An incremental GraphBLAS solution for the 2018 TTC Social Media case study

Márton Elekes
Budapest University of Technology and Economics
Department of Measurement and Information Systems
Email: elekes@mit.bme.hu

Gábor Szárnyas
Budapest University of Technology and Economics
Department of Measurement and Information Systems
MTA-BME Lendület Research Group on Cyber-Physical Systems
Email: szarnyas@mit.bme.hu

***Abstract*—Graphs are increasingly important for modelling and analysing connected data sets. Traditionally, graph analytical tools targeted global fixed-point computations, while graph databases focused on simpler transactional read operations such as retrieving the neighbours of a node. However, recent applications of graph processing (such as financial fraud detection and serving personalized recommendations) often necessitate a mix of the two workload profiles. A potential approach to tackle these complex workloads is to formulate graph algorithms in the language of linear algebra. To this end, the recent GraphBLAS standard defines a linear algebraic graph computational model and an API for implementing such algorithms. To investigate its usability and efficiency, we have implemented a GraphBLAS solution for the "Social Media" case study of the 2018 Transformation Tool Contest. This paper presents our solution along with an incrementalized variant to improve its runtime for repeated evaluations. Preliminary results show that the GraphBLAS-based solution is competitive but implementing it requires significant development efforts.**

I. Case study

This paper presents a GraphBLAS [8] solution for the "Social Media" case study of the 2018 Transformation Tool Contest [7]. The case study is defined using a familiar social network-like data model (Fig. 1), based on the schema of the LDBC Social Network Benchmark [5], and consists of Users and their Submissions. These submissions form a tree where the root node is a Post and the rest of the nodes are Comments. Users can like Comments and form "friends" relations with each other. Additionally, Comments have a direct pointer to the root Post to allow quick lookups. Fig. 3a shows an example graph with two Posts (`p1`, `p2`), three Comments (`c1`, `c2`, `c3`) and four Users (`u1`, …, `u4`). Solutions are required to compute two queries, denoted as Q1 and Q2.

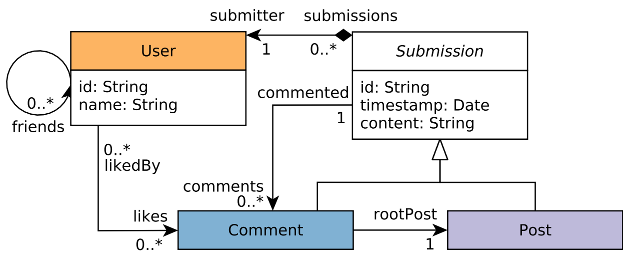
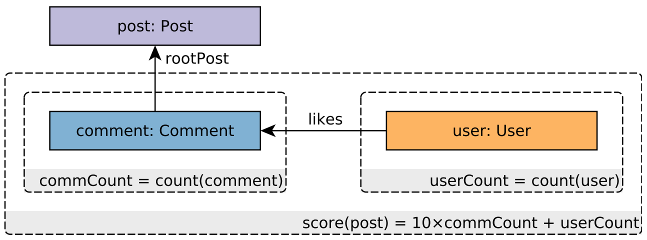
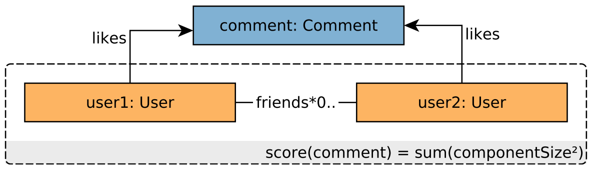


Fig. 1: Graph schema of the case study.



(a) Q1: influential posts.



(b) Q2: influential comments.

Fig. 2: Queries in the case study.

Q1: influential posts. Assign a score to each Post, defined as 10 times the number of their (direct or indirect) Comments plus the number of Users liking those Comments (Fig. 2a). Sort Posts according to their score and return the top 3.

