PARALLEL IMPLEMENTATION OF GRAPH ENCODER EMBEDDING IN THE LIGRA GRAPH FRAMEWORK

Ariel Lubonja

Johns Hopkins University
Dept. of Applied Mathematics and Statistics
Baltimore, MD, USA
alubonj1@jhu.edu

Randal Burns

Johns Hopkins University
Department of Computer Science
Baltimore, MD, USA
randal@cs.jhu.edu

ABSTRACT

In this paper we show how graph-parallel execution can deliver order of magnitude speedups to a linear time graph embedding task with few lines of code and little changes to the serial implementation. The target method is Graph Encoder Embedding, a graph encoder that converges asymptotically to the Spectral Embedding, while maintaining a linear runtime in the number of edges. Despite this linear complexity, it does not scale to large graphs due to limitations of running on a single-thread. We observe that this Encoder is very suitable to implementation in the Ligra graph-parallel framework and we contribute a fast, highly parallelizable implementation of Adjacency Matrix version of GEE and report on the speedup results. We also contribute a review of Dynamic graph algorithms. These algorithms study a graph not only along the spatial domain, but also across the temporal one. Measuring how attributes such as a node's importance changes as edges are added or removed, and generating insights from this information is a research area that is relatively unexplored, considering how ubiquitous graphs are in Computer Science. We will describe the motivation, challenges, approaches taken to overcome them, and promising future work in this area.

Index Terms— Graph Algorithms, Graph Embedding, Spectral Embedding, Temporal Graphs, Parallelism, Systems

1. INTRODUCTION

A graph $G(n,s)$ is a mathematical object that consist of a set of nodes n and edges s that connect these nodes. They are a concise way of capturing relationships between objects, and occur in many domains such as social networks, power grids, road, communication and citation networks, etc. Recently, the field of Graph Neural Networks is using graphs to get state of the art performance in many applications [1, 2]. However, two main challenges to learning on graphs exist. From a systems perspective, any operation involving graphs has to face the challenge that their binary adjacency matrix representation, which reveals whether two nodes are directly linked with an edge is $\Theta(V^2)$ in memory, and therefore infeasible to store for all but the smallest graphs. From a learning perspective, the challenge centers around incorporating existing Machine Learning methods, that are developed within domains where Euclidean distance is defined, to graphs, whose structure gives no indication as to the L_2 distance between any pair of nodes [2].

Graph Embedding is a popular solution to both these problems. It consists of building low-dimensional feature representations of the graph's nodes while encapsulating as much structural information (neighborhoods, shortest paths, centrality, etc.) as possible from the graph. Various embedding methods have been proposed: Spectral Clustering [2] is perhaps the most studied, and Deep Learning approaches such as DeepWalk [3], node2vec [4] and Graph Convolutional Networks (GCN) [5] are a more recent development.

We will focus our attention on a Spectral Embedding method called Graph Encoder Embedding (GEE) [6]. This approach addresses the computational concern of other embedders: many, such as spectral embedding and GCN run in $O(n^2)$ time, and others, like node2vec, run in $O(n)$ but have large hidden constants, making them impractical on consumer-grade PCs [6]. GEE proposes an $O(nK+s)$ algorithm for node embedding where s is the number of edges, n is vertices, and K is the number of classes we choose to project down to. Since in virtually all graphs $nK \ll s$, this algorithm is practically always linear in the number of edges. As a result, the authors of GEE observe a runtime of 10 minutes for a billion-edge graph.

Many of today's graphs, however, regularly run in the hundreds of billions of edges [7–9]. Further complicating matters are Dynamic Graphs - a graph $G(n,s)$

with a list of changes l at time t_l. Embedding these types of graphs requires computing the embedding several times along a graph's history, or alternatively incrementally updating the embedding for each change in the graph, requiring keeping the embedding in memory at all times.

