The Graph Based Benchmark Suite (GBBS)

Laxman Dhulipala
Carnegie Mellon University
ldhulipa@cs.cmu.edu

Jessica Shi
MIT CSAIL
jeshi@mit.edu

Tom Tseng
MIT CSAIL
tomtseng@csail.mit.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Julian Shun
MIT CSAIL
jshun@mit.edu

Abstract

In this demonstration paper, we present the Graph Based Benchmark Suite (GBBS), a suite of scalable, provably-efficient implementations of over 20 fundamental graph problems for shared-memory multicore machines. Our results are obtained using a graph processing interface written in C++, extending the Ligra interface with additional functional primitives that have clearly defined cost bounds. Our approach enables writing high-level codes that are simultaneously simple and high-performance by virtue of using highly-optimized primitives. Another benefit is that optimizations, such as graph compression, are implemented transparently to high-level user code, and can thus be utilized without changing the implementation. Our approach enables our codes to scale to the largest publicly-available real-world graph containing over 200 billion edges on a single multicore machine.

We show how to use GBBS to process and perform a variety of tasks on real-world graphs. We present the high-level C++ APIs that enable us to write concise, high-performance implementations. We also introduce a Python interface to GBBS, which lets users easily prototype algorithms and pipelines in Python that significantly outperform NetworkX, a mature Python-based graph processing solution.

ACM Reference Format:
Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E. Blelloch, and Julian Shun. 2020. The Graph Based Benchmark Suite (GBBS). In *3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'20), June 14, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3398682.3399168

GRADES-NDA'20, June 14, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-8021-8/20/06.
https://doi.org/10.1145/3398682.3399168

1 Introduction

Programming algorithms that can process massive graphs with billions to hundreds of billions of edges is a challenging task. To simplify the task, we have designed a problem-based benchmark and corresponding C++ library called Graph Based Benchmark Suite (GBBS) to make it easier to design *provably-efficient* and *scalable* shared-memory parallel graph algorithms [15, 16]. GBBS began as a project to benchmark parallel graph algorithms, but over time has evolved into a useful library for designing and implementing new highly performant parallel graph algorithms. In this short paper, we present an overview of the C++ library underlying GBBS, including the core techniques, system design, and APIs that enable us to achieve our results and enable the design of simple, efficient implementations. We have made GBBS publicly-available at https://github.com/ParAlg/gbbs, and provide a website documenting the benchmark at https://paralg.github.io/gbbs/. We hope our approach will be applicable to other algorithmic and data mining tasks on graphs in the future.

Graph Based Benchmark Suite (GBBS). In GBBS, we provide a high-level graph processing interface in C++ that extends the Ligra, Ligra+, and Julienne frameworks [14, 42, 45] with additional *functional primitives* that are *parallel by default*. We have found our approach to be broadly applicable as we have designed and implemented simple, fast, and provably-efficient multicore implementations of over 20 benchmark graph problems, ranging from standard graph kernels such as breadth-first search and connectivity, to more challenging problems such as biconnectivity, minimum spanning forest, k-clique enumeration, and strongly connected components. Importantly, the GBBS benchmarks provide clear input-output specifications enabling others to easily compare other algorithms and implementations with our results. Other benchmarks suites, such as the GAP [7], PBBS [44], and LDBC Graphalytics [24] benchmarks have inspired our work, but we extend the approach taken by these benchmarks to a much broader set of graph problems. We believe that the benchmark suite implemented as part of GBBS is one of the broadest set of high-performance graph algorithm implementations available today.

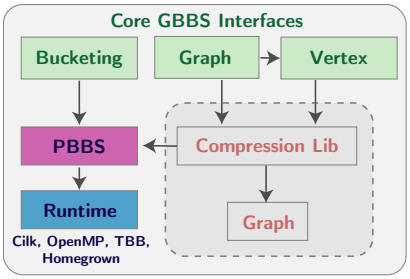

Figure 1: System architecture of GBBS. The core interfaces are the *bucketing*, *graph*, and *vertex* interfaces. These interfaces utilize parallel primitives and routines from PBBS. Parallelism is implemented using a parallel runtime system—Cilk Plus, OpenMP, TBB, or a homegrown scheduler that we wrote ourselves—and can be swapped using a command line argument. The vertex and graph interfaces use a compression library that mediates access to the underlying graph, which can either be compressed or uncompressed.

In this paper, we focus on describing our techniques, design principles, and APIs that enable short, reusable, and composable codes that simplify programming a wide range of fundamental graph problems. Importantly, all of our codes scale to the largest publicly-available real-world graph, the Hyperlink Web graph [33], with over 3.5 billion vertices and 128 billion edges (over 200 billion edges once symmetrized), on a commodity multicore machine. GBBS is an active bed for our ongoing research, and we hope to attract more users to use our system for both benchmarking existing graph algorithms as well as implementing new ones.

