A Comparison of Parallel Graph Processing Benchmarks

Samuel D. Pollard and Boyana Norris

University of Oregon
Eugene OR 97403, USA
spollard@cs.uoregon.edu
norris@cs.uoregon.edu

Abstract. The increasing popularity of large network analysis problems has led to the emergence of many parallel and distributed graph processing systems—one survey in 2014 identified over 80. Since then, the landscape has evolved; some packages have become inactive while more are being developed. Determining the best approach for a given problem is infeasible for most developers. To enable easy, rigorous, and repeatable comparison of the capabilities of such systems, we present an approach and associated software for analyzing the performance and scalability of parallel, open-source graph libraries. We demonstrate our approach on five graph frameworks: GraphMat, the Graph500, the Graph Algorithm Platform, GraphBIG, and PowerGraph using synthetic and real-world datasets. We examine previously overlooked aspects of parallel graph processing performance such as phases of execution and energy usage for three algorithms: breadth first search, single source shortest paths, and PageRank and compare our results to Graphalytics.

1 Introduction

Our research is motivated by the current state of parallel graph processing. The most comprehensive survey, released in 2014, identified and taxonomized over 80 different parallel graph processing systems without even including domain specific languages [4].

An overarching issue among these systems is the lack of comprehensive comparisons. One possible reason is the considerable effort involved in getting each system to run: satisfying dependencies and ensuring data are correctly formatted are generally nontrivial tasks. Additionally, some systems are developed with large, multi-node clusters in mind while others only work with shared memory, single node computers. An example of the former is GraphX [17], which incurs some overhead while gaining fault tolerance and an example of the latter is Ligra [15], a framework requiring a shared-memory architecture.

At the human level, there are optimizations for each system which may not be apparent. For example, the Parallel Boost Graph Library (PBGL) [6] provides generic implementations of their algorithms and the programmer must provide the template specializations. The optimal data structures may differ across algorithms and graphs and must be determined by the programmer.

To mitigate the inherent bias in hand-generated implementations, all the experiments performed use the author-provided implementations; we assume the developers of each system will provide the most performant implementation. While this limits the scope of the experiments it mitigates the bias inherent in our programming skills.

Our contributions can be summarized as follows.

- Automation of the installation, configuration, and dataset conversion to ensure the experiments execute in the same manner. Essentially, we provide a "level playing field" for graph algorithms.
- Comparison of algorithm runtime and scalability using a variety of data sets
- Comparison of our framework with the Graphalytics [3] project
- Inspection of the source code of the surveyed parallel graph processing systems to ensure the same phases of execution are measured across differing execution and programming paradigms
- Provision of a experimental framework distinguishing between the data structure generation and algorithm execution phases of each system
- Analysis of energy consumption during the algorithm execution phase

2 Related Work

A related performance analysis project is Graphalytics [3] which also attempts to automate the setup and execution of packages for performance analysis. However, in order for Graphalytics to analyze a graph processing system—Graphalytics calls these platforms—a programmer must implement Java classes as wrappers to call the particular implementations, satisfy dependencies, and set the correct shell commands to execute. These all require some knowledge of the inner workings of each system in addition to familiarity with the Graphalytics API.

The complexity of Graphalytics obfuscates the true behavior of the program. By default, Graphalytics generates an HTML report listing the runtimes for each dataset and each algorithm in seconds. For example, one run of a single source shortest paths algorithm took 5.6 seconds. However, perusing the log files reveals a more complete picture: of these 5.96 seconds, 0.21 was spent computing the shortest paths while the remaining time consisted of building the necessary data structures. Furthermore, data such as the number of iterations run for PageRank are not easily available using Graphalytics.

With a plugin to Graphalytics called Granula [13] one can explicitly specify a performance model to analyze specific execution behavior such as the amount of communication or runtime of particular kernels of execution. This requires in-depth knowledge of the source code and execution model. Furthermore, creating such a model requires a high level of expertise with the given system and with Granula[1].

As with Graphalytics, the initial development effort is high but Granula paired with graphalytics allows automatic execution and compilation of performance results. Likewise, our approach provides automatic execution and performance analysis without requiring a performance and execution model for each system.

Other related work includes that of [14], which analyzes performance of several systems on datasets on the order of 30 billion edges and [10] which uses six real-world datasets and focuses on the vertex-centric programming model.

