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 [15], which incurs some overhead while gaining fault tolerance and an example of the latter is Ligra [13], 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 is not easily available using Graphalytics.

With a plugin to Graphalytics called Granula [12] 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. The requires in-depth knowledge of the source code and execution model which is not always

available. 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.

3 Experimental Setup

3.1 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] [10]
2. The Graph Algorithm Platform (GAP) Benchmark Suite [1]
3. GraphBIG [11]
4. GraphMat [14]
5. PowerGraph [5]

3.2 Algorithms

We consider three algorithms though not all algorithms are available on all systems.

1. Breadth First Search (BFS)
2. Single Source Shortest Paths (SSSP)
3. PageRank

The canonical performance leaderboard for parallel graph processing is the Graph500 [10]. 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: BFS.

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

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

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

edge list stored in RAM and the actual BFS[3]. We run the BFS using 32 random roots.

One straightforward extension to BFS and our second algorithm is the Single-Source Shortest Paths algorithm (SSSP). We use the same graph and the same source vertices as in BFS.

The third algorithm is PageRank. These three algorithms are used because of their popularity; most libraries provide reference implementations.

3.3 Machine Specifications

Table 1 shows the specifications of the research computer.

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]

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`.

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 $D = 1 - (A + B + C) = 0.05$.

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 is sourced from the Game Trace Archive [7] and modified for Graphalytics[4].

4 Performance Analysis

4.1 Runtime

In Table 2 we show the results from running Graphalytics. For an explanation of each algorithm used, see [8].

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

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

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. It is difficult to make any claims about GraphMat from Graphalytics because of its lack of algorithm diversity.

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

Table 3, in contrast, gives an overview of the performance results of running graphalytics and our approach using the same dataset [TODO: GET DOTA LEAGUE WORKING FOR EPG!]. For compactness, Table 3 shows the average time across 32 roots, while Table 2 shows results for a single root. Figures 1 and 2 show the distribution of runtimes. However, even these do not give a complete picture of performance. Both of these show GraphMat to not be highly performant. However, a likely explanation is that GraphMat's underlying computation model (sparse matrix operations) paired the increased overhead of GraphMat's doubly-compressed sparse row (DSCR) graph representation is more conducive to larger-scale graphs.

Fig. 1. The y-axis is logarithmic. GraphBIG reads in the file and generates the data structure simultaneously so is omitted.

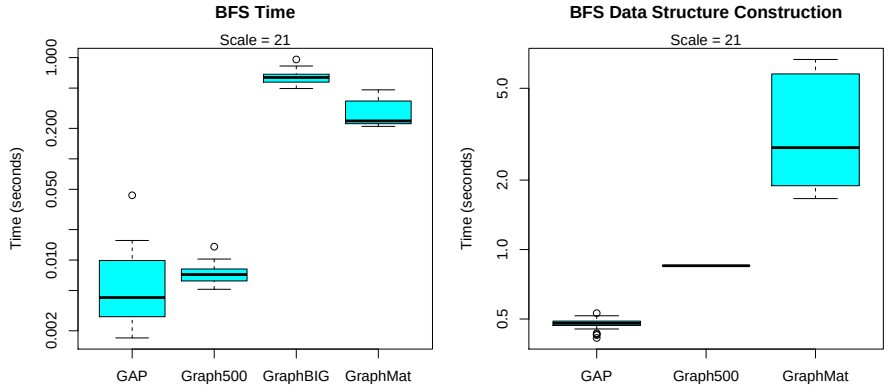


Fig. 2. The y-axis is logarithmic. [TODO: FIX IT SO GRAPHBIG HAS NO DSC]

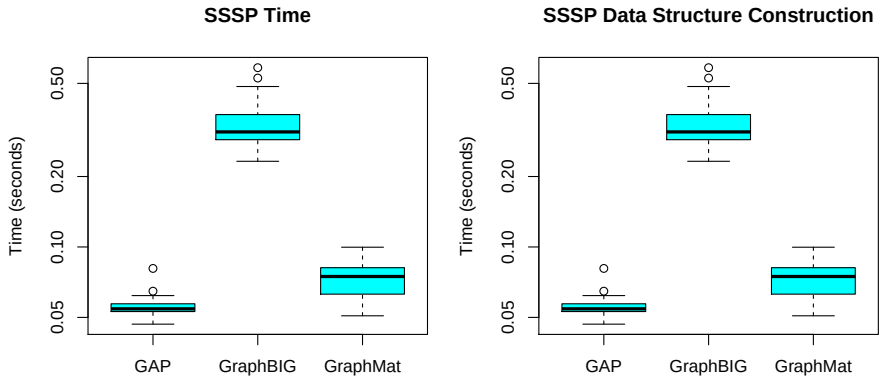


Table 3. The above table shows times for 2^{20} verties and the times are in seconds. The Graph500 generates the graph instead of loading it into a file. GraphBIG builds the graph and reads in the file simultaneously. These results were averaged across 32 roots.