Contributions. We make the following contributions:

(1) We describe GBBS, including the core APIs and the overall organization and design of the library.
(2) We describe how to implement an algorithm in GBBS and provide an example using CoSimRank, an important data mining algorithm.
(3) We provide a new Python-based API for GBBS and show that end-to-end processing of graphs using our API is orders of magnitude faster than NetworkX, a mature existing Python-based graph processing solution.

2 GBBS Design

GBBS is built as a number of layers, which we illustrate in Figure 1. We provide a detailed description of the library, our APIs and corresponding cost bounds, as well as the GBBS benchmarks, on our website (see Appendix F).

Parallel Runtime and Cost Model

GBBS uses a shared-memory approach to parallel graph processing in which the entire graph is stored in the main memory of a single multicore machine. Our codes exploit nested parallelism using scheduler-agnostic parallel primitives, such as ***fork-join*** and parallel-for loops. Thus, they can easily be compiled to use different parallel runtimes such as Cilk Plus, OpenMP, TBB, and also a custom work-stealing scheduler implemented by the authors. We analyze GBBS algorithms in the classic *work-depth model* for shared-memory algorithms, where the *work* is the number of operations used by the algorithm and the *depth* is the length of the longest sequential dependence in the computation [13, 25].

Parallel Datatypes and Primitives (PBBS)

We build on PBBS [44], a robust base layer providing parallel primitives and utilities, upon which we build the higher-level graph and vertex interfaces. PBBS provides the following utilities. A *sequence* is a generic parallel sequence datatype, similar to a parallel version of a C++ vector that provides parallel initialization and destruction. GBBS also uses generic implementations of a parallel linear-probing hash table [43]. Lastly, we import parallel primitives over sequences, including map, reduce, prefix-sum (scan), filter, pack, histogram, random shuffle, and a set of efficient sorting algorithm.

Graph Representations

Compressed Graphs. Graphs in GBBS are stored in the *compressed sparse row (CSR)* format. CSR stores two arrays, I and A, where the vertices are in the range $[0, n-1]$ and incident edges of a vertex v are stored in $\{A[I[v]], \ldots, A[I[v+1]-1]\}$ (with a special case for vertex $n-1$). The *uncompressed* format in GBBS is equivalent to the CSR format. GBBS also supports several *compressed* graph formats from the Ligra+ framework [45]. Specifically, we provide support for graphs where neighbor lists are encoded using byte codes and a parallel generalization of byte codes (see Appendix B).

Weighted Graphs. The graph and vertex datatypes used in GBBS are generic over the weight type of the graph. Graphs with arbitrary edge weights can be represented by simply changing a template argument to the vertex and graph datatypes. We describe how edge weights integrate with compression in Appendix B. We treat unweighted graphs as graphs weighted by an implicit null (0-byte) weight.

Vertex and Graph Datatypes

Next, we describe the core vertex and graph interfaces which mediate algorithms' and high-level routines' accesses to the underlying graph representation (which can either be compressed or uncompressed, and weighted or unweighted).

Vertex Datatypes and Primitives. GBBS provides vertex datatypes for both symmetric and asymmetric vertices, used for undirected and directed graphs, respectively. The vertex datatype interface (see Figure 2) provides functional primitives over vertex neighborhoods, such as MAP, REDUCE, SCAN, COUNT (a special case of reduce where the map function is a boolean function), as well as primitives to extract a subset of the neighborhood satisfying a predicate (FILTER) and a primitive to mutate the vertex neighborhood and delete edges that do not satisfy a given predicate (PACK). The interface also provides functions for computing the INTERSECTION, UNION,

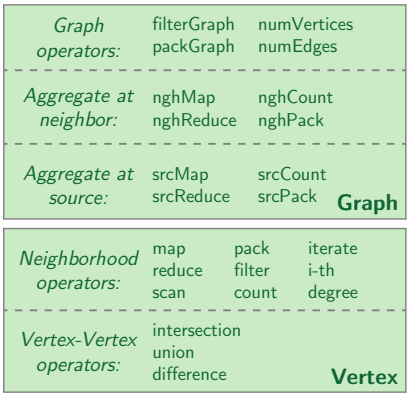

Figure 2: Core GBBS interfaces. We provide descriptions in the text.

or DIFFERENCE between the set of neighbors of two vertices. Due to space constraints, we provide the full interface on our website (see Appendix F).

