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*—In this paper we will analyze and compare the non-uniform memory access (NUMA) aware graph frameworks Galois, Ligra, Polymer and Gemini in their parformance.**

Giraph

All the frameworks will be tested in shared memory. Galois, Gemini and Giraph are tested on a distributed cluster as well. We will give some insight on the calculation performance and associated time overhead of each framework. For this we compare the frameworks in their performance with Single-source shortest-paths, Breadth-first search and PageRank algorithms.

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

I. Introduction

In recent years graph sizes have increased significantly, thus performance and memory efficiency of the graph analysis applications is now more important than ever.

We will 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[7] is included in the testing. Giraph itself has often been compared to other state of the art systems like Pregel or GraphX.

To this end several non-uniform memory access (NUMA) aware systems were proposed like Polymer [3], Galois [1] or Ligra [5].

This paper makes the following contributions:

- Comparison of some of the most widely used graph based calculation frameworks
-
-
-

II. Overview of the framworks

A. Galois and Gluon

Galois[1] is a general purpose library designed for parallel programming. The Galois system supports fine grain tasks, allows for autonomous, speculative execution of these tasks and grants control over the task scheduling policies to the application. It also simplifies the implementation of parallel applications by providing an implicitly parallel unordered-set iterators.

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

Gluon[2] is a framework written by the Galois team as a middleware to write graph analysis applications for distributed systems. It reduces the communication overhead needed in distributed environments by exploiting structural and temporal invariants.

The code of Gluon is embedded in Galois. It is possible to integrate Gluon in other frameworks too, which the Galois team showed in their paper[2].

B. Gemini

Gemini is a framework for parallel graph processing[6]. In comparison to the other NUMA-aware frameworks we discuss here, Gemini does not support shared memory parallel processing. Computation is entirely message based using MPI. Gemini is lightweight and only provides basic functionality. A basic API hides data and computation distribution details from the user writing applications. There are five algorithms (which ones?) already implemented.

C. Giraph

Apache Giraph is an iterative graph processing framework, built on top of Apache Hadoop[7]. Hadoop as a large MapReduce infrastructure provides a reliable (fault tolerant) basis for large scale graph processing. Because of the underlying Hadoop MapReduce infrastructure, expanding single-node processing to a multi-node cluster is almost seamless.

Giraph is based on computation units that communicate using messages and are synchonized with barriers.

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

Computation is vertex oriented and iterative. For each iteration step called superstep, the Compute method implementing the algorithm is invoked on each active vertex, with every vertex being active in the beginning. This method receives

messages sent in the previous superstep as well as its vertex value and the values of outgoing edges. With this data the values are modified and messages to other vertices are sent. Communication between vertices is only performed via messages, so a vertex has no access to values of other vertices or edges other than its own outgoing ones.

Supersteps are synchronized with barriers, meaning that all messages only get delivered in the following superstep and computation for the next superstep can only begin after every vertex has finished computing the current superstep. Edge and vertex values are retained across supersteps.

Any vertex can stop computing (i.e. setting its state to inactive) at any time but incoming messages will make the vertex active again. To end computation, a vote-to-halt method is applied. Each vertex outputs some local information (e.g. the final vertex value) as result.

Giraph is next to Galois probably the most actively developed framework in our comparison.

D. Ligra

Ligra[5] is a lightweight parallel graph processing framework for shared memory machines. It offers a programming inteface that allows expressing graph traversal algorithms in a simple way.

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

E. Polymer

Polymer is very similar to Ligra, in fact Polymer inherits the programming interfaces `EdgeMap` and `VertexMap` from Ligra as its main interface.[3]

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

III. And

A. An overview of some graph formats

A rather big portion of our time was invested in figuring out which graph framework requires which graph formats. We thus decided to give an overview over all the formats we encountered, with explanation on how they represent the graph.

Additionally, to make life in the future a little bit easier, we wrote multiple tools to convert graphs acquired from Snap or Konect to the required formats. Additional information on this is available in the section Supplementary Data at the end.

