A Comparison of Parallel Graph Processing Implementations

Samuel D. Pollard
Computer and Information Science
University of Oregon
Eugene, OR 97403
Email: spollard@cs.uoregon.edu

Boyana Norris
Computer and Information Science
University of Oregon
Eugene, OR 97403
Email: norris@cs.uoregon.edu

***Abstract*—The rapidly growing number 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 processing packages: GraphMat, the Graph500, the Graph Algorithm Platform Benchmark Suite, 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.**

I. Introduction

Our research is motivated by the current state of parallel graph processing. Fields such as social network analysis [13] and computational biology [24] require the analysis of ever-increasing graph sizes. The wide variety of problem domains is resulting in the proliferation of parallel graph processing frameworks. The most comprehensive survey, released in 2014, identified and categorized over 80 different parallel graph processing systems not even considering domain specific languages [7].

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 can be time consuming tasks. Moreover, 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 [30], which incurs some overhead while gaining fault tolerance and an example of the latter is Ligra [27], a framework requiring a shared-memory architecture. Additionally, there is a growing number of systems designed to run on GPUs [31].

From a graph algorithm user's point of view, optimizations for each system may not be apparent. For example, the Parallel Boost Graph Library (PBGL) [10] provides generic implementations of their algorithms and the programmer must provide the template specializations. Optimal data structures may differ across graphs and must be determined by the programmer.

We aim to simplify the decision making process by providing an easy-to-use framework for consistently and fairly evaluating the performance of different algorithms. To illustrate the use of this framework, we include results from our experiments on both real-world and synthetic datasets. We take inspiration from the Graph500 benchmark [20], which clearly specifies every step of the breadth first search (BFS) algorithm and how it should be timed. The Graph500 can be used to rank systems fairly on a single, well-defined benchmark. In this context, *system* is defined as the hardware, operating system, middleware, and algorithmic implementation which accomplishes a certain task, in this case BFS.

While leaderboards can indicate computational milestones, they are not particularly useful for the average user. Reference implementations, however, are critical for advancement; the complexity of today's hardware and operating systems requires empirical evidence to assess an algorithm's performance. In addition, there is motivation to define basic building blocks for graph computations [1], [5]. However, new algorithm designers and users alike have no easy way to accurately determine what constitutes a high-performance implementation.

To address these issues, we introduce *easy-parallel-graph-$*$*,[1] a framework which simplifies the installation, comparison, and performance analysis of the three most widely implemented graph algorithm building blocks: breadth first search (BFS), single source shortest paths (SSSP), and PageRank (PR). With this framework we select a small number of graph processing libraries over which we homogenize all aspects of execution to compare performance fairly. Another contribution of this paper is a demonstration of the use of this framework to analyze the

[1]Available at https://github.com/HPCL/easy-parallel-graph

performance, energy consumption, and scalability of specific algorithm implementations. This enables algorithm designers to select high performance reference implementations with which they can compare their improvements. To be clear, we are not proposing a new benchmark suite or providing any new reference implementations. Instead, we introduce a method for detailed and fair comparison of existing graph software packages without requiring the user to be familiar with them.

Section II describes previous performance analysis of parallel graph algorithms. Section III describes the architecture of *easy-parallel-graph-∗* and the algorithms, datasets, and graph processing implementations it uses. Section IV presents performance analysis and discusses possible explanations for the results. Section V describes future directions for this research and is followed by the conclusion.

II. Related Work

Most performance analysis of graph processing systems come from the empirical results of each new library designer's publications. For example, GraphMat [29], PowerGraph [9], and GraphBIG [22] present a comparison of performance between their system and a selection of other software packages and in general any new approach will compare itself to existing implementations. We assume the performance results which accompany a new library are at risk of bias because the authors typically know their own package better than the competition and as such are aware of more optimizations. For a collection of references to the original papers of each new library or framework, see [7]. Instead, we focus on related research which provide analysis of existing software.

Nai et al. [21] provide a detailed performance analysis using GraphBIG as their reference implementation. Their approach also compares GraphBIG to the GraphLab and Pregel execution models (Gather Apply Scatter). Their analysis considers architectural performance measurements such as cache miss rates to measure bottlenecks and computational efficiency for a variety of datasets. While incredibly thorough, these analyses focus on covering a wide range of datasets and appear difficult to apply to a new approach. For example, GraphMat reduces computation to sparse matrix operations which may not suffer from the same memory bottlenecks indicated in [21].

Research by Song et al. [28] and LeBeane et al. [14] considers the important problem of partitioning on heterogeneous architectures and provides profiling to reduce runtime, improve load balance, and reduce energy consumption. Our framework instead focuses on shared memory packages.

Beyond the aforementioned performance results, a number of reports focus on performance analysis not tied to a particular software package. Satish et al. [26] analyze the performance of several systems on datasets on the order of 30 billion edges and [18] uses six real-world datasets and focuses on the vertex-centric programming model. These implementations are hand tuned and provide recommendations for future improvements.

