Large-Scale Performance Characterization of Distributed Graph Processing Frameworks

Altan Haan, Manish Shetty
{altanh, manishs}@berkeley.edu

1 Abstract

Graph processing frameworks provide the necessary primitives and constructs to design and implement graph algorithms, which in turn can express many real-world applications. However, the landscape of graph processing frameworks is vast, and choosing the right or "best" framework for a given task and graph is nontrivial. Existing work has tried to address this by studying and comparing graph frameworks, but are limited to shared-memory settings and/or small scale of data (size of graphs). In this work, we aim to perform *large scale benchmark* of *distributed memory* graph processing frameworks, with the eventual goal of identifying performance characteristics *with respect to features of the input graph*. In that regard, we perform a sweep over diverse real-world graphs and compare two popular graph processing frameworks: ***Gemini*** (Zhu et al., 2016) and ***CombBLAS*** (Buluç & Gilbert, 2011). Through this study, we not only compare two frameworks, but also two paradigms of distributed graph processing: ***vertex centric*** and ***linear algebraic***. We find that CombBLAS scales significantly better than Gemini for smaller graphs, and neither frameworks successfully run on large web graphs. Furthermore, we identify significant memory bottlenecks in these frameworks during graph loading and construction.

2 Problem Description

The overarching problem we address in this work is the challenge of selecting the appropriate distributed graph processing framework for a given task and input graph. Graph processing frameworks provide essential constructs and primitives for implementing graph algorithms that can solve many real-world problems. However, the vast landscape of graph processing frameworks can make it difficult to choose the best one for a particular scenario.

The selection of the most appropriate framework depends on various factors such as the size and characteristics of the input graph, the algorithmic primitives required, the resources available, and the performance requirements. The choice of the wrong framework can lead to suboptimal performance, longer processing times, or even failure to complete the task.

Prior work has focused on comparing graph processing frameworks, but most of these studies are limited to small-scale data (Beamer et al., 2015) and shared-memory settings (Satish et al., 2014; Koch et al., 2016). Here, we aim to address these limitations by performing a large-scale benchmark of distributed memory graph processing frameworks. By identifying the strengths and weaknesses of each framework and paradigm, we aim to identify insights that could help researchers and practitioners choose the most appropriate framework for their specific use case. Additionally, this work can inform future research on developing more efficient, versatile, and user friendly graph processing frameworks.

3 Implementation Details

In the sections that follow, we discuss the various components of our study including frameworks, tasks, and inputs graphs. We begin by outlining the rationale and setup for each of these factors. Then, we provide an account of the necessary implementation details and modifications we made to integrate these components into each framework. Our evaluation and analysis code is available at `https://github.com/altanh/dgb`.

3.1 Framework Selection

We initially considered five distributed graph processing frameworks, in three broad categories:

1. **Vertex/edge-centric**: Galois (Dathathri et al., 2018), Gemini (Zhu et al., 2016);
2. **Hybrid linear-algebraic**: GraphMat (Sundaram et al., 2015), LA3 (Ahmad et al., 2018) - these frameworks reformulate message passing as generalized SpMV;

3. **Linear algebraic**: CombBLAS (Buluç & Gilbert, 2011).

Then, given our focus on *large* graphs, we discarded frameworks where vertex indices could not easily be modified to use 64-bit integers. This eliminated LA3 and GraphMat, which use hardcoded 32-bit integers pervasively (including SIMD optimizations in the case of GraphMat). Lastly, we ran basic correctness checks on the remaining frameworks, which led us to discard Galois as it could not produce correct results using either shared-memory or distributed-memory parallelism.[1] In the end, we were left with CombBLAS and Gemini as representative frameworks from the vertex/edge-centric and linear-algebraic approaches respectively. We modified Gemini to use 64-bit vertex indices by changing a single `typedef`.

3.2 Task Selection

We initially selected six common tasks (graph kernels) as mentioned in the GAP benchmarks suite to maintain comparability (Beamer et al., 2015). However, due to time constraints, we opted to drop the betweenness-centrality (BC) task for our benchmarks. These tasks are representative of many applications within social network analysis, engineering, and science. This set of tasks is computationally diverse, as it includes both *traversal-centric* and *compute-centric* tasks. We provide a brief description of the problem setup for each task below:

