Evaluation of Graph Analytics Frameworks Using the GAP Benchmark Suite

Ariful Azad*, Mohsen Mahmoudi Aznaveh†, Scott Beamer‡, Mark Blanco§, Jinhao Chen†, Luke D'Alessandro*, Roshan Dathathri¶, Tim Davis†, Kevin Deweese‖, Jesun Firoz**, Henry A Gabb††, Gurbinder Gill¶, Balint Hegyi‡‡, Scott Kolodziej†, Tze Meng Low§, Andrew Lumsdaine **‖, Tugsbayasgalan Manlaibaatar[x], Timothy G Mattson††, Scott McMillan§, Ramesh Peri††, Keshav Pingali¶, Upasana Sridhar§, Gabor Szarnyas‡‡, Yunming Zhang[x], Yongzhe Zhang[xi]

*Indiana University, †Texas A&M University, ‡University of California, Santa Cruz, §Carnegie Mellon University, ¶The University of Texas at Austin, ‖University of Washington, **Pacific Northwest National Laboratory, ††Intel Corporation, ‡‡Budapest University of Technology and Economics, [x]Massachusetts Institute of Technology, [xi]The Graduate University for Advanced Studies, SOKENDAI

***Abstract*—Graphs play a key role in data analytics. Graphs and the software systems used to work with them are highly diverse. Algorithms interact with hardware in different ways and which graph solution works best on a given platform changes with the structure of the graph. This makes it difficult to decide which graph programming framework is the best for a given situation. In this paper, we try to make sense of this diverse landscape. We evaluate five different frameworks for graph analytics: SuiteSparse GraphBLAS, Galois, the NWGraph library, the Graph Kernel Collection, and GraphIt. We use the GAP Benchmark Suite to evaluate each framework. GAP consists of 30 tests: six graph algorithms (breadth-first search, single-source shortest path, PageRank, betweenness centrality, connected components, and triangle counting) on five graphs. The GAP Benchmark Suite includes high-performance reference implementations to provide a performance baseline for comparison. Our results show the relative strengths of each framework, but also serve as a case study for the challenges of establishing objective measures for comparing graph frameworks.**

***Index Terms*—graph algorithms, benchmarking**

I. Introduction

A graph represents relationships among items. Mathematically, a graph is simply a set of vertices and a set of edges between vertices. How this mathematical definition translates into software systems for graph problems, however, is both diverse and complex. There are multiple ways to represent graphs, and for a given representation, multiple abstractions for defining graph algorithms.

In response, many software frameworks for implementing graph algorithms have emerged. Choosing between them can be overwhelming. Complicating matters further, the suitability of a graph framework for a class of problems can depend on the graph itself. A framework that performs well for a particular type of graph may perform poorly for another type.

A number of benchmarking projects exist to help choose between graph programming frameworks. One of the better known benchmarks in HPC is Graph 500 [36], [37], which generates a synthetic, scalable graph for a range of problem sizes. Two algorithms are considered, a breadth-first search (BFS) and multiple single-source shortest paths (SSSP). As implied by the name, the benchmark is often used to rank the top 500 HPC systems for performing graph computations.

A more recent benchmarking effort is the GraphChallenge [41]. These are a set of problems and a repository of large graphs to encourage "community approaches to develop new solutions for analyzing graphs."

Both the Graph 500 and GraphChallenge efforts have been impactful and have helped drive the state of the art in large-scale graph analytics. However, they only cover a small portion of the graph analytics landscape. They do not provide sufficient diversity of algorithms and graph topologies to represent the needs of the data analytics community. While their focus on large-scale systems is important, most graph problems are concerned with medium-sized graphs (i.e., a few billion edges) that fit on a single server. Also, with advances in memory technology, it is possible and even advantageous to process large graphs (i.e., > 100 billion edges) on a single system [22].

The LDBC Graphalytics [25] project provides a more diverse mix of graph algorithms and graph topologies. Graphalytics covers BFS, SSSP, PageRank (PR), weakly connected components (CC), community detection using label propagation (CDLP), and local clustering coefficient (LCC) over a mix of 39 medium-sized real and synthetic input graphs.

These benchmarking efforts, however, do not use the same hardware for each graph programming framework. This makes it difficult to distinguish between differences due to hardware from those due to the underlying algorithms. Hence, there is still considerable confusion about the fundamental performance differences between graph frameworks. In this study, we compare several frameworks for graph algorithms: the SuiteSparse implementation [18] of the GraphBLAS specification [12], [27], the Galois framework for data parallelism in irregular algorithms [30], the GraphIt domain-specific language [55], a new graph library called NWGraph that builds on the Parallel BGL [13], [23], plus hand-tuned implementations of specific graph algorithms in the Graph Kernel Collection (GKC) [2].

The benchmarks are run on the same system to eliminate

978-1-7281-7645-1/20/$31.00 ©2020 IEEE
DOI 10.1109/IISWC50251.2020.00029

hardware differences. The development team behind each framework ran the benchmarks to ensure correct and efficient usage. The results from each team were cross-validated by the other teams to minimize inconsistencies in both what is measured and how results are generated.

We used the GAP Benchmark Suite [7] in this effort. GAP consists of five medium-sized real and synthetic graphs, each with distinct characteristics. In addition to the benchmark specification, high-performance reference implementations of each graph algorithm are also provided. The 30 GAP tests (six graph algorithms times five input graphs) provide good coverage of the graph analytics landscape and reveal the relative strengths of each framework.

Our main contribution is the performance data in Table IV and Table V. Together these constitute the highest quality, consistent performance numbers that we are aware of for comparing graph frameworks. Such comparisons are vital to understanding ways these frameworks need to evolve.

II. Overview of the GAP Benchmark Suite

The GAP Benchmark Suite is designed to ease evaluating graph processing systems. Rather than leaving each effort to determine its own evaluation methodology and workload, a shared standard gives the community a common goal to work towards. The benchmark is described in a publicly-available specification [7]. Any implementation that follows those rules is thus easy to compare to other compliant evaluations. To establish baseline performance, the benchmark also provides high-performance reference implementations. Providing the specification and reference code separately best serves users who may only need only one of the two artifacts. For example, a graph framework developer may want to use the specification to ensure their results are easy to compare. Alternatively, a computer architecture researcher can use the reference code as a target software workload to accelerate.

The benchmark was designed in conjunction with a workload characterization [9] to ensure it exposes a range of computational demands. The graph kernels in the benchmark were selected based on their popularity in the research literature [6]. The benchmark does not require the use of specific algorithms to implement these kernels, but it does state the requirements of correct solutions to avoid ambiguity. The kernels contain an interesting mix of traits and are sufficiently scalable to run on large graphs. The emphasis on scalable algorithms focuses the benchmark on the more data-intensive traits that distinguish graph processing from other workloads.

The benchmark suite uses five input graphs selected for topological diversity and availability (Table I). One of the biggest takeaways from the workload analysis is that because graph processing is data-driven, the graph topology can have a bigger impact on the workload characteristics than the algorithm. The input graphs are from both real-world data (Road, Twitter, Web) and synthetic generators (Kron and Urand). The graphs[1] have been added to the widely-used SuiteSparse Library [19].