TABLE I
Graphalytics: tabulated sample run times (seconds) with 32 threads. Just one run per experiment is performed. Below the table is an excerpt from the GraphMat log file.

GraphBIG	BFS	CDLP	LCC	PR	SSSP	WCC
cit-Patents	0.8	11.8	15.5	4.5	N/A	1.3
dota-league	1.1	3.9	1073.7	**2.6**	3.0	1.0

PowerGraph	BFS	CDLP	LCC	PR	SSSP	WCC
cit-Patents	13.8	30.1	23.9	18.8	N/A	22.1
dota-league	25.6	31.2	458.1	26.7	28.9	22.9

GraphMat	BFS	CDLP	LCC	PR	SSSP	WCC
cit-Patents	7.5	20.1	9.8	8.1	N/A	6.6
dota-league	2.7	21.2	239.7	**6.3**	9.4	6.9

Timing results (for GraphMat PageRank on dota-league)
- Finished file read of dota-league. time: 2.65211
- load graph: 5.91229 sec
- initialize engine: 8.32081e-05 sec
- run algorithm 1 (count degree): 0.0555639 sec
- run algorithm 2 (compute PageRank): 0.149445 sec
- print output: 0.0641179 sec
- deinitialize engine: 0.00022006 sec

The most prominent example of a graph processing performance analysis tool not tied to a particular implementation is Graphalytics [6]. Similar to our work, Graphalytics also automates the setup and execution of graph packages for performance analysis. Graphalyics relies on Apache Maven and Java to wrap the execution of each graph processing software package.

Graphalytics generates an HTML report listing the runtimes for each dataset and each algorithm. Table I shows the results from a typical experiment run by Graphalytics.

However, perusal of the log files reveals a different story. The bullets shown below Table I summarize the output from GraphMat itself. Graphalytics reports a 6.3 second runtime but 2.7 seconds of that time GraphMat is simply reading the input file from disk. However, the GraphBIG timing (2.6 seconds) does not include the time to read the dota-league file. **If the time to read in the text file was ignored then GraphMat would complete nearly twice as quickly.** To call this a fair comparison is dubious at best when certain phases of execution are omitted for some packages but not others.

With a plugin to Graphalytics called Granula [23], 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 in addition to expertise with the Granula API but allows detailed performance analysis and automatic execution and compilation of performance results.[2].

[2]An example of Granula can be seen at http://bit.ly/granula-example

Beyond this plethora of performance data, the Graph Algorithm Platform Benchmark Suite (GAP) [3], GraphBIG [22], and the Graph500 [20] consider themselves reference implementations or benchmark suites with which other implementations can be compared. If no fewer than three software packages call themselves, "reference implementations," which one are we to trust? Our belief is that choosing a single reference implementation cannot capture the complexities inherent in graph processing.

III. Methodology

Our contribution strikes a balance between the highly detailed analysis such as those presented by Nai [21] (greater depth of analysis) and Graphalytics which compares only the runtimes of a greater number of software packages and algorithms (greater breadth of analysis). This middle-ground simplifies our interface which in turn allows users to perform their own analysis on their own datasets with greater ease than existing frameworks.

In addition, we analyze power usage, energy consumption, and details such as the number of iterations for PageRank or the time to construct the graph data structures. Moreover, our approach requires little knowledge of the inner workings of each system; the data are collected by either parsing log files (for execution time) or hardware counter sampling of model-specific registers (for power). This allows arbitrary datasets to be included as long as they are in the same format as those in the Stanford Network Analysis Project (SNAP) datasets [17].

Our framework breaks the process of characterizing performance into five principal phases shown in Fig. 1, each of which requires no more than a single shell command. The five phases are as follows.

1) Installing modified, stable forks of each software package to ensure homogeneity.
2) Given a synthetic graph size or a real-world graph file, generate the files necessary to run each software package.
3) Given a graph and the number of threads, run each algorithm using each software package multiple times.
4) Parse through the log files to compress the output into a CSV.
5) Analyze the data using the provided R scripts to generate plots.

A. Installing Libraries

The libraries are stable forks of the given repositories which are configured to ensure the experiments execute in the same manner.

Our experiments use the author-provided implementations with modifications only to insert performance analysis hooks or to ensure homogeneous stopping criteria; we assume the developers of each system will provide the best performing implementation. While this limits the scope of the experiments it mitigates the bias inherent in our programming skills in addition to the bias of library designers' performance analysis; a given library designer will understand his or her source code better than any other implementation.

B. Datasets

Homogenizing the datasets creates copies of the graph files and auxiliary files in various formats. This is both to ensure they are correctly formatted for each system and to speed up file I/O whenever possible by using the library designer's serialized data structure file formats.

We refer to a graph's *scale* when describing the size of the graph. Specifically, a graph with scale S has 2^S vertices. For example, many of our experiments were performed on graphs with $2^{22} = 4,194,304$ vertices and an average of 16 edges per vertex. We measure parallel efficiency and speedup by varying the number of threads from one to the total number of threads available on our research server, 72.

