Edge-Parallel Graph Encoder Embedding

Ariel Lubonja
Department of Computer Science
Johns Hopkins University
ariel@cs.jhu.edu

Cencheng Shen
Department of Applied Economics and Statistics
University of Delaware
shenc@udel.edu

Carey Priebe
Department of Applied Mathematics and Statistics
Johns Hopkins University
cep@jhu.edu

Randal Burns
Department of Computer Science
Johns Hopkins University
randal@cs.jhu.edu

***Abstract*—New algorithms for embedding graphs have reduced the asymptotic complexity of finding low-dimensional representations. One-Hot Graph Encoder Embedding (GEE) uses a single, linear pass over edges and produces an embedding that converges asymptotically to the spectral embedding. The scaling and performance benefits of this approach have been limited by a serial implementation in an interpreted language. We refactor GEE into a parallel program in the Ligra graph engine that maps functions over the edges of the graph and uses lock-free atomic instrutions to prevent data races. On a graph with 1.8B edges, this results in a 500 times speedup over the original implementation and a 17 times speedup over a just-in-time compiled version.**

***Index Terms*—graph embedding, graph processing, parallel programming**

I. Introduction

Graph embedding is a powerful technique for exploring the structure of graphs. It is the fundamental step in clustering [1], [2], feature learning [3], and representation learning [4] on graphs. It is used in the analysis of connectomes [5], cybersecurity threat detection [6], and community detection in social networks [7].

Spectral embedding learns a low-dimensional euclidean representation of a graph [5], [8] based on a singular-value decomposition (SVD) graph adjacency or graph Laplacian matrix. Spectral embedding has strong statistical guarantees; the resulting vertex embedding asymptotically converges to the latent positions under random dot product graphs [9] and, thus, is consistent for subsequent inference tasks such as hypothesis testing and community detection. However, spectral embedding is only as efficient as the SVD, which is $O(n^3)$ for n vertices and can be solved in $O(n^2 \log n)$ for restricted cases or in some approximations [10]. Other methods produce good results empirically, but they require parameter tuning or search spaces that are computationally expensive. Methods based on random walks [3], [11] are $O(n)$ but have large constants in the length and number of the walks. Graph convolutional neural networks [12] are quite expensive in practice.

Our work focuses on improving the computational performance of the recent one-hot graph encoder embedding (GEE) [13]. The algorithm is desirable for its convergence guarantees and because it performs a single pass over the edges. The base implementation is already an order of magnitude faster than spectral methods, GCN, or node2vec. The remaining gap this paper addresses is parallelism and memory efficiency. The current GEE implementation takes nearly an hour on a graph with 1.8B edges. We bring this down to 6.5 seconds.

We contribute an implementation of GEE that fully utilizes shared-memory hardware and scales to billions of edges. We reformulate the GEE algorithm into an edge-map program in Ligra's programming interface [14]. This *GEE-Ligra* implementation avoids data races using lock-free atomic updates, resulting in a 500 times speedup and good scalability. We also provide an optimized version of the original code using Numba just-in-time compilation. *GEE-Ligra* is 5 to 20 times faster than Numba, owing to more efficient memory usage and multicore parallelism. Our code is publicly available on GitHub: Numba and GEE-Ligra.

II. Background

GEE: For a graph $G(n, s)$ of nodes n connected by edges s, the One Hot Graph Encoder Embedding (Algorithm 1) builds an embedding matrix $\mathbf{Z}$ based on the edge list s and a vector of class labels $\mathbf{Y} \in \{0, \ldots, K\}^n$, where K is the number of classes. $\mathbf{Y}$ may represent the labels of a few known node ground truths or it may be derived from unsupervised clustering, such as by running the Leiden community detection algorithm [15]. GEE embeds the n nodes into k dimensions, where $k << n$. For brevity, our description does not include the preprocessing steps needed to compute the Laplacian version of the algorithm [13].