These two approaches represent the fundamental tradeoff in Dynamic graphs analysis: computation vs. memory footprint [10]. Methods to optimizing memory footprint are discussed in Section 5. Sections 2-4 are dedicated to the core issue of the first approach: the long running time of GEE for big graphs, and how it can be addressed using graph parallelism. Section 2 describes the relevant work - Graph Encoder Embedding and the Ligra [11] graph framework as well as optimizations we made in our implementation. Section 3 describes how we ran our tests and 4 presents the results and speedup observed from parallelizing GraphEncoderEmbedding, and limitations. Section 5 presents our Review work in the field of Dynamic graph algorithms.

2. PARALLELIZING GRAPH ENCODER EMBEDDING

We observe that Graph Encoder Embedding runs on all the edges of the graph independently, making it very suitable to parallel execution. To exploit this parallelism, we utilize the Ligra graph framework [11]. Ligra allows parallelism over both vertices and edges of a graph by exposing a user-defined EdgeMap and a VertexMap function, and a frontier of execution - a set of vertices or edges the corresponding function will be applied to.

2.1. Brief Description of the Graph Encoder Embedding (GEE) Algorithm

The semi-supervised GEE algorithm is shown in Algorithm 1. Since we're only implementing the adjacency matrix version, the only relevant lines are 14-24. The algorithm begins by counting the frequency n_k of each label in $\mathbf{Y}$ and initializes the corresponding nodes' embeddings in the projection matrix $\mathbf{W}$ to $\frac{1}{n_k}$ (lines 14-18). The $\mathbf{Z}$ embedding matrix is incrementally constructed by iterating over the edges (u, v) of the graph, adding the frequency of the destination node v's true label times the edge weight ($\mathbf{W}(v, \mathbf{Y}(v)) \cdot w$). Note $\mathbf{W}(v, \mathbf{Y}(v)) \in \{0, \frac{1}{n_k}\}$, so lines 22-23 only update $\mathbf{Z}$ if v's true label is known, and this happens in both directions of an edge for undirected graphs. Because $\mathbf{Z}$ is updated through addition, a commutative operation, we are not constrained to iterating over the edges in line 19 in any certain order. The loop at line 19, which constitutes the vast majority of the algorithm's runtime, can be safely run in parallel without compromising correctness. The resulting embedding vector $\mathbf{Z}$, which is a linear projection of the graph's adjacency matrix onto the span of $\mathbf{W}$, will be identical to the serial implementation.

Algorithm 1: Semi-Supervised Graph Encoder Embedding

input : $\mathbf{E} \in \mathbb{R}^{s\times 3}$: edge triples (source, destination, weight)
$\mathbf{Y} \in \{0, \dots K\}^n$: Ground truth of node embeddings (if available)
Lap: Boolean variable whether Adjacency Matrix or Graph Laplacian is to be used (optional)

output:
$\mathbf{Z} \in \mathbb{R}^{K\times n}$: node embeddings matrix
$\mathbf{W} \in \mathbb{R}^{n\times K}$: projection matrix

```
1  Function GEE(E, Y, Lap):
2      W = zeros(n, K)
3      if Lap == True then
4          D = zeros(n, 1); // Degree vector
5          for i=1:s do
6              D(E(i,1)) = D(E(i,1)) + 1;
7              D(E(i,2)) = D(E(i,2)) + 1;
8          end
9          D = D^{-0.5}; // Elementwise Sqrt.
10         for i = 1 : s do
11             E(i,3) = E(i,3) · D(E(i,1)) · D(E(i,2));
12         end
13     end
14     for k = 1 : K do
15         ind = find(Y = k);
16         n_k = sum(ind); // Count indices of class k
17         W(ind, k) = 1/n_k;
18     end
19     for i = 1 : s do
20         // (u-source, v-dest., w-weight)
21         u = E(i,1); v = E(i,2); w = E(i,3);
22         Z(u, Y(v)) += W(v, Y(v)) · w;
23         Z(v, Y(u)) += W(u, Y(u)) · w;
24     end
25 EndFunction
```

2.2. Ligra and the EdgeMap Model

Having established that GEE computation can be parallelized, we will use Ligra [11] to take advantage of this parallelism. Ligra is a graph-parallel framework that exploits the frontier-based nature of many graph algorithms. For example, Fig. 1 shows a simple Breadth-

