Scalable Node Embedding Algorithms using Distributed Sparse Matrix Operations

Isuru Ranawaka*, Ariful Azad§
* Indiana University, Bloomington, IN, USA (isjarana@iu.edu)
§ Indiana University, Bloomington, IN, USA (azad@iu.edu)

***Abstract*—We introduce a distributed memory parallel algorithm for force-directed node embedding that places vertices of a graph into a low-dimensional vector space based on the interplay of attraction among neighboring vertices and repulsion among distant vertices. We develop our algorithms using two sparse matrix operations, SDDMM and SpMM. We propose a configurable pull-push-based communication strategy that optimizes memory usage and data transfers based on computing resources and asynchronous MPI communication to overlap communication and computation. Our algorithm scales up to 256 nodes on distributed supercomputers by surpassing the performance of state-of-the-art algorithms.**

I. Introduction

The node embedding problem aims to map graph vertices into lower dimensional vector space by preserving the structural properties of the graph. Node embedding is used in graph visualization and machine-learning tasks like link prediction and node classification. We use force-directed node embedding, which is generally applicable to both visualization tasks and machine learning tasks. We model embedding operations with two linear algebra operations: sample dense-dense matrix multiplication (SDDMM) and sparse-dense matrix multiplication (SpMM). Their combination of SDDMM and SpMM (called FusedMM) can also be used [1]. This linear algebra formulation contributes to faster computations and reduced communication. As a result, our algorithm scales to thousands of processors and runs significantly faster than state-of-the-art (SOTA) distributed algorithms such as DistGER[2], and PyTorch-BigGraph (PGB)[3] with improved accuracy.

Distributed memory implementations such as DistGER, PGB, DistDGL[4], and KnightKing[5] were proposed to expedite the embeddings of large graphs. However, they scale well up a few tens of processors due to unnecessary data transfers, reliance on a central server, or preprocessing steps. Our main contribution is to develop a distributed-memory node embedding algorithm designed to scale effectively to thousands of processors. We achieved this goal through a cohesive optimization strategy spanning three interconnected aspects: (1) reduction of inter-process communication, (2) acceleration of local computations within each process, and (3) effective utilization of partitioned memory.

The graph's sparsity and patterns influence communication, computation, and memory optimization in force-directed algorithms in force calculations [6], [7]. In distributed-memory systems, retrieving embeddings from remote vertices in force-directed algorithms can lead to communication costs, particularly in mini-batch stochastic gradient descent. The pull-based strategy blindly fetches remote embedding vectors in each minibatch irrespective of their updated status. Hence, it leads to higher communication overhead. The push-based strategy, which involves updating and sending information to remote processes, is less memory-efficient than the pull-based approach but reduces communication overhead. An adaptive push-pull algorithm is proposed to minimize communication-based on available memory to address this trade-off. Additionally, shared-memory parallel operations accelerate local embedding calculations, and computation and communication are overlapped to mitigate communication overhead.

II. Force directed graph embedding

Let $G(V, E)$ be a graph where V is a set of vertices and E is a set of edges with $|V| = n$ and $|E| = m$. Let $\mathbf{A} \in R^{n \times n}$ be the sparse adjacency matrix of the graph where $\mathbf{A}_{ij} = 1$ if $\{v_i, v_j\} \in E$, otherwise $\mathbf{A}_{ij} = 0$. We consider a graph embedding problem where every vertex v is mapped to a vector $\mathbf{z}_v$ in d-dimensional vector space. We store the embeddings of all vertices in a dense matrix $\mathbf{Z} \in R^{n \times d}$ where $d \ll n$. When the embedding dimension is 2 or 3, graph embedding can be used to visualize a graph on a screen.

let $\delta : R^d \times R^d \rightarrow R$ be a function that computes the *similarity* between embeddings of two vertices. Let $N(u)$ be the set of vertices adjacent to u and $S(u)$ be a subset of vertices that are not adjacent to u. In this context, $S(u)$ is called *negative samples*. Then, embedding algorithms define the loss function for vertex u as the negative log likelihood with respect to $N(u)$ and $S(u)$:

$$\mathcal{L}(u) = -\sum_{v \in N(u)} \log \delta(\mathbf{z}_u, \mathbf{z}_v) - \sum_{w \in S(u)} \log(1 - \delta(\mathbf{z}_u, \mathbf{z}_w)). \tag{1}$$

We can then minimize the loss function using the Stochastic Gradient Descent (SGD) algorithm, where the embedding of vertex u is updated as follows:

$$\mathbf{z}_u = \mathbf{z}_u - \eta \frac{\partial \mathcal{L}(u)}{\partial \mathbf{z}_u}, \tag{2}$$

where η is the learning rate or step size. This calculates the coordinates of the vertices in lower dimensions such that it minimizes the energy of the entire graph. We use minibatch SGD for embedding calculation [8].