1. **Breadth-First Search (BFS)**: BFS is a traversal order starting from a source vertex. It traverses all vertices at the current depth (distance from the source vertex) before moving onto the next depth. Some uses of BFS only track reachability or depth, but we choose to track the parent. This is because we would like to verify the traversal was performed in a breadth-first manner, and reachability only returns a boolean value for each vertex.
2. **Single-Source Shortest Path (SSSP)**: SSSP computes the distances of the shortest paths from a given source vertex to every other reachable vertex. The distance between two vertices is the minimum sum of edge weights along a path connecting the two vertices. To maintain a unique solution to SSSP, we compute the distances and not the shortest paths themselves.
3. **PageRank (PR)**: PR computes the importance of all vertices within the graph, based on the number incoming edges and the importance of the corresponding source vertices. We run PR for 10 iterations.
4. **Connected Components (CC)**: CC labels all vertices by their connected component and each connected component is assigned its own unique label. If the graph is directed, we only require weakly connected components, so if two vertices are in the same connected component, it is equivalent to there being a path between the two vertices if the graph's edges are interpreted as undirected.
5. **Triangle Counting (TC)** TC computes the total number of triangles in a graph. A triangle is defined as three vertices that are directly connected to each other (clique of size 3). A triangle is invariant to permutation, so the same three vertices are counted as only one triangle no matter the order in which they are listed. The input graph is treated as undirected.

3.3 Algorithm Implementations

CombBLAS The CombBLAS implementation comes with applications for BFS and CC, so we implemented our own algorithms for SSSP, PR, and TC. For PR, we implement the standard power iteration method using repeated SpMV, following Algorithm 3 of (Kumar et al., 2018). For SSSP, we similiarly use the repeated SpMV formulation of Bellman-Ford in the min-plus tropical semiring. Finally, we implemented the lower-triangular masked SpGEMM algorithm for TC as described by Wolf et al. (Wolf et al., 2017).

Gemini. The Gemini implementation comes with applications for all tasks except TC. Unfortunately, *Gemini cannot to our knowledge support TC without significant modifications*. This is due to the following constraint: in Gemini, messages between vertices must have a fixed size. In TC, vertex-centric frameworks operate by sending neighborhood information to neighboring vertices, which is then intersected to count triangles. Since the neighborhood sizes vary, this cannot be implemented in Gemini without major reimplementation of the communication and storage codes.

3.4 Input Selection

For our evaluation, we conjectured that scaling experiments to large real-world graphs would reveal interesting insights about frameworks and paradigms. Our hypothesis originates from the fact that very large real-world graphs can reveal

[1]We opened an issue which received no reply: `https://github.com/IntelligentSoftwareSystems/Galois/issues/408`.

```
void SSSP(Mat &A, int64_t n, Vec &dist){
    dist = Vec(A.getcommgrid(), n, MAX_DIST);
    dist.SetElement(source, 0);
    for (int64_t i = 0; i < n - 1; i++){
        dist = SpMV<MinPlusSRing<double, double>>(A, dist);
    }
}

void TC(Mat &L){
    Mat Ltemp = L;
    Mat C = Mult_AnXBn_DoubleBuff<PlusTimesSRing<int64_t, int64_t>,
↪   int64_t,
        SpDCCols<int64_t, int64_t>>(L, Ltemp, false, true);
    C.EWiseMult(L, false);
    Vec triangles = C.Reduce(Column, plus<int64_t>(),
↪   static_cast<int64_t>(0));
    int64_t tc = triangles.Reduce(plus<int64_t>(),
↪   static_cast<int64_t>(0));
}
```

```
void PR(Mat &A, int64_t n, α){
    // ArithSR = ⟨ℝ,+,×,0⟩
    // err = inv_n = 1/n; p ← inv_n; od ← out degree of pages
    // b⟨od⟩ ← (1−α)/n, b⟨¬od⟩ ← 1/n
    // od_inv ← 1/od
    int iter = 0;
    while (err > eps && iter < max_iter){
        Vec p0 = p;
        Vec temp = b;
        temp.EWiseApply(p0, std::multiplies<double>());
        double t = temp.Reduce(std::plus<double>(), 0.0);
        p = t;
        p0.EWiseOut(od_inv, std::multiplies<double>(), temp);

        temp = SpMV<ArithSR>(A, temp);
        temp.Apply([](double x){ return x * alpha; });
        p += temp;

        // check convergence
        p.EWiseOut(p0, std::minus<double>(), temp);
        temp.Apply([](double x){ return x * x; });
        err = temp.Reduce(std::plus<double>(), 0.0);
        iter++;
    }
}
```