GEE first initiates a projection matrix $\mathbf{W}$ (lines 2-6). Then, in a single pass over the edges, GEE incrementally builds $\mathbf{Z}$ by adding the contribution of each edge to the embedding (lines 7-12). This contribution is a product

Algorithm 1: Semi-Supervised GEE

input : $\mathbf{E} \in \mathbb{R}^{s\times 3}$
$\mathbf{Y} \in \{0,\ldots,K\}^n$: class labels
output: $\mathbf{Z} \in \mathbb{R}^{n\times K}$: node embeddings

1 **Function** GEE($\mathbf{E},\mathbf{Y}$):
2 $\mathbf{W} = zeros(n,K)$ // Projection matrix
3 **for** $k = 1 : K$ **do**
4 // $k = 0$ means class unknown
$idx = n$ where $Y[n] = k$
5 $\mathbf{W}(idx,k) = \frac{1}{count(\mathbf{Y}=k)}$;
6 **end**
7 **for** $i = 1 : s$ **do**
8 // (u-source, v-dest., w-weight)
9 $u = \mathbf{E}(i,1);\ v = \mathbf{E}(i,2);\ w = \mathbf{E}(i,3)$;
10 $\mathbf{Z}(u,\mathbf{Y}(v))$ += $\mathbf{W}(v,\mathbf{Y}(v)) \cdot w$;
11 $\mathbf{Z}(v,\mathbf{Y}(u))$ += $\mathbf{W}(u,\mathbf{Y}(u)) \cdot w$;
12 **end**
13 **EndFunction**

Algorithm 2: *GEE-Ligra*

input : $\mathbf{E} \in \mathbb{R}^{s\times 3}$, $\mathbf{Y} \in \{0,\ldots,K\}^n$
output: $\mathbf{Z} \in \mathbb{R}^{n\times K}$, $\mathbf{W} \in \mathbb{R}^{n\times K}$

1 **Function** GEE($\mathbf{E},\mathbf{Y}$):
2 $\mathbf{W} = zeros(n,K)$
3 **ParallelFor** $k = 1 : K$ **do**
4 $idx = n$ where $Y[n] = k$
5 $\mathbf{W}(idx,k) = \frac{1}{count(\mathbf{Y}=k)}$;
6 **end**
7 EdgeMap(updateEmb, $\mathbf{Z},\mathbf{W},\mathbf{Y}$, frontier=$n$)
8 **EndFunction**

9 **Function** updateEmb($\mathbf{Z},\mathbf{W},\mathbf{Y},u,v,w$):
10 writeAdd ($\mathbf{Z}(u,\mathbf{Y}(v))$, $\mathbf{W}(v,\mathbf{Y}(v)) \cdot w$);
11 writeAdd ($\mathbf{Z}(v,\mathbf{Y}(u))$, $\mathbf{W}(u,\mathbf{Y}(u)) \cdot w$);
12 **EndFunction**

of the weight w and the corresponding coefficients in the projection matrix $\mathbf{W}$. The source node contributes to the class of the destination node (line 10) and vice-versa (line 11). This formulation is for weighted directed graphs. Unweighted graphs have unit weights. Undirected graphs are treated as two symmetric directed graphs.

Ligra: The edgeMap, vertexMap interface of Ligra [14] encodes fine-grained, asynchronous parallelism for shared-memory systems. The interface selects a vertex subset, called a *frontier*, and then calls a function for every vertex (vertexMap) or every outbound edge from the vertex subset (edgeMap). This captures almost all modern graph algorithms, including PageRank, Connected Components, and Betweenness Centrality. The frontier subset enables search-style algorithms like breadth-first search (BFS), that trigger computation on neighbors.

III. Methods

The *GEE-Ligra* implementation (Algorithm 2) runs the same algorithm as GEE, i.e. computes the same values on same input. Unlike GEE, which loops over the edges, *GEE-Ligra* uses a function map over the edges, which is parallelized and scheduled by the Ligra runtime. the frontier is the entire graph, i.e. all nodes are active. This invokes the updateEmb function to update the embedding on all edges in a single step [14]. This parallelizes GEE's $O(s)$ component. We also parallelize the initialization of the projection matrix, which costs $O(nk)$. For most graphs and choices of $K < 50$, $s > nk$. However, $O(nk)$ becomes the dominant component of the runtime when graphs have a high n and a very low average degree.