Each experiment uses 32 roots per graph. As with the Graph500, each root is selected to have a degree greater than 1. For PageRank, we simply run the algorithm 32 times. We plot many of the results as box plots with an implied 32 data points per box.

When possible, we measure the dataset 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.

We use the Graph500 synthetic graph generator which creates a Kronecker graph [16] 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. Kronecker graphs are a generalization of RMAT graphs.

Part of the Graphalytics results in Table I were performed on the Dota-League dataset. We use this dataset for other experiments and as well. It contains 61,670 vertices and 50,870,313 edges and models interactions between players in the online video game Defense of the Ancients. This was sourced from the Game Trace Archive [11] and is modified for Graphalytics[3]. 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.

We also present results for the `cit-Patents` dataset from SNAP [15]. This is a widely-used network of citations from the National Bureau of Economic Research (NBER) and is less dense than Dota-League with 3,774,768 vertices and 16,518,948 edges. We stress that though these two

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

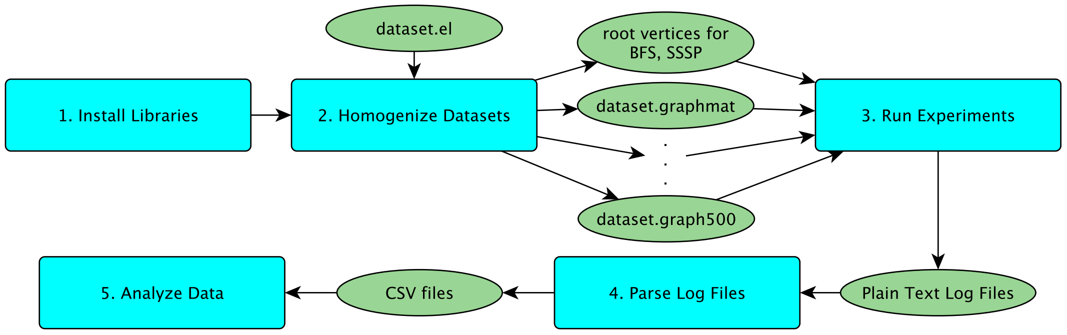


Fig. 1. Overview of *easy-parallel-graph-∗*. Each cyan box corresponds to a single shell script in the package while the green ellipses correspond to generated files.

datasets are presented, any network in the SNAP data format[4] can be used in *easy-parallel-graph-∗*.

C. Graph Processing Systems

This study explores four shared memory parallel graph processing frameworks and one distributed memory framework operating on a single node (Powergraph). The first three are so-called "reference implementations" while the remaining two are included because of their performance and popularity. Other popular libraries such as the Parallel Boost Graph Library [10] are not considered here because the authors do not provide reference implementations. The frameworks are as follows:

1) The Graph500 [20], the canonical BFS benchmark which consists of a specification and reference implementation. We use a modified version most similar to 2.1.4. The Graph500 provides reference implementations for MPI and OpenMP but since our framework focuses on shared memory we use only the OpenMP version. The Graph500 uses a compressed sparse row (CSR) representation.
2) The Graph Algorithm Platform (GAP) Benchmark Suite [3], a set of reference implementations for shared memory graph processing. The author of GAP has contributed to the Graph500 so the BFS implementations are similar. Additionally, GAP uses OpenMP to achieve parallelism and uses a CSR representation.
3) GraphBIG [22] benchmark suite. We consider only the shared memory solutions but GraphBIG also provides GPU benchmarks. GraphBIG uses a CSR representation for graphs and OpenMP for parallelism.
4) GraphMat [29], a library and programming model along with reference implementations of common algorithms. GraphMat uses a doubly-compressed sparse row representation and OpenMP for parallelism.
5) PowerGraph [9], a library and programming model for distributed (and shared memory) graph-parallel computation. PowerGraph includes additional toolkits for tasks such as clustering and computer vision. Parallelism is achieved via a combination of OpenMP and light-weight, user-level threads called fibers. Powergraph uses a novel storage scheme on top of CSR.

[4]A file in the SNAP format consists of one edge per line, with vertices separated by whitespace and lines which begin with # are comments.

Our approach is not specific or limited to these graph packages and can be extended to others. When comparing to Graphalytics, we use the most recent stable release, v0.3.

D. Algorithms

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

We select BFS because the canonical performance leaderboard for parallel graph processing is the Graph500 [20]. One strength 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 unsorted edge list stored in RAM and the actual BFS[5]. 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 and can use the same the root vertices from BFS. Furthermore, SSSP is used as a building block for other graph algorithms such as Betweenness Centrality.

PageRank is selected because of its popularity; most libraries provide reference implementations. One challenge

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

TABLE II
GRAPHALYTICS ON THE SAME KRONECKER GRAPH WITH SCALE 22 AS USED IN OTHER EXPERIMENTS. PERFORMANCE RESULTS ARE IN SECONDS WITH 32 THREADS. COMMUNITY DETECTION USES LABEL PROPAGATION.