[1]https://sparse.tamu.edu/GAP

We briefly describe the benchmark's graph kernels while highlighting some common differences, but we recommend consulting the specification for details [7]:

- **Breadth-First Search (BFS)** is a fundamental traversal order, and we track the parent vertices rather than depths.
- **Single-source Shortest Paths (SSSP)** finds the distances to all reachable vertices from a starting vertex.
- **PageRank (PR)** computes a popularity score for all vertices in the graph. We execute it until the scores are sufficiently close to convergence.
- **Connected Components (CC)** labels all vertices by which (weakly) connected component they are in.
- **Betweenness Centrality (BC)** determines a vertex's influence on the graph by the fraction of shortest paths that pass through it. Computing BC exactly requires an unreasonable amount of time, so we approximate it by considering only four root vertices per trial.
- **Triangle Counting (TC)** counts the number of triangles (cliques of size three) in the graph. It counts each triangle once regardless of the permutation of its constituent vertex identifiers.

The reference code included with the benchmark serves not only as a baseline, but also an educational tool. Internally, the code implements many leading algorithms. We recommend consulting the appendix of the specification for details [7].

The goal of the benchmark suite is to help the community develop new innovations. To ensure the innovations are practical and not overly specialized, the benchmark rules discourage optimizations that are infeasible in a general-purpose graph framework or that presuppose something about the structure of the answer. For example, all algorithm implementations of a framework must operate on the same graph format unless they include the time to convert the general-purpose graph format to the specific format used. The most common issue is when an optimization is only beneficial in some cases or requires an input-sensitive parameter. Practical implementations must determine which of these optimizations to use via run-time heuristics.

To capture the possibilities of both practical and specialized optimizations, we perform the benchmarks in this study under two different sets of requirements. The *Baseline Performance* data set (Section IV-A) captures the spirit of the GAP Benchmark Suite and disallows optimizations that overly specialize for the graph or kernel. The results from the Baseline give a sense of how frameworks will perform in practice. The *Optimized Performance* data set (Section IV-B) removes those restrictions, and gives a view into the peak performance of a framework.

III. Graph Analytics Frameworks

In this work, we compare the frameworks listed in Table II. They were created by a variety of different institutions for different purposes, and range from direct implementations of algorithms, to libraries to build algorithms, to compiled domain-specific languages. By evaluating this collection of

Name	Description	# Vertices (M)	# Edges (M)	Directed	Degree	Degree Distribution	Approx. Diameter	References
Road	Roads of USA	23.9	57.7	Y	2.4	bounded	6,304	[20]
Twitter	Twitter Follow Links	61.6	1,468.4	Y	23.8	power	14	[31]
Web	Web Crawl of .sk Domain	50.6	1,930.3	Y	38.1	power	135	[10]
Kron	Kronecker Synthetic Graph	134.2	2,111.6	N	15.7	power	6	[33], [37]
Urand	Uniform Random Graph	134.2	2,147.5	N	16.0	normal	7	[21]

TABLE I
GRAPHS USED FOR EVALUATION

Framework	GAP	GKC	Galois	NWGraph	SuiteSparse	GraphIt
Type	direct implementations	direct implementations	generic high-level library	header-only library	high-level library	domain-specific language compiler
Internal Graph Data Structure Provides	outgoing & incoming edges	outgoing & (opt.) incoming edges	outgoing and/or incoming edges	adjacency list as range of ranges	outgoing & incoming edges w/ (opt.) hypersparsity	outgoing & incoming edges w/ (opt.) blocking
Programming Abstraction	vertex-centric	arbitrary	vertex, edge, or chunked-edges centric	range-centric w/ tuple edge properties	sparse linear algebra	vertex or edge centric
Execution Synchronization	level-synchronous	algorithm-specific, level-synchronous	level-synchronous or asynchronous	algorithm-specific, level-synchronous	level-synchronous	level-synchronous
Dependences	C++11, OpenMP	C++11, OpenMP	C++17, boost, libllvm	C++17, libtbb	C11, OpenMP	C++11, OpenMP, cilk
Intended Users	researchers, benchmarkers	application developers	graph domain experts	practicing C++ programmers	graph/matrix domain experts	graph domain experts

TABLE II
MAIN ATTRIBUTES OF FRAMEWORKS CONSIDERED

frameworks on a common benchmark, we can assess the impact of design decisions on performance and programmability (e.g., whether the programming abstraction provided by a framework eases or complicates the implementation of some graph algorithms).

The benchmark only specifies the graph problems, so each framework is free to choose which algorithms it implements (Table III). Each framework's choice of algorithm is guided by many factors, including its intended audience, the framework's flexibility, and the developer's awareness of new algorithms. Additionally, some frameworks exploit implementation optimizations such as Galois' occasional use of asynchronous scheduling and GKC's use of SIMD instructions.

For some graph problems, such as BFS and BC, there are well-established algorithms such as Direction-Optimizing BFS [8] and Brandes [11]. For other problems such as SSSP, the established delta-stepping algorithm [35], has been recently improved with bucket fusion [54]. CC shows the greatest algorithmic diversity, ranging from the classic label-propagation approach, to revised versions [32], [53] of the prior standard Shiloach-Vishkin [42] algorithm, and the new Afforest algorithm [45]. For PR, all of the implementations repeatedly perform a sparse matrix vector multiply (SpMV), but they differ in whether the updated values are available immediately (Gauss-Seidel) [4] or after an iteration (Jacobi). For TC, most of the implementations reduce the search space by only counting one permutation of each triangle, and they use heuristics to consider whether to relabel/reorder the graph to further accelerate the search.

A. LAGraph/GraphBLAS

SuiteSparse:GraphBLAS [18] is an implementation of the GraphBLAS C API [12] that describes a set of sparse matrix operations over semirings. In a semiring, the multiplication $C = A * B$ of two matrices A and B is redefined. In the conventional semiring, $c_{ij} = \sum_k a_{ik} \times b_{kj}$. In a different semiring, the multiplicative operator ($\times$) can become any binary operator, and the reduction via the additive operator ($\sum$) becomes any *monoid* (associative and commutative operator with an identity element). The data types of the three matrices can change as well. For example, in the Boolean semiring, the three matrices are all Boolean, and $c_{ij} = \vee_k a_{ik} \wedge b_{kj}$. In the min-plus tropical semiring, $c_{ij} = \min_k a_{ik} + b_{kj}$.

Sparse linear algebra over a semiring is a powerful framework for expressing a wide range of graph algorithms, including those in the GAP benchmark. The sparse matrix becomes the adjacency matrix of the graph. For example, a single "push" step of a breadth-first-search can be written as the matrix-vector multiply `q'<!pi>=q'*A`, followed by the assignment `pi<q>=q`, where `q` is a vector of nodes in the current level, `A` is the adjacency matrix of the graph, and `pi` is a vector containing the parent of the node in the BFS tree. The operation `C<M>=...` is a *masked* assignment, where `C(i,j)` can be modified only if the mask `M(i,j)` allows it. Many graph algorithms include a conditional `if` statement in their innermost loops, and the masked assignment captures this behavior in a single bulk expression over the entire result. GraphBLAS does not include any graph algorithms directly; these are in algorithms that use GraphBLAS. For the GAP benchmark, we developed six algorithms in the LAGraph library [34]:

- **BFS:** The expression `q'<!pi>=q'*A` (written in a C API) forms the essential kernel of the direction-optimizing BFS, using the *any-secondi* semiring, where `A` has arbitrary type and `q` is `int64`. Assuming `A` and `A'` are in compressed sparse row (CSR) format, this is a *push* BFS step, while `q<!pi>=A'*q` is a *pull* step using same semiring. The `z=any(x,y)` function serves as the monoid for the semiring and is defined as `x` or `y` at the discretion of the operator itself. This allows the monoid to

