2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) | 979-8-3503-0548-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/SBAC-PAD59825.2023.00020

A Distributed Algorithm for Identifying Strongly Connected Components on Incremental Graphs

S. Srinivasan*, A. Khanda†, S. Srinivasan ◂ *, A. Pandey‡, S. K. Das†, S. Bhowmick‡, and, B. Norris*

*Department of Computer Science, University of Oregon, Eugene, OR 97403, USA
†Department of Computer Science, Missouri University of Science and Technology, Rolla, MO 65401, USA
‡Department of Computer Science and Engineering, University of North Texas, Denton, TX 76201, USA
◂Department of Management Information Systems, Bowie State University, MD 20715

***Abstract*—Incremental graphs that change over time capture the changing relationships of different entities. Given that many real-world networks are extremely large, it is often necessary to partition the network over many distributed systems and solve a complex graph problem over the partitioned network. This paper presents a distributed algorithm for identifying strongly connected components (SCC) on incremental graphs. We propose a two-phase asynchronous algorithm that involves storing the intermediate results between each iteration of dynamic updates in a novel meta-graph storage format for efficient recomputation of the SCC for successive iterations. To the best of our knowledge, this is the first attempt at identifying SCC for incremental graphs across distributed compute nodes. Our experimental analysis on real and synthesized graphs shows up to 2.8x performance improvement over the state-of-the-art by reducing the overall memory utilized and improving the communication bandwidth.**

***Index Terms*—Dynamic graphs, Distributed systems, Strongly connected components.**

I. Introduction

Detecting Strongly Connected Components (SCCs) in a large directed graph is a fundamental graph analytics problem. An SCC is defined as a subset of vertices in a directed graph with a path from any vertex to every other vertex in that subset. A graph can have many SCCs, but vertices are mutually exclusive to these SCCs. Detecting SCCs has many applications, such as pattern matching [1], topological sort [2], and graph analytics [3]. Although detecting SCCs by Depth First Search (DFS) on a directed graph works well for a sequential approach, performing a DFS can be expensive and computationally challenging in a parallel architecture [4].

Incremental graphs are extended graph data structures that undergo continuous updates, such as the addition of nodes and edges, which present numerous additional challenges. One of the foremost concerns is maintaining performance efficiency, as updates to the graph can impact the functionality of graph-based algorithms, potentially necessitating full recomputation. Another challenge lies in ensuring data consistency post-updates, as changes to a single node or edge could impact the overall graph or its segments. Memory management also poses a significant issue, particularly with large-scale graphs that rapidly consume memory resources, requiring efficient storage and retrieval methods such as graph partitioning or compression. Traditional graph algorithms designed for static graphs may not be practical for these dynamic, ever-evolving structures, hence calling for the development of dynamic algorithms. Query processing in such a fluid environment becomes complex, as results must be recalculated after every modification. If updates to the graph are performed by different processes or threads concurrently, managing these simultaneous alterations to maintain data integrity and consistency becomes a substantial challenge, often needing locking or transaction management mechanisms. Lastly, like all data types, incremental graphs have concerns regarding data privacy and security. These issues become even more pressing in distributed environments where it is imperative to ensure that updates are authorized, and information remains uncompromised. Overcoming these challenges necessitates a blend of advanced algorithms, effective data management practices, and adept software engineering techniques, as explained in [5].

To overcome the challenges of parallelizing DFS on static networks, FW-BW (Forward-backward) approach was proposed [6]. To further improve the performance, trim techniques which fast reduce a large number of trivial SCCs (e.g., with one or two vertices, called trim-1 and trim-2, respectively) are introduced by [7]. Machine learning based optimizations have also been proposed in [8] and [9] for shared memory systems. To the best of our knowledge, there has only been one attempt to detect SCC on distributed networks [10]. For the most part, the state-of-the-art parallel approaches for detecting SCCs are optimized for shared-memory systems.

Figure 1, shows an example workflow of computing SCC for an incremental graph that changes over time. At each timestep T^n, a new batch of edge insertions are applied over the graph at time T^{n+1}. This type of edge addition is a very common workflow in scientific simulations that periodically keep updating the graph databases. Currently, the standard approach to identify SCC in such a network is to recompute the SCC over the entire network every timestep, which is a costly operation. Although there have been attempts at an adaptive approach for other algorithms such as minimum spanning tree [11], single source shortest path [12], [13] and vertex coloring [14], adaptive approaches for identifying SCCs are limited in comparison. A recent approach for adaptive SCC detection is proposed in [15], which provides the Las Vegas algorithm for DAGs. However, they don't make considerations