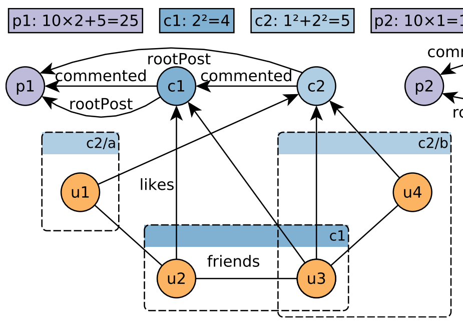
Q2: influential comments. Assign a score to each Comment, based on the friendships of the Users who like that Comment (Fig. 2b). Based on the graph formed by the User nodes and their friends edges, for every comment we define an induced subgraph which contains the Users who like the Comment and their "friends" relations. The subgraph contains connected components, i.e. groups of users who know each other directly or via friends. The score is defined as the sum of squared component sizes.

Updating the graph. The case study requires solutions to perform a number of inserts in the graph and return the results of the queries on the updated graph. Insertions are performed repeatedly, which favours solutions that use incremental maintenance techniques and avoid full recomputations.

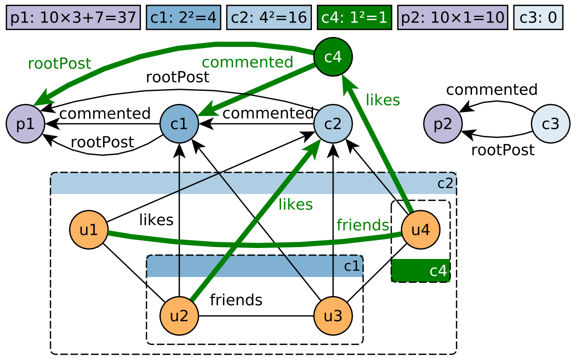
Fig. 3 shows the initial graph and the updated graph with the result scores of Q1 and Q2.

II. The GraphBLAS

A directed graph can be stored as a square adjacency matrix $\mathbf{A} \in \mathbb{N}^{n \times n}$, where rows and columns both represent nodes of



(a) Initial graph and scores. Comment `c2` has two components: `c2/a` consists of User `u1`, while `c2/b` consists of Users `u3` and `u4`. Its total score is the sum of the component sizes, i.e. $1^2 + 2^2 = 5$.



(b) Graph after performing an update that inserted six entities: (1) a friends edge between Users `u1` and `u4`, (2) a likes edge from User `u2` to Comment `c2`, (3) a Comment node `c4` with (4) an outgoing rootPost edge to Post `p1`, (5) an outgoing commented edge to Comment `c1`, and (6) an incoming likes edge from User `u4`. The changes have increased the score of Post `p1` and resulted in Comment `c2` having a single component of size 4, therefore receiving a score of $4^2 = 16$. Comment `c4` got a score of $1^2 = 1$.

Fig. 3: Example graphs: initial and updated versions.

the graph and cell A_{ij} contains the number of edges from node i to node j. If the graph is undirected, the matrix is symmetric. If the graph nodes and edges have type constraints, edges are stored per type, and the rows and columns of the matrix can represent source and target nodes of edges (resp.), whose number can differ.

An undirected graph can be stored as an incidence matrix $\mathbf{B} \in \{0,1\}^{n \times m}$, where rows and columns represent nodes and edges (resp.). Each column contains 1 for the source and the target vertex of the edge, otherwise 0.

GraphBLAS is a recently proposed standard built on the theoretical framework of matrix operations on arbitrary semirings [8], which allows defining graph algorithms in the language of linear algebra. The goal of GraphBLAS is to create a layer of abstraction between the graph algorithms and the graph analytics framework, separating the concerns of the algorithm developers from those of the framework developers and hardware designers. The GraphBLAS standard defines a C API that can be implemented on a variety of hardware components (including GPUs and FPGAs).