Task	GAP	GKC	Galois	NWGraph	SuiteSparse	GraphIt
BFS	Direction-optimizing	Direction-optimizing[3]	Direction-optimizing[4]	Direction-optimizing	Direction-optimizing	Direction-optimizing
SSSP	Delta-stepping[1]	Delta-stepping[3]	Delta-stepping[4]	Delta-stepping	Delta-stepping	Delta-stepping[1]
CC	Afforest	Shiloach-Vishkin Hybrid	Afforest[4]	Afforest	FastSV	Label Propagation
PR	Jacobi SpMV	Gauss-Seidel SpMV[3]	Gauss-Seidel SpMV	Gauss-Seidel SpMV	Jacobi SpMV	Jacobi SpMV
BC	Brandes	Brandes	Brandes[4]	Brandes	Brandes	Brandes
TC	Order invariant[2]	Lee & Low[2,3]	Order invariant[2]	Order invariant[2]	Order invariant[2]	Order invariant[2]

TABLE III
ALGORITHMS USED BY EACH FRAMEWORK WITH FOLLOWING ADDITIONS: 1 - BUCKET FUSION, 2 - HEURISTIC-CONTROLLED GRAPH RELABELLING, 3 - SIMD, 4 - AN ADDITIONAL ASYNCHRONOUS VARIANT

terminate as soon as any parent is found for a node in the next level. The multiplicative operator $secondi(a_{ik}, b_{kj})$ returns the row index of the second operand, which is the parent node id.

- **SSSP:** The SSSP implementation uses the delta-stepping algorithm [35], [44] and relies on the min-plus-int32 tropical semiring.
- **BC:** LAGraph includes Brandes' algorithm for computing betweenness centrality [11], which uses the plus-first-float semiring in GraphBLAS.
- **TC:** The LAGraph triangle counting method can be written in pseudo MATLAB notation as `L=tril(A,-1); U=triu(A,1); C<L>=L*U';`, based on the formulation in [49] using the plus-pair-int64 semiring. It is preceded by an optional permutation of `A`, decided by a heuristic.
- **CC:** LAGraph includes an implementation of the FastSV connected components algorithm [53], which uses the min-second-uint32 semiring.
- **PR:** PR can be written quite easily in terms of conventional linear algebra (plus-times-float). However, LAGraph uses the plus-second-float semiring so that only the structure and not the values of the adjacency matrix are accessed.

B. Galois

Galois [22], [38] is a C++-based general-purpose programming library and runtime for graph processing that permits optimizations to be specified in the program at compile- or run-time, giving the application programmer a large design space of implementations to explore. Galois supports a rich data-centric programming model called the *operator formulation* [40] that enables efficient, scalable graph analytics algorithms to be implemented without having to worry about concurrency bugs such as race conditions and deadlocks.

In typical graph analytics applications, each node has one or more labels (e.g., distance from the source node in the single-source shortest path), which are updated during algorithm execution until a global quiescence condition is reached. The labels are updated by repeatedly applying a computation rule, known as an *operator*, to *active* nodes in the graph. Each algorithm has its own set of operators. For example, SSSP problems are solved by applying the well-known *relaxation operator* to active vertices. When an operator is applied to an active vertex, this *activity* may read and update an arbitrary portion of the graph around the active vertex, known as the *neighborhood* of that activity.

Most graph analytics systems support only *vertex* programs in which operator neighborhoods are limited to the immediate neighbors of the active vertex. In contrast, the operator formulation does not restrict the neighborhoods of operators. Galois permits more efficient algorithms to be implemented such as the Afforest algorithm [45] for the CC problem. It also supports more complex algorithms that modify or mutate the graph like Delaunay mesh refinement [29] and METIS graph partitioning [26]. Due to its general non-vertex programming model, Galois has been used to build frameworks for more complex graph computations such as graph pattern mining [14].

Galois provides highly scalable concurrent data structures such as worklists to implement work-efficient *data-driven algorithms* [40] that dynamically track active nodes or the *frontier*. For data-driven algorithms (e.g., BFS, SSSP, BC, CC), Galois uses a sparse worklist (as large diameter graphs tend to have sparse frontiers), unlike most other frameworks (which use a dense bitvector). Galois uses a dense worklist to store the frontier only for *topology-driven algorithms* [40] (e.g., PR).

The concurrent sparse worklists also enable Galois to support *asynchronous* [22], [38] data-driven algorithms, which in contrast to bulk-synchronous algorithms do not have a notion of rounds. They maintain a single sparse worklist, pushing and popping active vertices from this worklist until it is empty. These algorithms have better work-efficiency and make fewer memory accesses, especially for BFS, SSSP, and BC on large diameter graphs which may need thousands of rounds in bulk-synchronous execution.

Another key factor impacting performance is memory allocation. Galois explicitly uses huge pages of size 2 MB and does not rely on the operating system to use *Transparent Huge Pages (THP)*. Huge pages can significantly reduce the cost of memory accesses over small pages even when THP is enabled [22], but we did not use it for this study. Galois provides non-uniform memory access (NUMA) blocked allocation (blocks the pages and distributes the blocks among NUMA nodes), which has been shown [22] to perform better for topology-driven algorithms over the NUMA local or interleaved policies provided by Linux utilities such as *numactl*. The *Galois runtime system* also optimizes program execution to exploit NUMA locality. For example, it performs NUMA-aware dynamic load balancing to ensure that computational load is spread evenly among the cores of a shared-memory

system.

Galois supports CPU and GPU [39] computation as well as distributed-memory CPU [15], [16] and GPU clusters [47]. The Galois source code [1] includes the Lonestar [29] suite of graph algorithm implementations.

C. NWGraph Library

The NWGraph library aims to fill the role of a reusable library of generic graph algorithms for C++, similar to the algorithms available in the standard template library. The library draws on the lessons learned from the Boost Graph Library (BGL) [43] and other libraries (PBGL [23], Galois, Gunrock [48], GraphX [50], et al.), the evolution of the C++ language, and the evolution of C++ practice over the last 20 years.

The underlying principle for NWGraph is that it is a *generic library*. That is, its algorithms are not written to use any particular graph data structures, but rather are written in terms of *properties of types* (aka concepts, finally to be available as a language feature in the upcoming C++20 standard). Following generic programming principles, the NWGraph *concepts are minimal*, enabling algorithms to be composed with *arbitrary types*. Users can therefore use NWGraph algorithms with the data types around which they have already structured their applications (which data structures are almost never graphs per se). Pragmatically, the algorithms in NWGraph are function templates written using modern C++ idioms, making them accessible to programmers already familiar with core language features and libraries and allowing them to leverage the full power of C++ and of other libraries, frameworks, and tools.

The fundamental interface abstraction to NWGraph algorithms is a "range of ranges" (expressed either as an iterator range or as C++20 ranges). The algorithms in turn are expressed using C++ standard library algorithms (transform(), reduce(), etc.). Again following modern C++ practice, parallelization of NWGraph algorithms is effected through mechanisms in the C++ standard. Since NWGraph algorithms in turn are based on C++ standard library algorithms, a modicum of shared-memory parallelization is immediately available through the standard library algorithms (specified with parallel execution policies). This parallelization approach will continue to be developed in future language standards and will be extended to encompass support for accelerators and FPGAs (e.g., Thrust, oneAPI, SYCL) and, accordingly, NWGraph will be able to take advantage of those advances.

Ideally, NWGraph prefers execution policies as the more "hands off" approach to parallelization (used in CC and BC implementations), but other approaches were explored as well. For best parallel performance it is necessary in some cases to manage parallelism directly through std::async (still a C++ standard feature) (used in TC implementation) or via Threading Building Blocks (TBB) primitives (used in BFS, SSSP, and PR implementations). These implementation details are not visible at the level of the library interface and the need for non-standard (and non-execution-policy) approaches is expected to dissipate over time. (Indeed, our experience with this benchmarking effort will aid future NWGraph development as the team continues to participate in the C++ standards committee.) Other non-standard atomic features were also required for competitive shared-memory performance, including atomic references, atomic bitmaps, and atomic operators for floats.