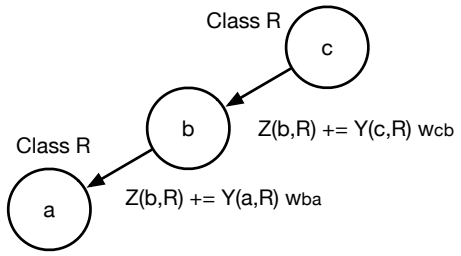

Figure 1. Race condition in which both inbound and outbound edges contribute to the emdedding and create conflicting updates.

The updateEmb function is mapped across all nodes and access the edge list of each node sequentially. Because the frontier is the entire graph, Ligra evaluates updateEmb using the edgeMapDense algorithm [14]. This schedules one worker for the edge list of each node to process all edges sourced from that node sequentially. The read accesses to $\mathbf{W}$ and write accesses to $\mathbf{Z}$ are fine-grained. $\mathbf{Z}(u,:)$ (line 10) and $\mathbf{W}(u,:)$ (line 11) are systematically reused during a edgeMapDense, and will be in the processor cache, however, access to $\mathbf{Z}(v,:)$ and $\mathbf{W}(v,:)$ will likely result in cache misses.

The Ligra writeAdd() function uses hardware support to perform a lock-free atomic increment on the embedding field. This protects the data from the race that occurs when two edges with nodes in the same class update the same entry (Figure 1). This is caused by GEE propagating class information from source to destination and destination to source. We expect such conflicts to happen infrequently, because it requires simultaneous scheduling of two separate edges on separate nodes with the same class label. Updates from the $\mathbf{Z}(u,\mathbf{Y}(v_1))$ and $\mathbf{Z}(u,\mathbf{Y}(v_2))$; $v_1 \neq v_2, \mathbf{Y}(v_1) == \mathbf{Y}(v_2)$ will not conflict. They are scheduled serially by edgeMapDense because they are successive entries in u's edge list.

Graph ($\|n\|$,$\|s\|$) \ Runtime (sec)	GEE-Python	Numba Serial	GEE-Ligra Serial	GEE-Ligra Parallel	Speedup (v. GEE)	Speedup (v. Numba)	Speedup (v. Ligra Serial)
Twitch ($n = 168$K, $s = 6.8$M)	12.18	0.20	0.11	0.013	936	15	8.5
soc-Pokec (1.6M, 30M)	133.21	1.68	0.99	0.12	1100	14	8.25
soc-LiveJournal (6.4M, 69M)	301.64	4.29	2.39	0.39	773	11	6.12
soc-orkut (3M, 117M)	499.83	4.48	2.97	0.26	1897	17	11.4
orkut-groups (3M, 327M)	595.29	11.43	6.06	2.36	252	4.8	2.6
Friendster (65M, 1.8B)	3374.72	112.33	77.23	6.42	525	17	12

Table I
RUNTIME (SECONDS) FOR GRAPHS OF VARIOUS SIZES. $k = 50$ IS USED FOR ALL GRAPHS. SPEEDUP (ROWS 5-7) SHOW THE PERFORMANCE IMPROVEMENT OF *GEE-LIGRA* RUN IN PARALLEL ON 24 CORES TO OTHER IMPLEMENTATIONS.

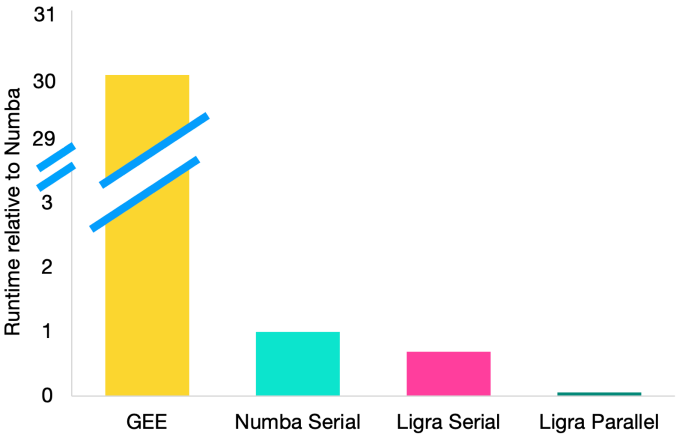

Figure 2. Runtimes for Friendster, normalized to Numba Serial.