Vertex Subsets. We use the vertexSubset datatype from Ligra, which represents a subset of vertices in the graph. A subset can either be *sparse* (represented as a collection of vertex IDs) or *dense* (represented as a boolean array or bit-vector of length n, the number of vertices in the graph). A $\text{vertexSubset}_{\mathsf{T}}$ is a generic vertexSubset, where each vertex is augmented with a value of some type T.

Bucketing. We use the bucketing interface from Julienne [14], which enables priority-based graph algorithms, including integer-weighted shortest paths, Δ-stepping for shortest paths, k-core decomposition, and others. Each bucket is represented as a vertexSubset, and the interface allows vertices to dynamically be moved through different buckets as priorities change. Algorithms using the interface iteratively extract the highest priority bucket, potentially update incident vertex priorities, and repeat until all buckets are empty.

Graph Datatypes and Primitives. GBBS provides graph datatypes for both symmetric and asymmetric graphs. The distinction is important for statically enforcing arguments to problems and routines that require a symmetric input (e.g., it does not make sense to call connectivity or biconnectivity on a directed input). Aside from standard functions to query the number of vertices and edges, the core graph interface is the set of functional operators defined on graphs, which extend and generalize the EDGEMAP primitive provided by Ligra, which we review for completeness in Appendix C.

Generalizing EDGEMAP. In GBBS, we generalize the EDGEMAP primitive (Appendix C) in two ways. First, we observe that EDGEMAP is a function from a vertexSubset to a vertexSubset containing *neighbors* of the input vertexSubset, and that it is often useful to apply a functional operator over a vertexSubset and return the results for the *same* vertexSubset. Second, we observe that we can generalize applying the map operation to perform reductions, counts, and packs using the same interface.

Graph Dataset	Num. Vertices	Num. Edges
com-Orkut	3,072,627	234,370,166
Hyperlink2012	3,563,602,789	128,736,914,167
Hyperlink2012-Sym	3,563,602,789	225,840,663,232

Table 1: Graph inputs, including vertices and edges.

Based on these observations, we provide versions of the EDGEMAP primitive that aggregate the results at the source: SRCMAP, SRCREDUCE, SRCCOUNT, and SRCPACK. We also provide generalizations of EDGEMAP that return a subset of the neighbors of the input vertexSubset, including NGHMAP (equivalent to EDGEMAP), NGHREDUCE, NGHCOUNT, and NGHPACK. We provide additional details about the generalized primitives in Appendix C.

Finally, we provide an operator for filtering edges out of a graph that returns a *new* graph, called FILTERGRAPH. The primitive is useful for codes such as triangle counting and k-clique enumeration, which require directing the edges of an undirected graph to eliminate redundant work. We provide a similar primitive which operates in-place called PACKGRAPH.

Python Interface

We have implemented a Python-based interface for GBBS that makes it easy for users to utilize our benchmark implementations and data structures. The library is implemented using pybind11 [26], which provides zero-copy interoperability between C++ and Python. We provide functionality to load graphs from a variety of formats and sources, including datasets from SNAP [29] and LAW [8], as well as the uncompressed and compressed formats in GBBS.

3 Demonstration Walkthrough

In this section, we demonstrate how to use the GBBS graph and vertex APIs, set up and implement CoSimRank, a new benchmark using GBBS, and demonstrate how to use GBBS to solve problems using a new zero-copy Python interface that we have implemented.

Using the Graph and Vertex APIs

The C++ graph and vertex APIs can be used as follows:

```
using sym_vertex_int = symmetric_vertex<int>;
using sym_graph_int = symmetric_graph<symmetric_vertex,
↪  int>;
```

After loading an integer-weighted symmetric graph G (please see our website for how to load graphs of different types from C++), we can now call various graph methods, and access a vertex object for the i'th vertex as follows:

```
size_t n = G.numVertices();
size_t m = G.numEdges();
sym_vertex_int vtx10 = G.get_vertex(10);
```

Similarly, methods on the vertex can be called as follows:

```
// Intersect vtx10 and vtx42 and return the size
sym_vertex_int vtx42 = G.get_vertex(42);
size_t intersection_size = vtx10.intersect(vtx42);
// Compute number of heavy edges incident to vtx10
auto pred = [](vtxid_t u, vtxid_t v, int wgh) {
```