D. GraphIt

GraphIt [54], [55] is a domain-specific language (DSL) that achieves consistent high-performance across different algorithms, graphs, and architectures while offering an easy-to-use high-level programming model. GraphIt achieves this by decoupling the algorithm specification from optimization strategies for graph applications. Many graph applications require different optimization techniques. Therefore, users normally have to try out a large set of such techniques to achieve performance. Separating the high-level algorithms from performance optimizations solves this problem.

Users specify graph algorithms using the algorithmic language involving just high-level operations on sets of vertices and edges. They use the separate scheduling language to compose different optimizations. The algorithmic language exposes different high-level optimization opportunities such as parallelization and edge traversal direction.

The scheduling language supports a large space of optimization techniques such as edge traversal direction, data layout, parallelization, cache efficiency, NUMA, and kernel fusion optimizations. GraphIt uses scoped labels to target specific operations to optimize. Moreover, it uses an abstract *graph iteration space* model to represent, compose, and ensure the correctness of edge traversal optimizations. The DSL guarantees correctness by imposing restrictions on the GraphIt language and automatically inserting atomic operations through dependency analysis. To make it more user-friendly, GraphIt also has a built-in autotuner based on Opentuner [3] that explores the optimization space and finds high-performance schedules quickly using methods such as AUC bandit and greedy mutation.

GraphIt also achieves portability across CPUs, GPUs, and domain-specific accelerators. GraphIt introduces a new intermediate representation, GraphIR, to provide a common interface across different hardware backends. This new IR lets the compiler separate hardware-independent and hardware-dependent optimizations and achieve consistent high-performance across CPUs and GPUs.

E. Graph Kernel Collection (GKC)

GKC is a collection of commonly used graph kernels that are designed as a black-box library. These graph kernels are designed by applying traditional high performance computing techniques used in linear algebraic libraries to graph workloads. GKC embodies the hardware-software co-design philosophy by taking into account algorithmic properties and hardware features. This is implemented by identifying core primitives used in graph algorithms and designing high performance implementations of these primitives that leverage hardware features, such as instruction set capabilities and

aspects of the memory hierarchy of a given platform. Below we detail some techniques used in GKC.

1) Reducing false sharing: For implementations other than TC, each thread allocates its own memory buffer. The local buffer stores intermediate outputs (e.g., the next frontier for breath-first traversal-based algorithms). This buffer is explicitly flushed back to the global buffer accessed by all threads to form the global frontier for the next iteration of the algorithm. The local buffer reduces false sharing because threads can still read information stored in the global buffer while updating values maintained in separate local buffers.

2) Hardware-aware implementations: Implementations of the algorithms are tuned to the specific architecture on which the benchmarks were run. Local buffers are sized according to either the L1 or L2 cache sizes to ensure that they remain in the appropriate cache level. SIMD instructions are used to load and store data to and from local buffers. Computations, where appropriate, also use SIMD. Notably, higher performance was attained with AVX-256 over AVX-512 on the test platform. We intend to examine this in greater detail in the future.

3) Use of inline assembly: We observed that the Intel® C++ compiler (`icpc` 19.1) occasionally replaces code with calls to libraries (e.g., CLib). These libraries are efficient but include additional code for general cases. As such, GKC contains specialized kernels to handle specific tasks such as flushing local buffers of specific sizes using techniques introduced by Veras et al. [46]. C/C++ macros provide an intrinsics-like interface and expand to one or more inline assembly instructions surrounded by the `volatile` keyword. This ensures that the desired sequence and selection of instructions are untouched by the compiler.

IV. Evaluation Methodology

A. Baseline Performance

This data set is intended to be a uniform comparison of each framework. In a sense, it represents the performance that a typical end-user would achieve after installing the framework and running GAP using default parameters. Each framework used the same number of processors (32 physical cores) and NUMA policy (interleave=all). Frameworks with existing internal auto-tuners and heuristics were allowed to use them, but hand-tuning algorithms based on the graph topology was not allowed for this data set. The only exception is the *delta* parameter for SSSP. GAP allows customization of this parameter based on the graph topology because it can lead to orders of magnitude difference in performance otherwise.

Graph transposition was not included in the timing data because the GAP reference implementations store both forms of the graph. However, the cost of restructuring, relabeling, or other graph transforms was included in the timing data.

B. Optimized Performance

This data set represents the best performance that each framework can currently achieve for each GAP test. Teams were free to optimize thread count, thread placement and affinity, NUMA policy, etc. They could even tune for graph characteristics. They were not required to include the time for such tuning efforts in the timing data, but optimization details and settings must be reported.

C. Benchmarking System

All performance measurements were collected on Intel® Xeon®-based servers hosted in the Intel® DevCloud[2]. Each server contains two Intel® Xeon® Platinum 8153 processors, each with 16 physical cores (32 logical cores) running at 2.0 GHz. Each processor has 22 MB L3 cache. The total system memory of each server is 384 GB DDR4 running at 2.6 GHz.

V. Benchmark Results

With six algorithms, five graphs, and two ways of running the benchmarks, the quantity of data is daunting. We present our results in Table IV and Table V[3]. Table IV shows the best time (in seconds) for each algorithm/graph pair. The color coding of the cells indicate which framework achieved the best result for each GAP test. In Table V, results for all algorithms, graphs, and frameworks are presented as the ratio of the time for the GAP reference implementation to the time for a particular case (speedup). A value of 100% indicates that a particular case matched the time for GAP, a value of 50% indicates a case took twice as long as GAP, 200% indicates half as long, and so forth. Color coding as a heat map visually conveys trends: green indicates results faster than GAP while red indicates results slower than GAP.

Before discussing the results for each of the graph kernels, we start with some high level comments. Three of the frameworks (GAP, GKC, and NWGraph) directly code the graph kernels case-by-case in low-level programming languages (C or C++ with parallel programming models such as TBB, OpenMP). Galois, GraphIt, and GraphBLAS, however, are high-level abstractions specialized for expressing graph algorithms. There are overheads associated with these additional layers of abstraction which may cause performance challenges for the GAP kernels with low runtimes (BFS, SSSP and CC). Furthermore, all frameworks sort the adjacency list of each vertex based on the destinations and remove duplicate edges.

General frameworks for graph algorithms must handle problems well beyond those addressed by GAP. GraphBLAS, for example, is designed to handle graphs with up to 2^{60} nodes [28] with up to 2^{60} entries, so it uses 64-bit integer indices throughout. The other frameworks use 32-bit indices throughout by default (which can be easily changed), and that size easily accommodates the graphs evaluated. Thus, they can be tuned to this limited size of problem sets. They can all use 32-bit integers, while GraphBLAS must use 64-bit integers.

