A Performance and Recommendation System for Parallel Graph Processing Implementations: Work-In-Progress

Samuel D. Pollard
Sudharshan Srinivasan
Boyana Norris
spollard@cs.uoregon.edu
University of Oregon
Eugene, Oregon

ABSTRACT

There are nearly one hundred parallel and distributed graph processing packages. Selecting the best package for a given problem is difficult; some packages require GPUs, some are optimized for distributed or shared memory, and some require proprietary compilers or perform better on different hardware. Furthermore, performance may vary wildly depending on the graph itself. This complexity makes selecting the optimal implementation manually infeasible. We develop an approach to predict the performance of parallel graph processing using both regression models and binary classification by labeling configurations as either well-performing or not. We demonstrate our approach on six graph processing packages: GraphMat, the Graph500, the Graph Algorithm Platform Benchmark Suite, GraphBIG, Galois, and PowerGraph and on four algorithms: PageRank, single-source shortest paths, triangle counting, and breadth first search. Given a graph, our method can estimate execution time or suggest an implementation and thread count expected to perform well. Our method correctly identifies well-performing configurations in 97% of test cases.

CCS CONCEPTS

• **General and reference** → **Performance**; • **Computing methodologies** → *Supervised learning by classification*; *Supervised learning by regression*.

KEYWORDS

Parallel Graph Processing; Performance Engineering; Performance Analysis; Recommender System; Classification; Regression

ACM Reference Format:
Samuel D. Pollard, Sudharshan Srinivasan, and Boyana Norris. 2019. A Performance and Recommendation System for Parallel Graph Processing Implementations: Work-In-Progress. In *Tenth ACM/SPEC International Conference on Performance Engineering Companion (ICPE '19 Companion), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3302541.3313097

ICPE '19 Companion, April 7–11, 2019, Mumbai, India

ACM ISBN 978-1-4503-6286-3/19/04…$15.00
https://doi.org/10.1145/3302541.3313097

1 INTRODUCTION

Our research is motivated by the current state of parallel graph processing. Fields such as social network analysis [6] and computational biology [15] require the analysis of ever-increasing graph sizes. The wide variety of problem domains is resulting in the proliferation of Parallel Graph Processing Implementations (PGPI). The most comprehensive survey, released in 2014, identified and categorized over 80 different PGPIs [4] not including domain-specific languages such as Gremlin [16].

We take inspiration from the Graph500 benchmark [12] and its complete specification of graph kernels to fairly compare performance. We present EPG*, a framework which simplifies the installation, comparison, and performance analysis of four widely-implemented graph algorithm building blocks: breadth first search (BFS), single-source shortest paths (SSSP), PageRank (PR), and Triangle Counting (TC).

This paper describes the following contributions.

- Reproducible analysis of the performance and scalability of PGPIs.
- Recommendation of well-performing algorithms across PGPIs based on properties of the input graphs. To illustrate our approach, we show that for a common performance objective, Traversed Edges Per Second, the average improvement recommended by the model is between 6% (for PR) and 700% (for BFS) better than the average across all configurations. The same approach can be applied to other objectives such as energy or power.

To be clear, we are not proposing a new benchmark suite or providing new implementations. Instead, we introduce a method to compare and recommend existing software packages without requiring the user to be familiar with the underlying implementation. We provide EPG* open-source at https://github.com/HPCL/easy-parallel-graph.

2 METHODOLOGY

EPG* works by automatically downloading or generating graph datasets and converting them to the correct formats, running experiments for different thread counts, then using

this as training data to a recommender system. This system either uses regression to predict runtime or classifies configurations as "well-performing" or not.

At each stage we ensure the comparison is fair; we use undirected graphs with no duplicate edges, the same graph files, random seeds, and stable repository forks. We manually inspect source code to ensure timing data and stopping criteria (e.g. termination condition for PR) are consistent. Namely, we split total execution time into data structure construction, input file reading, and algorithm runtime.

Each experiment is run multiple times to account for algorithmic and OS variation and uses different starting vertices (roots) for BFS and SSSP for each trial. For building the machine learning model, we use a grid search on Kronecker [8] synthetic graphs and 25 real-world graphs from the Stanford Network Analysis Project (SNAP) [9] or KONECT [7].