Figure 1: CombBLAS SSSP and TC Implementation

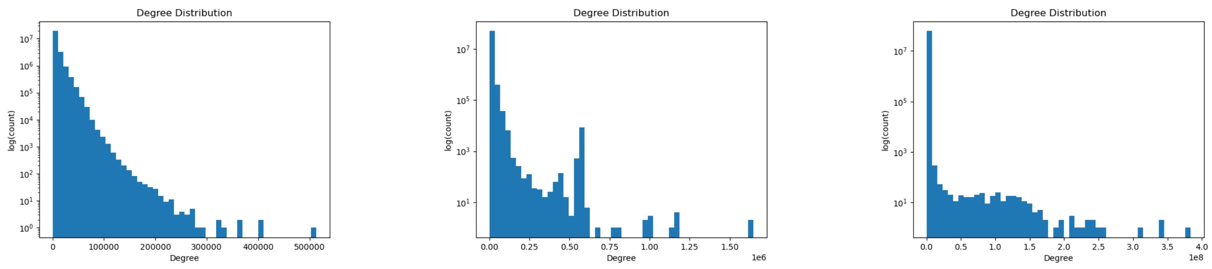


Figure 2: Degree distribution for (a) GAP-road, (b) GAP-web, and (c) GAP-twitter

bottlenecks in not just the compute operations, but also loading, intermediate results, communication, synchronization, and other memory/network related overheads that must be carefully managed to ensure scalability. Furthermore, these graphs may exhibit complex patterns of connectivity – such as community structures and small-world phenomena (Watts & Strogatz, 1998) – which can impact performance of different frameworks and algorithms. To perform this experiment, we picked 3 graphs from the Laboratory for Web Algorithmics: (1) *gsh-2015*, (2) *eu-2015*, and (3) *hyperlink2012* (Boldi & Vigna, 2004). Additionally, to perform quicker checks of basic performance characteristics and correctness, we also picked 3 real-world graphs from the GAP benchmark suite (Beamer et al., 2015).

Note: However, we encountered significant challenges with loading large graphs into memory, even while utilizing parallel loading strategies provided by the frameworks. Furthermore, we faced `Out of memory` errors when fitting some larger graphs into memory, even when utilizing more than 16 nodes. We discuss some of these issues as part of our performance analysis in Sections 5.1 and 5.2. As a result, we could run experiments on only the smallest of the three large web graphs (*gsh-2015*).

In the end, we were left with four input graphs for our experiments that are quite diverse in both topology and origin. They model connections between people, websites, and roads. The graph sizes were selected to be small enough to fit comfortably in memory yet large enough to reveal some performance characteristic differences between frameworks and paradigms.

1. **`GAP-twitter`**: ($|V| = 61.6M$, $|E| = 1.468B$, directed): graph of a social network topology. Since it comes from real-world data, it has interesting irregularities and a *skewed degree distribution* that can be a potential challenge for some implementations and frameworks.
2. **`GAP-web`**: ($|V| = 50.6M$, $|E| = 1.949B$, directed): graph of a web-crawl of the .sk domain (sk-2005). Despite its large size, it exhibits substantial locality due to its topology and *high average degree*.
3. **`GAP-road`** ($|V| = 23.9M$, $|E| = 58.3M$, directed): graph of distances of all of the roads in the USA. Although it is substantially smaller than the rest of the graphs, it has a *high diameter* which can cause some synchronous implementations to have long runtimes.

4. **gsh-2015**: ($|V| = 988M$, $|E| = 33.87B$, directed): graph of a large snapshot of the web taken in 2015 by BUbiNG starting from the site http://europa.eu/ without any domain restriction.

3.5 Additional Implementation Effort

In addition to the setup described above, we also had to make some additional changes.

Graph formats and loading We observed that virtually every graph processing framework requires a bespoke input format, typically a variation of the COO (triplets) format. For consistency, we opted to use the CombBLAS binary format, which is a binary COO format with a metadata header. We thus modified Gemini's parallel loading code to use this format as well. Finally, as all the large web graphs are stored in the compressed WebGraph format (Boldi & Vigna, 2004), we also wrote a converter that translates WebGraph files into the CombBLAS binary format.

Bug-fixing and improvements In order to run Gemini, we had to fix some compilation bugs. For CombBLAS, we noticed some issues that we worked around: (1) in-place permuting a matrix seems to give incorrect results with a single MPI rank (in the CC application), (2) using the add method of a semiring class to perform a reduction causes a segfault with mutiple MPI ranks, but using `std::plus` is fine, and (3) for faster debugging, we improved MatrixMarket loading to support transposing during load.

