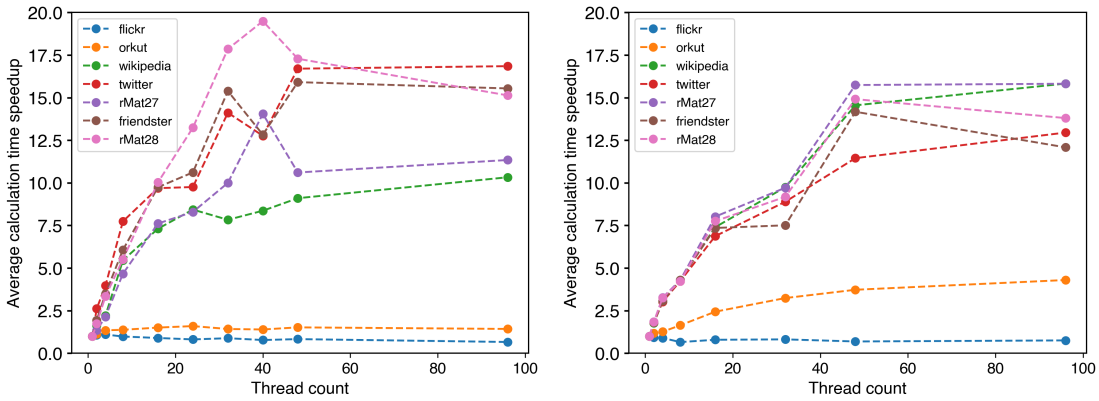
Keep in mind, that we have shown the results of 96 threads in this section. The following sections compare the behavoiur of Galois on different numbers of threads.

b) Single-Source Shortest-Paths: Starting with SSSP, an algorithm that greatly benefits from many available threads. The results can be seen in Figure 7. We first look at the speedups without hugepages, seen in Figure 7a. 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 between 10× (96 threads, wikipedia) and 19× (40 threads, rMat28). In some cases, increasing the thread count can even decrease the speedup again. 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. But on orkut, the speedup is always greater or equal to 1.

Upon activating the hugepages, we acquired the results seen in Figure 7b. Here, the overall results are similar to the findings without hugepages. Table V shows the mean speedups and variances over the different graphs with and without hugepages. We see that the mean speedup is either very similar or slightly reduced by the hugepages. But in all cases with hugepages, the variance is significantly smaller. This proves a slightly smaller but more reliable speedup with hugepages. Examples for this are on the one hand, orkut that could not reach a speedup above 1.6×. Now, with hugepages it is 4.3× faster with 96 threads compared to one thread. On the other hand, twitter reached a speedup of 17× without hugepages and only 13× with. While the hugepages did not neccessarily help improve the speedup value, they helped bring the different graphs together.