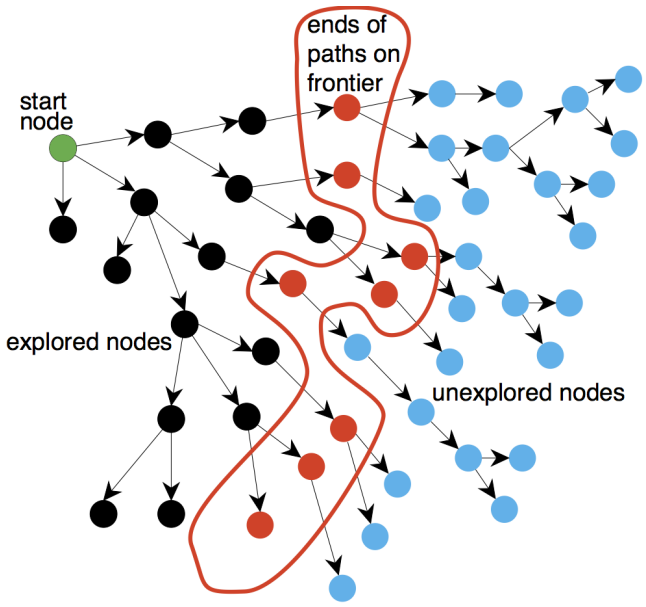

Fig. 1. Frontier nature of Breadth-First Search. Image credit: artint.info

First Search implementation. Initially, all nodes start off in the set of unexplored nodes (blue), with the chosen starting point in the "to be explored" active set (green). After one iteration, the starting node is put in the visited set (black), and its unseen but reachable neighbors are added to the active set (red), iterating until all nodes reachable from the starting node have been visited.

One can observe that, at any iteration of the algorithm, only the outgoing edges of the nodes in the active set will be used, and the other nodes and edges are not involved in the calculation. This active set of graph entities is called the *frontier*. One could parallelize each iteration of BFS by mapping the computation over the set of outgoing edges of the nodes in the active set across multiple processing units. Some caution must be taken to deal with race conditions, as two edges from different sources might arrive at the same node simultaneously, and a decision must be made as to which source node will be chosen as the parent.

Ligra's authors observed that many graph algorithms have this frontier-based nature and they developed a graph framework that exposes an EdgeMap and a VertexMap function to the user. The user needs to define the algorithmic logic that applies across a set of Edges or Vertices by filling in the implementation of the corresponding function. The user also need to define and maintain the frontier the chosen function will be applied to, and deal with race-conditions and other potential correctness issues arising from an arbitrary order of execution. The benefit of using Ligra is often an order of magnitude or more speedup over most serial algorithms. The actual observed speedup is determined by the underlying hardware in terms of memory and CPU core count, the efficiency of the user's implementation, and by Amdahl's Law.

2.3. Implementation

Graph Encoder Embedding's semi-supervised algorithm, when viewed from the EdgeMap point of view, functions very similarly to PageRank. At each iteration, PageRank processes all the edges of the graph. Since PageRank is a property of the graph's nodes, not edges, the algorithm then pushes or pulls (depending on the implementation) the result computed over edges into the nodes. This is exactly what GEE does in lines 14-18. The main difference is that, in contrast to PageRank which takes multiple iterations to converge, GEE iterates only once over the graph's edges. Since Ligra already provides a PageRank implementation, we use it as a starting point and morph it into GraphEncoderEmbedding. In the EdgeMap model, GEE is an algorithm with all the graph's edges in the active frontier.

2.4. Optimizations

Various optimizations were implemented to further speed up computation. Ligra is written in C++ - a high-level compiled language. This makes it compare favorably to interpreted languages like Python and MatLab, since the compiler can introduce optimizations into the code such as unrolling loops, removing unnecessary variables and unused code branches etc. Crucially, Clang allows for OpenMP [12] support, which enables parallel execution across CPU cores.