Graphalytics	GraphMat	GraphBIG	PowerGraph
Community Detection	45.8	7.4	55.6
PageRank	8.9	4.7	46.4
Local Clustering Coeff.	401	1,802.7	299.8
Weakly Conn. Comp.	7.4	2.4	40.5
BFS	10.3	1.8	43

with using PageRank 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 page rank at step t. Verification of the PageRank results is beyond the scope of this paper, although 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 applied easily to other algorithms. We select some representative algorithms here to demonstrate the process, which we hope will motivate others to use the framework to add more algorithms.

E. Parsing and Data Analysis

Our example analyzes data using the R programming language and shell (Bash and AWK) parsers. The example workflow provided in our framework yields the figures and tables generated in this paper.

F. Machine Specifications

We performed experiments on our 36-core (72 thread) Intel Haswell server with 256GB DDR4 RAM, with two Intel Xeon E5-2699 v3 CPUs. The operating system is GNU/Linux version 4.4.0-22. Our code was compiled with GCC version 4.8.5 with the exception of GraphMat which was compiled with ICPC version 17.0.0.

IV. PERFORMANCE ANALYSIS

We analyze the performance of the algorithms described in Sec. III-D in terms of execution time, scalability over multiple threads, and power and energy consumption. All figures except those measuring scalability (Figs. 5 and 6) use 32 threads.

A. Runtime for Synthetic Graphs

In Table II we show the results from running Graphalytics on a Kronecker graph of scale 22 (4,194,304 vertices and approximately $16 \times 2^{22} \approx 33,500,000$ edges). An explanation of each algorithm is given in [12]. Graphalytics by default does not perform SSSP on unweighted, undirected graphs. The discrepancy between PageRank values in Table II and Fig. 4 is a result of the differing stopping criterion and the aforementioned inconsistency of Graphalytics's performance collection scheme.

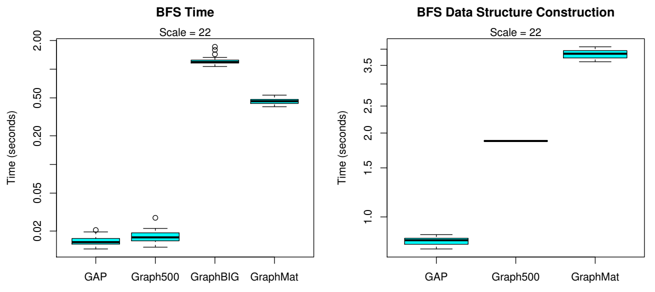


Fig. 2. The y-axes are logarithmic. The left box plot shows the time to compute BFS on 32 random roots while the right plot shows the times to construct the graph for each system. The Graph500 only constructs its graph once. GraphBIG reads in the file and generates the data structure simultaneously so is omitted.

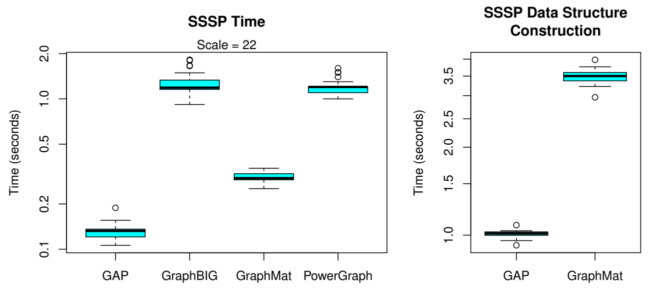


Fig. 3. The y-axes is logarithmic. The left box plot shows the time to compute the SSSP starting at the same 32 roots as Fig. 2. Both PowerGraph and GraphBIG construct their data structures at the same time as they read the file.

In contrast, our results for the Kronecker graph of scale 22 are presented in Figs. 2, 3, and 4.

Figures 2 and 3 show performance results for a Kronecker graph with scale 22. The box plots give an idea of the runtime distributions. There is less variance in 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 [29] which lists GraphMat as more higher performing than PowerGraph in SSSP.

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 define 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 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 criterion, 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 to make the stopping criteria for all implementations as

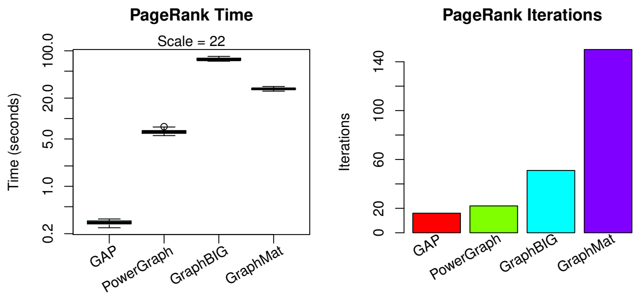


Fig. 4. The y-axis is logarithmic only for the left figure. GraphMat continues to run until none of the vertices' ranks change. For the others, we use the stopping condition that the sum of the changes in the weights is no more than 6×10^{-8}, or approximately machine epsilon for single precision floating point numbers.