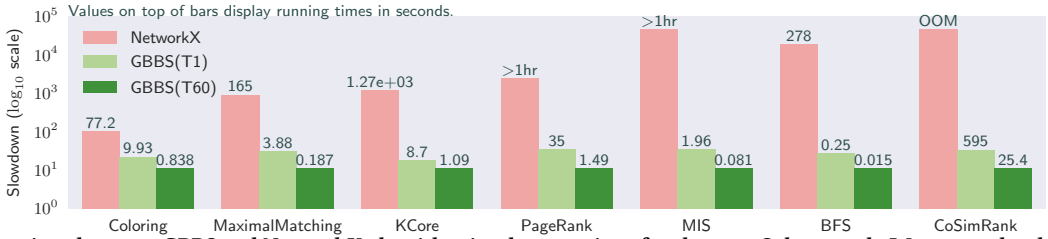

Figure 3: Comparison between GBBS and NetworkX algorithm implementations for the com-Orkut graph. We report the slowdown relative to GBBS(T60), which is GBBS on 60 threads. The experiments are run on a c2-standard-60 Google Cloud instance, which consists of 60 cores (with two-way hyper-threading), with 3.8GHz Intel Xeon scalable (Cascade Lake) processors and 240 GiB of main memory.

```
  return wgh > 5;};
size_t num_heavy = vtx10.countOutNgh(pred);
```

Here, `vtxid_t` is the numeric type for vertex IDs.

Benchmark Implementation: CoSimRank

CoSimRank [38] is a local version of SimRank [27] that allows the similarity of a pair of vertices to be computed without computing the similarity of all pairs of vertices in the graph. Its computation involves a simplified Personalized PageRank computation [10], without the use of a damping factor. We provide the pseudocode and description for our implementation using GBBS in Algorithm 1 in Appendix D.

Using the Python Interface

Next, we illustrate how to use GBBS in an end-to-end fashion to rapidly import a graph from the SNAP benchmark and run the CoSimRank algorithm on it. Extending the Python bindings after implementing a new benchmark requires only a few lines of code to add an extra method to the graph object exported by the library.

We first build the bindings using Bazel [6] and add the compiled libraries to the Python path:

```
> bazel build //pybindings/...
> export PYTHONPATH=$(pwd)/bazel-bin/pybindings/:$PYTHONPATH
```

Next, we launch the Python REPL, import the library, and import a downloaded graph from the SNAP dataset [29].

```
>>> import gbbs
>>> G = gbbs.loadSNAP("com-youtube.ungraph.txt",
↪  undirected=True)
```

This command creates an uncompressed graph in the GBBS format at the same location as the input (compression can optionally be enabled using a separate flag). We can then apply the CoSimRank method defined on graphs:

```
>>> sim = G.CoSimRank(src=10, dest=82)
>>> print(sim)
0.0002881
```

Other primitives can be applied similarly. For example:

```
>>> components = G.Connectivity()
>>> print(components[10] == components[82])
True
>>> cores = G.KCore() # Computes coreness values
>>> print(cores[10], cores[82])
(41, 50)
```

Comparison with NetworkX

We compared the performance of our implementations with that of NetworkX [23]. We ran our experiments on a 60-core, 2-way hyper-threaded c2-standard-60 Google Cloud instance, with 3.8GHz Intel Xeon Scalable (Cascade Lake) processors and 240 GiB of memory.

Figure 3 shows the results of the comparison for the com-Orkut graph from SNAP. For PageRank and maximal independent set (MIS), the NetworkX implementation did not finish after 1 hour. For CoSimRank, the NetworkX library calls an all-pairs implementation of SimRank [27], which runs out of memory as it materializes an $n \times n$ matrix. We find that even for this small input which has about 3 million vertices and 234 million edges (see Table 1), GBBS is significantly faster than NetworkX even for GBBS running on a single thread. In particular, we demonstrate significant speedups of 7.77x to over 1836.73x running our benchmarks on a single thread, and of 92.12x to over 4444.44x running our benchmark on 60 threads, as shown in Figure 3.

Although this comparison is not apples-to-apples, since our implementations are run in parallel using a highly-optimized C++ library, and NetworkX is implemented in Python, we believe that our approach and our Python bindings make high-performance algorithm implementations more accessible to the broad Python community.

4 Conclusion and Future Work

We have presented the Graph Based Benchmark Suite (GBBS), a benchmark suite of over 20 fundamental graph problems, and an overview of the techniques and interfaces enabling our implementations. In future work, we intend to implement our interface on a recent system for streaming graphs called Aspen [17] in a way that enables all GBBS codes to work without modifications over the Aspen graph representations. We encourage others to use GBBS for both benchmarking existing graph algorithms as well as implementing new ones.

Acknowledgements. This research was supported by DOE Early Career Award #DE-SC0018947, NSF Graduate Research Fellowship #1122374, NSF CAREER Award #CCF-1845763, NSF grants CCF-1910030 and CCF-1919223, and Google Faculty Research Award.