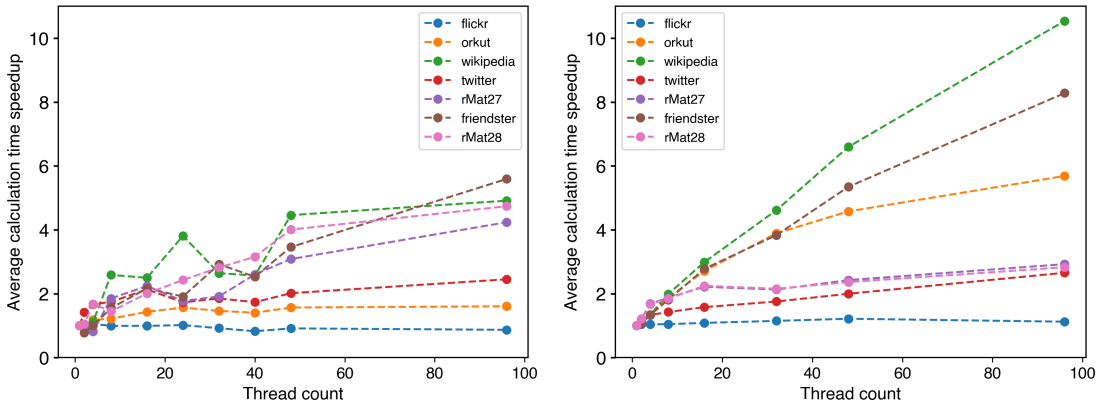
c) Breadth-First Search: For our speedup results on BFS, Figure 8 shows the calculation time speedup of Galois' BFS with and without hugepages. If we look at the results without hugepages first, we see most significantly, that the speedup never exceeds 6× even when using 96 threads (cf. Figure 8a). For both flickr and orkut, we have the same behaviour as on SSSP without hugepages. Speedup is close to 1 in all cases, with orkut reaching a maximum speedup of 1.6× at 24 threads. That said, the larger graphs are not benefitting from more threads as much as they did with SSSP. Twitter for example, reaches a speedup of 2× only with 48 or more threads. Meanwhile on SSSP, twitter reached a speedup of around 17× on those thread counts. For the other graphs, the speedup is between 4.2× (rMat27) and 5.5× (friendster) at 96 threads. So while speedups are possible, not even remotely to the same degree as on SSSP. This, in turn is not intuitive, one would assume these two algorithms to perform similarly. Both algorithms are iterative traversal algorithms with comparable computation and synchronization complexity. But the worse speedup behaviour extends even to the case with hugepages (cf. Figure 8b). While the results are generally better, still not to the same degree as on SSSP. With hugepages, BFS reaches a maximum speedup of 10.5× on wikipedia. The two graphs with largest speedup, namely wikipedia and friendster roughly follow a line with slope $1/8$. So with every 8 threads, the speedup is increased by about 1. Orkut follows the same line up to about 32 threads, slowly flattening off above that. On the other graphs, a speedup of 3× is hardly reached, even at 96 threads.

One might assume the two algorithms SSSP and BFS to behave very similarly, yet they produce very different results. With SSSP, we observed a decrease in the variance of the speedups, while the average speedup decreased or stayed the same. This provided a more reliable speedup. But hugepages did not necessarily increase the maximum possible speedup. For 4 of 7 graphs the larger speedup value is reached without hugepages (cf. Table VI). However exactly this is happening with BFS, we measured a larger speedup with hugepages on all real-world graphs (i.e. 5 of 7). But as can also be seen in



(a) without hugepages (b) with hugepages

Fig. 7: Calculation time speedups on SSSP



(a) without hugepages (b) with hugepages

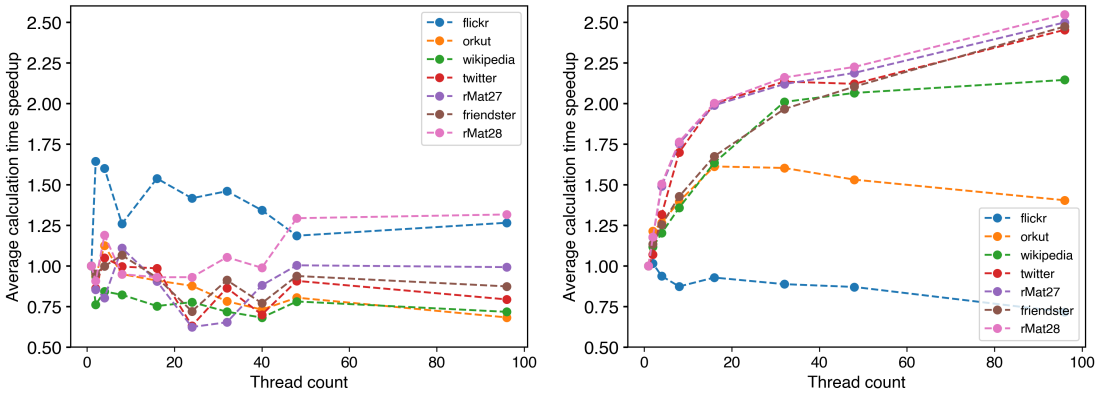
Fig. 8: Calculation time speedups on BFS

Figure 8b, the variance of the speedup increases, making the possible speedup very dependent on the graph.

d) PageRank Pull: We want to first take a look at the results for PageRank in Pull mode, seen in Figure 9. Without hugepages, computation time is hardly reduced on any graph other than flickr, where the reached maximum is 1.6× (cf. Figure 9a). 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 1.3× 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 1.1× and 1.05× at 4 threads, while being less or equal to 1 in all other cases. Speedup on the wikipedia graph is never grater than one. Friendster and rMat27 can be sped up to 1.07× or 1.1× respectively on 8 threads.