Further, to maximize cache utilization, we flattened the matrices $\mathbf{Z}, \mathbf{W}$ into vectors. Because C is a row-major order language, matrices are accessed row-by-row, and switching rows would incur a cache miss. $\mathbf{Z} \in \mathcal{R}^{K \times n}$ and $\mathbf{W} \in \mathcal{R}^{n \times K}$, so one row would be either $O(n)$ or $O(K)$, both far shorter than a CPU cache line (15MB in our machine), so the algorithm would cost either $O(K)$ or $O(n)$ cache misses. Since accessing an item from main memory is orders of magnitude slower than from cache, this severely hampers performance. By flattening these matrices into a single long row vector, we can fill the cache with either matrix, minimizing cache misses and maximizing the throughput of the processor. Despite this, our code maintains the simplicity of much Ligra code: the nontrivial parts run to less than 50 lines.

3. SETUP FOR BENCHMARKS

In order to benchmark the speedup we expect, we ran both the original GEE code and our Ligra implemen-

tation on graphs of various sizes, up to the limit our machine's memory would allow. We ran Ligra in both serial and parallel fashion, to isolate the effects of using C++ and vectorizing, from that of scaling to many processors. Results on the most powerful server-grade machine, with 2 Intel Xeon E5-2420v2 CPUs, each of 6 cores and 12 threads, and 32GB of shared RAM, are presented in this section, using the fastest available GEE code - its MatLab version. Benchmarks run on a consumer-grade desktop computer and on the slower Python version of GEE can be viewed in Appendix A.2.

Note that our Ligra implementation assumes an undirected and unweighted graph even if the graph is otherwise: we preprocess the input graph to drop the weights are dropped if they exist, and edges are set to be bidirectional (redundancy caused by directed graphs on this is accounted for). The implementation can trivially be modified to handle weighted graphs, since the code simply assumes the weights are set to 1 - one only need initialize this weight vector with the desired edge weights. Implementing directed graphs is slightly trickier, since one has to make sure to not perform the calculation twice on bidirectional edges. However, this is irrelevant for the purposes of this benchmarks, as both implementations iterate over the exact same number of edges.

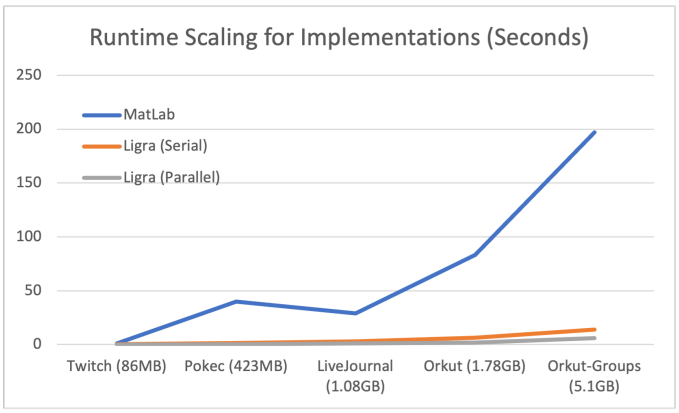

Fig. 2. Relative time for the 3 implementations in order of Graph Size

Implemen. (seconds) / Graph	GEE-MatLab	Ligra Serial	Ligra Parallel
Twitch (86MB)	0.91	0.17	0.038
soc-Pokec (423MB)	39.7	1.32	0.37
soc-LiveJournal (1.08GB)	29	2.87	1.11
soc-orkut (1.78GB)	83.22	6.35	1.71
orkut-groups (5.1GB)	197	14	5.9

Table 1. Runtimes in seconds for graphs of various sizes

4. FINDINGS

We tested on graphs of various sizes and found that the runtime of all implementations of GEE scales linearly with the number of edges of the graph as expected (Table 1), however, the runtime of both the serial and parallel Ligra implementations is well ahead of the paper's MatLab implementation. We observe an order of magnitude improvement going from MatLab to a single-core Ligra implementation: this can most likely be attributed to the benefits of vectorizing the matrices, and using a compiled language. We see a further 2.5-4.5x improvement when comparing the serial version of Ligra to the parallel implementation.

