# Batch Updates of Distributed Streaming Graphs using Linear Algebra

Elaheh Hassani, Md Taufique Hussain, and Ariful Azad
*Dept. of Intelligent Systems Engineering*
*Indiana University Bloomington*, IN, USA
Email: {ehassani, mth, azad}@iu.edu

***Abstract*—We develop a distributed-memory parallel algorithm for performing batch updates on streaming graphs, where vertices and edges are continuously added or removed. Our algorithm leverages distributed sparse matrices as the core data structures, utilizing equivalent sparse matrix operations to execute graph updates. By reducing unnecessary communication among processes and employing shared-memory parallelism, we accelerate updates of distributed graphs. Additionally, we maintain a balanced load in the output matrix by permuting the resultant matrix during the update process. We demonstrate that our streaming update algorithm is at least 25 times faster than alternative linear-algebraic methods and scales linearly up to 4,096 cores (32 nodes) on a Cray EX supercomputer.**

## I. Introduction

Streaming graphs, where edges and vertices change continuously, are widely used to model dynamic systems, including social networks [1], [2], communication networks [3], and biological networks [4]. In this paper, we consider the problem of updating streaming graphs where edges arrive in batches. As the size of streaming graphs increases, storing them in distributed systems and performing parallel updates across distributed nodes become essential. To address this challenge, we develop distributed-memory parallel algorithms for batch updates of streaming graphs.

Streaming graph updates are well-studied [5], but most approaches target shared-memory systems [6] or event-streaming platforms like Kafka [7]. We use an alternative approach by representing the existing graph and streaming updates as two sparse matrices distributed across $P$ processes. These two sparse matrices are then added under a user-defined monoid [8]. In a distributed memory setting, this operation presents two main challenges: (1) *Data Communication*: Since the input and output matrices can have drastically different shapes and non-zero elements, data communication among processes significantly impacts the scalability and runtime of distributed updates. (2) *Load Imbalance*: Streaming subgraphs may add or delete vertices and edges, causing the output matrix to expand or shrink. This can lead to load imbalances in the distributed system. To address these challenges, we develop a distributed-memory parallel algorithm that reduces unnecessary communication among processes and permutes the output matrix to maintain load balance without extra communication costs. To this end, the primary contribution of this paper is the use of a matrix-based approach for handling batched updates, along with distributing the computations to ensure scalability for large-scale graphs.

We implemented the streaming update operations within the CombBLAS library [9], utilizing the existing sparse matrix data structures provided by CombBLAS. Our results show that our streaming update algorithm is at least 25 times faster than the alternative approach in CombBLAS which relies on sparse matrix-matrix multiplication [10]. Additionally, our new algorithm maintains perfect load balance in the output, which is beneficial for downstream analysis. We demonstrate that the algorithm scales linearly up to 4,096 cores (32 nodes) on the Big Red 200 supercomputer at Indiana University.

## II. Related Work

Recent research on streaming graph updates has focused on efficient data structures and supporting frameworks. STINGER [6] uses a linked list of edge blocks for graph streaming updates on a single node, while Aspen [11] offers a tree-based update framework. LLAMA [12] and PCSR [13] are CSR format-based data structures for graph streaming. However, these frameworks and data structures primarily target shared-memory environments. DISTINGER [14], an extension of STINGER, is a distributed framework for large graphs using STINGER's linked list structure. Sallinen et al. [15] presents a dynamic graph data structure for distributed memory with near real-time updates. CuSTINGER [16] and Hornet [17] are GPU-based data structures designed for efficient graph updates, optimizing memory by transferring only updates. Makkar et al. [18] introduce a GPU implementation using cuSTINGER to update large graphs and adjust triangle counts.

## III. Method

### A. Notations

Let $G(V, E, W)$ be a weighted graph where $V$ is a set of vertices, $E$ is a set of edges, and $W_{ij}$ denotes the weight of the edge between vertices $v_i$ and $v_j$. The graph has $n$ vertices, meaning $|V|{=}n$. Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be the sparse adjacency matrix of the graph where $\mathbf{A}[i,j] = W_{ij}$ if $\{v_i, v_j\}{\in}E$, otherwise $\mathbf{A}[i,j]{=}0$. We represent a batch of edges as a graph $G_B(V_B, E_B, W_B)$. A batch is then added to the original graph $G_A(V_A, E_A, W_A)$ to create an updated graph $G_C(V_A \cup V_B, E_A \cup E_B, W_C)$.