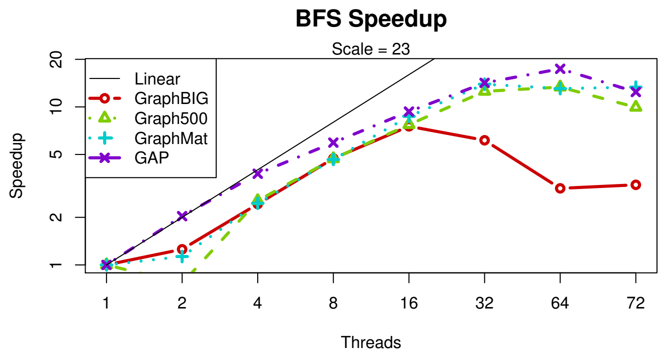


Fig. 5. Speedup of BFS for a scale-23 graph, black solid line represents ideal speedup. Both axes are logarithmic with an exception at 72 threads for readability.

similar as possible. However, with GraphMat there is no computation of $|p_k^{(i)} - p_k^{(i-1)}|$.

Moreover, the logarithmic scale in Figures 2 and 4 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.

The difficulty 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 can be recompiled to store weights as integers or floating-point values. 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.

B. Scalability

Figures 5 and 6 illustrate the parallel strong efficiency of the BFS algorithm in different packages on a Kronecker graph of scale 23. The parallel speedup shown in Fig. 5 is computed as T_1/T_n where T_1 is the sequential time and T_n is the execution time on n threads. Because of timing considerations, only four trials were run for these experiments.

Figure 6 shows the parallel efficiency, $T_1/(nT_n)$ for different implementations of BFS. Ideal efficiency is defined as $T_n = T_1/n$ and is the horizontal line near the top of Fig. 6.

These plots show generally poor scaling for this size problem, a challenge facing most parallel graph algorithms. In addition, 2^{23} vertices is a relatively small graph by today's standards and thus library designers focus on scalability for larger graphs. Another limitation may be the ability for OpenMP to efficiently handle such a large number of threads per machine. GCC version 4.8.5 supports OpenMP version 3.1 but the most recent release of OpenMP is 4.5.

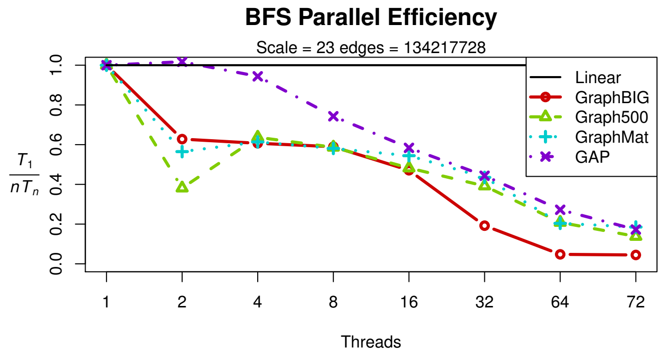
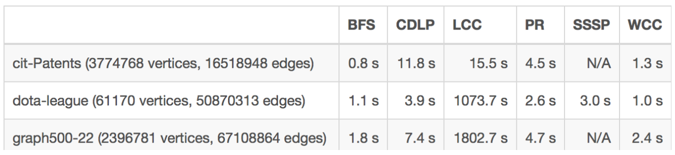


Fig. 6. Parallel efficiency for a scale-23 graph. Graph500 dips below 1 because it is slower for 2 threads than for 1. T_1 is the serial time, n is the number of threads, and T_n time with n threads.

Overall, GAP is the most scalable with GraphMat close behind for larger threads and even slightly beating GAP at 72 threads. While care was taken to ensure there was no other load on the system when the experiments were performed, the efficiency for two threads is surprisingly low for the Graph500. Because the Graph500 spends a shorter amount of time executing in general (there is no file I/O performs multiple BFS passes one right after another), it is more sensitive to spikes in CPU usage.

C. Real-World Datasets

In addition to the data presented in Table II, one may want to know the cost of performing additional computations on a graph once the data structures have been built. Figure 8 compiles average performance results for



	BFS	CDLP	LCC	PR	SSSP	WCC
cit-Patents (3774768 vertices, 16518948 edges)	0.8 s	11.8 s	15.5 s	4.5 s	N/A	1.3 s
dota-league (61170 vertices, 50870313 edges)	1.1 s	3.9 s	1073.7 s	2.6 s	3.0 s	1.0 s
graph500-22 (2396781 vertices, 67108864 edges)	1.8 s	7.4 s	1802.7 s	4.7 s	N/A	2.4 s

Fig. 7. Real-world and synthetic experiments using Graphalytics on GraphBIG using 32 threads. Graphalytics outputs one HTML page per software package.