Our approach differs from the previously mentioned appproaches because it focuses on the depth of analysis rather than the breadth via diversity of datasets or graph processing systems. We analyze power and energy consumption in addition to details such as the number of iterations or the time to construct the data structures. Additionally, our approach requires little knowledge of the inner workings of each system; the data are collected using either sampling of model specific registers (for power) or parsing log files (for timing).

3 Experimental Setup

We refer to a graphs's *scale* when describing the size of the graph. Specifically, a graph with scale S has 2^S vertices. The datasets are described further in Sect. 3.4. We measure strong scaling and speedup by varying the number of threads from one to the total number of threads on the reasearch machine, 72.

Each experiment uses 32 roots per graph. As with the Graph500, each root is selected to have a degree greater than 1. In the case of PageRank, we simply run PageRank 32 times.

When possible, we measure the data set construction time as the time to translate from the unstructured file data in RAM to the graph representation on which the algorithm can be performed. This is not possible for PowerGraph and GraphBIG because they read in the input file and build a graph simultaneously. The Graph500 builds the graph once and performs the BFS on each root in turn, speeding up experimentation but only giving a single data point for the data set construction.

We plot many of the results as box plots with an implied 32 datapoints per box.

3.1 Machine Specifications

Table 1 shows the specifications of the research computer.

[1] An example of Granula can be seen at `https://github.com/tudelft-atlarge/graphalytics-platforms-graphx/tree/master/granula-model-graphx`

3.2 Graph Processing Systems

This report explores four shared memory parallel graph processing platforms. The first three are so-called “reference implementations” while the remaining two are included because of their performance and popularity. Our target is shared memory CPU processing. Other popular libraries such as the Parallel Boost Graph Library [6] are not considered here because the authors do not provide reference implementations. The systems are:

1. The Graph500[2] [11]
2. The Graph Algorithm Platform (GAP) Benchmark Suite [1]
3. GraphBIG [12]
4. GraphMat [16]
5. PowerGraph [5]

3.3 Algorithms

We consider three algorithms: Breadth First Search (BFS), Single Source Shortest Paths (SSSP), and PageRank, though not all algorithms are implemented on all systems.

We select BFS because the canonical performance leaderboard for parallel graph processing is the Graph500 [11]. The advantage of the Graph500 is it provides standardized measurement specifications and dataset generation. The primary drawback with the Graph500 is it measures a single algorithm.

Our work aims to add similar rigor to other graph algorithms by borrowing heavily from the Graph500 specification. The Graph500 Benchmark 1 (“Search”) is concerned with two kernels: the creation of a graph data structure from an edge list stored in RAM and the actual BFS[3]. We run the BFS using 32 random roots with the exception of PowerGraph which doesn’t provide an reference implementation of BFS in its toolkits.

We select SSSP because of the straightforward extension from BFS; we need not modify the graph nor the root vertices from BFS.

PageRank is selected because of its popularity; most libraries provide reference implementations. One challenge with using PageRank is the stopping criterion; this is addressed in Sect. 4. Verification of the PageRanks is beyond the scope of this paper though this may explain some of the large performance discrepancies.

This approach is not specific to a particular algorithm; measuring the execution time, data structure construction time, and power consumption can be easily extended to other algorithms.

[2] We used the most recent version from `https://github.com/graph500/graph500`, most similar to release 2.1.4.

[3] For a complete specification, see `http://graph500.org/specifications`

Table 1. The operating system is GNU/Linux version 4.4.0-22. The disparity between the CPU's advertised clock speed and the "CPU Clock" row is a result of the Turbo Boost technology which can increase the clock speed to a limit. We use the manufacturer's published maximum clock speeds which can be found at `http://ark.intel.com`.

CPU Model	Intel Xeon(R) E5-2699 v3 @ 2.30GHz
CPU Sockets	2
CPU Cores	72
CPU Clock	3600MHz
RAM Size	256GB
RAM Freq	1866MHz
GPU Model	GM204 [GeForce GTX 980]

3.4 Datasets

We use the Graph500 synthetic graph generator which creates a Kronecker graph [9] with initial parameters of $A = 0.57, B = 0.19, C = 0.19$, and $D = 1 - (A + B + C) = 0.05$ and set the average degree of a vertex as 16. Hence, a Kronecker graph with scale S has 2^S vertices and approximately 16×2^S edges.