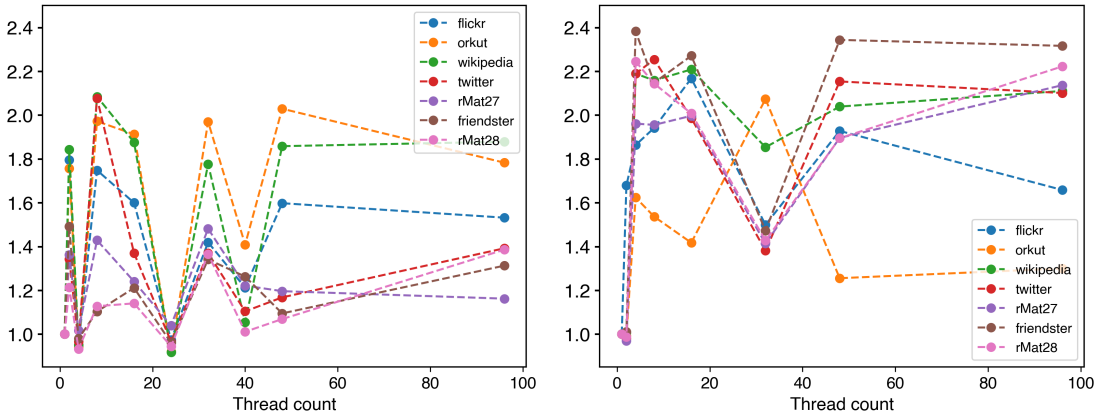
Most of this changes drastically with the activation of hugepages, our data can be seen in Figure 9b. Here actually all graphs except flickr reach a speedup greater than 1.5×. Remember, that 1.6× was the maximum possible speedup without hugepages. Furthermore, orkut is the only graph that never reaches a speedup of 2×. Twitter, rMat27, friendster and rMat28 all reach a speedup of around 2.5× at 96 threads. Only with hugepages enabled do we observe a considerable increase in performance and thus a benefit of utilizing many threads. Without hugepages though, having many threads available is in most cases slowing down the performance. Hence, enabling hugepages is very much recommended to be able to use a machine with many cores.

e) PageRank Push: The speedup results on the Push variant show odd behaviour in the Galois implementation, both with and without hugepages. Especially without hugepages though, there is a significant performance loss on 4, 24 and 40 threads that is far from the expected behaviour. This is most visible in Figure 10a, we validated the shown results with multiple samples each. The speedup for 24 threads is, by interpolating between 16 and 32 threads, expected to be anywhere between 1.3× and 1.9×. Actually however, the system does not reach a speedup of more than 1.04× 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. We intially assumed that this is due to the missing hugepages, Galois is recommended to be run with. This seems to be only partly true, since the results with hugepages are generally



(a) without hugepages (b) with hugepages

Fig. 9: Calculation time speedups on PR Pull



(a) without hugepages (b) with hugepages

Fig. 10: Calculation time speedups on PR Push

better, but still show some of this behaviour (cf. Figure 10b). There is only one significant anomaly at 32 threads, while the points where we observed the performance drop without hugepages look normal. At 32 threads, speedup for 5 of 7 graphs drops to a value between 1.4× and 1.5×. Yet those five graphs reach a speedup larger than 1.8× both at 16 and 48 threads (i.e. immediately before and after 32).