1) AdjacencyList: The AdjacencyList and WeightedAdjacencyList formats[4] are used by Ligra and Polymer. They represent the directed edges of a graph as a number of offsets that point to a set of target nodes in the file. First the file contains the number of vertices n and edges m, followed by an offset for each vertex. This offset specifies at what point in the following list of numbers the information for a node begins. Lastly the file format contains a list of target nodes. The numbers are all separated by newlines.

$$\begin{array}{c} n \\ m \\ o_1 \\ o_2 \\ \vdots \\ o_n \\ t_1 \\ t_2 \\ \vdots \\ t_m \end{array}$$

The offsets $o_i = k$ and $o_{i+1} = k + j$ mean that vertex i has j outgoing edges, these edges are

$$(i, t_k), (i, t_{k+1}), \ldots, (i, t_{k+j-1})$$

For the WeightedAdjacencyList format, the weights are just appended to the end of the file in the same order as the edges.

2) EdgeList: The EdgeList format is probably the easiest to understand and is one of the most commonly used in the online graph repositories. The directed eges $(s_1, t_1), (s_2, t_2), \ldots$ or $(s_1, t_1, w_1), (s_2, t_2, w_2), \ldots$ are represented in the following way.

$$\begin{array}{ccc} s_1 & t_1 & w_1 \\ s_2 & t_2 & w_m \\ & \vdots & \\ s_m & t_m & w_m \end{array}$$

The weights are optional, everything is ASCI encoded and the inline delimiter is a variable amount of any whitespace.

3) Binary EdgeList: The binary EdgeList format is used by Gemini. Finding information on this format required reverse engineering of the Gemini code.

We found that Gemini requires the following input format

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

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

4) Giraph's I/O formats: Giraph is capable of parsing many different input and output formats. All of those are explained in Giraph's JavaDoc. 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 the existing `JsonLongDoubleFloatDoubleVertexInputFormat`[**?**]. Our graph conversion tool outputs this format.

In this format, the vertex IDs are specified as `long` with `double` vertex values, `float` out-edge weights and the messages are of type `double` (all are Java data types). All this is in a JSON-like format, so each line in the graph file looks as follows

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

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

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

IV. Testing methods

A. Hardware and Software

For testing all systems we used 5 machines with 96 CPU cores each (48 physical) and 256 GB of RAM. 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[1].

All benchmarks were initiated by our benchmark script that is available in our repository. Galois, Gemini and Giraph were benchmarked on both the 5-node cluster and a singe machine. Since Galois supports this parameter, we ran multiple tests comparing Galois' performance with different thread counts on a single machine.

The complete benchmark log files and extracted raw results will be available in our repository as well.

B. Measurements

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

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

- For Galois, we resulted to extracting console log time stamps. Galois outputs `Reading graph complete.`. Calculation time is the time from this output to the end of execution.
  We know that this is not the most realiable way for measuring the calculation times. Not only due to unavoidable buffering in the console output we expect the measured time to be larger than the actual. First, it is not clear that all initialization is in fact complete after reading the graph. Second, we include time that is used for cleanup after calculation in the measurement.
- 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) which we summed up to extract the computation time.

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

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

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

TABLE I: Size comparison of the used graphs

Algorithm	Galois	Gemini	Giraph	Ligra	Polymer
PageRank	ΔPush, ΔPull				
SSSP	?, Push, Pull				
BFS	Yes		*		

TABLE II: Tested algorithms of the frameworks
* Algorithm not natively supported

TODO

C. Graphs

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 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[2] associated with the Koblenz Network Collection (KONECT)[8]. 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.

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. For frameworks that support multiple implementations (e.g. PageRank in push and pull modes), we included the alternatives in our testing. Table II shows the tested algorithms in detail.

[1]The setup guides are available at https://github.com/SerenGTI/Forschungsprojekt/tree/master/documentation

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

E. Comparison of setup

V. Results

A. The frameworks during setup and benchmark

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

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

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

B. Single-source Shortest-paths

1) Single-node: Figure 1 shows the average calculation times (time without initialization overhead), execution time and the normalized overhead for SSSP on the different frameworks. In these figures, Galois with 96 threads is shown. We show the impact of Galois' thread count in subsection V-E.