The Graphalytics results in Table 2 were performed on the Dota-League dataset. This dataset contains 61,670 vertices and 50,870,313 edges. This dataset comes from the Game Trace Archive [7] and is modified for Graphalytics[4]. This dataset is useful because it is both weighted and more dense than the usual real-world dataset with an average out-degree of 824.

4 Performance Analysis

4.1 Runtime

In Table 2 we show the results from running Graphalytics on a single dataset. For an explanation of each algorithm used, see [8]. Table 2 shows only a broad overview of the two systems: GraphBIG and PowerGraph. In general, we see GraphBIG is more performant. However, results such as these are preliminary and do not show the complete picture.

Table **??**, in contrast, gives an overview of the performance results of running graphalytics and our approach using the same dataset [TODO: Get the Dota-League Dataset working with easy-parallel-graph and replace Table **??**]. Table **??** shows the average time across 32 roots, whereas Graphalytics by default shows results for a single root.

rework this. It would be nice to compare Graphalytics and our approach on the real-world dataset.

Figures 1 and 2 show more detail, with box plots for the runtime distributions. However, even these do not give a complete picture of performance. Both of these show GraphMat to not be highly performant. One explanation is GraphMat's underlying computation model (sparse matrix operations) paired the increased overhead of GraphMat's doubly-compressed sparse row (DSCR)

[4] This dataset is available at `https://atlarge.ewi.tudelft.nl/graphalytics/`.

Table 2. Performance results are in milliseconds. Community detection is performed using label propagation. At the time of this writing, Graphalytics only supports SSSP for GraphMat.

	GraphMat	GraphBIG	PowerGraph
Community Detection	N/A	212.5	1,167
PageRank	N/A	293.5	983
Local Clustering Coefficient	N/A	316.5	1,011
Weakly Connected Components	N/A	89	767.5
Single-Source Shortest Paths	11,298.5	6,378.5	34,654.5

graph representation is more conducive to larger-scale graphs. There is less discrepancy between the runtimes of SSSP (between 0.1 and 1.7 seconds) compared to BFS (0.01 and 1.7 seconds) but GAP is the clear winner in both cases. The data structure construction times for GAP and GraphMat are consistent; in both cases the platforms create the same data structure for both algorithms.

These results are consistent with [16] which lists GraphMat as more performant than PowerGraph in SSSP.

Make the text bigger for these

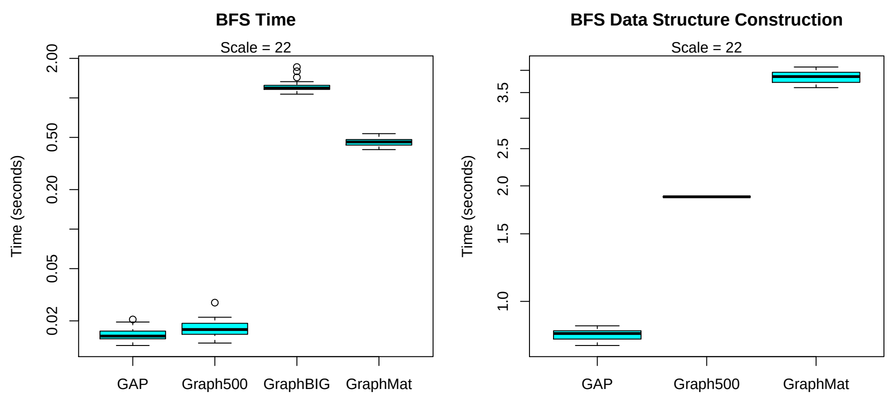


Fig. 1. The y-axes are logarithmic. The left box plot shows the time to compute BFS on 32 random roots while the right plot shows the times to construct the graph for each system. The Graph500 only constructs its graph once. GraphBIG reads in the file and generates the data structure simultaneously so is omitted.

The behavior of PageRank is slightly different. As with SSSP and BFS, the GAP Benchmark Suite is the fastest but it also requires the fewest iterations.