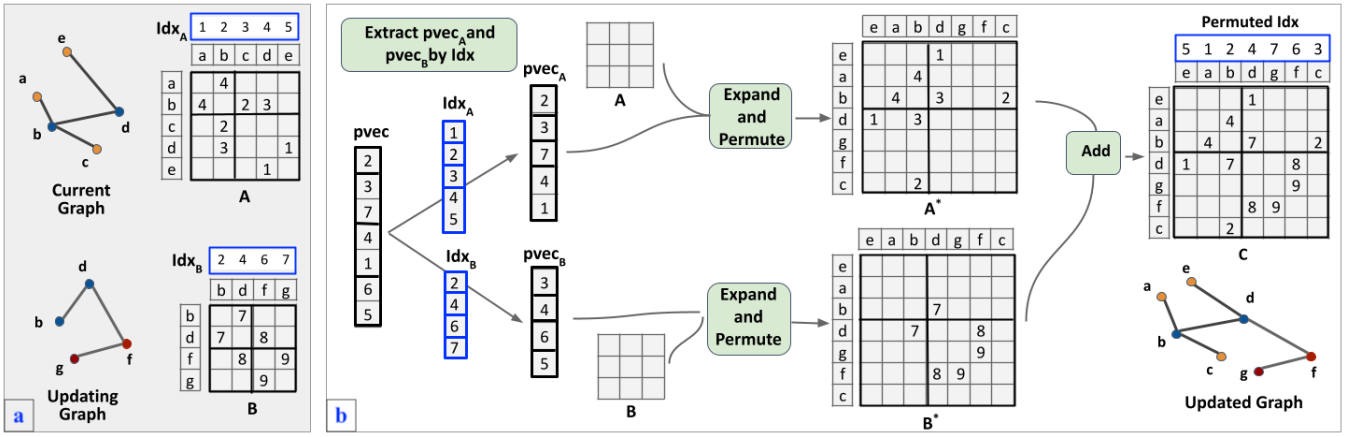


Fig. 1. The graph update pipeline. (a) Illustrate the current graph as the one to be updated, and the updating graph as introducing new edges and possibly new vertices. New edges can connect existing or new vertices. In the diagrams, yellow represents vertices only in the current graph, blue indicates vertices shared by both current and updating graphs and red signifies new vertices from the update. The adjacency matrices, $\mathbf{A}$ for the current graph and $\mathbf{B}$ for the updating graph are shown. Assuming a 2×2 process grid, the submatrix owned by each process is highlighted with bold black lines. (b) Given matrices $\mathbf{A}$, $\mathbf{B}$, and vector $\mathbf{Idx_B}$ (mapping indices from $\mathbf{B}$ to the updated matrix), the algorithm pipeline begins by extracting subvectors from the random permutation vector $\mathbf{pvec}$. $\mathbf{pvec_A}$ and $\mathbf{pvec_B}$ are the permutation vectors for matrices $\mathbf{A}$ and $\mathbf{B}$ respectively. Each matrix will be expanded and permuted given its relative permutation vector to get $\mathbf{A}^*$ and $\mathbf{B}^*$. Finally, $\mathbf{A}^*$ and $\mathbf{B}^*$ will be added with a select-second operation to get the updated matrix $\mathbf{C}$.

### B. Data Distribution

We distribute matrices and vectors across $p$ processes in a $\sqrt{p} \times \sqrt{p}$ process grid. $P_{ij}$ denotes the process in the $i$th row and $j$th column of the grid. The adjacency matrix is partitioned such that each process $P_{ij}$ stores an $n/\sqrt{p} \times n/\sqrt{p}$ submatrix, denoted by $\mathbf{A}_{ij}$. Figure 1(a) illustrates the distribution of matrices $\mathbf{A}$ and $\mathbf{B}$ across 4 processes in a $2 \times 2$ grid. Within each process, the local matrix is stored in a doubly compressed sparse column (DCSC) format, an efficient variant of CSC that stores only columns containing at least one nonzero element [19].

Vectors (e.g., permutation vectors) are distributed across all processes without replication. In a $\sqrt{p} \times \sqrt{p}$ grid, a vector $\mathbf{v}$ is stored such that all processes in the $i$th row of the grid collectively hold subvector $\mathbf{v}_i$ of length $\lfloor \frac{|\mathbf{v}|}{\sqrt{p}} \rfloor$, which is further divided among the processes within the same row, each holding a smaller subvector of length $\lfloor \frac{|\mathbf{v}_i|}{\sqrt{p}} \rfloor$.