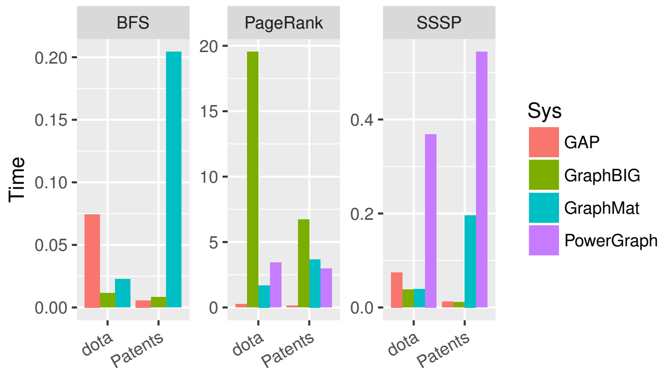


Fig. 8. Real world experiments using *easy-parallel-graph-∗*. The leftmost plot does is missing Powergraph because Powergraph does not provide BFS.

easy-parallel-graph-∗. We contrast this with Graphalytics which outputs an HTML page resulting from a single trial. We display a partial screenshot in Fig. 7.

Comparison of these two datasets in Fig. 8 brings up some surprising variations in the data. For example, PowerGraph is faster for SSSP. This could be because of the efficient edge-cut partitioning scheme in place on PowerGraph which can more efficiently deal with the high degree vertices present on the denser Dota-League graph. However, this comes with a significant overhead; PowerGraph is slower for SSSP than the other platforms. GraphBIG seems to have the widest variation; it is by far the slowest for PageRank but is the fastest for the Dota-League BFS, beating out even GAP which uses a more modern algorithm for BFS called Direction-optiming BFS [2]. One reason for this lack of performance from GAP is our lack of tuning; we use the default parameterization of $\alpha = 15$ and $\beta = 18$, which may not be optimal for all graphs. One possible explanation for the poor performance of GraphMat compared is the overhead of the sparse matrix operations. These may pay off for larger datasets and we see good performance on the denser Dota-League dataset for GraphMat across all algorithms.

D. Power and Energy Consumption

We use the Performance Application Programming Interface (PAPI) [4] to gain access to Intel's Running Average Power Limit (RAPL), which provides a set of hardware counters for measuring energy usage. We use PAPI to obtain 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 III. In our case, the fastest code is also the most energy efficient, although with this level of granularity we could detect circumstances where one could make a tradeoff between energy and runtime.

RAPL also allows the measurement of DRAM power, the results of which are shown in Fig. 9. The right plot describes the second row of Table III in more detail. We

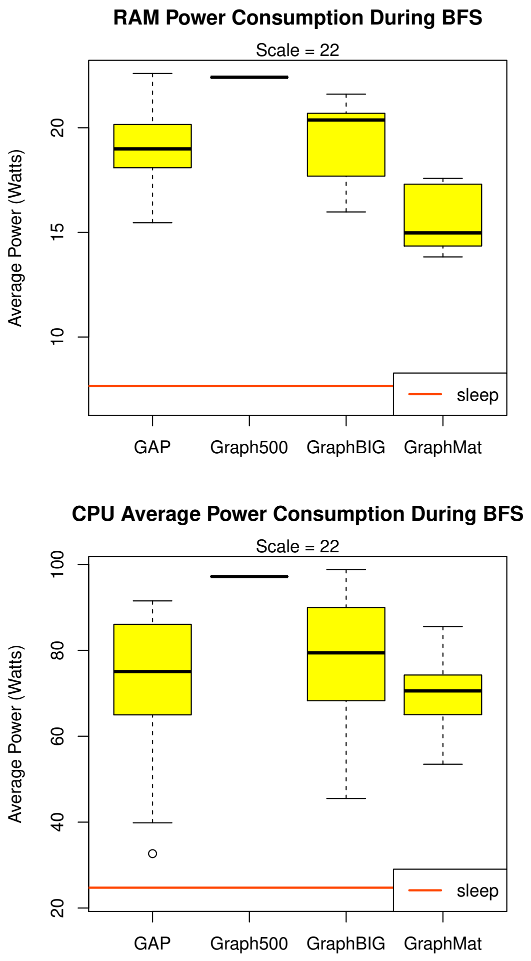


Fig. 9. Similar to Fig. 2 we plot RAM and CPU Power Consumption for each of the 32 roots. Since the Graph500 runs multiple roots per execution, we only get a single data point. The baseline was computed by monitoring power consumption while a program containing only one call to the C `unistd` library function `sleep(10)` (ten seconds).

notice a smaller spread of RAM power consumption, but still a noticeable difference. GraphMat exhibits the lowest average RAM power consumption (but not the shortest execution time). This information can be useful when choosing an algorithm to use in limited power scenarios, where a slower algorithms that will not exceed the power cap is preferred to a faster one that may exceed it.

Listing 10 shows the simplicity of adding RAPL profiling to existing code. Once the source code is modified, it is compiled with our provided header file and linked with a small library. Alternatively, there may be scenarios where this approach is not feasible. For example, in some cases, if the user does not have root privileges s/he may not be able to access the model specific registers storing the energy usage. One alternative is to use TAU paired with PAPI to instrument the source code, and if the counters are available on the host CPU then the power and energy can still be measured.