References

[1] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. CRONO: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*, IISWC '15, Washington, DC, USA, 2015. IEEE Computer Society.

[2] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: A database benchmark based on the facebook social graph. In *ACM SIGMOD International Conference on Management of Data*, pages 1185–1196, 2013.

[3] D. A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, and E. Loh. HPC scalable graph analysis benchmark.

[4] D. A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *International Conference on High-Performance Computing (HiPC)*, pages 465–476, 2005.

[5] G. Bagan, A. Bonifati, R. Ciucanu, G. H. Fletcher, A. Lemay, and N. Advokaat. gMark: Schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, 29(4):856–869, 2016.

[6] Bazel. https://bazel.build/.

[7] S. Beamer, K. Asanovic, and D. A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[8] P. Boldi and S. Vigna. The Webgraph framework I: compression techniques. In *International World Wide Web Conference (WWW)*, pages 595–602, 2004.

[9] A. Bonifati, G. Fletcher, J. Hidders, and A. Iosup. A survey of benchmarks for graph-processing systems. In *Graph Data Management*, pages 163–186. Springer, 2018.

[10] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Computer Networks and ISDN Systems*, pages 107–117, 1998.

[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference (ATC)*, pages 49–60, 2013.

[12] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, Feb. 1985.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[14] L. Dhulipala, G. Blelloch, and J. Shun. Julienne: A framework for parallel graph algorithms using work-efficient bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 293–304, 2017.

[15] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 393–404, 2018.

[16] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. *CoRR*, abs/1805.05208, 2018.

[17] L. Dhulipala, G. E. Blelloch, and J. Shun. Low-latency graph streaming using compressed purely-functional trees. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 918–934, 2019.

[18] L. Dhulipala, C. McGuffey, H. Kang, Y. Gu, G. E. Blelloch, P. B. Gibbons, and J. Shun. Sage: Parallel semi-asymmetric graph algorithms for NVRAMs. *Proceedings of the VLDB Endowment*, 13(9), 2020.

[19] A. Dubey, G. D. Hill, R. Escriva, and E. G. Sirer. Weaver: a high-performance, transactional graph database based on refinable timestamps. *Proceedings of the VLDB Endowment*, 9(11):852–863, 2016.

[20] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC social network benchmark: Interactive workload. In *ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 2015.

[21] L. Gao, L. Golab, M. T. Özsu, and G. Aluç. Stream WatDiv: A streaming RDF benchmark. In *Proceedings of the International Workshop on Semantic Big Data*, pages 1–6, 2018.

[22] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 17–30, 2012.

[23] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[24] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardto, H. Chafio, M. Capotă, N. Sundaram, M. Anderson, I. G. Tănase, Y. Xia, L. Nai, and P. Boncz. LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proceedings of the VLDB Endowment*, 9(13):1317–1328, Sept. 2016.

[25] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[26] W. Jakob, J. Rhinelander, and D. Moldovan. pybind11 – seamless operability between C++11 and Python, 2017. https://github.com/pybind/pybind11.

[27] G. Jeh and J. Widom. SimRank: a measure of structural-context similarity. In *ACM SIGKDD International conference on Knowledge Discovery and Data Mining*, pages 538–543, 2002.

[28] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica. ZipG: A memory-efficient graph store for interactive queries. In *ACM SIGMOD International Conference on Management of Data*, pages 1149–1164, 2017.

[29] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2019.

[30] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new framework for parallel machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 340–349, 2010.

[31] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of Data*, pages 135–146, 2010.

[32] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, Oct. 2015.

[33] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer. The graph structure in the web–analyzed on different aggregation levels. *The Journal of Web Science*, 1(1), 2015.

[34] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin. GraphBIG: understanding graph computing in the context of industrial solutions. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[35] Neo4j. http://neo4j.com.

[36] D. Nguyen, A. Lenharth, and K. Pingali. A lightweight infrastructure for graph analytics. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 456–471, 2013.

[37] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *USENIX Conference on Annual Technical Conference (ATC)*, pages 41–52, 2012.

[38] S. Rothe and H. Schütze. CoSimRank: A flexible & efficient graph-theoretic similarity measure. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1392–1402, 2014.