4 Evaluation

4.1 Setup and Methodology

In the following discussion, let N denote the number of machines (or "nodes"), T the number of MPI ranks *per node*, and C the number of threads (including hyperthreads) *per rank*. For our experiments, we evaluate N in powers of two. For CombBLAS, we use $T = 4$ and $C = 64$ when $4N$ is square (i.e., $N = 1, 4, 16, \dots$). Otherwise we use $T = 2$ and $C = 128$. This is because, CombBLAS expects a square logical processor grid (i.e., $\sqrt{TN} \times \sqrt{TN}$ where TN is the total number of MPI ranks). On the other hand, Gemini uses only one MPI rank per node (i.e., $T = 1$) with all threads utilized. Internally, Gemini also performs NUMA-aware allocation and chunking. Within a node, Gemini applies NUMA-aware sub-partitioning across multiple sockets: On Perlmutter, for each node containing 8 NUMA domains, the vertex chunk is further cut into 8 sub-chunks, one for each domain. Lastly, experiments are run for a maximum of one hour.

4.2 Results

4.2.1 GAP Benchmark Suite

While we aimed to identify performance characteristics on larger graphs, the GAP benchmark provided a quick smoke test to make observations on how the two frameworks scale on relatively small graphs. Here are some key observations from our experiments:

1. As shown in Figures 3 and 4, Gemini is significantly faster than CombBLAS when running on a single node.
2. However, the scalability of Gemini is generally poor as the number of nodes is increased. In fact, the runtime of Gemini tends to increase as more computing nodes are used.
3. Lastly, we observe that on the GAP-road graph, which is well-known to have a high diameter, Gemini has worse single-node and multi-node performance for connected components (CC).

4.2.2 Large Graphs

We observe in Figure 5 that on larger graphs, neither framework was able to successfully complete all tasks, although we tested on the smallest "large" graph (gsh-2015). CombBLAS only successfully runs CC (on 64 nodes and 256 nodes) and PR (on 64 nodes; 256 nodes raised an MPI error). The failures for both frameworks arise *mainly* due to Out-of-memory (OOM) errors while loading the graph. We discuss more on the cost of data loading in Section 5.1 and the memory footprint in Section 5.2. Specifically for Gemini, we observe that it segfaults when run with lesser number of nodes (likely OOM) and encounters an internal assertion error when run with higher number of nodes.