We attempt to use similar stopping criteria for each system, but GraphMat executes until no vertices change rank; effectively its stopping criterion requires the ∞-norm be less than machine epsilon. This could account for the increased

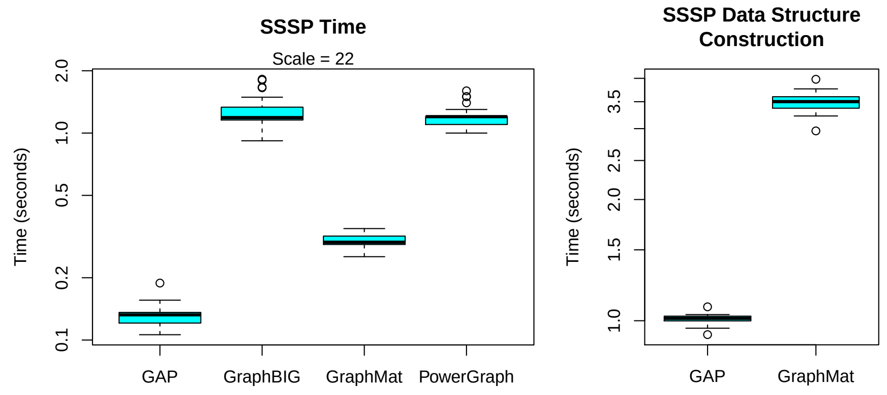


Fig. 2. The y-axes is logarithmic. The left box plot shows the time to compute the SSSP starting at the same 32 roots as Fig. 1. Both PowerGraph and GraphBIG construct their data structures at the same time as they read in the file.

number of iterations. We adjusted the other systems to use $\sum_{k=1}^{n} |p_k^{(i)} - p_k^{(i-1)}| < \epsilon$ as the stopping criteria, where i is the iteration and n is the number of vertices. We use $\epsilon = 6 \times 10^{-8}$ because this value is approximately machine epsilon for a single precision floating point number. This is done with the goal of making the stoppping criteria as similar as possible.

Furthermore, the logarithmic scale in Figs. 4 and 1 compresses the apparent variance in runtime. Each platform in Fig. 4 has a relative standard deviation between 1/4 and 1/2 that of the same system executing SSSP.

Check this again—that's a huge discrepancy!

The challenge in comparing iteration counts for PageRank underscores an important challenge for any comparison of graph processing systems. The assumptions under which the various platforms operate can have a dramatic effect on the program. For example, the GAP Benchmark Suite stores vertex weights as 32-bit integers. However, other systems store them as floating point numbers. This may affect performance in addition to runtime behavior in cases where weights like 0.2 are cast to 0. Similarly, how a graph is represented in the system (e.g. weighed or directed) may have performance and algorithmic implications but is not always readily apparent.

Figure 5 shows the scalability and speedup for BFS. We define the strong scaling as $T_1/(nT_n)$ where T_1 is the serial time and n is the number of threads. A linear scalability is when $T_n = T_1/n$ and is the horizontal line on the bottom plot in Fig. 5. These plots show in general poor scaling throughout, though GraphBIG has the best scaling. A potential explanation is GraphBIG is the slowest system so there is more room for improvement. Additionally, a scale of 20 generates a small graph by today's standards and thus the focus is on scalability for larger graphs. Another limitation may be the ability for OpenMP to efficiently handle such a large number of threads per machine.

Rerun GraphMat on scale 22 with 2 threads. Surprising that it slows down

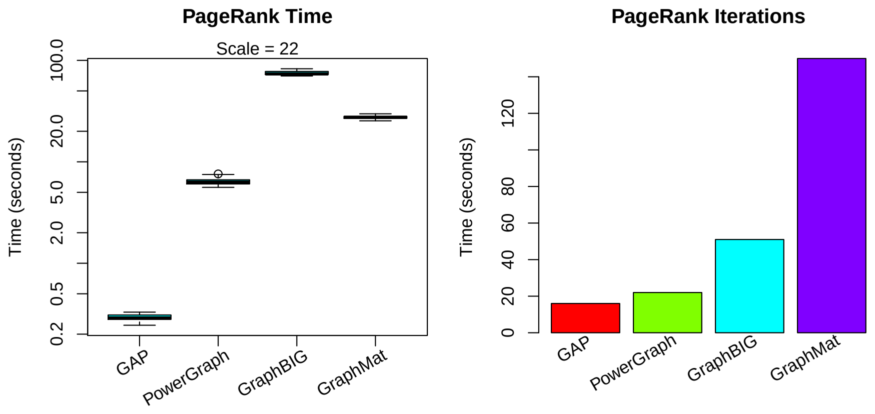


Fig. 3. The y-axis is logarithmic only for the left figure. GraphMat continues to run until none of the vertices' ranks change. For the others, we use the stopping condition that the sum of the changes in the weights is no more than 6×10^{-8}, or approximately machine epsilon for single precision floating point numbers.