2) Distributed: For the distributed scenario, Figure 2 shows the benchmark results as calculation and execution times.

Results are especially interesting for Giraph since it seems to not cope well with synthetic graphs. 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 (Gemini on flickr) up to Galois Pull needing 18× longer on wikipedia. On both synthetic graphs however, Giraph is the slowest to compute. Giraph requires 12× or even 15× the computation time of Gemini on rMat27 or rMat28 respectively.

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

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

C. Breadth-first search

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

D. PageRank

E. Galois speedup

For Galois we

F. Execution times and overhead

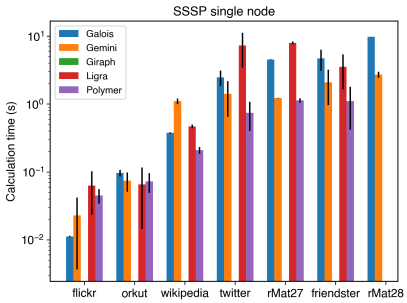
hier erhoffe ich mir einen Vergleich der Ladezeiten und erwarte, dass Systeme wie Giraph, die erstmal auf irgendwas warten schlecht abschneiden. Aber vielleicht ist auch die setup time bei gleichen frameworks zwischen verteilt und shared memory ganz interessant zu vergleichen.

VI. Discussion

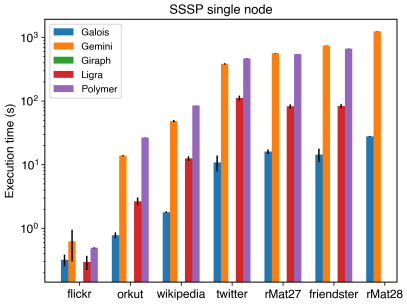
VII. Conclusion

The conclusion goes here.

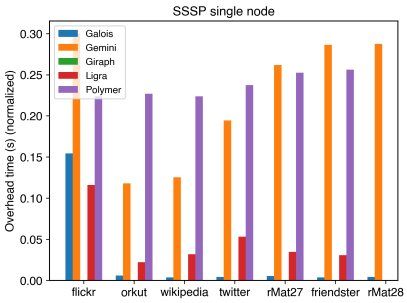
[3] https://github.com/jasc7636/GeminiGraph