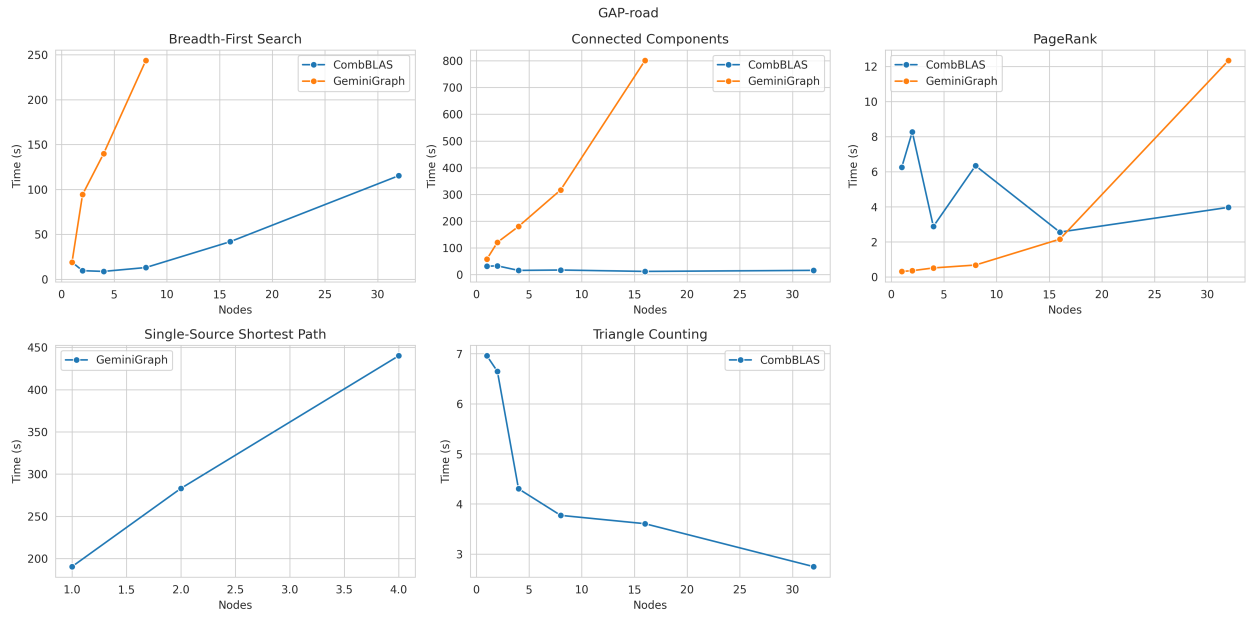
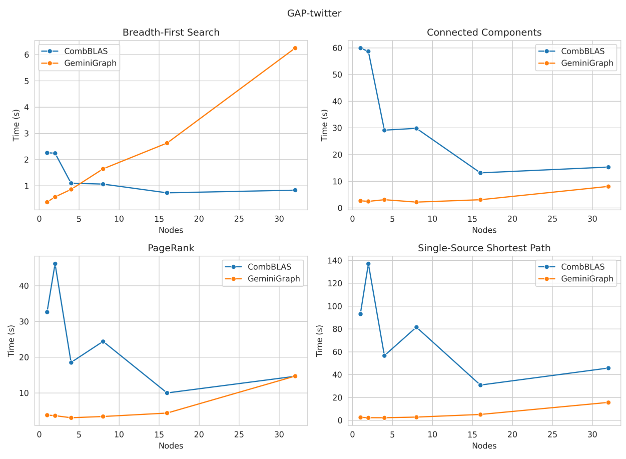
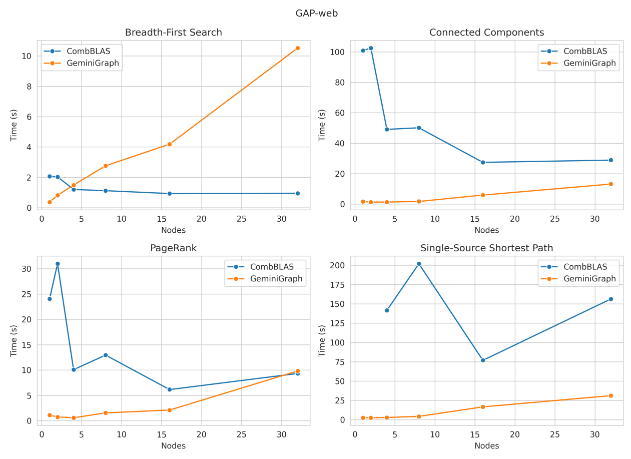


Figure 3: Runtime on GAP-road



(a) GAP-twitter



(b) GAP-web

Figure 4: Runtime on GAP-twitter and GAP-web

5 Discussion

In this section, we analyze the results and discuss possible causes.

5.1 Data Loading Cost

As Figure 6 shows, loading the input graph from disk to memory often makes up most of the time spent in a graph processing application. For example, CombBLAS spends 87% of time loading the `gsh-2015` graph on 256 nodes for CC. Considering the cost of running 256 nodes, this means **nearly 60 node hours** were used just to load the graph. This imbalance is even more pronounced on tasks that have locality (where only a subset of the graph is relevant), such as BFS on `GAP-twitter` where loading is 99% of the runtime.

One may ask if this is expected, and simply a result of the graphs themselves being large. Figure 8 shows the *effective bandwidth* achieved *globally across all nodes*. As we can see, both frameworks achieve poor effective bandwidth - CombBLAS reaches 800MB/s *total* effective bandwidth on only one benchmark configuration (across 32 nodes), while