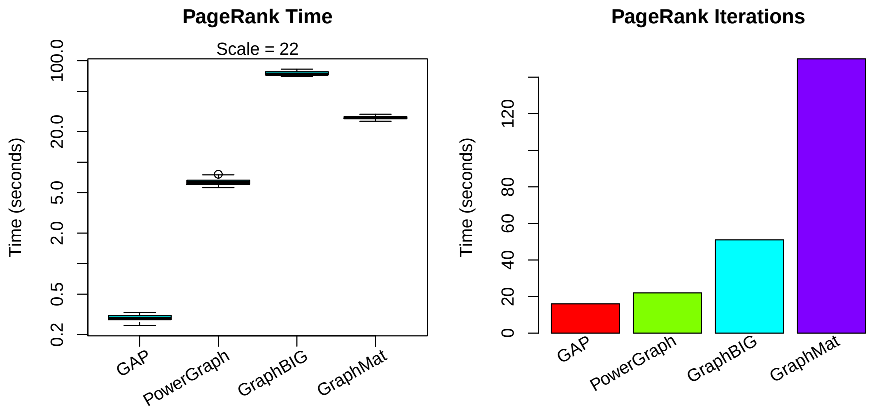
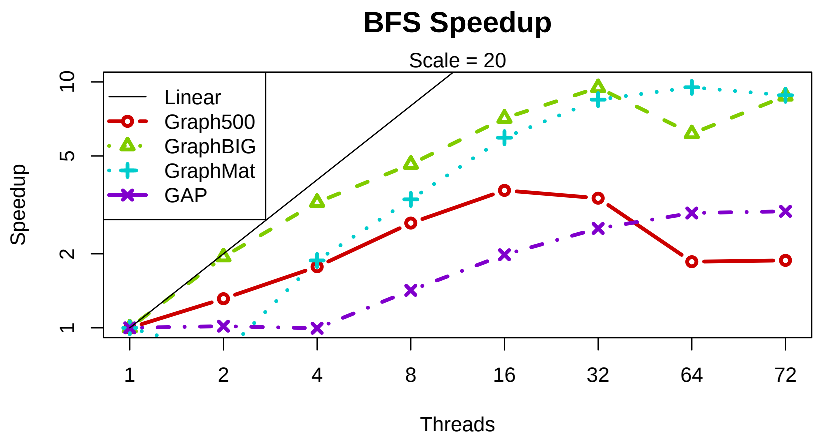


Fig. 4. The y-axis is logarithmic. GraphMat iterations are measured differently because of the "think like a vertex" paradigm and runs until none of the vertices' ranks change.



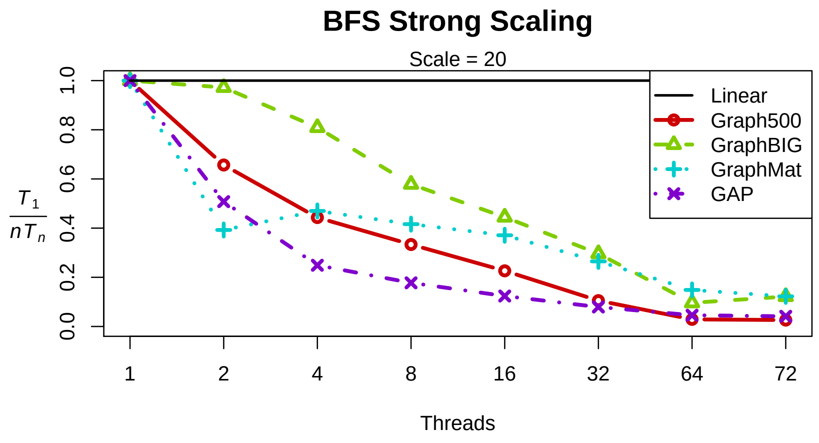


Fig. 5. The top figure has both axes logarithmic. The exception is the extra space at 72 threads for readability. Linear speedup is. GraphMat dips below 1 because it is slower for 2 threads than for 1. T_1 is the serial time, n is the number of threads, and T_n time for n threads.