When considering such shortcomings in high-level frameworks, however, it is important to consider their advantages. With GraphBLAS, for example, algorithms can be developed quickly [17], often with good performance using productivity languages such as MATLAB or Python. GraphIt separates the

[2]https://devcloud.intel.com

[3]The complete timing data is available at https://tinyurl.com/eval-graph-frameworks

	Baseline (seconds)					Optimized (seconds)				
	Real Graphs			Synthetic Graphs		Real Graphs			Synthetic Graphs	
Kernel	Web	Twitter	Road	Kron	Urand	Web	Twitter	Road	Kron	Urand
BFS	0.329	0.248	0.130	0.365	0.570	0.300	0.214	0.109	0.308	0.486
SSSP	0.900	2.217	0.269	4.566	6.438	0.603	2.174	0.272	3.810	5.199
CC	0.219	0.246	0.060	0.691	0.670	0.167	0.209	0.045	0.479	0.606
PR	2.554	10.268	0.338	11.050	12.143	2.737	5.405	0.267	6.960	9.499
BC	3.178	8.237	2.431	13.300	16.389	2.978	5.215	1.876	11.240	14.040
TC	9.358	62.356	0.028	207.627	24.716	8.650	42.486	0.021	160.593	15.985
Fastest	GAP Reference		SuiteSparse	Galois		GraphIt		GKC	NWGraph	

TABLE IV
FASTEST TIMES FOR BASELINE AND OPTIMIZED DATA SETS. COLOR INDICATES WHICH FRAMEWORK ACHIEVED THE FASTEST RESULT FOR EACH CASE.

		Baseline (speedup over GAP reference)					Optimized (speedup over GAP reference)				
		Real Graphs			Synthetic Graphs		Real Graphs			Synthetic Graphs	
		Web	Twitter	Road	Kron	Urand	Web	Twitter	Road	Kron	Urand
SuiteSparse GraphBLAS	BFS	39.98%	60.50%	13.74%	58.14%	51.09%	36.38%	54.04%	8.02%	53.71%	46.48%
	SSSP	8.50%	32.23%	0.35%	32.10%	40.51%	5.84%	31.18%	0.43%	23.95%	32.56%
	CC	12.66%	18.87%	7.40%	20.13%	43.45%	11.08%	15.65%	6.30%	15.96%	33.05%
	PR	92.86%	87.92%	137.50%	91.04%	91.45%	85.02%	91.21%	173.42%	96.53%	97.81%
	BC	54.00%	70.93%	3.96%	80.38%	92.40%	42.69%	69.64%	3.46%	85.74%	84.95%
	TC	48.76%	31.92%	12.86%	34.01%	61.51%	55.53%	34.49%	12.47%	37.46%	61.04%
Galois	BFS	54.18%	44.77%	351.04%	57.14%	8.93%	58.55%	41.88%	220.92%	62.16%	77.85%
	SSSP	46.13%	55.94%	54.40%	41.76%	49.47%	26.62%	45.11%	67.37%	58.06%	53.53%
	CC	64.43%	114.02%	84.11%	85.22%	66.06%	113.94%	75.16%	90.16%	85.53%	49.16%
	PR	157.54%	84.36%	331.66%	106.15%	117.35%	154.67%	108.96%	456.72%	110.63%	125.71%
	BC	102.90%	68.88%	54.66%	71.36%	30.88%	105.52%	73.18%	43.83%	72.87%	75.12%
	TC	113.14%	108.29%	111.57%	98.02%	81.26%	235.19%	140.02%	130.04%	106.39%	90.62%
GraphIt	BFS	64.24%	86.40%	37.14%	84.29%	88.59%	54.11%	83.92%	74.34%	88.59%	95.14%
	SSSP	106.50%	110.96%	94.74%	112.40%	107.56%	86.17%	104.35%	93.88%	96.13%	106.48%
	CC	19.60%	8.86%	0.17%	7.06%	16.92%	16.10%	19.55%	0.45%	16.45%	27.85%
	PR	194.40%	109.23%	307.38%	102.72%	101.64%	149.14%	196.47%	350.03%	211.61%	186.20%
	BC	73.23%	100.23%	45.98%	224.15%	272.49%	75.85%	189.21%	34.67%	223.41%	251.01%
	TC	99.30%	108.45%	67.67%	113.89%	101.73%	98.72%	107.06%	98.41%	106.97%	104.38%
Graph Kernel Collection (GKC)	BFS	68.68%	67.33%	157.85%	61.20%	67.47%	74.44%	60.29%	83.29%	56.75%	64.35%
	SSSP	113.22%	89.68%	18.38%	86.72%	119.25%	115.98%	98.23%	18.53%	77.29%	118.17%
	CC	31.87%	26.53%	14.29%	32.95%	295.12%	27.69%	19.76%	10.82%	23.46%	214.27%
	PR	191.32%	105.56%	358.54%	136.28%	142.03%	125.03%	104.14%	324.19%	137.15%	150.24%
	BC	106.98%	100.30%	101.55%	101.60%	102.33%	106.23%	97.49%	77.15%	101.34%	102.76%
	TC	107.36%	157.92%	149.43%	197.51%	123.19%	106.98%	160.46%	176.41%	187.20%	113.98%
NWGraph	BFS	23.78%	65.85%	53.02%	65.34%	42.54%	26.59%	66.57%	33.97%	67.28%	48.74%
	SSSP	47.62%	85.35%	4.61%	114.69%	54.25%	46.33%	109.46%	6.58%	102.53%	55.39%
	CC	59.89%	69.09%	62.36%	61.50%	99.63%	49.60%	64.33%	60.34%	57.21%	87.41%
	PR	230.67%	110.38%	373.94%	108.16%	120.65%	175.33%	119.14%	499.59%	112.20%	124.68%
	BC	139.07%	135.88%	41.49%	163.21%	92.44%	117.33%	139.02%	38.15%	151.84%	90.77%
	TC	249.06%	132.30%	60.61%	108.27%	124.01%	228.14%	129.97%	51.35%	109.45%	112.77%

TABLE V
SPEEDUPS OVER THE GAP REFERENCE IMPLEMENTATION FOR THE BASELINE AND OPTIMIZED DATA SETS. PERCENTAGES REPRESENT THE RATIO OF THE TIME RELATIVE TO THE GAP REFERENCE FOR A PARTICULAR TEST. THE COLOR-CODED HEAT MAP INDICATES WHERE PERFORMANCE IS LOWER THAN (RED), EQUAL TO (WHITE), OR HIGHER THAN (GREEN) THE GAP REFERENCE.

algorithm expression from the schedule which makes it much easier to adapt to features of different platforms.

It is interesting to consider the changes made for the different frameworks when moving from the Baseline to the Optimized cases. Some of the frameworks made only minimal changes in moving between the two. The improvements in performance in NWGraph and GKC for the Optimized cases are almost entirely from taking advantage of hyperthreading and using all 64 logical cores. The general improvements in the hyperthreaded performance suggest that hardware resources remain underutilized, and techniques introduced by the other frameworks could be leveraged for better performance. The NWGraph developers consider the low requirement for parameter tuning to be a feature of their library as users are not required to tune for optimal performance. Instead, this burden falls to the implementors of STL and TBB.

GraphIt also benefited from the use of hyperthreading, but in addition, it used schedules/optimizations specialized for the size and structure of the graphs for the Optimized case. This was not allowed for the Baseline data set. These optimizations resulted in general improvements for PR, BFS, TC, and BC even though some of the schedules remained the same.

Galois stood out by making extensive changes between the Baseline and Optimized cases. For BFS, SSSP, and BC, the relative performance of different algorithm implementations can vary significantly for high diameter and low diameter graphs. Hence, in the Optimized case, the Galois team chose one algorithm for Road because it is known to have a high diameter (Table I), and another algorithm for the other inputs because they are known to have low diameters. It is not trivial to estimate the diameter of a graph. They used a vertex sampling scheme (similar to that in GAP for TC) to determine whether a graph has power-law degree distribution (Web, Twitter, Kron) or uniform-degree distribution (Road, Urand). In the Baseline case, they assumed the graph had a low diameter if it has power-law degree distribution and a high diameter otherwise[4], and then automatically picked the algorithm based on the assumed diameter.