(a) Calculation times for SSSP on a single node

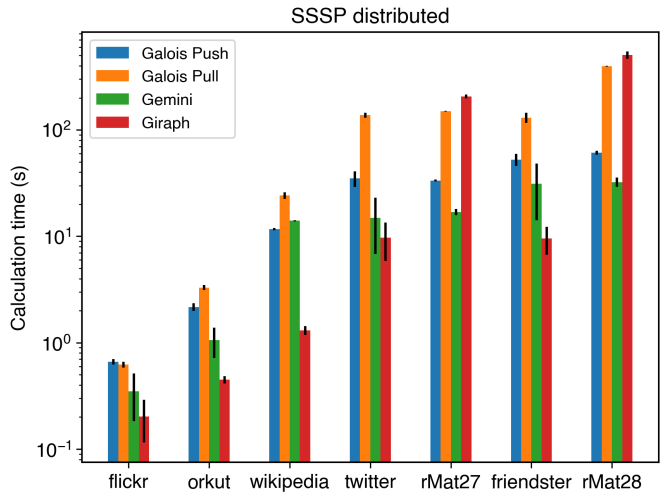


(b) Execution times for SSSP on a single node

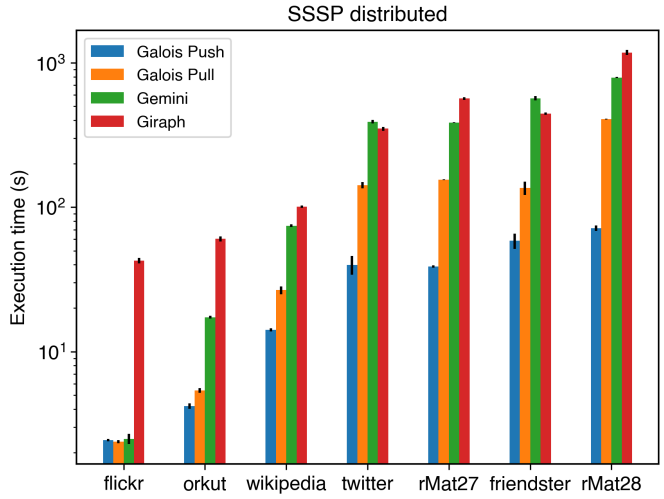


(c) Overhead time normalized by the graph size in million edges

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



(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.

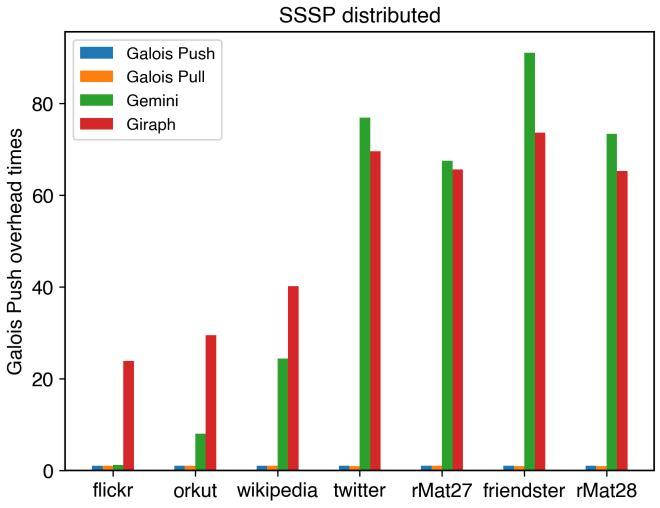
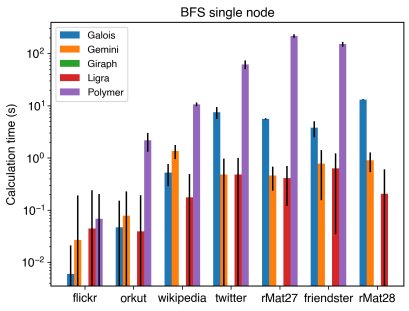
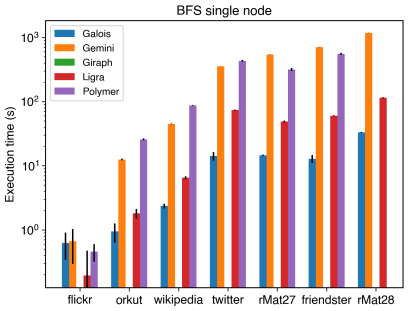


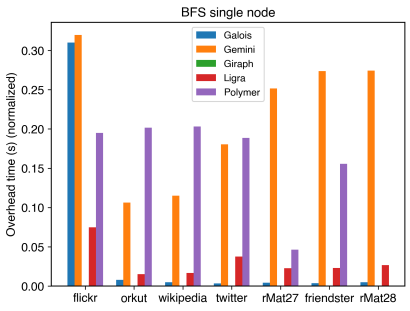
Fig. 3: Overhead times of each framework normalized by the overhead time of Galois Push