4.2 Power and Energy Consumption

We use the Performance Application Programming Interface (PAPI) [2] to gain access to Intel's Running Average Power Limit (RAPL), which provides access to a set of hardware counters measuring energy usage. PAPI provides convenient access to these values and average energy in nanojoules for a given time interval. We modify the source code for each project to time only the actual BFS computation and give a summary of the results in Table 3. In our case, the fastest code is also the most energy efficient though with this level of granularity we could detect circumstances where one could make a tradeoff between energy and runtime.

Table 3. The data are generated from a Kronecker graph with 2^{22} vertices. Sleeping Energy refers to the power (in Watts) consumed during the Unix `sleep` function, multiplied by time. Effectively, this measures the amount of energy that would have been consumed even if nothing was running. The increase over sleep is the ratio of the first and third columns. These are all averaged over the 32 roots.

	GAP	Graph500	GraphBIG	GraphMat
Time (s)	0.01636	0.01884	1.600	1.424
Average Power per Root (W)	72.38	97.17	78.01	70.12
Energy per Root (J)	1.184	1.830	112.213	111.104
Sleeping Energy (J)	0.4046	0.4660	39.591	35.234
Increase over Sleep	2.926	3.928	2.834	3.153

RAPL also allows the measurement of DRAM power, the results of which are shown in Fig. 6. The left plot describes the second row of Table 3 in more detail. We notice a smaller spread of RAM power consumption, but still a noticeable difference. GraphMat results in the lowest RAM and power consumption despite not having the largest runtime.

Make text bigger

5 Future Work and Conclusion

We have presented an approach to analyze parallel graph processing performance at fine detail, both measuring power consumption and the two fundamental phases of execution. Still, the problem of choosing a system for graph processing at large scale is far from being simple. We make our code available at `https://github.com/HPCL/easy-parallel-graph` and encourage further experimentation. Our results required minor changes to the original projects to add calls to PAPI sampling and those changes are also available on forked versions of the repositories.

Overall, the GAP Benchmark Suite was the highest-performing system across the given datasets and the most scalable. However, this is only for relatively small graphs; all the graphs used here had at most 2^{22} vertices. It should be noted that

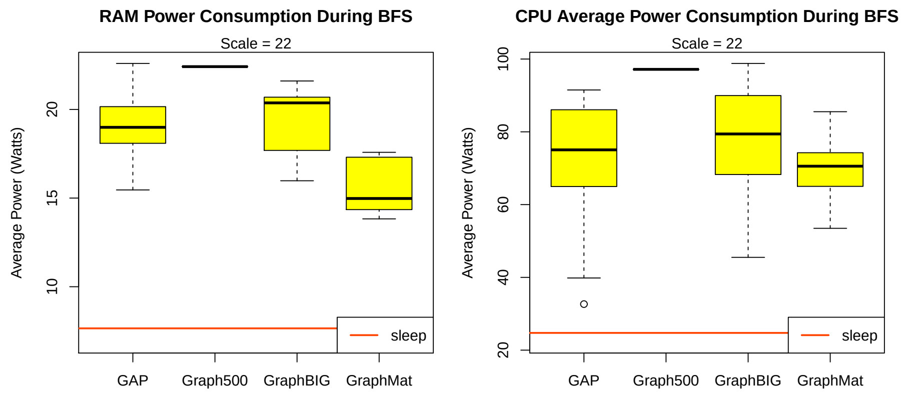


Fig. 6. Similar to Fig. 1 we plot RAM and CPU Power Consumption for each of the 32 roots. Since the Graph500 runs multiple roots per execution, we only get a single datapoint. The baseline was computed by monitoring power consumption while a program containing only one call to the C `unistd` library function `sleep(10)` (ten seconds).

GAP is also the most recent of projects. Thus, we recommend graph processing algorithm designers compare their implementations against the GAP Benchmark Suite to as a good test of performance.

There are a vast number of directions this work could go. To ensure fairness, each platform must be configured to use the same graphs and the same roots. In the case of GraphBIG this required the file to be loaded in for each experiment. This was by far the most time consuming of the experiments; the file reading for GraphBIG being done serially on an uncompressed ASCII text format for graphs. This limited the sizes of the experiments we could run. Graphalytics also had circumstances with the more computationally expensive algorithms where certain experiments fail [8], so determining whether an algorithm will finish given a particular machine, input size, runtime limit, and resources is an important unanswered question.