GraphBLAS stores graphs as sparse matrices which contain

GraphBLAS method	name	notation
`GrB_mxm`	matrix-matrix multiplication	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A} \oplus.\otimes \mathbf{B}$
`GrB_vxm`	vector-matrix multiplication	$\mathbf{w}^\intercal\langle\mathbf{m}^\intercal\rangle = \mathbf{u}^\intercal \oplus.\otimes \mathbf{A}$
`GrB_mxv`	matrix-vector multiplication	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{A} \oplus.\otimes \mathbf{u}$
`GrB_eWiseAdd`	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A} \oplus \mathbf{B}$
	set union	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{u} \oplus \mathbf{v}$
`GrB_eWiseMult`	element-wise,	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A} \otimes \mathbf{B}$
	set intersection	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{u} \otimes \mathbf{v}$
`GrB_extract`	extract submatrix	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A}(I, J)$
	extract subvector	$\mathbf{w}\langle\mathbf{m}\rangle = \mathbf{u}(I)$
`GrB_apply`	apply unary operator	$\mathbf{C}\langle\mathbf{M}\rangle = f(\mathbf{A})$
		$\mathbf{w}\langle\mathbf{m}\rangle = f(\mathbf{u})$
`GxB_select`	apply select operator	$\mathbf{C}\langle\mathbf{M}\rangle = f(\mathbf{A}, k)$
		$\mathbf{w}\langle\mathbf{m}\rangle = f(\mathbf{u}, k)$
`GrB_reduce`	reduce to vector	$\mathbf{w}\langle\mathbf{m}\rangle = [\oplus_j \mathbf{A}(:, j)]$
	reduce to scalar	$s = [\oplus_{ij} \mathbf{A}(i, j)]$
`GrB_transpose`	transpose	$\mathbf{C}\langle\mathbf{M}\rangle = \mathbf{A}^\intercal$
`GrB_build`	matrix from tuples	$\mathbf{C} \hookleftarrow \{\langle i, j, C_{ij}\rangle\}$
	vector from tuples	$\mathbf{w} \hookleftarrow \{\langle i, w_i\rangle\}$
`GrB_extractTuples`	extract $\langle i, j, A_{ij}\rangle$ tuples	$\{\langle i, j, A_{ij}\rangle\} \hookleftarrow \mathbf{A}$
	extract $\langle i, u_i\rangle$ tuples	$\{\langle i, u_i\rangle\} \hookleftarrow \mathbf{u}$

TABLE I: Notation of the GraphBLAS operations used in this paper (based on [3]). Matrix $\mathbf{A}$ contains scalar elements A_{ij}, vector $\mathbf{u}$ contains scalar elements u_i, i and j are row and column indices, I and J are subset of indices, $\oplus$ and $\otimes$ are addition and multiplication operators of an arbitrary semiring, mask $\langle\mathbf{M}\rangle$ is used to selectively write to the result.

elements as $\langle i, j, A_{ij}\rangle$ tuples. An optional mask can be used for operations, which limits the evaluation to the non-empty positions of the mask. Table I contains the list of GraphBLAS operations used in this paper.

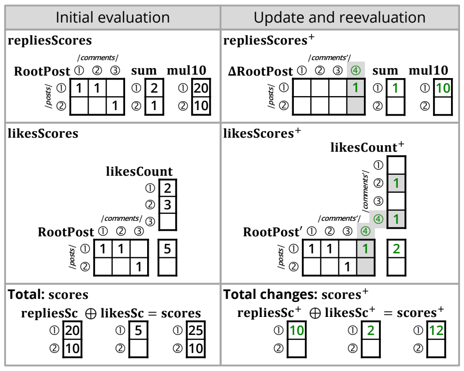
III. Solution

Q1 Batch. Alg. 1 computes the score for every post, then selects the top 3 posts. In Line 6 row-wise summation of **RootPost** matrix produces the number of comments per post, then a `GrB_apply` operation multiplies the vector elements by 10. Line 8 sums up the number of likes the post has via its comments. For each post, the **RootPost** adjacency matrix selects the cells of **likesCount** vector corresponding to the comments of the post, then sums up the values. The score for each post is the element-wise sum of the vectors (Line 9). Fig. 4a shows an example calculation.