2.1 Graph Processing Systems

This study explores five shared-memory parallel graph processing packages and one distributed memory framework operating on a single node (Powergraph). They are:

(1) The Graph500 [12]
(2) The Graph Algorithm Platform (GAP) Benchmark Suite [2]
(3) GraphBIG benchmark suite [13]
(4) GraphMat [19] PowerGraph [5]
(5) Galois [14]
(6) Powergraph [5]

2.2 Algorithms

We consider four parallel algorithms: breadth first search (BFS), single-source shortest paths (SSSP), triangle counting (TC), and PageRank (PR), though not all algorithms are implemented on all systems.

One challenge with PR is the stopping criterion; all implementations have been modified to use $\|p_t - p_{t-1}\|_1$ (the absolute sum of differences) where p_t is the PageRank vector at step t. Verification of the PR results is beyond the scope of this paper though this may explain performance discrepancies.

Our approach is not specific to a particular algorithm; measuring the execution time, data structure construction time, and power consumption can be applied easily to other implementations. We hope this motivates others to use the framework for their own work.

2.3 Machine Specifications

We performed experiments on a 56-core (28 thread) node with 128GB DDR4 RAM and two Intel Xeon E5-2690 v4 CPUs. The operating system is GNU/Linux 3.10.0. Our code was compiled with GCC version 4.8.5 except GraphMat which was compiled with the Intel compiler version 17.0.0.

3 MACHINE LEARNING RECOMMENDATIONS

3.1 Graph Features

We use as features the numbers of threads, edges, vertices, average clustering coefficient, number of triangles, fraction of closed triangles, diameter (longest shortest path between two vertices), 90th percentile effective diameter, and the fractions of nodes and edges in largest strongly and weakly connected components (SCC and WCC) for a total of 12 features.

For regression, we attempt to predict runtime. However, for small graphs or some roots, the runtime is effectively 0 which skews our models so we instead calculate the z-score of the runtimes for each algorithm (normalization).

For classification, we normalized using Traversed Edges Per Second (TEPS) which is runtime divided by the number of edges in the graph. This is a misnomer for algorithms like PR which touch edges multiple times, but to account for this we train separate models for each algorithm.

3.2 Models

The goal of EPG* is to suggest the best package for a given graph, algorithm, and hardware configuration. To accomplish this we model performance using linear regression and binary classification.

We split the data into a 70%-30% training and test set. The linear model was then trained for each algorithm with the normalized and unnormalized version of the training set. To further improve the accuracy, we incorporated a ridging based linear regression model.

Since predicting runtime is a secondary goal (the user just wants the best-performing package), we also built classifiers to determine if a package is well-performing. We began with binary classification using logistic regression. Each data point in the learning set is tagged either "good" or "bad" based on its TEPS. We weighted these tags such that roughly 1/3 are considered "good." (See the sums along the rows of Table 1)

We also implemented a supervised decision tree-based model, random forests, and achieved higher accuracy.

4 MACHINE LEARNING RESULTS

4.1 Regression

We built linear regression models for four different configurations: with and without ridging on normalized and unnormalized runtimes. Figure 1 shows how the results progressively improved with each configuration leading up to the best one being ridging on the normalized learning set (R+N). However, the R^2 values for R+N, while an improvement, are still relatively low.

Although R^2 is a common metric for evaluating a linear model, it is not sufficient for our purposes. This can be intuited by noting our goal is to predict runtime; a linear model can have a low R^2 yet still predict runtime with reasonable accuracy. We measure Root Mean Square Deviation (RMSD) to measure the degree of difference in runtime (the middle plot in Figure 1). However, RMSD is hard to compare across

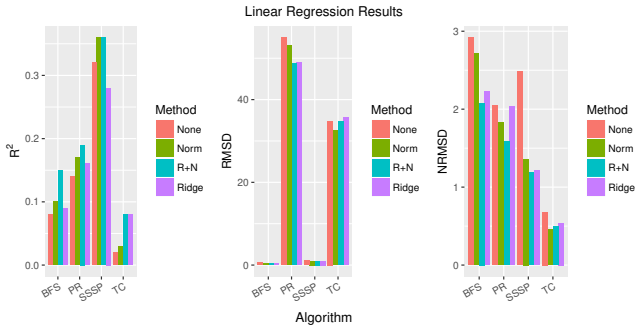

Figure 1: Linear regression with and without ridging and normalization

Table 1: Confusion matrices for all algorithms. The columns are predictions, the rows are observations.