Using the largest graph our machine could load: orkut-groups, we managed to show how the runtime can be reduced from more than 3 minutes in MatLab to around 6 seconds, a 33x speedup. The improvement versus the MatLab version for the soc-Pokec graph is more than 2 orders of magnitude.

4.1. Application to Temporal Graph Analysis

In a temporal graph setting, where one would need to run such algorithms repeatedly after a certain amount of changes has been made (i.e. a checkpoint), this runtime difference becomes emphasized. For example, if one were to need to run this algorithm 100 times, the time to run the MatLab version balloons to 6h, whereas the parallel Ligra version is under 10m. At this point, loading the graph's checkpoints from disk, or incrementally maintaining them becomes the limiting factor to performance. In Section 5, we include a review of current approaches to this problem and discuss their relative strengths and weaknesses.

4.2. Factors against Parallelism

The benefits of parallelism in our case are counterintuitively most pronounced when the graphs are small. We believe this is most likely because our Ligra implementation unfortunately needs to load the $\mathbf{Y}$ labels from disk every time it is run, unlike the MatLab version. Even though $\mathbf{Y} \in O(n)$, though much smaller than the edgelist $O(s)$, can still be up tens of MB in size for bigger graphs, which requires a non-insignificant load time on a mechanical hard drive. Fixing this requires changes to the internal workings of the Ligra engine, which is impractical for our purposes. Other factors such as the computation required to distribute and collect work to and from all cores and collect them, the limited mem-

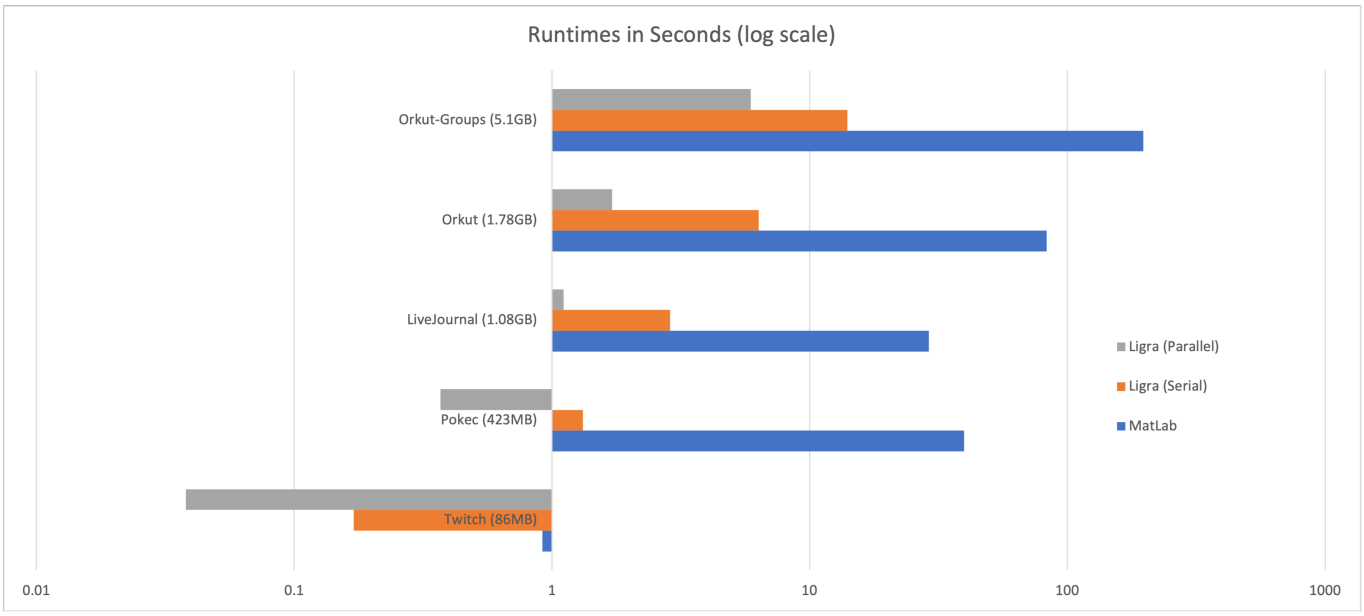

Fig. 3. Log-Scale plot of the improvement in runtimes observed by switching to Ligra

ory bandwidth of the machine, and Amdahl's Law contribute to us only seeing a fraction of the possible 12x speedup expected when running on 12 cores.