[39] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *ACM SIGMOD International Conference on Management of Data*, pages 505–516, 2013.
[40] J. Shi, L. Dhulipala, and J. Shun. Parallel clique counting and peeling algorithms. *arXiv preprint arXiv:2002.10047*, 2020.
[41] J. Shun. Practical parallel hypergraph algorithms. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 232–249, 2020.
[42] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 135–146, 2013.
[43] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 96–107, 2014.
[44] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 68–70, 2012.
[45] J. Shun, L. Dhulipala, and G. E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+. In *IEEE Data Compression Conference (DCC)*, pages 403–412, 2015.
[46] J. Shun and K. Tangwongsan. Multicore triangle computations without tuning. In *IEEE International Conference on Data Engineering (ICDE)*, pages 149–160, 2015.
[47] Z. Xu, X. Chen, J. Shen, Y. Zhang, C. Chen, and C. Yang. Gardenia: A graph processing benchmark suite for next-generation accelerators. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 15(1):1–13, 2019.
[48] D. Yan, Y. Bu, Y. Tian, and A. Deshpande. Big graph analytics platforms. *Foundations and Trends in Databases*, 7(1-2):1–195, 2017.

A Related Work

Graph Processing Frameworks and Systems. There has been a wealth of work on designing efficient parallel graph frameworks and systems over the past two decades ([22, 30, 31, 36, 42] among many others). We refer the reader to [32, 48] for excellent surveys of this growing literature.

The approach used in the GBBS library crucially depends on the line of work on Ligra [42], and subsequent systems, including Ligra+ [45], and Julienne [14]. Recently, the line of work on Ligra was generalized for NVRAM-based systems [18], and to support hypergraphs [41]. An interesting question is whether the extended EDGEMAP primitives used in GBBS make it easier to implement a broad class of parallel hypergraph algorithms.

Parallel Graph Algorithm Benchmarks. Many parallel graph algorithm benchmarks have been proposed. A recent survey by Bonifati et al. [9] provides a good overview of many existing benchmarks. SSCA [3, 4] is an early benchmark specifying four graph kernels including graph generation, subgraph extraction, and clustering. The Problem Based Benchmark Suite (PBBS) [44] is a more general parallel algorithm benchmark that includes six problems on graphs (BFS, spanning forest, minimum spanning forest, MIS, maximal matching, and graph separators). The PBBS benchmarks are problem-based in that they are defined only in terms of the input and output without specifying the algorithm used to solve the problem. We follow the style of PBBS in GBBS of defining the input and output requirements for each problem. The LDBC Graphalytics benchmark [24] includes 6 algorithms including BFS, PageRank, connected components, label propagation, local clustering coefficient, and SSSP. The Graph Algorithm Platform (GAP) Benchmark [7] specifies six kernels for BFS, SSSP, PageRank, connectivity, betweenness centrality, and triangle counting. GBBS implements a superset of the GAP benchmarks, and supports a much broader set of problems than both the LDBC and GAP benchmarks.

Several recent benchmarks focus on the architecural properties of parallel graph algorithms. CRONO [1] implements 10 graph algorithms, including all-pairs shortest paths, exact betweenness centrality, traveling salesman, and depth-first search, and performs an architectural analysis of their implementations. GraphBIG [34] describes 12 algorithms, including several problems that we consider, like k-core and graph coloring (using the Jones-Plassmann algorithm), but also problems like depth-first search, which are difficult to parallelize. GARDENIA [47] provides a benchmark with 9 algorithms, including connectivity, BFS, betweenness centrality, PageRank, and triangle counting. Compared with these architectural benchmarks, GBBS implements a much broader set of graph problems. It would be interesting to study our implementations from an architectural perspective.

Graph Databases and Streaming Systems. A related line of research has been on graph databases (e.g., [11, 19, 28, 35, 37, 39]). Graph databases support dynamically updating the graph, usually through transactions (i.e., multi-writer concurrency) and are thus more general than the Ligra system and its descendants. However due to the overheads of supporting transactions, they are generally slower for static graph algorithms, which is what GBBS targets. Aspen [17] is a recent system supporting dynamic graph updates that also supports a Ligra-like interface. An interesting question that we plan to investigate is whether implementing the core interfaces from GBBS on top of Aspen would enable the growing number of GBBS implementations to automatically run on both static and dynamically evolving graphs.

Graph Database Benchmarks. Recently, there has been interest in designing benchmarks for graph databases [2, 5, 20, 21]. The LinkBench benchmark [2] generates a synthetic graph database, update stream, and operations that simulate the workload observed at Facebook. The LDBC Social Network Benchmark [20] presents a broad benchmark with many graph-based queries, and supports generating synthetic networks at various scales. The gMark benchmark [5] is a domain- and query-language-independent benchmark for generating graphs and query workloads. Stream WatDiv [21] is a streaming RDF benchmark for benchmarking