III. Distributed Memory Algorithms

A. Data distribution and storage

We use 1D row partitioning of the both dense embedding matrix $\mathbf{Z}$ and the sparse adjacency matrix $\mathbf{A}$ such that each process owns $\mathbf{Z_p} \in R^{\frac{n}{p} \times d}$ and $\mathbf{A_p} \in R^{\frac{n}{p} \times n}$. When $\mathbf{A}$ is unsymmetric, we additionally keep a column partition of $\mathbf{Z^T}$ where each process stores $\mathbf{Z_p^T} \in R^{d \times \frac{n}{p}}$ and its corresponding column partition $\mathbf{A_p^c} \in R^{n \times \frac{n}{p}}$ of $\mathbf{A}$. This representation creates a $ptimesp$ virtual partitioning of the sparse matrix, in which two processes own each non-diagonal block. Local sparse matrices are stored in the compressed sparse row (CSR) format.

B. Communication patterns

We employ minibatch SGD for computing embedding vectors, parallelizing the computation of vertex embeddings in each minibatch. We explore three communication patterns: pull-based, push-based, and an adaptive pull-push strategy. Pull-based communication involves fetching remote computing vectors using two-way communication. We optimize this to one-round communication by leveraging reverse graph structure. However, in a minibatch setting, redundant communication and increased overhead occur due to fetching the same vectors multiple times. Alternatively, push-based communication only sends embedding vectors after updating and self-calculates the remote processes needing updated embeddings after each minibatch update. This approach reduces communication overhead at the expense of increased memory usage for caching. To strike a balance between communication and memory overhead, we propose an adaptive pull-push strategy that combines both approaches, categorizing subsets of remote processes for pull or push-based interactions, allowing for control over memory and communication overhead.

We enhance computation efficiency by dividing the computation and communication into multiple segments, and concurrently executing computation and communication through asynchronous MPI communication.

IV. Results

Experimental platform. We ran all experiments on the CPU partitions of the Perlmutter supercomputer at NERSC. Shared memory comparisons with other state-of-the-art methods were conducted on a single Perlmutter node that has AMD EPYC CPU with 128 cores and 512GB memory. The algorithms are written in C/C++ with OpenMP for shared-memory multi-threading and Cray's MPI implementation for inter-process communication. Unless otherwise stated, our distributed algorithm used 4 MPI processes per node (i.e., 32 OpenMP threads per process).

Datasets. We use *arabic-2005*, *it-2004*, *GAP-web*, *uk-2002* and *twitter-7* are used for large scale experiments. *flickr*, *pubmed*, *youtube*, and *com-Orkut* are used for small-scale and medium-scale experiments. We collected these graphs from SNAP [9] and Suitesparse Matrix Colelction [10]. In addition, we generated two K-nearest neighbor graphs (KNNGs) from the MNIST and BIGANN datasets. KNNGs are generated with 10 edges per vertex using the DRPT software [11].

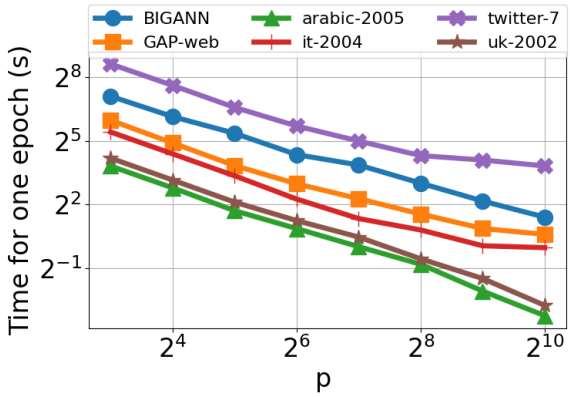

Fig. 1. Strong scaling experiments with 16K vertices per process in a mini-batch. The total number of vertices in a mini-batch across all processes increases linearly with p.