A. Breadth First Search (BFS)

GraphBLAS - The BFS relies on three internal data structures in GraphBLAS, which are opaque to the LAGraph library: a bitmap, a sparse list (CSR), and a full matrix. The vector `q` is converted to bitmap for the "pull" step, and converted to a sparse list for the "push" step. This conversion time is included in the total run time. The parent `pi` is held as a full vector, while the adjacency matrix `A` and its transpose are held in CSR format. The BFS achieves competitive performance, except for Road. The same algorithm is used for all graphs. Road has high diameter, so many iterations in LAGraph are needed, with smaller and lighter-weight calls to GraphBLAS kernels. GraphBLAS does include a non-blocking mode that could in theory allow for kernels to be fused, but this is not fully implemented yet. A truly asynchronous BFS that can work on multiple levels at a time is likely beyond the scope of GraphBLAS+LAGraph.

Galois - For power-law graphs, both Galois and GAP use the same bulk-synchronous direction-optimizing algorithm. As the runtime is very small, the overheads of a generic library such as Galois are significant. For Urand, the Optimized case in Galois uses the same bulk-synchronous algorithm but the Baseline case uses asynchronous execution, which increases redundant work significantly because Urand is a low diameter graph. In contrast, asynchronous execution for Road increases parallelism with a small increase in redundant work, so Galois is $3.6\times$ and $2.2\times$ faster than GAP for Road in the Baseline and Optimized cases, respectively.

GraphIt - GAP has better performance on Road because of a more efficient way of creating a new frontier/vertexset than GraphIt. For social networks (e.g., Twitter), the difference can also be attributed to different frontier creation mechanisms and a more efficient way of counting the number of active vertices in GAP. For the Optimized case, GraphIt is faster than GAP on Road by 40% because it does not use direction optimization (always push). This eliminates the runtime overhead of checking the number of active vertices.

GKC - Because Road is a small, large-diameter graph, the BFS algorithm will be particularly sensitive to overheads associated with higher level abstractions. Hence, a hand-optimized approach as used with GKC has an advantage as shown by the high performance with BFS for Road.

NWGraph - The BFS algorithm used with NWGraph is a straightforward, initial implementation with a simple direction optimized search and no fine tuning of the switching criteria. Performance is sensitive to the heuristic that controls the switch between the pull and push portions of the algorithm. Overheads due to NWGraph's reliance on STL vectors over more lightweight vectors was particularly noticeable for Road.

B. Single Source Shortest Paths (SSSP)

GraphBLAS - SSSP uses a delta-stepping method, and has similar characteristics to that seen with BFS, except that it is slower because it cannot yet exploit the bitmap data structure. The bitmap data structure has not yet been fully incorporated into GraphBLAS so it is currently available only in the BFS.

Galois - Galois uses a delta-stepping algorithm with a bulk-synchronous variant for power-law graphs and an asynchronous variant for the uniform graphs in the Baseline case. Although GAP uses a bulk-synchronous delta-stepping algorithm for all graphs, GAP is faster than Galois due to the bucket fusion optimization. Asynchronous execution in Galois for Road reduces this performance gap. For the Optimized case, the bulk-synchronous variant with Urand ran better and reduced the performance gap relative to GAP.

GraphIt - GraphIt is comparable to GAP on all the graphs because GAP incorporated GraphIt's bucket fusion optimization, which significantly reduces synchronization [54]. GAP is slightly faster by further reducing overhead in the optimization implementation. Note that GraphIt was more than $7\times$ faster

[4]Real-world graphs that do not have power-law degree distribution typically have a high diameter because they are planar graphs, but the synthetic Urand graph is not planar and has a low diameter without having power-law degrees.

on Road before the bucket fusion optimization was integrated into the current GAP benchmark. Relative performance for the Baseline and Optimized cases were the same.

C. Connected Components (CC)

GraphBLAS - The CC algorithm in LAGraph+GraphBLAS is based on a high-performance algorithm [52] but the implementation in GraphBLAS has some issues that need to be resolved. One issue is that the matrix assignment with the *MIN* operator as the accumulator does not take the minimum of multiple entries assigned into the same location; the GraphBLAS C API specifies that the result of this kind of assignment is undefined. As a result, the CC method in LAGraph uses its own implementation of this kernel.

Galois - Galois and GAP both use the same Afforest algorithm. For the Baseline case, GAP is faster than Galois except for Twitter. For the Optimized case and Web, the edge blocking variant of the Afforest algorithm used in Galois performs much better due to better load balancing.

GraphIt - GraphIt is slower than GAP for CC due to differences in the algorithms they used. GAP uses the sampling-based Afforest algorithm which runs in $O(V)$ where V is the number of vertices. GraphIt does not yet support sampling algorithms and uses a label-propagation approach which runs in $O(ED)$ where D is the diameter of the graph and E is the number of edges. For the Optimized data set, GraphIt used label propagation with a short-circuiting approach on Road as the vertex chains tended to go longer on high-diameter graphs. This resulted in a 3x speedup but it was still slower than GAP. The GraphIt CC implementation also used cache optimizations similar to PR for speedups on the social network graphs.

GKC - CC performance is dependent on the algorithm. The observation by Sutton et al. [45] that the Afforest algorithm is less effective on the Urand graph is replicated here. However, the performance gap between Afforest and the hybrid algorithm used with GKC is significantly smaller (at most 7x speedup on Road) than the performance reported in the original paper (up to 100x speedup on Road). The narrowing of the performance gap is most likely due to the use of SIMD instructions and local intermediate buffers for GKC.

D. PageRank (PR)

GraphBLAS - GraphBLAS does well fairly well for PR, taking about as much time as the GAP benchmark. This is expected, since it is using the same basic algorithm. The main constraint that PR in LAGraph faces in the future is that an asynchronous Gauss-Siedel method is likely beyond the scope of the GraphBLAS API. There is no mechanism in the C API Specification for partially computing a vector x in $x = A * x$, in asynchronous parallelism with other threads.

Galois - Galois is faster than GAP because its Gauss-Seidel-style algorithm converges faster and performs fewer operations than the Jacobi algorithm. The benefits increase with the diameter of the graph, so Galois is $3.6\times$ faster than GAP for Road. For the Optimized case, Galois uses NUMA blocked allocation for the graph topology and vertex labels. When combined with the Gauss-Seidel algorithm, this resulted in a $4.7\times$ speedup relative to GAP for Road.

GraphIt - GraphIt is comparable to GAP on Kron, Urand, and Twitter, and faster on Web and Road due to better scaling for the same amount of work. For the Optimized cases, GraphIt is faster than GAP due to cache optimization from tiling the graph [51]. Web had good locality and did not benefit as much from cache optimization. In general, the preprocessing time to construct cache efficient subgraphs from CSR format is small compared to the performance gains, so it is amortized within 2 - 5 iterations. This is helpful for algorithms like PR that require around 20 iterations to converge.

NWGraph - NWGraph used the Gauss-Seidel algorithm and saw performance in line with that observed for the other frameworks using that algorithm.

E. Betweenness Centrality (BC)

GraphBLAS - BC is competitive versus the GAP benchmark, except for Road, where it shares the same limitations as BFS and SSSP. Most of the operations are matrix-matrix, where one matrix is dense and 4-by-n. LAGraph implements the batch Brandes algorithm, in a mere 97 lines of very readable code (47 in the MATLAB interface, including error checks on the inputs). It cannot yet use the newly-developed bitmap structure internal to GraphBLAS, as this is only partially developed.