5. TEMPORAL GRAPH ALGORITHMS REVIEW

5.1. Motivation

This project started off as a literature review of Temporal Graph Algorithms. Although graphs and graph algorithms are ubiquitous in the fields of Computer Science and Applied Mathematics, less work has been dedicated to graphs that evolve over time. This is curious because the definitive examples of a graph: social networks, road and citation networks, electricity grids etc. are not static.

We should emphasize that we are not referring to incremental or online algorithms that focus on computing a target value once, then saving computation by updating this value as new data is introduced, usually with complexity proportional to the amount of new data. Examples of incremental work on graphs include [13–22].

Static graph algorithms extract information from the structure of a graph: node importance metrics such as Betweenness Centrality, PageRank, distance measures such as Bellman-Ford and Dijkstra, Max-Flow Min-Cut which is popular in the Computer Vision community etc. all take advantage of the spatial relations in the graph. However, Dynamic or Temporal graphs introduce the time-domain component: how the centrality of nodes or graph diameter changes as edges are added or deleted, how communities are formed or broken up etc. Algorithms in this domain have sparked considerable recent interest [23–30]. However, in order to run such algorithms across a graph's history, an efficient memory layout of must be created and maintained. This task is abstracted away by dynamic graph frameworks. Because of the large size of graphs and their histories, one cannot keep multiple copies in memory to make them readily accessible. As a result, dynamic graph engines suffer from a fundamental tradeoff in access speed versus memory usage.

5.2. Challenges

This tradeoff stems from the fact that the two most straightforward ways to store dynamic graphs is either as the graph $G(n,s)$ at time t_0 and a list of l changes corresponding to times $t_0,\ldots,t_l$, or as a full copy of the entire graph at every time point $G_0,\ldots G_l$.

The latter is ideal in terms of computation - any version of the graph is available at any time, but, storing multiple copies in memory is impossible with any large-scale problem, and results in much duplication. The more granular the graph history, the smaller the changes as a proportion to the overall graph, leading to a large amount of duplication.

The former is optimal in terms of storage since it stores the minimum amount of information to reconstruct the graph at any point along its history. However reconstructing the graph at any point involves re-

building the history of the graph since the start up to the desired point in time. This is a linear-time scan of the list of changes and is computationally prohibitive since analyzing the change in a graph is most likely to prove insightful when studied across a large number of snapshots, with sufficiently granular changes to the graph.

An ideal solution would allow for the execution of time-continuous queries - any granularity of changes across any arbitrary window or multiple windows in the graph's history. Then, the change in metrics such as Betweenness Centrality, Connected Components, PageRank etc. can be expected to reveal interesting insights.

5.3. Current Approaches - Dynamic Graph Frameworks

Graph Engines that attempt to solve this problem can be classified into those that accept real-time updates to the graph, and those that do not. The second group assumes that the graph and the whole graph history is available ahead of time. These involve works such as ImmortalGraph, and Chronos [31, 32].

In the first group, two approaches stand out and are our focus as of writing: Aspen [33] and TEGRA [10]. Aspen is a single-machine graph framework that extends Ligra and uses a functional compressed tree structure to handle streaming updates and simultaneously run queries. Aspen adds the user-defined Update function to Ligra that is executed as graph data is fed into the system. Every time a property is changed, the functional tree is traversed, new nodes are created along the traversed path and linked to the unchanged part of the tree. These trees are kept in-memory until any queries that are running on them finish executing. Because each of these trees takes $O(\log n)$ space in memory, this approach relies on aggressively garbage collecting old trees. For this reason, Aspen supports analysis only on the latest version of the graph, which doesn't fulfil our criteria of arbitrary time-history queries.