(a) Logistic Regression

		good	bad
TC	good	0	49
	bad	0	113
BFS	good	133	9
	bad	91	4
PR	good	161	0
	bad	81	0
SSSP	good	27	85
	bad	17	113

(b) Random Forest

		good	bad
TC	good	43	1
	bad	0	118
BFS	good	110	0
	bad	0	147
PR	good	75	0
	bad	1	193
SSSP	good	51	11
	bad	9	198

algorithms; BFS has a much lower runtime than TC. Thus, we use and Normalized Root Mean Square Deviation (NRMSD) (the right plot). This gives us a better idea of how much the runtimes vary as a percentage of overall runtime. We note that our model predicts runtime within a factor of 2 in the worst case (BFS) and a factor of 0.5 in the best case (TC).

4.2 Classification

The classification model is more accurate than linear regression in identifying the well-performing graph, algorithm, and hardware configuration. Table 1(a) shows the confusion matrix for logistic regression across all algorithms. Although the model achieves an overall accuracy of 64%, it categorizes all data points as "bad" for TC and "good" for PR, resulting in a high percentage of false negatives and false positives for TC and PR, respectively.

This motivated us to consider methods which work well even with a relatively small training set. Table 1(b) shows the confusion matrix of a random forest classifier across all algorithms. Random forest provides a significant improvement over logistic regression. Not only does it achieve a mean accuracy of 97% across all algorithms, the percentage of false positives and negatives is also significantly reduced; the model achieves perfect classification for BFS and a near perfect classification for TC and PR.

Table 2: TEPS for each algorithm. Improvement is computed as the mean TEPS of data labeled as good divided by the mean TEPS for all data. Larger TEPS indicates better performance.

Algorithm	Min	Max	Mean (all)	Mean (good)	Improvement (%)
TC	1.2e2	2.3e3	6.8e2	1.6e3	66.6
BFS	2.8e3	3.1e10	1.1e8	8.1e8	700.7
PR	7.6e1	1.1e5	4.4e3	4.7e3	6.8
SSSP	2.3e2	2.6e10	8.2e7	1.1e8	34.1

Even when the model wrongly classifies a data point as good, it is only negligibly slower than average. With SSSP, the mean TEPS for the nine false positives is slower than the mean overall TEPS by only 0.03%. The singular false positive for PR is 11% slower, but we used the worst results from eight random seeds for Table 1; the rest had no false positives.

Similar to performance penalty, performance improvement for true positive predictions is also better for random forests. Performance improvement determines by how much the proposed model outperforms a hypothetical strategy which picks configurations at random. We see from Table 2 that for each algorithm, the mean TEPS across all experiments is significantly lower than the mean for experiments labeled "good" and this improvement ranges from 7% to 700%. The performance improvement for PR is not as high as the other algorithms. This is explained by the low variance in runtime—and therefore TEPS—of PR: the coefficient of variation (standard deviation/mean) of TEPS for BFS it is 1.5 and PR is only 0.21.

5 RELATED WORK

Most performance analysis of graph processing systems come from the empirical results of each new library designer's publications, like for GraphMat [19], PowerGraph [5], and GraphBIG [13]. Other work uses hand-tuned implementations and provides performance improvement recommendations [10, 17]. Additionally, Graphalytics [3] automates the setup and execution of graph packages for performance analysis. These projects accomplish a different goal of comparing performance between packages for a given dataset rather than building models of performance characteristics.

With respect to algorithm classification, the work most similar to ours appears in [11], where machine learning models are used to predict the scalability of the Graph500 which supports only BFS and SSSP.

6 CONCLUSION & FUTURE WORK

We presented a method for predicting the performance of parallel graph processing implementations (PGPI). We make a fair comparison by ensuring graphs, stopping criterion, and runtime configurations are uniform across PGPIs. We use EPG* to run a large number of experiments and from them generate models of performance based on the graph algorithm, hardware, and input graph.

We demonstrated its effectiveness in recommending implementations, resulting in 97% classification accuracy and mean performance improvement ranging between 7% and 700%. To improve this further, we note each package uses different timing software and poorly-chosen roots may be measured as 0. Even with good roots, BFS only takes 0.2 seconds on a graph with ~ 64 million edges so is more susceptible to OS jitter and measurement error. Future work includes running experiments on larger graphs and to better select roots. To improve the robustness of our models we will run on different hardware. We should reproduce the results on the Graph500 [11] to more critically evaluate our approach.