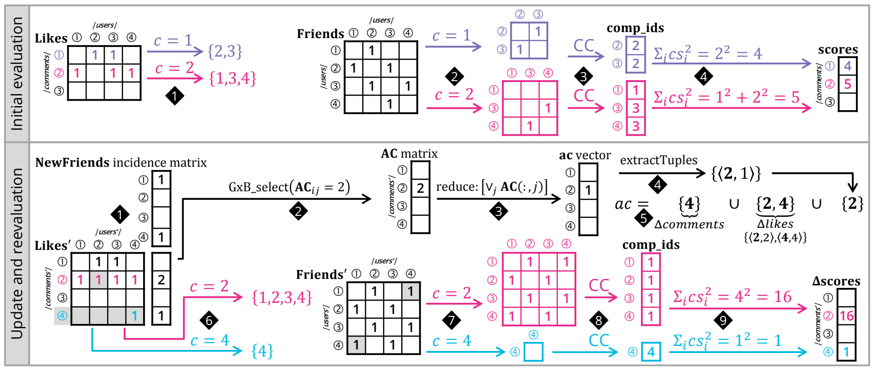
Algorithm 1 Calculate scores of query 1

```
 1: Input
 2:     RootPost ∈ $\mathbb{B}^{|posts| \times |comments|}$          ▷ adjacency matrix
 3:     likesCount ∈ $\mathbb{N}^{|comments|}$                  ▷ # of incoming likes
 4: Output
 5:     scores ∈ $\mathbb{N}^{|posts|}$
 6: sum ← $[\oplus_j$ RootPost$(:, j)]$                        ▷ row-wise sum
 7: repliesScores ← 10 × sum                          ▷ apply mul-by-10 op.
 8: likesScore ← RootPost $\oplus.\otimes$ likesCount
 9: scores ← repliesScores $\oplus$ likesScore
10: return scores
```

Q1 Incremental. To incrementally evaluate Q1, Alg. 2 updates the scores for next evaluations and returns the posts with new scores. Merging the previous top 3 scores and the new ones yields the new result (new scores overwrite existing ones). Lines 9 and 10 compute the increment of the score induced by new comments. In Line 11 the number of likes the comments



(a) Q1.



(b) Q2. CC: connected components, **comp_ids**: component ids, cs_i: size of component i.

Fig. 4: Execution of the algorithms on the example graph: initial evaluation and incremental maintenance. Recall that the update in the example inserts the following relevant entities (highlighted with grey background): a friends edge between Users `u1` and `u4`, a likes edge from User `u2` to Comment `c2`, a Comment node `c4` with an outgoing rootPost edge to Post `p1` and an incoming likes edge from User `u4`.

newly received are summed up per post. Two types of increments are summed up in Line 12. For subsequent evaluations the scores are updated using the increment vector (Line 13). To find the top 3 scores only the previous maximum values and the posts with updated scores are considered. Line 14 yields the score values which changed by assigning the $\mathbf{scores}'$ vector via the $\mathbf{scores}^{+}$ increment vector as a mask, which allows values in the result only if the mask has a value at the corresponding position. Fig. 4a shows an example calculation.

Algorithm 2 Update scores of query 1

```
 1: Input
 2:     $\mathbf{scores} \in \mathbb{N}^{|posts'|}$                                   ▷ previous scores
 3:     $\mathbf{likesCount}^{+} \in \mathbb{N}^{|comments'|}$                        ▷ new incoming likes
 4:     $\mathbf{\Delta RootPost} \in \mathbb{B}^{|posts'|\times|comments'|}$         ▷ new rootPost edges
 5:     $\mathbf{RootPost}' \in \mathbb{B}^{|posts'|\times|comments'|}$               ▷ all rootPost edges
 6: Output
 7:     $\mathbf{\Delta scores} \in \mathbb{N}^{|posts'|}$                            ▷ only changed scores
 8:     $\mathbf{scores}' \in \mathbb{N}^{|posts'|}$                                  ▷ all scores
 9: $\mathbf{sum} \leftarrow \left[\bigoplus_j \mathbf{\Delta RootPost}(:, j)\right]$   ▷ # of new comments
10: $\mathbf{repliesScores}^{+} \leftarrow 10 \times \mathbf{sum}$
11: $\mathbf{likesScore}^{+} \leftarrow \mathbf{RootPost}' \oplus.\otimes \mathbf{likesCount}^{+}$
12: $\mathbf{scores}^{+} \leftarrow \mathbf{repliesScores}^{+} \oplus \mathbf{likesScore}^{+}$   ▷ score increment
13: $\mathbf{scores}' \leftarrow \mathbf{scores} \oplus \mathbf{scores}^{+}$                   ▷ update scores
14: $\mathbf{\Delta scores}\langle\mathbf{scores}^{+}\rangle \leftarrow \mathbf{scores}'$      ▷ updated scores where changed
15: return $\mathbf{\Delta scores}, \mathbf{scores}'$
```

Q2 Batch. The batch evaluation of Q2 is depicted in the upper part of Fig. 4b. The algorithm computes the score for every comment, then selects the top 3 comments. To collect the users of each subgraph, Step ⟨1⟩ extracts the elements of **Likes** matrix as $\langle c, u, 1\rangle$ tuples and collects them into sets of user IDs (u) per comment (c). To produce the subgraph, for each comment Step ⟨2⟩ extracts a submatrix based on the users selected. Step ⟨3⟩ finds connected components in the subgraph using the FastSV algorithm [11] of the LAGraph library [9]. This produces a vector containing the component id for every user. Step ⟨4⟩ yields the squared sum of component sizes, i.e. the score for each comment.