Nevertheless, if we disregard these anomalies, we see a significant performance increase with hugepages compared to without them. Actually, all graphs show an improvement in their maximum reached spedups. The largest improvements are from 1.5× to 2.1× for rMat27, 1.5× to 2.4× for friendster and 1.4× to 2.2× for rMat28, just by switching hugepages on. Interestingly, with hugepages all graphs reach a speedup at 4 threads that is very similar, if not equal to the speedup at 96 threads. The largest discrepancy between the two thread counts is rMat27, with a performance uplift from 1.96× to 2.14× at 4 and 96 threads.

Using hugepages is very effective and yet again increases performance by a considerable margin. Though, they do not completely remove the performance drops on some thread counts we observed without hugepages, the abnormal behaviour is at least less severe. But even with hugepages, a thread count larger than 4 did often not improve performance at all. This is due to the great synchonization overhead of a push based implementation. Generally, the push based application requires many more write operations, with each requiring synchronization. So, on push based applications a large thread pool is not helpful, rather should the framework itself be optimized. In this optimization, hugepages are a great factor for Galois.

VII. Discussion

//TODO: unfertig

There are two main cases that should be regarded in a comparison of the frameworks. One, the case of individual calculations on a graph, i.e. for each calculation, the graph has to be loaded. This is called the *research* case. For research, frameworks with small execution times and small overhead are preferred. Case two on the other hand is the case of a running system, that performs multiple calculations on a single graph, without the need of reloading graph data with every calculation. This is the *production* case. In production, frameworks with short calculation times should be preferred because the overhead time is only spent once on startup and amortizes quickly.

Orthogonal to this, the graph size and topology of the system is a deciding factor. Large graphs often require a distributed cluster, because the graph is simply too large to fit in the RAM of just one machine. We observed this for example with Giraph on multiple cases. Furthermore, some environments require a distributed framework setup regardless of any other factors.

For algorithms that perform a traversal through the graph, with a concentrated and relativly small amount of active vertices, our results for SSSP and BFS are most relevant. If the problem is however of a nature, where many vertices are active, thus many vertices, widely spread across the graph performe some kind of calculation, the results for PR are most applicable. Further, some considerations whether a push or pull-style implemenation is more efficient should be made. Generally, we expect pull-style implementations to be faster on a single-node, like for example PR with Galois. On a distributed cluster though, the synchronization time is not as severely impacting performance because communication over the network is needed, which is slower than local synchonization in any case.

We already know push implementations to be more efficient on problems with only few active vertices or where the affected push-areas are not overlapping. Because SSSP and BFS often have few active vertices with few overlapping vertices (i.e. little synchronization needed), synchronization is often not impacting performance as much, making push often the better implementation for those algorithms. PageRank on the other hand has many active vertices and is thus expected to perform much better with pull implementations, because less synchronization is needed in that case. Hence, a good guideline is pull-style on single-node systems with PR-like applications and push-style for SSSP and BFS, especially on distributed systems. Our data backs this up, we observed single-node PageRank to be fastest with Galois Pull, while the push based algorithms were fastest on the distributed systems. In general however, some tests determining which implementation is faster can be beneficial.

For graphs that are not too large for single node RAM, Galois as the fastest single-node framework is recommended. It provides excellent performance, being orders of magnitude faster than the competing frameworks. And when it is possible to use hugepages, these improve performance even further, sometimes up to a factor of 2 times faster compared to Galois without hugepages. Also, we have shown that most algorithms are able to utilize many threads in favour of much smaller calculation times. Graph-traversal algorithms like BFS or SSSP shown here are examples for such algorithms. Supplying a large thread pool for Galois is most likely improving performance on those algorithms. However, especially when using push-style algorithms, a large thread count is most likely not going to significantly improve performance over single threaded performance. The reason for this is the synchronization required for push applications. This is true not only for single-node Galois but any framework.

On production systems, we expect it to be possible to determine what configuration is best before putting the system in service. But not only the absolute runtime of each computation has to be taken into account. Production systems have to be very reliable because downtime often directly correlates to financial loss.