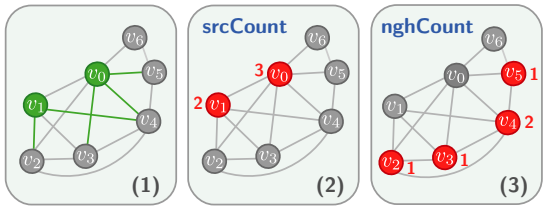

Figure 4: Illustration of srcCount and nghCount primitives. The input is illustrated in Panel (1), and consists of a graph and a vertexSubset, with vertices in the vertexSubset illustrated in green. The green edges are edges for which the condition function C returns true**. Panel (2) and Panel (3) show the results of applying srcCount and nghCount, respectively. Both primitives emit an augmented vertexSubset$_{\text{int}}$, illustrated in red, where each vertex has an associated count of the number of edges satisfying C.**

SPARQL-based streaming systems, which are closely related to graph databases. Compared with these efforts, the main difference in GBBS is that we focus on static graphs, and focus on problems from the parallel graph algorithms literature.

B Compression

Byte Codes. In byte codes, we store a vertex's neighbor list by difference encoding consecutive vertices, with the first vertex difference encoded with respect to the source. Decoding is done by sequentially uncompressing each difference, and summing the differences into a running sum which gives the ID of the next neighbor. As this process is sequential, graph algorithms using the byte format that map over the neighbors of a vertex will have poor depth bounds.

Parallel Byte Codes. We enable parallelism using the parallel-byte format from Ligra+. This format breaks the neighbors of a high-degree vertex into blocks, where each block contains a constant number of neighbors. Each block is difference encoded with respect to the source, and the format stores the blocks in a neighbor list in sorted order. As each block can have a different size, it also stores offsets that point to the start of each block. Using the parallel-byte format, the neighbors of a high-degree vertex can then be done in parallel over the blocks. We refer the reader to Ligra+ [45] for a detailed discussion of this idea.

Integrating Edge Weights. Both schemes above provide support for compressing weighted graphs. If the graph weight type is E, the encoder simply interleaves the weighted elements of type E with the differences generated by the byte or parallel byte code. GBBS supports compressing integer weights using variable-length coding, similar to Ligra+ [45].

C edgeMap and Generalizing edgeMap

edgeMap. Next, we review the edgeMap primitive from Ligra, which is the basis for the generalized interface used in GBBS. edgeMap is a basic graph processing primitive useful for performing graph traversal. The edgeMap primitive takes as input a frontier, or subset of seed vertices. It then applies a user-defined function to generate a new frontier consisting of neighbors of the input frontier. For example, in a breadth-first search, the user-defined primitive emits a neighbor in the output frontier if it has not yet been visited.

More formally, given a graph $G(V, E)$, edgeMap takes as input a vertexSubset U, and two boolean functions F (the *map* function) and C (the *cond* or condition function). edgeMap applies F to $(u, v) \in E$ such that $u \in U$ and $C(v) = true$ (call this subset of edges E_a), and returns a vertexSubset U', where $u \in U'$ if and only if $(u, v) \in E_a$ and $F(u, v) = true$.

Generalizing edgeMap. Here we provide some additional details about our generalizations of the edgeMap primitive. The interface for these primitives is similar to edgeMap, but the return types differ depending on the functional operation:

- The map operator (like edgeMap) returns a vertexSubset.
- The reduce operator returns an vertexSubset$_{\text{E}}$, where E is the result type of the reduction operation.
- The count operator is a specialization of reduce, returning a vertexSubset$_{\text{int}}$, where each vertex is augmented with the number of incident edges satisfying the condition function (defined in the description of edgeMap above).
- The pack operator preserves edges satisfying an input predicate P and deletes edges that do not satisfy P. It returns a vertexSubset$_{\text{int}}$ containing the new vertex degrees of affected vertices (either the sources or the neighbors).

Figure 4 illustrates the two count operator variants used in GBBS, srcCount and nghCount. Both primitives are generalizations of the edgeMap primitive.

D CoSimRank Pseudocode

Algorithm 1 Parallel CoSimRank

```
1:  procedure CoSimRank(G, u, v, c = 0.85, max_iter= 100, ε = 1e-6)
2:      cur_u ← e_u, cur_v ← e_v                ▷ e_w is the standard basis vector
3:      C: w → true                             ▷ All neighbors are valid for nghReduce
4:      S_u ← {u}, S_v ← {v}                    ▷ Initial frontier is u and v
5:      map_u: (src, ngh) → cur_u[src]/deg(src) ▷ Contributions for u,v
6:      map_v: (src, ngh) → cur_v[src]/deg(src)
7:      reduce: (ℓ, r) → ℓ + r
8:      i ← 0
9:      sim ← cur_u · cur_v
10:     while i < max_iter do
11:         S_u ←nghReduce(S_u, map_u, C, reduce)
12:         S_v ←nghReduce(S_v, map_v, C, reduce)
13:         ∀i, nxt_u[i] ← S_u[i].value
14:         ∀i, nxt_v[i] ← S_v[i].value
15:         sim ← sim + c^i · (nxt_u · nxt_v)   ▷ Compute similarity
16:         if ‖nxt_u − cur_u‖_1 < ε and ‖nxt_v − cur_v‖_1 < ε then break
17:         Swap cur_u and nxt_u
18:         Swap cur_v and nxt_v
19:     return sim
```