References

1. Beamer, S., Asanovic, K., Patterson, D.A.: The GAP benchmark suite. CoRR abs/1508.03619 (2015), `http://arxiv.org/abs/1508.03619`
2. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. 14(3), 189–204 (Aug 2000), `http://dx.doi.org/10.1177/109434200001400303`
3. Capotă, M., Hegeman, T., Iosup, A., Prat-Pérez, A., Erling, O., Boncz, P.: Graphalytics: A big data benchmark for graph-processing platforms. In: Proceedings of the GRADES'15. pp. 7:1–7:6. GRADES'15, ACM, New York, NY, USA (2015), `http://doi.acm.org/10.1145/2764947.2764954`

4. Doekemeijer, N., Varbanescu, A.L.: A survey of parallel graph processing frameworks. Tech. rep., Delft University of Technology (2014)
5. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). pp. 17–30. USENIX, Hollywood, CA (2012), `https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez`
6. Gregor, D., Edmonds, N., Barrett, B., Lumsdaine, A.: The parallel boost graph library. `http://www.osl.iu.edu/research/pbgl` (2005)
7. Guo, Y., Iosup, A.: The game trace archive. In: Proceedings of the 11th Annual Workshop on Network and Systems Support for Games. pp. 4:1–4:6. NetGames '12, IEEE Press, Piscataway, NJ, USA (2012), `http://dl.acm.org/citation.cfm?id=2501560.2501566`
8. Iosup, A., Hegeman, T., Ngai, W.L., Heldens, S., Pérez, A.P., Manhardt, T., Chafi, H., Capota, M., Sundaram, N., Anderson, M., Tanase, I.G., Xia, Y., Nai, L., Boncz, P.: Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms, a technical report. Tech. rep., Delft University of Technology (2016)
9. Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., Ghahramani, Z.: Kronecker graphs: An approach to modeling networks. J. Mach. Learn. Res. 11, 985–1042 (Mar 2010), `http://dl.acm.org/citation.cfm?id=1756006.1756039`
10. Lu, Y., Cheng, J., Yan, D., Wu, H.: Large-scale distributed graph computing systems: An experimental evaluation. Proc. VLDB Endow. 8(3), 281–292 (Nov 2014), `http://dx.doi.org/10.14778/2735508.2735517`
11. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Arg, J.A.: Introducing the graph500. Tech. rep., Cray User's Group (2010)
12. Nai, L., Xia, Y., Tanase, I.G., Kim, H., Lin, C.Y.: GraphBIG: Understanding graph computing in the context of industrial solutions. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 69:1–69:12. SC '15, ACM, New York, NY, USA (2015), `http://doi.acm.org/10.1145/2807591.2807626`
13. Ngai, W.L.: Fine-grained Performance Evaluation of Large-scale Graph Processing Systems. Master's thesis, Delft University of Technology (2015)
14. Satish, N., Sundaram, N., Patwary, M.M.A., Seo, J., Park, J., Hassaan, M.A., Sengupta, S., Yin, Z., Dubey, P.: Navigating the maze of graph analytics frameworks using massive graph datasets. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. pp. 979–990. SIGMOD '14, ACM, New York, NY, USA (2014), `http://doi.acm.org/10.1145/2588555.2610518`
15. Shun, J., Blelloch, G.E.: Ligra: A lightweight graph processing framework for shared memory. SIGPLAN Not. 48(8), 135–146 (Feb 2013), `http://doi.acm.org/10.1145/2517327.2442530`
16. Sundaram, N., Satish, N., Patwary, M.M.A., Dulloor, S.R., Anderson, M.J., Vadlamudi, S.G., Das, D., Dubey, P.: Graphmat: High performance graph analytics made productive. Proc. VLDB Endow. 8(11), 1214–1225 (jul 2015), `http://dx.doi.org/10.14778/2809974.2809983`
17. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: GraphX: A resilient distributed graph system on spark. In: First International Workshop on Graph Data Management Experiences and Systems. pp. 2:1–2:6. GRADES '13, ACM, New York, NY, USA (2013), `http://doi.acm.org/10.1145/2484425.2484427`