TEGRA shares some similarities with Aspen but is different in two crucial ways: it supports arbitrary time queries, and it is geared towards distributed computing. It also uses a tree-based structure - a Persistent Adaptive Radix Tree which shares the drawback of Aspen, mitigated to a certain extent by inferring the needed granularity of changes from a user's query. The unnecessary parts of the graph history are instead stored as leaves to disk. With the observation that users tend to run multiple queries along relatively small windows of snapshots, TEGRA also utilizes an incremental computation model. However, the leaves need to be reloaded and graph rebuilt if higher change granularity is required. Therefore, there is a tradeoff between memory usage and granularity of visible changes.

6. CONCLUSION

We demonstrated how GraphEncoderEmbedding's frontier nature can be exploited for graph-parallel execution and used the Ligra framework's EdgeMap interface to implement a significantly faster version of the algorithm. We empirically benchmarked the alternative implementations on graphs of various sizes and reported on the results. We have made the code available on GitHub. We also contributed a review of dynamic graph algorithms, describing their potential utility, challenges in execution, dynamic graph engine research, and the tradeoff between access speed and memory efficiency.

7. APPENDIX

A. BENCHMARKS

A.1. Description of Graphs Used

All graphs with the exception of orkut-groups were sourced from [34]. Orkut-groups was sourced from [35]. GEE has a single parameter - the desired dimension of the embedding K. Values used are shown in Table 2. Class labels were generated using the utility provided by GraphEncoderEmbedding and used as ground-truth for the semi-supervised GEE algorithm. 90% of the labels were deleted uniformly at random.

Graph Name	N. Vertices n	N. Edges s	N. classes K	Size
Twitch	168K	6.8M	20	86MB
soc-Pokec	1.6M	30.6M	50	423MB
soc-LiveJournal	4.8M	68.9M	50	1.08GB
soc-orkut	3M	117M	50	1.78GB
orkut-groups	8.7M	327M	40	5.1GB

Table 2. Size description of graphs used for benchmarking

A.2. Results on a Consumer grade Intel i7-4770 and GEE-Python

We found that the performance of GEE on Python was slower than expected. After contacting the author, we ran the tests on MatLab 2022a which significantly improved performance. The results can be seen in Table 1. Here we present the original runtimes run on the Python version of GEE on a commodity Intel i7-4770 desktop computer. Because this machine has less RAM than our server machine, we could not manage to run the orkut-groups graph.

Implemen. (seconds) / Graph	GEE-Python	Ligra Serial	Ligra Parallel
Twitch (86MB)	29.5	0.3	0.014
soc-LiveJournal (1.08GB)	298 (4:58)	2.06	1.23
soc-orkut (1.78GB)	483 (8:03)	4.23	1.86

Table 3. Runtimes in seconds and minutes a commodity desktop machine

B. REFERENCES

[1] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andrew J Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis, "Highly accurate protein structure prediction with AlphaFold," *Nature*, vol. 596, no. 7873, pp. 583–589, Aug. 2021.

[2] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[3] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, USA, 2014, KDD '14, p. 701–710, Association for Computing Machinery.

[4] Aditya Grover and Jure Leskovec, "node2vec: Scalable feature learning for networks," 07 2016, vol. 2016, pp. 855–864.

[5] Thomas Kipf and Max Welling, "Semi-supervised classification with graph convolutional networks," *ArXiv*, vol. abs/1609.02907, 2017.

[6] Cencheng Shen, Qizhe Wang, and Carey E. Priebe, "Graph encoder embedding," 2021.

[7] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue B. Moon, "What is twitter, a social network or a news media?," in *WWW '10*, 2010.

[8] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer, "The graph structure in the web - analyzed on different aggregation levels," *J. Web Sci.*, vol. 1, pp. 33–47, 2015.

[9] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *Proc. VLDB Endow.*, vol. 11, no. 4, pp. 420–431, dec 2017.

[10] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica, "TEGRA: Efficient Ad-Hoc analytics on evolving graphs," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. Apr. 2021, pp. 337–355, USENIX Association.

[11] Julian Shun and Guy E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *SIGPLAN Not.*, vol. 48, no. 8, pp. 135–146, feb 2013.

[12] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[13] Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis, "Parallel algorithm for incremental betweenness centrality on large graphs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 659–672, 2018.