(a) Calculation times for BFS on a single node

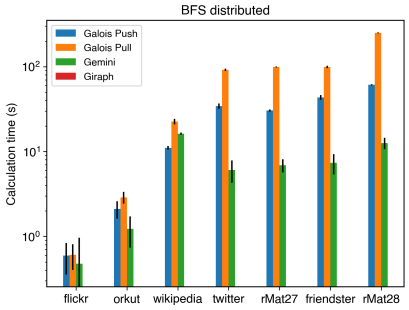


(b) Execution times for BFS on a single node

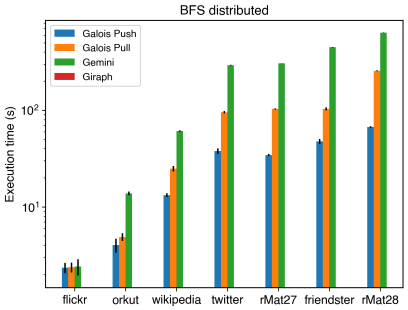


(c) Overhead time normalized by the graph size in million edges

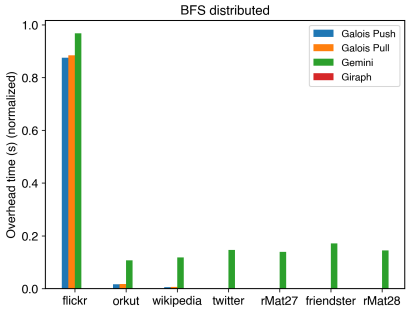
Fig. 4: Average times on a single computation node, black bars represent one standard deviation in our testing



(a) Calculation times for distributed BFS

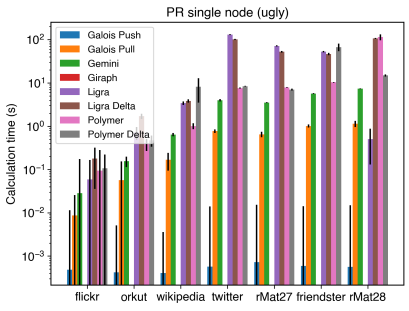


(b) Execution times for distributed BFS

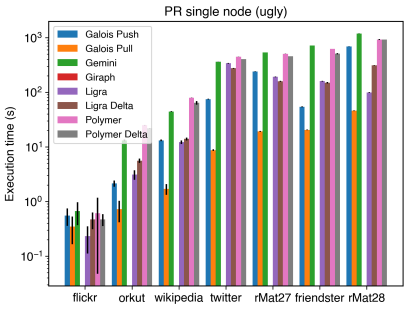


(c) Overhead time normalized by the graph size in million edges

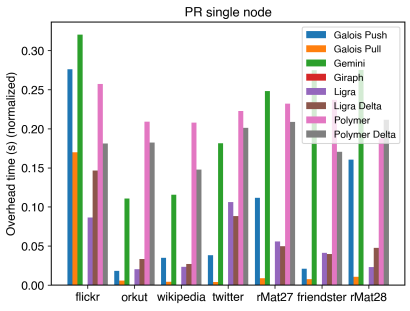
Fig. 5: Average times on the distributed cluster, black bars represent one standard deviation in our testing



(a) Calculation times for PR on a single node

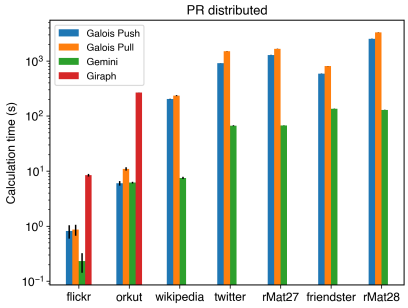


(b) Execution times for PR on a single node

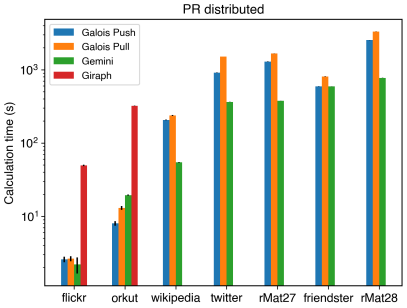


(c) Overhead time normalized by the graph size in million edges

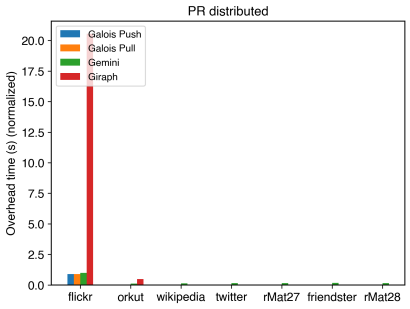
Fig. 6: Average times on a single computation node, black bars represent one standard deviation in our testing