### C. The Graph Update Pipeline

**The problem setting.** Let $G_A(V_A, E_A, W_A)$ be an existing graph with $n$ vertices and $G_B(V_B, E_B, W_B)$ a new graph $m$ vertices. We aim to add $G_A$ and $G_B$ to create an updated graph $G_C(V_A \cup V_B, E_A \cup E_B, W_C)$. Let the updated graph has $l$ vertices, meaning $|V_A \cup V_B| = l$. In our algorithm, we maintain all three graphs as sparse matrices: $G_A$ is represented by $\mathbf{A} \in \mathbb{R}^{n \times n}$, $G_B$ by $\mathbf{B} \in \mathbb{R}^{m \times m}$, and $G_C$ by $\mathbf{C} \in \mathbb{R}^{l \times l}$. Since there could be common vertices in $\mathbf{A}$ and $\mathbf{B}$, we need a vertex mapping vector $\mathbf{idx_B} \in \mathbb{N}^{m \times 1}$ that maps vertices from $\mathbf{B}$ to $\mathbf{A}$ if any. As discussed in the previous section, all matrices are distributed in a $\sqrt{p} \times \sqrt{p}$ process grid. To ensure load balance in $\mathbf{C}$, we apply a symmetric random permutation while constructing $\mathbf{C}$, relabeling the graph's vertices. The permutation vector $\mathbf{pvec} \in \mathbb{N}^{l \times 1}$ is used to permute $\mathbf{C}$.

**Steps in the graph update pipeline.** Figure 1 outlines the streaming graph update process using example matrices distributed across four processes. The pipeline has three steps: the first two reshape $\mathbf{A}$ and $\mathbf{B}$ to the same dimensions, and the final step combines them locally without communication.

1) **Expand and permute $\mathbf{A}$.** We reshape $\mathbf{A}$ to $\mathbf{A}^* \in \mathbb{R}^{l \times l}$ while permuting it using the permutation vector $\mathbf{pvec}$. This step requires AllToAll communication.
2) **Expand and permute $\mathbf{B}$.** We reshape $\mathbf{B}$ to $\mathbf{B}^* \in \mathbb{R}^{l \times l}$ and permute it using $\mathbf{idx_B}$ and $\mathbf{pvec}$. This step requires AllToAll communication.
3) **Add $\mathbf{A}^*$ and $\mathbf{B}^*$.** We add $\mathbf{A}^*$ and $\mathbf{B}^*$ to form $\mathbf{C}$ with a user-defined monoid, requiring no communication. Afterward, $\mathbf{C}$ remains load balanced, as shown in Figure 1.

For completeness, Algorithm 1 details the steps. It takes $\mathbf{A}$, $\mathbf{B}$, and the vertex mapping vector $\mathbf{idx_B}$ as inputs. A random permutation vector $\mathbf{pvec}$ is generated (line 4), and the three steps produce the permuted result matrix $\mathbf{C}$.

### D. Expand and Permute

The most expensive step in our graph update pipeline is the matrix expansion and permutation operation. Algorithm 2 described the *ExpandAndPermute* operation that expands (i.e., reshape) a matrix to a larger size and permutes its rows and columns according to a given permutation vector. The inputs to the *ExpandAndPermute* algorithm are matrix $\mathbf{A}$, permutation vector $\mathbf{pvec}$ and $l$, the size of output matrix $\mathbf{A}^*$. The algorithm starts with each process $P_{ij}$ gathering necessary subvectors of $\mathbf{pvec}$ from other processes, for the row and column indices of submatrix $\mathbf{A_{ij}}$ (line 3 of Algorithm 2). Next, each process performs parallel local computations, processing nonzero elements of $\mathbf{A_{ij}}$ and determining their new location in $\mathbf{A}^*_{ij}$ based on $\mathbf{pvec_{(i:)}}$, $\mathbf{pvec_{(:j)}}$ and $l$ (line 4 of Algorithm 2). We preprocess the data for load balancing,

**Algorithm 1** Algorithm for graph update

**Input and Output:** Input adjacency matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, adjacency matrix $\mathbf{B} \in \mathbb{R}^{m \times m}$, vector $\mathbf{idx_B}$ and output updated matrix $\mathbf{C} \in \mathbb{R}^{l \times l}$