[14] Nicolas Kourtellis, Gianmarco De Francisci Morales, and Francesco Bonchi, "Scalable online betweenness centrality in evolving graphs," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, 2016, pp. 1580–1581.

[15] Rishi Singh, Keshav Goel, Sudarshan Iyengar, and Sukrit Gupta, "A faster algorithm to update betweenness centrality after node alteration," 12 2013, vol. 11.

[16] Min-Joong Lee, Sunghee Choi, and Chin-Wan Chung, "Efficient algorithms for updating betweenness centrality in fully dynamic graphs," *Inf. Sci.*, vol. 326, no. C, pp. 278–296, jan 2016.

[17] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard, "Differential dataflow," .

[18] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos, "Facilitating real-time graph mining," in *Proceedings of the Fourth International Workshop on Cloud Data Management*, New York, NY, USA, 2012, CloudDB '12, p. 1–8, Association for Computing Machinery.

[19] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen, "Kineograph: Taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM European Conference on Computer Systems*, New York, NY, USA, 2012, EuroSys '12, p. 85–98, Association for Computing Machinery.

[20] Hongyang Zhang, Peter Lofgren, and Ashish Goel, "Approximate personalized pagerank on dynamic graphs," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, USA, 2016, KDD '16, p. 1315–1324, Association for Computing Machinery.

[21] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse, "Towards large-scale graph stream processing platform," in *Proceedings of the 23rd International Conference on World Wide Web*, New York, NY, USA, 2014, WWW '14 Companion, p. 1321–1326, Association for Computing Machinery.

[22] Keval Vora, Rajiv Gupta, and Guoqing Xu, "Synergistic analysis of evolving graphs," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 4, oct 2016.

[23] Ahmad Alsayed and Desmond J. Higham, "Betweenness in time dependent networks," *Chaos, Solitons & Fractals*, vol. 72, pp. 35–48, 2015, Multiplex Networks: Structure, Dynamics and Applications.

[24] Dylan Walker, Huafeng Xie, Koon-Kiu Yan, and Sergei Maslov, "Ranking scientific publications using a model of network traffic," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2007, no. 06, pp. P06010–P06010, jun 2007.

[25] Derek Greene, Dónal Doyle, and Pádraig Cunningham, "Tracking the evolution of communities in dynamic social networks," in *2010 International Conference on Advances in Social Networks Analysis and Mining*, 2010, pp. 176–183.

[26] Jaewon Yang and Jure Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, 05 2012.

[27] William Brendel, Mohamed Amer, and Sinisa Todorovic, "Multiobject tracking as maximum weight independent set," in *CVPR 2011*, 2011, pp. 1273–1280.

[28] David Gleich and Ryan Rossi, "A dynamical system for pagerank with time-dependent teleportation," *Internet Mathematics*, vol. 10, 11 2012.

[29] Polina Rozenshtein and Aristides Gionis, "Temporal pagerank," in *Machine Learning and Knowledge Discovery in Databases*, Paolo Frasconi, Niels Landwehr, Giuseppe Manco, and Jilles Vreeken, Eds., Cham, 2016, pp. 674–689, Springer International Publishing.

[30] Elise Henry, Loïc Bonnetain, Angelo Furno, Nour-Eddin El Faouzi, and Eugenio Zimeo, "Spatio-temporal correlations of betweenness centrality and traffic metrics," in *2019 6th International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*, 2019, pp. 1–10.

[31] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen, "Immortalgraph: A system for storage and analysis of temporal graphs," *ACM Trans. Storage*, vol. 11, no. 3, jul 2015.

[32] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen, "Chronos: A graph engine for temporal graph analysis," in *Proceedings of the Ninth European Conference on Computer Systems*, New York, NY, USA, 2014, EuroSys '14, Association for Computing Machinery.

[33] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun, "Low-latency graph streaming using compressed purely-functional trees," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2019, PLDI 2019, p. 918–934, Association for Computing Machinery.

[34] Jure Leskovec and Andrej Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, June 2014.

[35] Ryan A. Rossi and Nesreen K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.