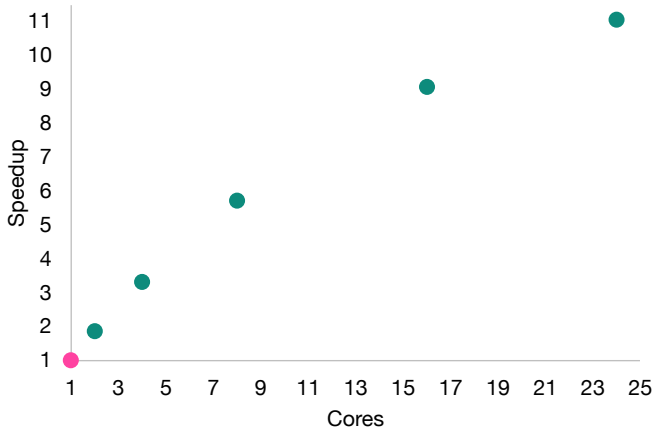

Figure 3. Speedup of Ligra on Friendster by nr. cores.

IV. FINDINGS

We evaluate the parallel implementation of *GEE-Ligra* against the reference implementation in Python and our Numba just-in-time compiled implementation. Experiments were performed using a machine with a 24-core, 48-thread Intel Xeon Platinum 8259CL with 192GB main memory. We used Python 3.10.12 and compiled C++ code with the GNU G++ 11.4.0 compiler. Ligra was compiled from source. We experimented with a variety of graphs collected from the SNAP repository [16] that vary between 6.8-327M edges and the Friendster graph [17] as an example of an Internet-scale dataset with 1.8B edges. We generated the $\mathbf{Y}$ labels uniformly at random from $[0, K = 50]$ for 10% of nodes, which were also selected uniformly at random. This practice aligns with GEE's experimental configuration [13].

Table I summarizes performance results. Massive performance gains are realized by moving to compiled code. This can be seen through our Numba results which show a 30-50 times speedup. *GEE-Ligra* obtains a further 10-15 times performance improvement over Numba. This results from a combination of asynchronous execution in the Ligra graph engine and parallelism. Ligra run in serial improves performance over Numba by less than a factor of 2.

The results comparing compiled code to *GEE-Ligra* on the largest graph (Figure 2) provide insights into the benefit of using a declarative graph engine. On a single thread on a single core, *GEE-Ligra* reduces runtime by 31%. Running *GEE-Ligra* in parallel produces a 17 times speedup over Numba and 12 times over *GEE-Ligra* serial.

We study the strong-scaling performance of *GEE-Ligra* on the Friendster graph. The algorithm exhibits good scalability, realizing 11 times speedup over 24 cores (24 threads with hyperthreading disabled). Atomic updates could lead to interference between threads and limit scalability. However, we ran the program with atomics off, performing unsafe updates, and saw no appreciable performance difference. We expect this workload to be memory bound, because there is so little computation per edge. GEE-Ligra performs two fused-multiply adds per edge and two memory writes, one of which is likely to miss. These scaling results are consistent with other graph algorithms [14], [18] which have been shown to be memory bound.

We also study a large range of graph sizes to demonstrate that we preserve performance as inputs grow. We generate Erdős-Rényi random graphs increasing numbers of edges and run *GEE-Ligra* using all 24 cores. Fig-

ure 4 shows that GEE-Ligra's runtime increases linearly with the number of edges.

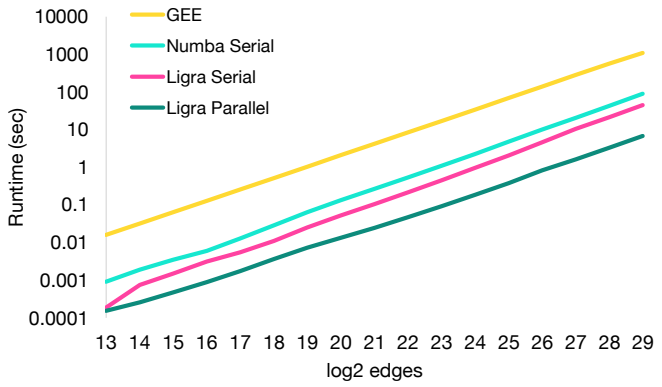

Figure 4. Runtime as the number of edges increase on Erdős-Rényi graphs.