For a pair of vertices u and v, CoSimRank starts with the standard basis vectors e_u and e_v respectively (Line 2), and iteratively applies PageRank using nghReduce for u and v

Problem	Work	Running Time (s)
Breadth-First Search	$O(m)$	8.44
Weighted Breadth-First Search	$O(m)^*$	58.1
Bellman-Ford	$O(d(G)m)$	59.4
Single-Source Widest Path	$O(d(G)m)$	48.4
Single-Source Betweenness	$O(m)$	37.1
$O(k)$-Spanner	$O(m)^*$	36.5
Low-Diameter Decomposition	$O(m)^*$	16.6
Connectivity	$O(m)^*$	25.0
Spanning Forest	$O(m)^*$	35.8
Biconnectivity	$O(m)^*$	165
Strongly Connected Components	$O(m \log m)^*$	185
Minimum Spanning Forest	$O(m)^*$	187
Maximal Independent Set	$O(m)^*$	32.2
Maximal Matching	$O(m)^*$	108
Graph Coloring	$O(m)^*$	158
Approximate Set Cover	$O(m)^*$	90.4
Triangle Counting	$O(m^{3/2})$	1168
4-Clique Counting	$O(m\alpha(G)^2)$	$1.62 \cdot 10^5$
k-core	$O(m)^*$	184
Approximate Densest Subgraph	$O(m)$	51.4
PageRank Iteration	$O(m)$	13.1

Table 2: Work bounds for GBBS implementations, and parallel running times in seconds on the Hyperlink2012 web graph. All benchmarks other than strongly connected components are run on the undirected version of the graph. * denotes that a bound holds in expectation. m is denotes the number of edges in the graph, and we assume that $m = \Omega(n)$, where n is the number of vertices. $d(G)$ is the diameter of the graph, and $\alpha(G)$ is the arboricity of the graph (the minimum number of spanning forests needed to cover the graph). The depth bounds for most implementations is poly-logarithmic in m, and we defer a full list of our depth bounds to the GBBS website.

(Lines 11–12). Specifically, starting with an initial singleton frontier for each of u and v, for each traversed edge nghReduce computes the PageRank contribution from each source (Lines 5–6), and reduces the sum to the neighbors (Line 7). The values on the new frontier (Lines 13–14) are incorporated into the similarity score through an inner product. The algorithm stops after a maximum number of iterations (Line 10) or when the ℓ_1-distances between consecutive PageRank vectors for both vertices are below a threshold (Lines 16–17).

E Results on Hyperlink2012

Table 2 shows the experimental results for the graph benchmarks currently supported in GBBS on the Hyperlink2012 Web graph [33]. The benchmarks are run on a 72-core Dell PowerEdge R930 (with two-way hyper-threading) with 4 × 2.4GHz Intel 18-core E7-8867 v4 Xeon processors (with a 4800MHz bus and 45MB L3 cache) and 1TB of main memory. Table 1 lists the number of vertices and edges in both the directed, and undirected (-Sym) versions of this graph.

The times reported here are from our earlier work [15, 16], and the clique-counting time is from [40]. We note that for triangle counting [46] and higher-clique counting [40], the bounds that we achieve are work-efficient with respect to existing, highly-optimized sequential algorithms [12]. Other than the running times for triangle counting and 4-clique counting, all of our times run in just a few minutes, which is surprising given the size of this graph. For triangle and 4-clique counting, our times are slower due to the sheer number of triangles and 4-cliques supported by this graph—9.648×10^{12} and 7.306×10^{15}, respectively. All of our benchmarks implementations have strong bounds on their work and depth.

F GBBS Website

To make it easier to access GBBS documentation and also view our benchmark specifications, we have built a website for GBBS, which can be found at https://paralg.github.io/gbbs/. The website contains:

(1) Input-output specifications for each problem currently included in GBBS. Each specification page also includes information on how to compile the benchmark and run it on supported graph inputs.
(2) A getting-started guide, which explains the requirements for GBBS, how to install the library, and how to compile and run the codes on different graph formats.
(3) Instructions for how to use the Python bindings and update the bindings with new benchmarks.