1: **procedure** UPDATEGRAPH($\mathbf{A}, \mathbf{B}, \mathbf{idx_B}$)
2: $l \leftarrow$ max (Max_element($\mathbf{idx_B}$), n)
3: $\mathbf{idx_A} \leftarrow [1...n]$
4: $pvec \leftarrow$ Generate a random permutation of length $l$
5: $\mathbf{pvec_A} \leftarrow$ EXTRACTSUBVECTOR($\mathbf{pvec}, \mathbf{idx_A}$)
6: $\mathbf{A}^* \leftarrow$ EXPANDANDPERMUTE($\mathbf{A}, l, \mathbf{pvec_A}$)
7: $\mathbf{pvec_B} \leftarrow$ EXTRACTSUBVECTOR($\mathbf{pvec}, \mathbf{idx_B}$)
8: $\mathbf{B}^* \leftarrow$ EXPANDANDPERMUTE($\mathbf{B}, l, \mathbf{pvec_B}$)
9: $\mathbf{C} \leftarrow$ ADD WITH MONOID($\mathbf{A}^*, \mathbf{B}^*, monoid$)
10: **return** $\mathbf{C}$
11: **end procedure**

**Algorithm 2** ExpandAndPermute

**Input and Output:** Input adjacency matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, permutation vectors **pvec**, and integer $l$ for output matrix size. Output matrix $\mathbf{A}^* \in \mathbb{R}^{l \times l}$

1: **procedure** EXPANDANDPERMUTE($\mathbf{A}, l, \mathbf{pvec}$)
2: **for** Every process $P_{ij}$ in **parallel do**
3: $\mathbf{pvec}(\mathbf{i}, :)$ and $\mathbf{pvec}(:, \mathbf{j}) \leftarrow$ Gather subvectors
4: $sendBuf \leftarrow LocalComputation($
$\mathbf{A_{ij}}, \mathbf{pvec}(\mathbf{i}, :), \mathbf{pvec}(:, \mathbf{j}), l)$
5: **end for**
6: $recvBuf \leftarrow$ EXCHANGE($sendBuf$)
7: **for** Every process $P_{ij}$ in **parallel do**
8: $\mathbf{A^*_{ij}} \leftarrow$ BUILDLOCALMATRIX($recvBuf, l$)
9: **end for**
10: **return** $\mathbf{A}^*$
11: **end procedure**

ensuring an equal distribution of local computation across threads. All processes then perform an Alltoall communication to exchange nonzeros (line 6 of Algorithm 2). Finally, each process constructs $\mathbf{A}^*_{ij}$ from the received data, using parallel sorting and multiway merging (line 8 of Algorithm 2).

Figure 2 illustrates of the *ExpandAndPermute* module, using an example where the local matrix and its corresponding computations are highlighted to demonstrate the process.

## IV. RESULT

### A. Experimental Setup

We evaluated the performance of our algorithm on Big Red 200, an HPE Cray EX supercomputer at Indiana University. Each compute node is equipped with 256 GB of memory and two 64-core 2.25 GHz AMD EPYC 7742 processes. Our algorithm was implemented in C/C++ and it was compiled with gcc compiler version 11.2.0. We used Cray's MPI implementation for inter-node communication and OpenMP for intra-node multithreading. In all experiments, we used 8 MPI processes per node and 16 OpenMP threads per process, distributing the matrices using 2D square process grids.

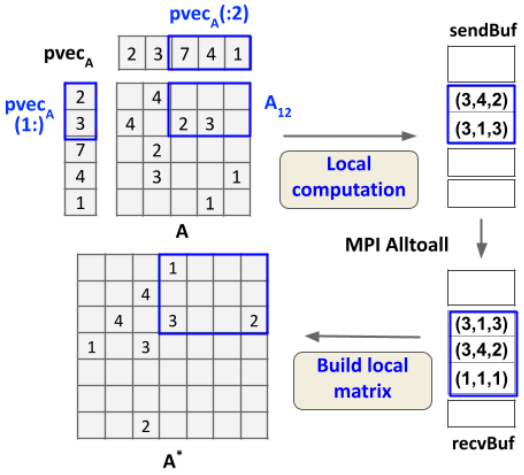


Fig. 2. The *ExpandAndPermute* procedure for matrix $\mathbf{A}$ is illustrated. Matrix $\mathbf{A}$ is distributed among 4 processes, with $\mathbf{A_{12}}$ stored on process $P_{12}$. The gathered subvectors $\mathbf{pvec_A}(1:)$ and $\mathbf{pvec_A}(:2)$ are highlighted. Each nonzero in $\mathbf{A_{12}}$ is processed to find its new location, placed in $sendBuf$, and exchanged via Alltoall communication. Finally, $\mathbf{A^*_{12}}$ is rebuilt from the received nonzeros and reshaped to fit the new matrix dimensions.