Galois - For power-law graphs, both Galois and GAP use the bulk-synchronous Brandes algorithm, but GAP is faster because it saves the list of successors for each vertex using a bitmap. Because of this optimization, GAP is faster than Galois for uniform graphs, even though Galois uses the asynchronous Brandes algorithm. Asynchronous execution hurts performance for Urand in the Baseline case because Urand is a low diameter graph. For the Optimized case, results on Urand are better because the bulk-synchronous algorithm is used.

GraphIt - Unlike GAP's implementation, GraphIt transposes the graph for the backward pass gaining speed for larger graphs but running slower for the smaller graphs. GraphIt uses a bitvector to represent the frontier, which is advantageous when there are many active elements in the frontier. For the Optimized case, the GraphIt algorithm reduces overhead by not using a bitvector for the frontier on Road, resulting in a modest speedup.

NWGraph - The BC kernel did not use direction optimized breadth-first search. Performance, however, is still competitive, with the exception of Road. Results on Road are often worse for NWGraph. As Road is a smaller graph, overheads due to NWGraph's reliance on STL vectors are more significant compared to frameworks that use more lightweight vectors.

F. Triangle Counting (TC)

GraphBLAS - TC is very simple in LAGraph+GraphBLAS: except for the optional presort, it is a single masked matrix-matrix multiply, followed by a reduction to a single scalar. To accomplish this, the entire

matrix is first formed, then summed to a single scalar and discarded. It would be much faster to skip construction of the matrix and simply sum up its entries as they are computed. If this kernel were fused (which could be done in non-blocking mode), this would improve TC performance by a factor of two or more. Also yet to be implemented is a fast SIMD set intersection method for the dot product-based matrix-matrix multiply.

Galois - Galois uses the same TC algorithm as GAP. For Web, which has power-law degrees, Galois performance benefits from better work stealing and load balancing. For Urand, which has uniform degrees, Galois is slower due to the overheads of work stealing when the load is already well balanced. For the Optimized case, we excluded the time to preprocess and relabel the graph so Galois is much faster than GAP.

GraphIt - GraphIt is slightly faster than GAP on Kron and Twitter. The GraphIt algorithm is observed to have less branch misprediction [24], which is important for the larger graphs. For the Optimized data set, GraphIt was originally slower than GAP on Road because it used a set intersection method that was inefficient for smaller graphs. Changing back to the naive intersection method used in GAP improved performance.

GKC - GKC sorts vertices depending on degree skewness, then uses SIMD instructions depending on average degree and available hardware features. It performs set intersections with vectors that were previously visited, thereby increasing data reuse in caches. The combination of algorithm-enabled cache reuse, heuristic-driven relabeling, and appropriate use of hardware capability such as SIMD set intersection results in GKC outperforming GAP for both cases on all graphs.

NWGraph - The TC numbers were quite competitive, especially for Web, whose skewed degree distribution makes load balancing difficult. NWGraph's cyclic distribution of rows across threads led to near optimal load balancing. TC also benefits from sorting and relabeling the edge list (which is included in the timing results) before compressing to a sparse adjacency format (which is not timed per benchmark timing rules). This is a much more efficient strategy than sorting and relabeling on the compressed graph.

VI. Discussion

This paper was born from frustration. We tried repeatedly to produce high quality, reproducible performance numbers for key graph algorithm frameworks. There were so many ways to use each framework that we could never be sure we were using any given system to its full advantage. The result was little hard data to gauge relative performance.

The solution was to bring the groups behind each of the frameworks together to run their software with a common benchmark on the same hardware. We negotiated rules for two cases. The *Baseline* case tried to replicate the performance an unsophisticated user might see (the "out of the box" experience). The *Optimized* case urged each group to carry out optimizations for each algorithm to understand the best performance available from their framework.

The heat maps shown in Table V suggest that no framework is best for all graphs or algorithms. This is apparent in that none of the rows or columns are fully green, and is well supported by anecdotal evidence from graph algorithm researchers; no single graph framework can equally handle the full diversity of graph problems.

Road in particular was difficult for many of the frameworks because of its small size and high diameter. Many graph algorithms are iterative with synchronization required at each iteration. With the small size of Road, there was little useful work to amortize synchronization overhead. For the *Optimized* data set, the GAP reference implementations often did better on Road with fewer cores precisely because it would reduce the synchronization burden. Finally, timings for algorithms on Road were more unstable compared to other cases. This was most likely due the short runtimes making the results more sensitive to sequential startup overheads.

Three frameworks stood out in their performance on Road: Galois for BFS, GraphIt for SSSP, and GKC for TC.

- Galois makes heavy use of asynchronous execution. This helps algorithms converge sooner because they can update information faster without waiting at the bulk synchronous (frontier-based) iteration boundaries. This effect would be particularly notable for large diameter graphs such as Road. Galois preforms better for PR due to the Gauss-Seidel approach with in-place updates, which is more efficient due to a reduced number of operations.
- GraphIt used a new bucket fusion optimization for SSSP with delta stepping. It is based on the bucketing-based priority queue. The gist of the optimization is if a thread sees that the next bucket has the same priority as the current bucket, it can process the next bucket without synchronizing with other threads. This way, GraphIt is able to reduce the number of rounds/synchronizations by a factor of ten while maintaining a strict priority order. It sets a threshold on the next bucket size to avoid load imbalance [54]. The bucket fusion optimization has been incorporated into the GAP reference implementation.
- GKC heuristically applies SIMD kernels and graph relabeling, based on hardware capability (e.g., SIMD length and ops per second), graph skewness, and size. In Road, this is relevant because the overheads of sorting and using SIMD are avoided due to the heuristics. Further, Road benefits from GKC's algorithm because of its small size, resulting in higher cache-reuse.

This study revealed several potential improvements to the GAP Benchmark Suite.

- We identified and fixed a bug in the implementation of BC's path counting algorithm.
- The GAP reference implementations try to establish a reasonable performance target for graph frameworks. However, the reference PR implementation is no longer performance competitive with leading frameworks. It can be accelerated with blocking, but the resulting code might be too platform-specific [5]. Alternatively, switching to

a Gauss-Seidel approach for PR is far more practical, and the results of this study demonstrate the performance advantages of that approach.
- We found considerable ambiguity in the procedures to validate results and validation is important to assure that all frameworks are converging on consistently meaningful results. We recommend more formally specified verification and validation procedures for GAP. This needs to include both correctness and timing guidelines.

We see two productive avenues for future work. First, the most difficult part of this project was to work out procedures required to generate consistent results. Those same procedures can be used with other graph frameworks, allowing us to expand these data sets. Second, we did not analyze the complexity of the algorithms from one framework to the next. This is the ever-challenging "programmability problem" all too often overlooked due to the ambiguities inherent in measuring programmability. This is still, however, a critical issue to explore as we work to improve the quality of frameworks used to create graph algorithms.

VII. Acknowledgements

This work was supported by the National Science Foundation (grants: 1835499, 1618425, 1705092, and 1725322), a DARPA ERI contract (26-3511-51), a DARPA SDH Award (HR0011-18-3-0007), and grants from Intel, NVIDIA, IBM, and Redis Labs. It was also supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM20-0760]. M.B. is supported by the National Science Foundation Graduate Research Fellowship Program (grant DGE 1745016) and by the Carnegie Mellon University Jack and Mildred Bowers Scholarship in Engineering. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Galois. http://iss.oden.utexas.edu/?p=projects/galois, 2019.

[2] Graph Kernel Collection. https://github.com/CMU-SPEED/GKC, 2019.

[3] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*. PACT, 2014.