Additionally, we plan to combine regression and classification to produce a ranking of configurations similar to work by Sood et al. [18]. We also seek to refine our features to speed up our recommendations; of the 12 features, some should have a greater effect on runtime.

We plan to add more common graph algorithms such as betweenness centrality and minimum spanning tree. Additionally, EPG* measures six PGPI. Adding more is not difficult; our goal is to motivate the developers of these packages to add to EPG*. One potential motivator for community contribution would be a leaderboard in which our recommendation system ranks implementations by their aggregate performance. The difference between this and the Graph500 is our leaderboard would measure more algorithms and datasets.

Lastly, parallel SSSP and BFS algorithms contain parameters (Δ for SSSP and α and β for BFS) which affect performance. We plan to add parameter tuning as performed in [1].

Our vision is this approach can change the way users select an implementation for graph computations; instead of tracking down performance results and repositories, they can simply provide a description of their problem(s) and receive a recommendation, installed and configured for their machine, ready to use.

ACKNOWLEDGMENTS

This material is supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Contract DE-AC02-06CH11357.

REFERENCES

[1] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-Optimizing Breadth-First Search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society, Salt Lake City, UT, USA, Article 12, 10 pages. http://dl.acm.org/citation.cfm?id=2388996.2389013

[2] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). http://arxiv.org/abs/1508.03619

[3] Mihai Capotă, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. 2015. Graphalytics: A Big Data Benchmark for Graph-Processing Platforms. In *Proceedings of the Graph Data Management Experiences and Systems (GRADES '15)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2764947.2764954

[4] Niels Doekemeijer and Ana Lucia Varbanescu. 2014. *A Survey of Parallel Graph Processing Frameworks.* Technical Report. Delft University of Technology.

[5] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (ODSI '12)*. USENIX, Hollywood, CA, USA, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[6] U. Kang, Brendan Meeder, and Christos Faloutsos. 2011. Spectral Analysis for Billion-Scale Graphs: Discoveries and Implementation. In *Advances in Knowledge Discovery and Data Mining (LNCS)*, Vol. 6635. Springer, Berlin. https://doi.org/10.1007/978-3-642-20847-8_2

[7] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web (WWW '13 Companion)*. ACM, Rio de Janeiro, Brazil, 1343–1350. https://doi.org/10.1145/2487788.2488173

[8] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker Graphs: An Approach to Modeling Networks. *Journal of Machine Learning Research* 11 (March 2010), 985–1042. http://dl.acm.org/citation.cfm?id=1756006.1756039

[9] Jure Leskovec and Andrej Krevl. [n. d.]. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. [Accessed 6/2018].

[10] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale Distributed Graph Computing Systems: An Experimental Evaluation. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 281–292. https://doi.org/10.14778/2735508.2735517

[11] S. Medya, L. Cherkasova, and A. Singh. 2017. Predictive modeling and scalability analysis for large graph analytics. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 63–71. https://doi.org/10.23919/INM.2017.7987265

[12] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Arg. 2010. *Introducing the Graph500.* Technical Report. Cray User's Group.

[13] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, NY, USA, Article 69, 12 pages. https://doi.org/10.1145/2807591.2807626

[14] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP '13)*. 456–471. https://doi.org/10.1145/2517349.2522739

[15] Georgios A Pavlopoulos, Maria Secrier, Charalampos N Moschopoulos, Theodoros G Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G Bagos. 2011. Using Graph Theory to Analyze Biological Networks. In *BioData Mining*. PubMed Central, Bethesda, MD. https://doi.org/10.1186/1756-0381-4-10

[16] Marko A. Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language. *CoRR* abs/1508.03843 (2015). http://arxiv.org/abs/1508.03843

[17] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 979–990. https://doi.org/10.1145/2588555.2610518

[18] K. Sood, B. Norris, and E. Jessup. 2017. Comparative Performance Modeling of Parallel Preconditioned Krylov Methods. In *IEEE 19th International Conference on High Performance Computing and Communications; 15th International Conference on Smart City; 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 26–33. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.4

[19] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *The Proceedings of the VLDB Endowment* 8, 11 (July 2015), 1214–1225. https://doi.org/10.14778/2809974.2809983