### B. Dataset

Table I lists the datasets used in our experiments, including both synthetic and real-world problems. For synthetic datasets, we use sparse matrices from the SuiteSparse collection [20] as $\mathbf{A}$ and generate Erdős-Rényi random graphs for $\mathbf{B}$ (first four rows). The real-world dataset includes archaeal protein similarity graph generated from the isolate genomes [21] for $\mathbf{A}$ and Uniparc metagenomic sequence dataset [22] for $\mathbf{B}$ (last row). This graph is ever-growing as the metagenomic database expands, requiring updates with new vertices and edges before any analysis, such as clustering [23], [24], can be performed efficiently. Given the computational cost of all-vs-all sequence alignment [25], incremental updates are more practical than rebuilding the graph from scratch.

### C. Relative performance of algorithms

We compare our algorithm against an implementation using currently supported operations in the CombBLAS library [9]. $\mathbf{A} \in \mathbb{R}^{n \times n}$ is expanded to $\mathbf{A}^* \in \mathbb{R}^{l \times l}$ using two matrix multiplications. Then, $\mathbf{B}^*$ is generated by assigning $\mathbf{B}$ to an empty matrix of size $l \times l$ based on the vertex mapping in $idx_B$ using the *SpAsgn* operation, which also requires two matrix multiplications. Existing edges that need to be updated are then removed using the *SetDifference* operation, which is a local operation and does not involve any communication. Finally, $\mathbf{C}$ is obtained by adding $\mathbf{A}^*$ and $\mathbf{B}^*$, followed by permutation, which involves two more matrix multiplications.

Table I highlights the speedup of our graph update algorithm achieved by two main key factors. Firstly, our *ExpandAndPermute* algorithm combines permutation and expansion into a single operation. Both tasks involve similar local computations, MPI Alltoall communication, and matrix reconstruction. By integrating these, we reduce communication overhead and computational load. Secondly, as we have

TABLE I
PEFORMANCE OF OUR UPDATE ALGORITHM IN TERMS OF EXECUTION TIME AND UPDATED MATRIX LOAD IMBALANCE FACTOR COMPARED WITH COMBBLAS USING 64 PROCESSES ON 8 NODES AND 16 THREADS PER PROCESS.

| Graph | A #vertices | A #edges | B #vertices | B #edges | C #vertices | C #edges | Our update(s) | CombBLAS (s) | C load-imbalance with our updates | C load-imbalance without permutation |
|---|---|---|---|---|---|---|---|---|---|---|
| wb-edu | 9.8M | 57.1M | 4.2M | 24.6M | 11.8M | 81.8M | 1.324 | 71.42 | 1.18 | 7.61 |
| uk-2002 | 18.5M | 298.1M | 8.4M | 121.7M | 25.2M | 419.8M | 4.984 | 395.3 | 1.1 | 8.14 |
| GAP-web | 50.6M | 1.9B | 16.7M | 292.8M | 58.2M | 2.2B | 23.068 | 2056.5 | 1.09 | 9 |
| GAP-urand | 134.2M | 4.2B | 33.5M | 588.8M | 161.1M | 4.3B | 31.66 | 1552.79 | 1 | 1.67 |
| Archaea | 5.5M | 148.1M | 4.4M | 34.9M | 8.4M | 183M | 2.03 | 49.8 | 1.3 | 2.19 |

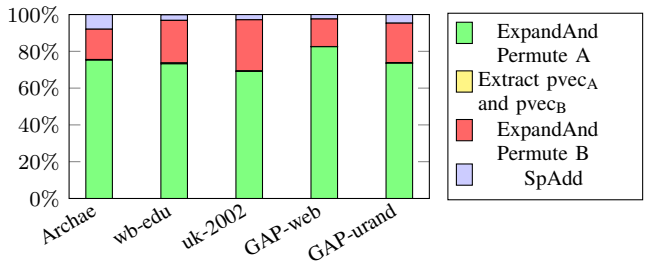

Fig. 3. Breakdown of runtimes for our update algorithm.