[4] Arvind Arasu, Jasmine Novak, Andrew Tomkins, and John Tomlin. PageRank computation and the structure of the web: Experiments and algorithms. In *WWW*, pages 107–117, 2002.

[5] S. Beamer, K. Asanović, and D. Patterson. Reducing pagerank communication via propagation blocking. In *IPDPS*, pages 820–831, 2017.

[6] Scott Beamer. *Understanding and Improving Graph Algorithm Performance*. PhD thesis, University of California, Berkeley, 2016.

[7] Scott Beamer, Krste Asanović, and David Patterson. The GAP Benchmark Suite. *arXiv preprint arXiv:1508.03919*, 2015.

[8] Scott Beamer, Krste Asanović, and David A. Patterson. Direction-optimizing breadth-first search. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[9] Scott Beamer, Krste Asanović, and David A. Patterson. Locality exists in graph processing: Workload characterization on an Ivy Bridge server. *International Symposium on Workload Characterization (IISWC)*, 2015.

[10] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. *WWW*, pages 595–601, 2004.

[11] Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[12] Aydın Buluç, Timothy Mattson, Scott McMillan, José Moreira, and Carl Yang. The GraphBLAS C API specification. *GraphBLAS. org, Tech. Rep.*, version 1.3.0, 2019.

[13] V. G. Castellana, M. Drocco, J. Feo, J. Firoz, T. Kanewala, A. Lumsdaine, J. Manzano, A. Marquez, M. Minutoli, J. Suetterlein, A. Tumeo, and M. Zalewski. A parallel graph environment for real-world data analytics workflows. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1313–1318, 2019.

[14] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. Pangolin: An efficient and flexible graph pattern mining system on CPU and GPU. *PVLDB*, 13(8), 2020.

[15] R. Dathathri, G. Gill, L. Hoang, V. Jatala, K. Pingali, V. K. Nandivada, H. Dang, and M. Snir. Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–28, 2019.

[16] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *PLDI*, pages 752–768. ACM, 2018.

[17] T. A. Davis, M. Aznaveh, and S. Kolodziej. Write quick, run fast: Sparse deep neural network in 20 minutes of development time via suitesparse:graphblas. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2019.

[18] Timothy A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Trans. Math. Softw.*, 45(4), December 2019.

[19] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38:1:1 – 1:25, 2011.

[20] 9th DIMACS implementation challenge - Shortest paths. `http://www.dis.uniroma1.it/challenge9/`, 2006.

[21] Paul Erdős and Alfréd Rényi. On random graphs. I. *Publicationes Mathematicae*, 6:290–297, 1959.

[22] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using Intel Optane DC persistent memory. *PVLDB*, 13(8):1304–1318, 2020.

[23] Douglas Gregor and Andrew Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[24] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with SIMD instructions by reducing branch mispredictions. *PVLDB*, 8(3):293–304, November 2014.

[25] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardt, Hassan Chafi, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. LDBC Graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. In *PVLDB*, volume 9, pages 1317–1328, 2016.

[26] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.

[27] Jeremy Kepner, Peter Aaltonen, David Bader, Aydın Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, José Moreira, John Owens, Carl Yang, Marcin Zalewski, and Timothy Mattson. Mathematical foundations of the GraphBLAS. In *HPEC*. IEEE, 2016.

[28] Jeremy Kepner, Tim Davis, Chansup Byun, William Arcand, David Bestor, William Bergeron, Vijay Gadepally, Matthew Hubbell, Michael Houle, Michael Jones, Anna Klein, Peter Michaleas, Lauren Milechin, Julie Mullen, Andrew Prout, Antonio Rosa, Siddharth Samsi, Charles Yee, and Albert Reuther. 75,000,000,000 streaming inserts/second using hierarchical hypersparse GraphBLAS matrices. 2020.

[29] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*. IEEE, 2009.

[30] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222. ACM, 2007.

[31] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? *WWW*, 2010.

[32] Matthew Lee and Tze Meng Low. A family of provably correct algorithms for exact triangle counting. In *CORRECTNESS*, page 14–20. ACM, 2017.

[33] Jurij Leskovec, Deepayan Chakrabarti, Jon Kleinberg, and Christos Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. *PKDD*, 2005.

[34] Tim Mattson, Timothy A. Davis, Manoj Kumar, Aydin Buluc, Scott McMillan, José Moreira, and Carl Yang. LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS. In *GrAPL at IPDPS*, pages 276–284. IEEE, 2019.

[35] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114 – 152, 2003. 1998 European Symposium on Algorithms.

[36] Richard C. Murphy, Jonathan Berry, William McLendon, Bruce Hendrickson, Douglas Gregor, and Andrew Lumsdaine. DFS: A simple to write yet difficult to execute benchmark. In *IEEE International Symposium on Workload Characterizations (IISWC)*. IEEE, 2006.

[37] Richard C. Murphy, Kyle B. Wheeler, Brian W Barrett, and James A. Ang. Introducing the Graph 500. In *Cray User's Group*. CUG, 2010.

[38] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *SOSP*, pages 456–471. ACM, 2013.

[39] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on GPUs. *SIGPLAN Not.*, 51(10):1–19, October 2016.

[40] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The TAO of parallelism in algorithms. In *PLDI*, pages 12–25, 2011.

[41] Siddharth Samsi et al. GraphChallenge.org: Raising the bar on graph analytic performance. In *HPEC*. IEEE, 2018.

[42] Yossi Shiloach and Uzi Vishkin. An O(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57 – 67, 1982.

[43] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. *The boost graph library: user guide and reference manual*. Addison-Wesley, 2002.

[44] U. Sridhar, M. Blanco, R. Mayuranath, D. G. Spampinato, T. M. Low, and S. McMillan. Delta-stepping SSSP: From vertices and edges to GraphBLAS implementations. In *GrAPL at IPDPS*, pages 241–250, 2019.

[45] Michael Sutton, Tal Ben-Nun, and Amnon Barak. Optimizing parallel graph connectivity computation via subgraph sampling. In *IPDPS*, pages 12–21. IEEE, 2018.

[46] Richard Veras, Thom Popovici, Tze-Meng Low, and Franz Franchetti. Compilers, hands-off my hands-on optimizations. In *Workshop on Programming Models for SIMD/Vector Programming (WPMVP) at PPoPP*, 2016.

[47] Jatala Vishwesh, Dathathri Roshan, Gill Gurbinder, Hoang Loc, Nandivada V. Krishna, and Pingali Keshav. A study of graph analytics for massive datasets on distributed GPUs. In *IPDPS*, 2020.

[48] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the GPU. In *PPoPP*, 2016.

[49] M. M. Wolf, M. Deveci, J. W. Berry, S. D. Hammond, and S. Rajamanickam. Fast linear algebra-based triangle counting with KokkosKernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[50] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. GraphX: A resilient distributed graph system on Spark. In *Graph Data-management Experiences & Systems (GRADES) at SIGMOD*, 2013.

[51] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. Making caches work for graph analytics. In *Big Data*, pages 293–302, 2017.

[52] Yongzhe Zhang, Ariful Azad, and Zhenjiang Hu. *FastSV: A Distributed-Memory Connected Component Algorithm with Fast Convergence*, pages 46–57.

[53] Yongzhe Zhang, Ariful Azad, and Zhenjiang Hu. Fastsv: a distributed-memory connected component algorithm with fast convergence. In *PP*, pages 46–57. SIAM, 2020.

[54] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with GraphIt. In *CGO*, page 158–170. ACM, 2020.

[55] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. GraphIt: A high-performance graph DSL. *PACMPL/OOPSLA*, 2:121:1–121:30, October 2018.