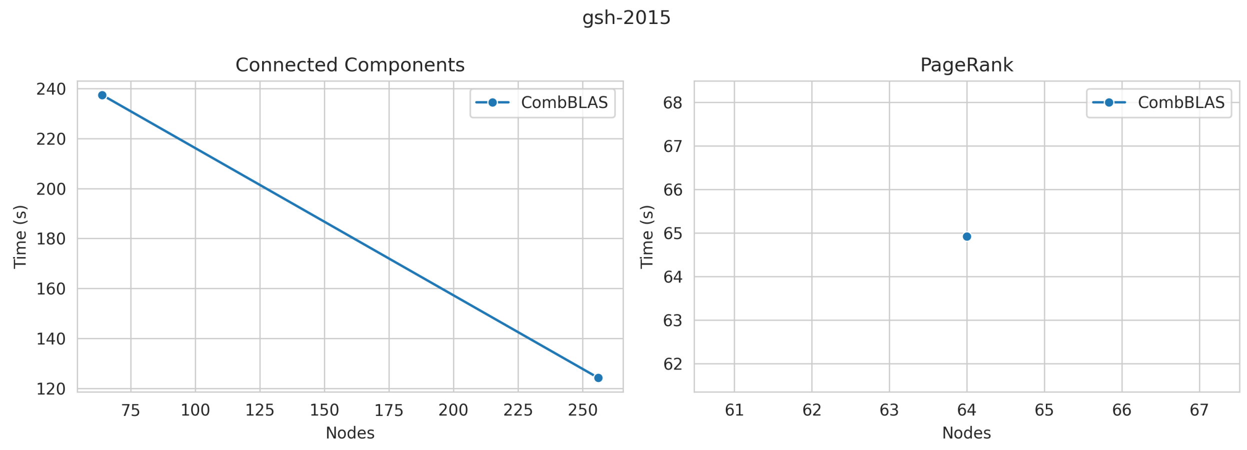
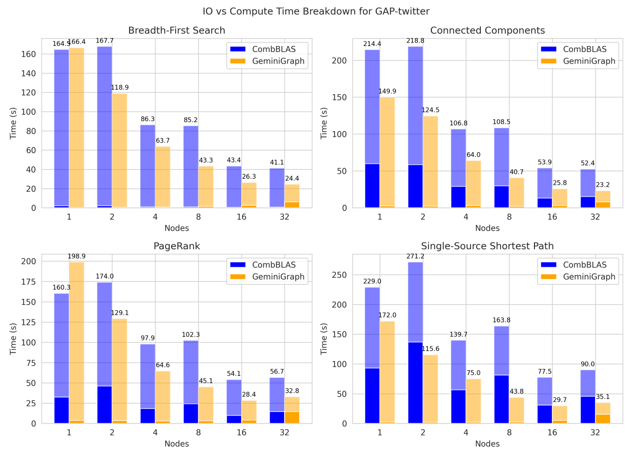
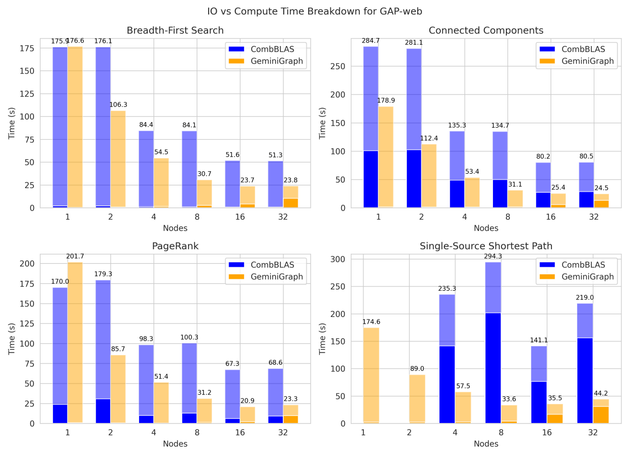


Figure 5: Runtime on gsh-2015



(a) GAP-twitter



(b) GAP-web

Figure 6: IO vs Compute Breakdown on GAP-twitter and GAP-web

Gemini is better but still suboptimal. Indeed, the flash storage used by Perlmutter supports up to 7GB/s of sequential read *per drive*.[2]

It is important to note that "loading" is really comprised of two interlinked tasks: (1) bringing the nonzeros (i.e. edges) into memory from disk, and (2) constructing the in-memory representation of the graph (which may involve inter-node communication depending on how the data is partitioned). Thus, improving loading performance will require carefully coordinating the two components.

5.2 Memory Footprint

In Table 1, we show the per-node peak memory usage as calculated from the CrayPat profiling tool. When executed on four nodes, CombBLAS shows a lower per-node memory usage of 8.61 GB compared to Gemini's incredibly

[2]See https://cug.org/proceedings/cug2021_proceedings/includes/files/pap120s2-file1.pdf.

Framework	CC ($N = 4$)	CC ($N = 8$)	BFS ($N = 4$)	BFS ($N = 16$)
CombBLAS	8.61	7.03	15.42	40.44
Gemini	166.54	229.77	30.30	25.35

Table 1: Per-node peak memory usage (GB), calculated from CrayPat profiling.