TABLE II
IMPACT OF MULTIPLE STEPS OF GRAPH UPDATES ON LOAD IMBALANCE FACTOR FOR THE uk−2002 DATASET USING 64 PROCESSES.

| Updating method | Step 0 | Step 1 | Step 2 | Step 3 |
|---|---|---|---|---|
| Update and Permute | 9 | 1.2 | 1.3 | 1.4 |
| Update without permute | 9 | 10.49 | 11.23 | 11 |

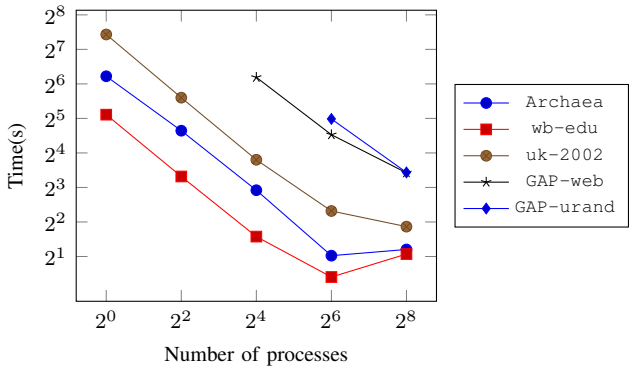

Fig. 4. Scalability of the algorithm.

discussed above, many of these operations in CombBLAS rely on matrix multiplication. While using matrix multiplication is mathematically elegant, it introduces redundant computation and communication. Our approach removes this redundancy, making the algorithm more efficient.

The runtime breakdown of our update algorithm across different datasets on 8 nodes (each with 8 processes and 16 threads) is shown in Figure 3. We observe that more than 50% of the time is spent on expanding and permuting matrix $\mathbf{A}$, due to its larger size and higher number of nonzeros compared to matrix $\mathbf{B}$.

### D. Effect of permutation on load-balance

In Table II we demonstrate the load imbalance of matrix distribution over several steps of update operation on the graph. The load imbalance factor is defined as the ratio of the largest to the average number of nonzeros across all processes. We observe that without permuting the matrix during updates, significant load imbalance develops over time. If $\mathbf{B}$ has a different number of nonzeros per row than $\mathbf{A}$, the last rows/columns become sparser or denser, causing load imbalance in the 2D process grid of the sparse matrix. Permuting the matrix resolves this issue, as shown in our experiments.

### E. Scalability

Figure 4 shows the scalability of our algorithm on all 5 datasets. Some cases were omitted due to memory limitations. We observe that our algorithm exhibits near-linear scalability. However, for the wb-edu and Archaea datasets, an increase in execution time is observed beyond 64 processes. This observation can be attributed to the nature of these smaller datasets as the time is already very small to observe any benefit of increased parallel resources.

### F. Application: Updating Protein Similarity Graph

We emphasize the application of our work for metagenomic protein similarity graphs. It has been demonstrated that a metagenomic sequence analysis pipeline consisting of preparing a graph and applying graph clustering techniques enables new scientific discovery [23]. Such an analysis pipeline involves analyzing a very large metagenomic database that is ever-increasing. For example, the database from which the above-mentioned discovery was made, currently contains more than 83 billion metagenomic sequences [21]. The same database contained 70 billion sequences in October 2022 and 77 billion sequences in August 2023. Generating a sequence similarity graph from such databases involves performing all-vs-all sequence alignment which is computationally very expensive for large databases like this. Hence, instead of building the graph from scratch whenever the database increments, it is more practical to update the existing graph with new vertices and edges.

## V. Conclusions

This paper introduces a linear-algebraic method for updating distributed streaming graphs. To address potential load imbalances caused by these updates, we incorporate a random permutation within the graph update process to maintain balanced loads across the graph. Our algorithm consists of three primary steps: matrix expansion, permutation, and addition. Although standard GraphBLAS operations like `GrB_assign` can be used for graph updates, our three-step approach is at least 25 times faster than the alternative method in CombBLAS, which relies on sparse matrix-matrix multiplication. By minimizing communication overhead and leveraging multithreaded local computations, our algorithm scales linearly up to thousands of cores on modern supercomputers.

## VI. Acknowledgment

This research was funded in part by DOE grants DE-SC0022098 and DE-SC0023349; by NSF grants PPoSS CCF 2316233 and OAC-2339607

## References

[1] A. Grewal, J. Jiang, G. Lam, T. Jung, L. Vuddemarri, Q. Li, A. Landge, and J. Lin, "RecService: Distributed real-time graph processing at twitter," in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, 2018.