(a) Calculation times for distributed PR

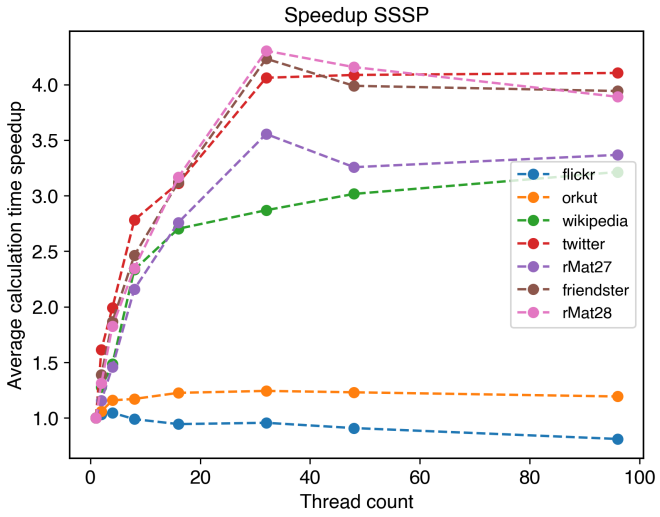


(b) Execution times for distributed PR

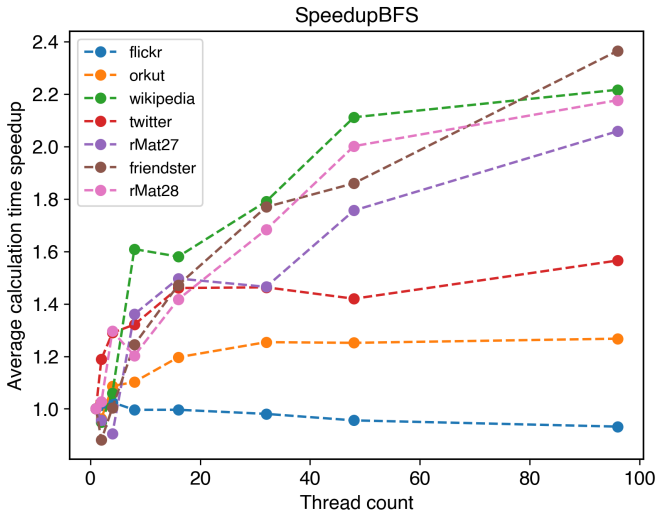


(c) Overhead time normalized by the graph size in million edges

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

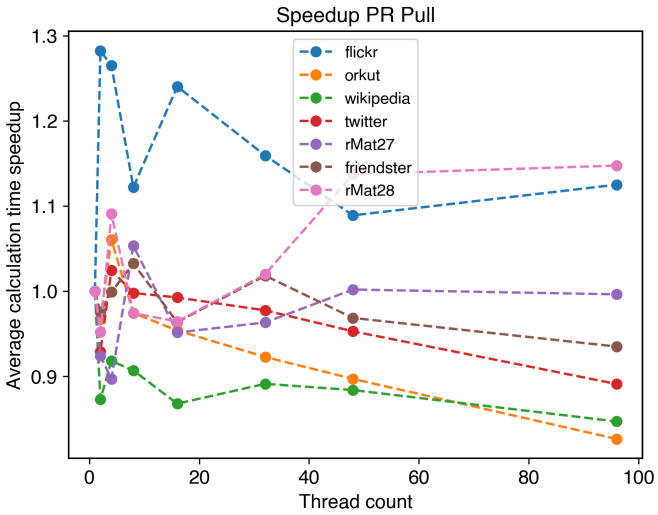


(a) Single-source Shortest-paths

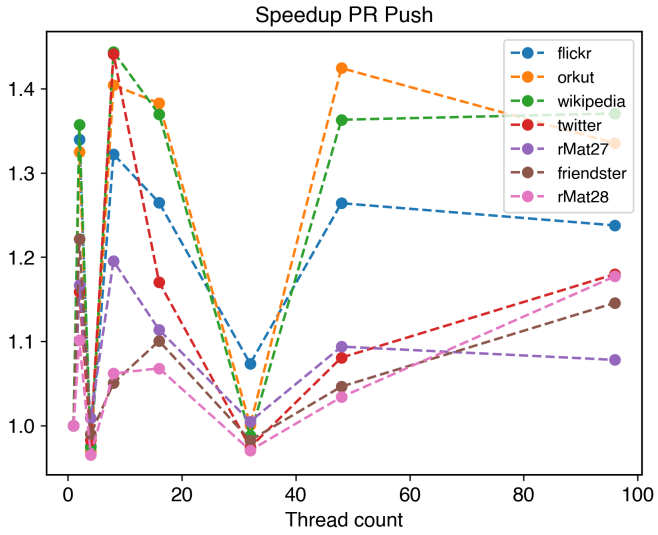


(b) Breadth-first search

Fig. 8: Calculation time speedup with increasing thread count for Galois Single-source Shortest-paths and Breadth-first search algorithms.



(a) PageRank Pull



(b) PageRank Push

Fig. 9: Calculation time speedup with increasing thread count for Galois PageRank Push and Pull algorithms.

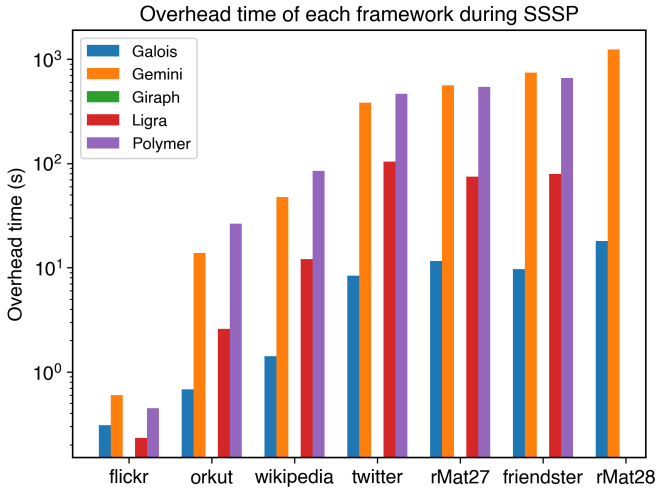


Fig. 10: Overhead SSSP single node

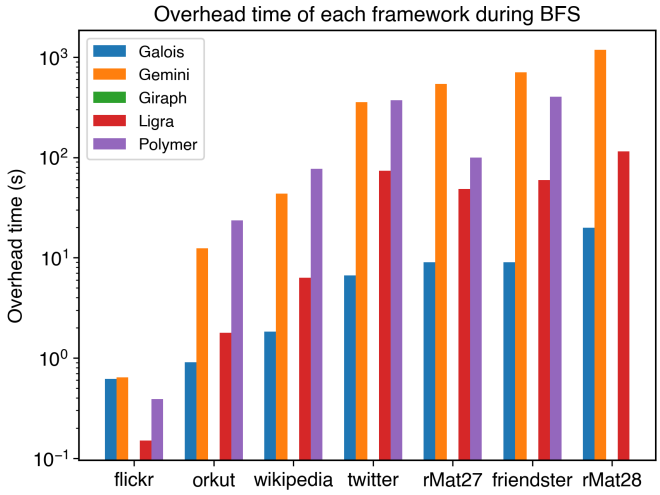


Fig. 11: Overhead BFS single node

ACKNOWLEDGMENT

We are using the graph frameworks Galois [1], Ligra [5], Polymer [3], Gemini [6] as well as Apache Giraph [7].

Also we use Gluon [2] for the distributed Galois setups.

Gemini [6]

SUPPLEMENTARY DATA

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] 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

[2] 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

[3] 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

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

[5] 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

[6] 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

[7] A. S. Foundation. (2020, Jun.) Apache Giraph. [Online]. Available: https://giraph.apache.org

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