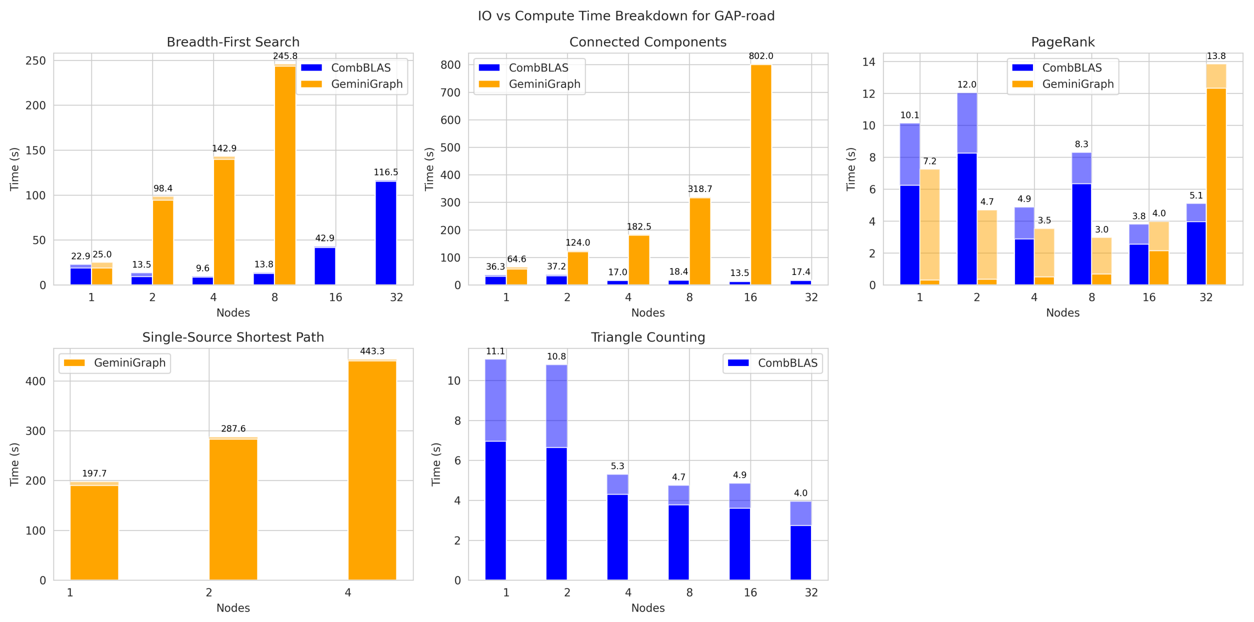
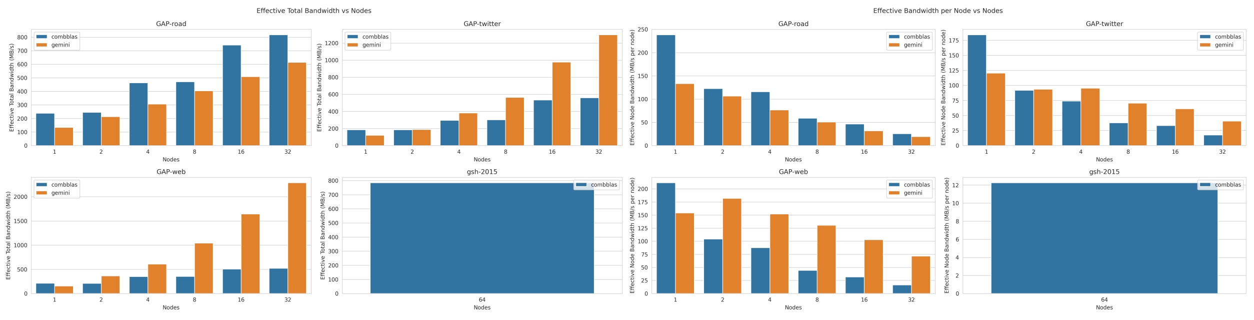


Figure 7: IO vs Compute Breakdown on GAP-road



(a) Total Bandwidth

(b) Bandwidth per node

Figure 8: Effective bandwidth vs. nodes, where effective bandwidth is defined as the time to load and construct the in-memory representation of the graph divided by on-disk filesize of the graph.

large 166.54 GB. However, when the number of nodes is increased to eight, CombBLAS's per-node memory usage reduces by 18%, while Gemini's increases by nearly 40%. Overall, for CC, CombBLAS is significantly more memory-efficient. However, for BFS, we see a completely flipped scenario. We think these results are due to the differences in implementations (CombBLAS provides highly custom implementations for both CC and BFS) and algorithms used by the two frameworks. Overall, these results suggest that the choice of algorithms and how they are implemented can greatly impact the memory usage for a task. We discuss this further in Section 6.