[2] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 85–98.

[3] J. Park and K. Nahrstedt, "Navigation graph for tiled media streaming," in *Proceedings of the 27th ACM International Conference on Multimedia*, 2019, pp. 447–455.

[4] B. Schiller, S. Jager, K. Hamacher, and T. Strufe, "Stream-a stream-based algorithm for counting motifs in dynamic graphs," in *Algorithms for Computational Biology: Second International Conference, AlCoB 2015, Mexico City, Mexico, August 4-5, 2015, Proceedings 2*. Springer, 2015, pp. 53–67.

[5] A. McGregor, "Graph stream algorithms: a survey," *ACM SIGMOD Record*, vol. 43, no. 1, pp. 9–20, 2014.

[6] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–5.

[7] T. P. Raptis and A. Passarella, "A survey on networked data streaming with apache kafka," *IEEE access*, 2023.

[8] M. T. Hussain, G. S. Abhishek, A. Buluç, and A. Azad, "Parallel algorithms for adding a collection of sparse matrices," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2022, pp. 285–294.

[9] A. Azad, O. Selvitopi, M. T. Hussain, J. R. Gilbert, and A. Buluç, "Combinatorial BLAS 2.0: Scaling combinatorial algorithms on distributed-memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 989–1001, 2021.

[10] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[11] L. Dhulipala, J. Shun, and G. E. Blelloch, "Low-latency graph streaming using compressed purely-functional trees," *CoRR*, vol. abs/1904.08380, 2019. [Online]. Available: http://arxiv.org/abs/1904.08380

[12] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "LLAMA: Efficient graph analytics using large multiversioned arrays," in *2015 IEEE 31st International Conference on Data Engineering*, 2015, pp. 363–374.

[13] B. Wheatman and H. Xu, "Packed compressed sparse row: A dynamic graph representation," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–7.

[14] G. Feng, X. Meng, and K. Ammar, "DISTINGER: A distributed graph data structure for massive dynamic graph processing," in *2015 IEEE International Conference on Big Data (Big Data)*, 2015, pp. 1814–1822.

[15] S. Sallinen, R. Pearce, and M. Ripeanu, "Incremental graph processing for on-line analytics," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 1007–1018.

[16] O. Green and D. A. Bader, "cuSTINGER: Supporting dynamic graph algorithms for GPUs," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–6.

[17] F. Busato, O. Green, N. Bombieri, and D. A. Bader, "Hornet: An efficient data structure for dynamic sparse graphs and matrices on gpus," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, 2018, pp. 1–7.

[18] D. Makkar, D. A. Bader, and O. Green, "Exact and parallel triangle counting in dynamic graphs," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, 2017, pp. 2–12.

[19] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–11.

[20] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[21] I.-M. A. Chen, K. Chu, K. Palaniappan, A. Ratner, J. Huang, M. Huntemann, P. Hajek, S. J. Ritter, C. Webb, D. Wu *et al.*, "The IMG/M data management and analysis system v. 7: content updates and new features," *Nucleic Acids Research*, vol. 51, no. D1, pp. D723–D732, 2023.

[22] R. Leinonen, F. G. Diez, D. Binns, W. Fleischmann, R. Lopez, and R. Apweiler, "Uniprot archive," *Bioinformatics*, vol. 20, no. 17, pp. 3236–3237, 2004.

[23] G. A. Pavlopoulos, F. A. Baltoumas, S. Liu, O. Selvitopi, A. P. Camargo, S. Nayfach, A. Azad, S. Roux, L. Call, N. N. Ivanova *et al.*, "Unraveling the functional dark matter through global metagenomics," *Nature*, pp. 1–9, 2023.

[24] A. Azad, G. A. Pavlopoulos, C. A. Ouzounis, N. C. Kyrpides, and A. Buluç, "HipMCL: A high-performance parallel implementation of the markov clustering algorithm for large-scale networks," *Nucleic acids research*, vol. 46, no. 6, pp. e33–e33, 2018.

[25] O. Selvitopi, S. Ekanayake, G. Guidi, M. G. Awan, G. A. Pavlopoulos, A. Azad, N. Kyrpides, L. Oliker, K. Yelick, and A. Buluç, "Extreme-scale many-against-many protein similarity search," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–12.