V. Conclusion

We present an edge-parallel implementation in a shared-memory graph engine of the One Hot Graph Encoding Embedding (GEE) algorithm. The original algorithm provides an order of magnitude performance improvement over spectral methods, random walks, and graph convolutional networks [13]. Our implementation provides a speedup of 500 times over the base implementation and 17 times speedup over a compiled version of the algorithm. This allows us to embed graphs of 1.8B nodes in 6.5 seconds. Our treatment describes how to parallelize over the edge lists of each node and use atomic instructions to avoid race conditions.

References

[1] Di Jin, Xinxin You, Weihao Li, Dongxiao He, Peng Cui, Francoise Soulie Fogelman, and Tanmoy Chakraborty, "Incorporating network embedding into markov random field for better community detection," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 160–167, 07 2019.

[2] Di Jin, Zhizhi Yu, Pengfei Jiao, Shirui Pan, Dongxiao He, Jia Wu, Philip S. Yu, and Weixiong Zhang, "A survey of community detection approaches: From statistical modeling to deep learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 2, pp. 1149–1170, 2023.

[3] Aditya Grover and Jure Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.

[4] William L. Hamilton, Rex Ying, and Jure Leskovec, "Representation learning on graphs: Methods and applications," *IEEE Data Eng. Bull.*, vol. 40, pp. 52–74, 2017.

[5] Carey E. Priebe, Youngser Park, Joshua T. Vogelstein, John M. Conroy, Vince Lyzinski, Minh Tang, Avanti Athreya, Joshua Cape, and Eric Bridgeford, "On a two-truths phenomenon in spectral graph clustering," *Proceedings of the National Academy of Sciences*, vol. 116, no. 13, pp. 5995–6000, 2019.

[6] Benjamin Bowman and H. Howie Huang, "Towards next-generation cybersecurity with graph AI," *SIGOPS Oper. Syst. Rev.*, vol. 55, no. 1, pp. 61–67, jun 2021.

[7] Vince Lyzinski, Minh Tang, Avanti Athreya, Youngser Park, and Carey E. Priebe, "Community detection and classification in hierarchical stochastic blockmodels," *IEEE Transactions on Network Science and Engineering*, vol. 4, no. 1, pp. 13–26, 2017.

[8] Daniel L. Sussman, Minh Tang, Donniell E. Fishkind, and Carey E. Priebe, "A consistent adjacency spectral embedding for stochastic blockmodel graphs," *Journal of the American Statistical Association*, vol. 107, no. 499, pp. 1119–1128, 2012.

[9] Avanti Athreya, Donniell E. Fishkind, Minh Tang, Carey E. Priebe, Youngser Park, Joshua T. Vogelstein, Keith Levin, Vince Lyzinski, Yichen Qin, and Daniel L Sussman, "Statistical inference on random dot product graphs: a survey," *Journal of Machine Learning Research*, vol. 18, no. 226, pp. 1–92, 2018.

[10] Dinesh Ramasamy and Upamanyu Madhow, "Compressive spectral embedding: sidestepping the SVD," in *Advances in Neural Information Processing Systems*, 2015, vol. 28.

[11] 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*, 2014, p. 701–710.

[12] Thomas N Kipf and Max Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[13] Cencheng Shen, Qizhe Wang, and Carey E. Priebe, "One-hot graph encoder embedding," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 45, no. 6, pp. 7933–7938, 2023.

[14] Julian Shun and Guy E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *SIGPLAN Notices*, vol. 48, no. 8, pp. 135–146, 2013.

[15] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck, "From louvain to leiden: guaranteeing well-connected communities," *Scientific reports*, vol. 9, no. 1, pp. 5233, 2019.

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

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

[18] Brian Wheatman, Randal Burns, Aydın Buluç, and Helen Xu, "CPMA: An efficient batch-parallel compressed set without pointers," *arXiv preprint arXiv:2305.05055*, 2023.