6 Takeaways and Future Directions

As this project has shown, running distributed graph processing applications at large scales is a challenging task. Many frameworks were not designed for extremely large graphs in mind (despite being distributed), and even when run successfully, have poor scaling and costly IO. In the remainder of this section, we look to the future and identify promising directions for robustifying large-scale distributed graph processing.

6.1 Optimizing Data Loading and Graph Construction

As discussed in Section 5.1, the cost of loading and constructing the in-memory graph representation is often extreme. One possible approach to solving this problem is trading off space for compute, by utilizing compression. In fact, the WebGraph format that gsh-2015 is distributed in uses only around 11GB of disk space, while the uncompressed binary triplets takes over 500GB. It would be entirely feasible to simply load the compressed graph into memory on *every node*, before locally decompressing the graph. However, the WebGraph format does not provide a way to extract equal-sized chunks of edge data, which makes parallel IO nontrivial. An impactful direction for future research is to design a compressed graph format designed with large scale parallel processing in mind.

Another challenge comes from the initial partitioning cost, where the edges are (re)distributed across the nodes. A partitioning-aware graph format would be of high value.

6.2 Separation of Concerns

When implementing a graph processing application, there are at least 4 things the developer must keep in mind:

1. the high-level graph task (e.g. BFS, CC, etc.);
2. the partitioning/sharding of the graph across the available logical/physical processors;
3. the *algorithm* used to solve the task (e.g. Bellman-Ford for SSSP);
4. and the *low-level implementation* of the algorithm (e.g. SpGEMM for TC, but using SUMMA internally).

The task determines the algorithms, which in turn determines the set of implementations. However, the implementation also constrains the partitioning strategy, making modular design difficult. In the case of CombBLAS, the low-level implementations are fixed, and this is reflected in the fixed partitioning strategy (square grid, fixed nonzero distribution).

We believe linear-algebraic graph processing is well suited to address this difficulty. Indeed, a major motivation of GraphBLAS is to separate the algorithm from the low-level implementation by a standardized API. Another promising direction is that of sparse tensor algebra (beyond matrices), which has been explored by the TACO line of work (Henry et al., 2021). Recently, SpDISTAL (Yadav et al., 2022) has succesfully formulated sparse tensor partitioning as a *scheduling language*, complementing their existing scheduling approach for lowering shared-memory kernels for sparse tensor algebra. Then, we might imagine writing a TC application by specifying the computation

$$C(i,j) = L(i,k) \cdot L(k,j) \cdot L(i,j),$$

together with a set of schedules that would correspond to different variations of masked SpGEMM. Perhaps these schedules could even be automatically synthesized by searching over possible partitioning strategies and local schedules.

References

Yousuf Ahmad, Omar Khattab, Arsal Malik, Ahmad Musleh, Mohammad Hammoud, Mucahid Kutlu, Mostafa Shehata, and Tamer Elsayed. La3: A scalable link- and locality-aware linear algebra-based graph analytics system. *Proc. VLDB Endow.*, 11(8):920–933, apr 2018. ISSN 2150-8097. doi: 10.14778/3204028.3204035. URL https://doi.org/10.14778/3204028.3204035.

Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite, 2015. URL https://arxiv.org/abs/1508.03619.

Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pp. 595–601, Manhattan, USA, 2004. ACM Press.

Aydın Buluç and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

*Roshan Dathathri, *Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali (*Both authors contributed equally). Gluon: A Communication Optimizing Framework for Distributed Heterogeneous Graph Analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pp. 752–768, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192404. URL http://doi.acm.org/10.1145/3192366.3192404.

Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. Compilation of sparse array programming models. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021. doi: 10.1145/3485505. URL https://doi.org/10.1145/3485505.

Jannis Koch, Christian L. Staudt, Maximilian Vogel, and Henning Meyerhenke. An empirical comparison of big graph frameworks in the context of network analysis, 2016. URL https://arxiv.org/abs/1601.00289.

Manoj Kumar, José E Moreira, and Pratap Pattnaik. Graphblas: Handling performance concerns in large graph analytics. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pp. 260–267, 2018.

Nadathur Satish, Narayanan Sundaram, Md Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 979–990, 2014.

Narayanan Sundaram, Nadathur Rajagopalan Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *arXiv preprint arXiv:1503.07241*, 2015.

Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

Michael M Wolf, Mehmet Deveci, Jonathan W Berry, Simon D Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with kokkoskernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7. IEEE, 2017.

Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. Spdistal: Compiling distributed sparse tensor computations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022. ISBN 9784665454445.

Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *OSDI*, volume 16, pp. 301–316, 2016.