Q2 Incremental. The incremental evaluation of Q2 is depicted in the lower part of Fig. 4b. The algorithm returns the comments with new scores ($\mathbf{\Delta scores}$) by reevaluating the comments which the updates might impact on. Merging the previous top 3 scores and the new ones yields the new result (new scores overwrite existing ones). The first phase of the algorithm (Steps ⟨1⟩–⟨5⟩) collects the comments which might be affected by the updates (ac set), then the second phase (Steps ⟨6⟩–⟨9⟩) computes the new scores of these comments using the batch algorithm already mentioned.

A comment might be affected by an update if (1) it is a new comment, (2) the comment receives a new incoming likes edge from a user, resulting in a new component or the expansion of an existing one, or (3) two users who like the comment become friends, which merges the components the users belong to (if the components differ). Step ⟨5⟩ collects the IDs of these comments.

Steps ⟨1⟩–⟨4⟩ compute the comments which might be affected by new friends edges. **NewFriends** incidence matrix represents each new friendship by a column having two 1-valued cells for the two users. For every new friendship (i.e. pair of users) Step ⟨1⟩ computes how many user of the pair likes each comment (0, 1, or 2). During the matrix-matrix multiplication each new column of friendships selects two columns of $\mathbf{Likes}'$ matrix and sums them up into **AC** matrix. Step ⟨2⟩ keeps only 2-valued cells, i.e. where both users of a friendship liked the comment, so they are present in the subgraph and the new friendship might merge components. Then Step ⟨3⟩ produces a row-wise sum using *binary or* operation. Step ⟨4⟩ extracts $\langle c, 1\rangle$ tuples from the result vector and collects the comment IDs from these tuples. Step ⟨5⟩ collects all the comments which might be affected by the update. The next steps reevaluate the scores of these comments.

IV. Evaluation

To evaluate the performance and scalability of our solution, we have used the benchmark framework of the case study [7].

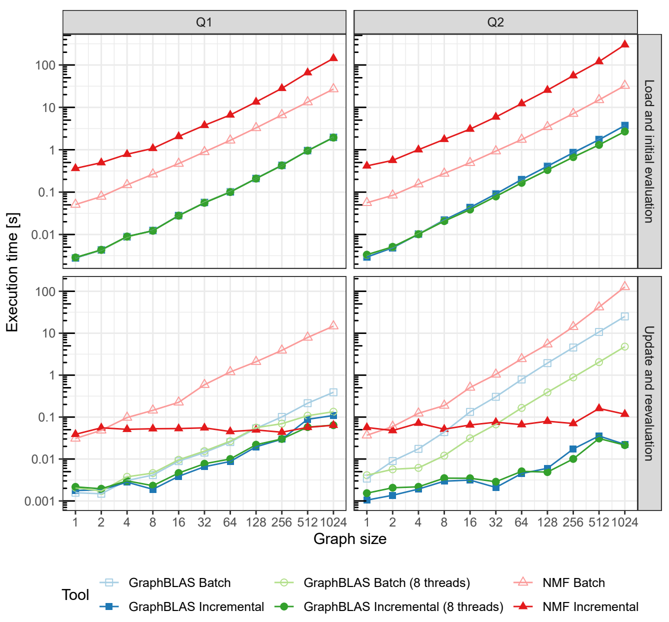


Fig. 5: Execution times of the queries with respect to the graph sizes in the *load and initial evaluation*, and the *update and reevaluation* phases. Both axes are logarithmic.

	1	2	4	8	16	32	64	128	256	512	1024
#nodes	1274	2071	4350	7530	15k	30k	58k	115k	225k	443k	859k
#edges	2533	4207	9118	18k	35k	71k	143k	287k	568k	1.1M	2.3M
#inserts	67	120	132	104	110	117	68	86	45	112	74

TABLE II: Graph sizes w.r.t. to the scale factor.