TABLE III

The data are generated using a Kronecker graph with a scale of 22. The program is executed with 32 threads. Sleeping Energy refers to the power (in Watts) consumed during the `unistd` C `sleep` function, multiplied by the Time row. Essentially, this measures the energy that would have been consumed when the CPU and memory are (nearly) idle. The increase over sleep is the ratio of the first and third columns. These are all averaged over 32 roots.

*easy-parallel-graph-**	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

```
#ifdef POWER_PROFILING
power_rapl_t ps;
power_rapl_init(&ps);
power_rapl_start(&ps);
#endif
        <region of code to profile>
#ifdef POWER_PROFILING
power_rapl_end(&ps);
power_rapl_print(&ps);
#endif
```

```
CFLAGS += -I$(PAPI)/include -DPOWER_PROFILING=1
LDLIBS += -L$(PAPI)/lib
LDLIBS += -Wl,-rpath,$(PAPI)/lib -lpapi
```

Fig. 10. The changes required to monitor energy usage for a C or C++ file and its corresponding Makefile, given the papi header file is included and $PAPI is the directory where PAPI is installed.

V. Future Work

Our choice of algorithms is based on the common subset of methods analyzed in prior work and implemented in the selected software packages. The standardization of graph algorithm building blocks (graph kernels) is being developed by the Graph BLAS Forum [19]). Once this standardization is finalized there is motivation from both library designers and performance analyzers to implement and profile each kernel. Regardless, algorithms like triangle counting and betweenness centrality are widely implemented but not supported by either Graphalytics nor *easy-parallel-graph-**.

Advances in parallel SSSP and BFS contain parameterizations (Δ for SSSP and α and β for BFS) which affects performance depending on graph structure. These are provided in GAP. We plan to add some level of heuristic parameter tuning as performed in [2] to the next iteration of our framework to take advantage of these algorithmic advances.

Graphalytics encountered circumstances with the more computationally expensive algorithms fail [12], so determining whether an algorithm will finish given a particular machine, input size, runtime limit, and resources is an important unanswered question we plan to pursue further.

Even though most software packages represent graphs using CSR format, the implementation details differ across packages. There may be significant performance differences among the various packages between using directed or undirected, or weighted and unweighted graphs.

One overarching issue with these software packages is the speed at which they change. On one end of the spectrum, the code base of PowerGraph was made closed-source in 2015. Simple fixes such as increasing the maximum number of threads per process and updating dependency URLs are not merged and as a result there is much duplicated effort among more than 400 forks of PowerGraph. On the other end, GraphMat's update to version 2.0 is in general incompatible with version 1.0 and as such has not yet been adopted by Graphalytics or our framework. One potential solution would be to add each framework to a package management system such as Spack [8] which supports packaging of high performance computing software.

With respect to power and energy profiling, while our current implementation supports measurements based on PAPI's interface to RAPL, which is only available on Intel platforms, the interface is simple and easy to adapt to other platforms in future without requiring PAPI support. In particular, fine-grained measurements provided through potentially available custom hardware [25] can be enabled through the same interface.

Ultimately, the automated aspect of *easy-parallel-graph-** allows us to generate a large amount of performance data which can in turn be fed into machine learning models. These models can then be used to predict performance given a new dataset. This involves feature selection, model selection, and training, but once this is complete this will allow a model to be used in lieu of expensive experimental runs.

VI. Conclusion

We have presented a tool called *easy-parallel-graph-**, where $*$ = installer, comparator, performance analyzer, dataset homogenizer, or performance visualizer. This framework can analyze both power consumption and the two fundamental phases of execution: data structure construction and algorithm execution. While the comparison presented here can help one choose among alternatives for the selected packages and algorithms, the problem of selecting a graph processing framework for a given large-scale problem remains far from simple. Furthermore, increasing hardware heterogeneity demands performance analysis be easily repeatable on the target architecture. Because of this, our framework is designed to be easy to use and we make our code freely available at https://github.com/HPCL/easy-parallel-graph to encourage further experimentation.

Overall, the GAP Benchmark Suite is the best-performing system across all chosen datasets and in general the most scalable for the graphs used in this study (at most 2^{23} vertices). In addition, GAP is the most recent project and also the most robust in terms of the input graphs accepted. Powergraph and Graphmat are designed for larger graphs (it is possible to run in distributed mode for both) so the overhead of these frameworks may dominate for smaller problem sizes. Two more potentially important considerations are cost and portability: GraphMat requires the Intel compiler collection which may be cost-prohibitive for some users. Despite the GAP Benchmark Suite being the overall best performing benchmark, it was not the best performing in every case so we unsurprisingly recommend the more comprehensive comparison using all five benchmarks offered by *easy-parallel-graph-∗*.

References

[1] "Graph algorithm building blocks workshop," in *30th IEEE International Parallel and Distributed Processing Symposium*, ser. GABB '16, D. A. Bader, A. Buluç, J. Gilbert, and J. Kepner, Eds., 2016.