System	Load Graph	Construct Data Structure	Run BFS
Graph500	0.3474	0.3971	0.003380
GAP	2.351	0.1935	0.001393
GraphMat	0.1511	1.101	0.1031
GraphBIG	37.22	N/A	0.1528

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 provides average energy in nanojoules for a given time interval.

5 Future Work and Conclusion

The problem of choosing a system for graph processing at large scale is far from being simple. We make our source code open source at `github.com/HPCL/easy-parallel-graph` and encourage further experimentation.

Fig. 3. Both figures use the kronecker graph with 2^{20} vertices with an average of 16 edges per vertex.

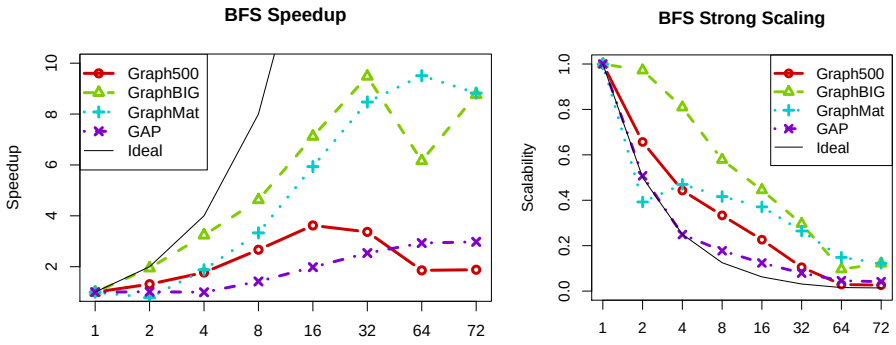
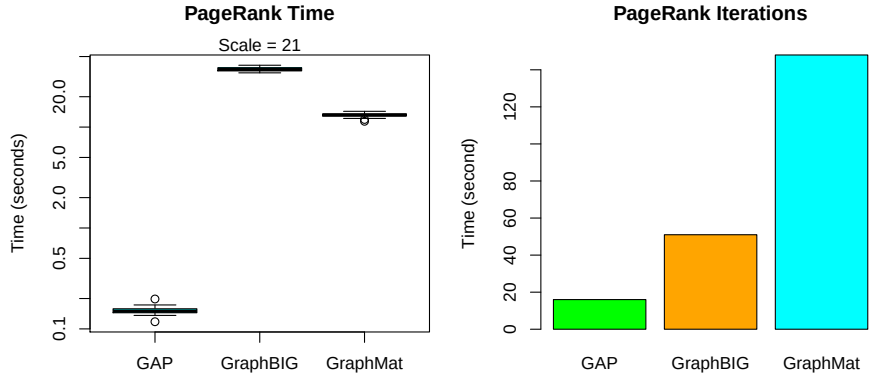


Table 4. The timeframe measured here is only when the BFS is performed. Sleeping Energy refers to the power (in Watts) consumed during the Unix `sleep` function, multiplied by time. Effectively, this measures the amount of energy that would have been consumed even if nothing was running. The increase over sleep is the ratio of the first and third columns. These are all averaged over the 32 roots.

	GAP	Graph500	GraphMat	GraphBIG
Energy Per Root (J)	1.184	1.830	112.213	111.104
Time (s)	0.01636	0.01884	1.600	1.424
Sleeping Energy (J)	0.4046	0.4660	39.591	35.234
Increase over Sleep	2.926	3.928	2.834	3.153

Fig. 4. The y-axis is logarithmic. GraphMat continues to run until none of the vertices' ranks change. [There is only one data point] The y-axis *not* logarithmic. GraphMat iterations are measured differently because of the "think like a vertex" paradigm and runs until none of the vertices' ranks change. [Note that for figure Fig.4, mention time per iteration]



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. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Arg, J.A.: Introducing the graph500. Tech. rep., Cray User's Group (2010)
11. 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
12. Ngai, W.L.: Fine-grained Performance Evaluation of Large-scale Graph Processing Systems. Master's thesis, Delft University of Technology (2015)
13. 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
14. 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

15. 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