This is often a reason for choosing a distributed system. Many nodes together are less prone to failure of the entire system. Thus, there is an argument to be made for systems like Hadoop. It automatically handles node failures, a node failure does not immediately result in loss of data or a faulty computation result. This is especially important on very large clusters, that can not easily be monitored by humans. A user defining an application does not need to know the topology of the system that runs the application later on. Hadoop transparently splits the input data and distributes work among the worker nodes. Furthermore, Giraph using hadoop shows a bias towards larger graphs, we observed an improvement in performance over single-node Giraph, when increasing the size of the input graph. This shows the hadoop infrastructure to be working well and is exactly what the distributed system should be used for anyways. Only a system like hadoop can reasonably be used on such large clusters. Any of the other framewors will just result in large administration overhead due to node failures or other problems of distributed systems in general. Furthermore, the calculation times of Giraph proved to be very fast in comparison to the other distributed frameworks, with Giraph often being one of the fastest framework in computation time. Very short calculation times along with automatic handling of the computation cluster make Giraph using Hadoop a good choice for production systems. Pregel is a similarl, fault-tolerant and scalable system but it is contrary to Giraph closed-source.

But not only distributed production systems are possible, of course. Single-node systems are possible, as long as the graph data fits in local memory. Further, horizontal scaling can for example be used to provide fault-tolerance of the entire system. While Giraph has very short computation times even on a single node, we would advise against it. Giraph requires a lot more memory than any of the other frameworks and ran out of memory on multiple occasions. Thus it should only really be considered on distributed setups.

There are multiple possibilities to be accounted for **//TODO: unfertig**

If the graph data becomes too large for one node or the computation cluster is distributed anyways, there are multiple possibilities.

On research systems however, single computations are performed and node failures are not as common, because the system is most likely not in-use constantly. Hence, Hadoop handling all this in the background is a nice-to-have but most likely, a plain faster framework would be preferred. Furthermore, administration is most likely needed frequently because system reconfiguration is happening often? Because the system is most likely reconfigured frequently. **//TODO: unfertig** Here, the fastest framework really depends on the use case. Graph traversal algorithms prefer

Here, Galois is the fastest distributed framework for SSSP and BFS, while Gemini outperforms Galois on PR.

VIII. Conclusion

//TODO: Section

ACKNOWLEDGMENTS

We would like to thank our supervisor Heiko Geppert for his continued guidance and support.

APPENDIX

We provide a set of installation guides, describing setup and usage of each of the presented frameworks. In case of 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 is provided as well. Furthermore, the conversion tool we described to convert between the different graph input formats is available alongside the guides in our repository[6].

REFERENCES

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

[2] 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

[3] 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

[4] 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

[5] M. Newman, *Networks: An Introduction*. OUP Oxford, 2010. [Online]. Available: https://books.google.de/books?id=-DgTDAAAQBAJ

[6] 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

[7] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1, pp. 107 – 117, 1998, proceedings of the Seventh International World Wide Web Conference. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016975529800110X

[8] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 12–25. [Online]. Available: https://doi.org/10.1145/1993498.1993501

[9] 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

[10] S. Forster and D. Nanongkai, "A faster distributed single-source shortest paths algorithm," *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, Oct 2018. [Online]. Available: http://dx.doi.org/10.1109/FOCS.2018.00071

[11] 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.

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

[12] 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

[13] 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.

[14] E. Abrossimov, M. Rozier, and M. Shapiro, "Generic virtual memory management for operating system kernels," in *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, ser. SOSP '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 123–136. [Online]. Available: https://doi.org/10.1145/74850.74863

[15] A. Panwar, A. Prasad, and K. Gopinath, "Making huge pages actually useful," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 679–692. [Online]. Available: https://doi.org/10.1145/3173162.3173203

[16] M. Gorman and P. Healy, "Performance characteristics of explicit superpage support," in *Computer Architecture*, A. L. Varbanescu, A. Molnos, and R. van Nieuwpoort, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 293–310.

[17] 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

[18] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what COST?" in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, May 2015. [Online]. Available: https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

[19] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 211–222. [Online]. Available: https://doi.org/10.1145/1250734.1250759

[20] 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

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

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

[23] 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/