[2] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 12:1–12:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389013

[3] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[4] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, Aug. 2000. [Online]. Available: http://dx.doi.org/10.1177/109434200001400303

[5] A. Buluç and J. R. Gilbert, "The combinatorial blas: design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011. [Online]. Available: http://dx.doi.org/10.1177/1094342011403516

[6] M. Capotă, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. Boncz, "Graphalytics: A big data benchmark for graph-processing platforms," in *Proceedings of the Graph Data Management Experiences and Systems*, ser. GRADES '15. New York, NY, USA: ACM, 2015, pp. 7:1–7:6. [Online]. Available: http://doi.acm.org/10.1145/2764947.2764954

[7] N. Doekemeijer and A. L. Varbanescu, "A survey of parallel graph processing frameworks," Delft University of Technology, Tech. Rep., 2014.

[8] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The spack package manager: Bringing order to hpc software chaos," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 40:1–40:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807623

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "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)*, ser. ODSI '12. Hollywood, CA: USENIX, 2012, pp. 17–30. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[10] D. Gregor, N. Edmonds, B. Barrett, and A. Lumsdaine, "The parallel boost graph library," http://www.osl.iu.edu/research/pbgl, 2005.

[11] Y. Guo and A. Iosup, "The game trace archive," in *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games*, ser. NetGames '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 4:1–4:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2501560.2501566

[12] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. P. Pérez, T. Manhardt, H. Chafi, M. Capota, N. Sundaram, M. Anderson, I. G. Tanase, Y. Xia, L. Nai, and P. Boncz, "Ldbc graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms, a technical report," Delft University of Technology, Tech. Rep., 2016.

[13] U. Kang, B. Meeder, and C. Faloutsos, "Spectral analysis for billion-scale graphs: Discoveries and implementation," in *Advances in Knowledge Discovery and Data Mining*, ser. Lecture Notes in Computer Science, vol. 6635. Berlin: Springer, 2011.

[14] M. LeBeane, S. Song, R. Panda, J. H. Ryoo, and L. K. John, "Data partitioning strategies for graph workloads on heterogeneous clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 56:1–56:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807632

[15] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *International Conference on Knowledge Discovery and Data Mining*, ser. SIGKDD. ACM, 2005.

[16] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, "Kronecker graphs: An approach to modeling networks," *J. Mach. Learn. Res.*, vol. 11, pp. 985–1042, Mar. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756006.1756039

[17] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[18] Y. Lu, J. Cheng, D. Yan, and H. Wu, "Large-scale distributed graph computing systems: An experimental evaluation," *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 281–292, Nov. 2014. [Online]. Available: http://dx.doi.org/10.14778/2735508.2735517

[19] T. Mattson, D. Bader, J. Berry, A. Buluç, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo, "Standards for graph algorithm primitives," in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2013, pp. 1–2.

[20] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Arg, "Introducing the graph500," Cray User's Group, Tech. Rep., 2010.

[21] L. Nai, Y. Xia, I. G. Tanase, and H. Kim, "Exploring big graph computing — an empirical study from architectural perspective," *Journal of Parallel and Distributed Computing*, August 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731516300910

[22] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, "GraphBIG: Understanding graph computing in the context of industrial solutions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 69:1–69:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807626

[23] W. L. Ngai, "Fine-grained performance evaluation of large-scale graph processing systems," Master's thesis, Delft University of Technology, 2015.

[24] G. A. Pavlopoulos, M. Secrier, C. N. Moschopoulos, T. G. Soldatos, S. Kossida, J. Aerts, R. Schneider, and P. G. Bagos, "Using graph theory to analyze biological networks," in *BioData Mining*. Bethesda, MD: PubMed Central, 2011.

[25] M. Rashti, G. Sabin, and B. Norris, "Power and energy analysis and modeling of high performance computing systems using WattProf," in *Proceedings of the 2015 IEEE National Aerospace and Electronics Conference (NAECON)*, July 2015.

[26] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 979–990. [Online]. Available: http://doi.acm.org/10.1145/2588555.2610518

[27] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *SIGPLAN Not.*, vol. 48, no. 8, pp. 135–146, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2517327.2442530

[28] S. Song, M. Li, X. Zheng, M. LeBeane, J. H. Ryoo, R. Panda, A. Gerstlauer, and L. K. John, "Proxy-guided load balancing of graph processing workloads on heterogeneous clusters," in *2016 45th International Conference on Parallel Processing (ICPP)*, Aug 2016, pp. 77–86.

[29] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *The Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1214–1225, jul 2015. [Online]. Available: http://dx.doi.org/10.14778/2809974.2809983

[30] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on spark," in *First International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '13. New York, NY, USA: ACM, 2013, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/2484425.2484427

[31] J. Zhong and B. He, "Medusa: Simplified graph processing on gpus," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1543–1552, Jun. 2014. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2013.111