Baselines. We use PBG [3] and DistGER [2] as our distributed memory baseline models to compare the runtime and quality of embeddings. Force2vec [6], HARP [12], and DeepWalk [13] are used as the baseline for shared memory implementation, in addition to the PBG and DistGER. Furthermore, to compare the efficiency of communication, we exploit the 1.5D Dense Shifting algorithm [14] with SpMM.We download each software from the respective GitHub repositories.

Our embedding generation demonstrates comparable or superior accuracy in node classification and link prediction tasks compared to the previously mentioned algorithms. It exhibits remarkable efficiency by operating at a faster pace. Moreover, our algorithm effortlessly scales beyond 1024 MPI processes, distinguishing itself from the limitations of PBG and DistGER, which struggle to scale beyond 32 MPI processes (See Figure 1). This scalability enhancement results in an impressive 8X speedup, perfectly aligned with an 8X increment in resources.

Incorporating hybrid communication schemes further contributes to our algorithm's prowess, showcasing superior runtime for balanced memory and minimizing data transfer overhead, which is particularly beneficial for minibatch SGD. On the other hand, the pure pull-based strategy excels in runtime efficiency when implementing a fullbatch scheme.

The asynchronous communication with computation and communication overlapping further reduces the runtime of our algorithm. We implemented the SpMM operation on our communication kernel and tested it with the STOA 1.5D dense shifting algorithm. Our algorithm achieved a 5X speedup.

V. Acknowledgements

This research is partially supported by the Applied Mathematics Program of the DOE Office of Advanced Scientific Computing Research under contracts numbered DE-SC0022098 and DE-SC0023349 and by NSF grants CCF-2316234 and OAC-2339607.

REFERENCES

[1] M. K. Rahman, M. H. Sujon, and A. Azad, "FusedMM: A unified sddmm-spmm kernel for graph embedding and graph neural networks," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 256–266.

[2] P. Fang, A. Khan, S. Luo, F. Wang, D. Feng, Z. Li, W. Yin, and Y. Cao, "Distributed graph embedding with information-oriented random walks," *Proc. VLDB Endow.*, vol. 16, no. 7, p. 1643–1656, 2023.

[3] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich, "PyTorch-BigGraph: A Large-scale Graph Embedding System," in *Proceedings of the 2nd SysML Conference*, Palo Alto, CA, USA, 2019.

[4] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "Distdgl: Distributed graph neural network training for billion-scale graphs," in *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2020, pp. 36–44. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IA351965.2020.00011

[5] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang, "Knightking: a fast distributed graph random walk engine," ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 524–537. [Online]. Available: https://doi.org/10.1145/3341301.3359634

[6] M. K. Rahman, M. H. Sujon, and A. Azad, "Force2vec: Parallel force-directed graph embedding," in *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2020, pp. 442–451.

[7] H. Lotfalizadeh and M. A. Hasan, "Force-directed graph embedding with hops distance," *arXiv preprint arXiv:2309.05865*, 2023.

[8] M. K. Rahman, M. H. Sujon, and A. Azad, "Scalable force-directed graph representation learning and visualization," *Knowledge and Information Systems*, vol. 64, no. 1, pp. 207–233, 2022.

[9] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, pp. 1–20, 2016.

[10] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw. (TOMS)*, vol. 38, no. 1, pp. 1–25, 2011.

[11] I. Ranawaka, M. K. Rahman, and A. Azad, "Distributed sparse random projection trees for constructing k-nearest neighbor graphs," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2023, pp. 36–46.

[12] H. Chen, B. Perozzi, Y. Hu, and S. Skiena, "Harp: Hierarchical representation learning for networks," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence*. AAAI Press, 2018.

[13] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: online learning of social representations," ser. KDD '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 701–710. [Online]. Available: https://doi.org/10.1145/2623330.2623732

[14] V. Bharadwaj, A. Buluc, and J. Demmel, "Distributed-memory sparse kernels for machine learning," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2022, pp. 47–58. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPS53621.2022.00014