Our GraphBLAS solution was implemented using SuiteSparse:GraphBLAS [3]. The complete solution consists of approx. 1100 lines of C++ of code and is available open-source[1]. As a performance baseline, we used the reference implementation of the case study, written in the .NET Modeling Framework [6] (NMF Batch) and its incremental version (NMF Incremental). As described in Sec. III, we have implemented two variants: GraphBLAS Batch always performs a full evaluation, while GraphBLAS Incremental performs a full evaluation during the first step, then switches to incremental maintenance for the subsequent steps. We compared single- and multi-threaded performance of our GraphBLAS solution using 8 threads for the latter. The GraphBLAS implementation we used has built-in parallelization of the operators [1], additionally, we parallelized Q2 using OpenMP constructs at the granularity of comments. We ran the benchmark on synthetic graphs of increasing sizes following powers of 2. The elements in the graphs follow the Facebook-like distribution enforced by the LDBC Datagen [5]). For each graph, the number of nodes/edges and the number of inserted elements are shown in Table II. We ran the computation on each graph size 5 times and report the geometric mean value of these runs.

[1]https://github.com/TransformationToolContest/ttc2018liveContest/

We executed the benchmark on a cloud virtual machine with a 24-core Intel® Xeon® Platinum 8167M CPU with Hyper Threading at 2.00 GHz, 320 GB RAM, and HDD storage. The machine was running the Ubuntu 18.04 operating system, and the .NET Core 3.1.100 runtime. The GraphBLAS solution was using SuiteSparse:GraphBLAS 3.2.0draft20 compiled with GCC/G++ 7.4.0.

The execution times are shown in Fig. 5. Both tools scale similarly for the *load and initial evaluation* phase. GraphBLAS is the fastest, while the incremental NMF variant is the slowest as it initially builds a dependency graph from the query to assist incremental change propagatation. During the *update and reevaluation*, both tools gain significant performance benefits from incrementalization as they scale better for large graph sizes. GraphBLAS has similar execution times for Q1 as NMF, and outperforms NMF for Q2. Parallel processing of updates in GraphBLAS has a small performance gain for the incremental version as the updates are small. However, for GraphBLAS Batch, the difference is half of an order of magnitude in favour of the parallel version as it requires a costly recomputation over the whole graph, which negates the parallelization overhead.

V. Conclusion and Future Work

This paper presented a linear algebraic solution for the "Social Media" case study of the 2018 Transformation Tool Contest. While the presented solution already exhibits good performance and scalability compared to the reference implementation, a number of optimizations could be applied as future work: (1) using updatable compressed matrix representation formats such as faimGraph [10] or Hornet [2] and (2) running an incremental connected components algorithm [4] in Step ⟨8⟩ of Q2. Additionally, it would be interesting to investigate the performance of the solution in the presence of more realistic update operations, including both insertions and removals.

References

[1] M. Aznaveh et al. Parallel GraphBLAS with OpenMP. In *CSC at PP*. SIAM, 2020.
[2] F. Busato et al. Hornet: An efficient data structure for dynamic sparse graphs and matrices on GPUs. In *HPEC*. IEEE, 2018.
[3] T. A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM TOMS*, 2019.
[4] D. Ediger et al. Tracking structure of streaming social networks. In *IPDPS*, pages 1691–1699. IEEE, 2011.
[5] O. Erling et al. The LDBC Social Network Benchmark: Interactive workload. In *SIGMOD*, pages 619–630, 2015.
[6] G. Hinkel. NMF: A multi-platform modeling framework. In *ICMT*, 2018.
[7] G. Hinkel. The TTC 2018 Social Media case. In *TTC at STAF*, 2018.
[8] J. Kepner et al. Mathematical foundations of the GraphBLAS. In *HPEC*, 2016.
[9] T. Mattson et al. LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS. In *GrAPL at IPDPS*, 2019.
[10] M. Winter et al. faimGraph: High performance management of fully-dynamic graphs under tight memory constraints on the GPU. In *SC*. IEEE / ACM, 2018.
[11] Y. Zhang, A. Azad, and Z. Hu. FastSV: A distributed-memory connected component algorithm with fast convergence. In *PP*. SIAM, 2020.