2643-3001/23/$31.00 ©2023 IEEE
DOI 10.1109/SBAC-PAD59825.2023.00020

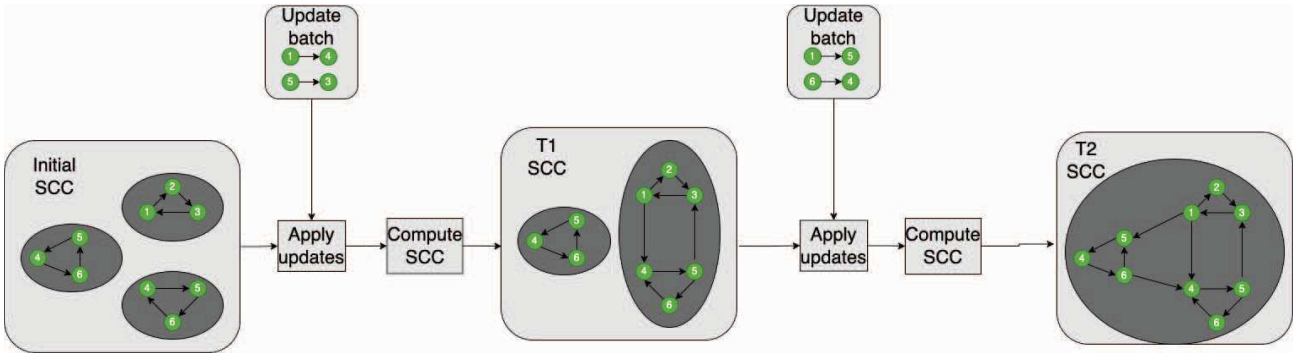


Fig. 1: Example workflow for computing SCC on incremental graphs. At each timestep T, a new batch of updates is added and SCC is recomputed.

for parallel or distributed scalability. Considering incremental SCC detection is an unbounded problem, as explained in [16], there isn't a theoretical algorithm that solves it in polynomial time when there are both vertex and edge updates. As a result, we focus our work strictly on edge additions alone, keeping the number of vertices constant.

Our key contributions are as follows:

- A novel meta-graph storage format for caching intermediate SCC results after every new insertion batch. This format gives us a reduced-size graph for computing further SCCs after dynamic edge additions.
- DistSYNC, an asynchronous and distributed memory algorithm for identifying new SCCs after dynamic edge additions on the meta-graph format.
- A distributed memory implementation of DistSYNC using YGM [17], an asynchronous communication framework on top of MPI.

To the best of our knowledge, we propose the first distributed algorithm for incremental SCC detection. We also propose the first asynchronous incremental algorithm for SCC detection. The central concept behind meta-graph storage is the fact that we can treat identified components as meta-vertices and traverse over them to find new components rather than traversing every vertex in the graph. The rest of the paper is organized into five sections. In section II, we introduce the required background information and related works. Section III dives into explaining the DistSYNC algorithm with the help of examples. The implementation details and experimental evaluations are covered in sections IV and V, respectively. The final concludes the paper with directions for future works.

II. Background and Related Work

Detecting SCCs in a network is a well-studied problem. This section discusses related work on detecting SCCs in a large-scale network. Our approach utilizes some of the concepts, so this section also provides prerequisite background information.

A. Sequential SCC

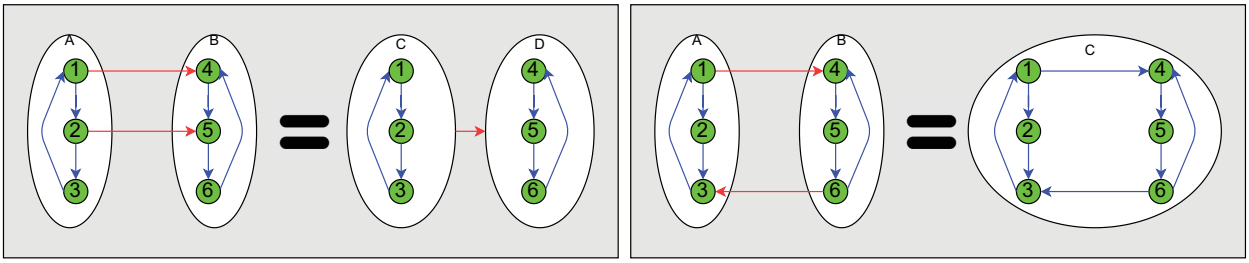
Tarjan's [18] implementation is the well-known sequential algorithm for detecting SCC. It uses DFS (depth-first search), and the complexity is $\mathcal{O}(V+E)$, where V is the number of vertices and E is the number of edges. There are many attempts to parallelize Tarjan's approach; however, all demonstrate poor scalability.

The Forward-Backward (FW-BW) algorithm [6] uses a recursive approach. The algorithm can be described as follows: Let V be the set of the vertices in the graph $G(V,E)$, $O(V)$ be the set of all outgoing edges in the graph, and $I(V)$ set of all incoming edges. Now for a given graph $G(V,O(V))$, a random pivot vertex u is selected, and then a BFS is performed on $G(V,O(V))$ from a pivot vertex u to detect vertices (Let the set of these vertices be D) that can be reached from u. Next, another BFS is performed on $G(V,I(V))$ from the pivot vertex u, and a backward search is done where those vertices that can reach u are selected and inserted into P. The intersection of D and P forms SCC, which has the pivot element. Now from the original graph, the vertices identified in SCC are removed, and the FW-BW approach is recursively called on the remaining sets and the disjoint sets obtained after removing the vertices part of SCC from D and P. In the best-case scenario, it takes $\mathcal{O}(n \log n)$ to detect SCC. This approach was further improvised using trimming, which removes the vertices with zero in-degree and out-degree. Trimming reduces the number of vertices in FW-BW sets and speeds up the overall performance.

B. Shared Memory SCC

Ji et al. [10] proposed a novel synchronization paradigm, called R-sync, to spanning tree-based detecting of SCC. This approach provides many benefits, such as early termination for conventional bottom-up traversal. The early termination allows them to check only a few neighbors and reduces the traversal compared to the conventional synchronization approach.

Hong et al. [7] identified the potential limitation of the FW-BW-Trim approach on large real-world networks. They proposed an extension of the FW-BW approach, which considers the characteristics of the dataset instances, such as the small-world property. Their implementation was the first attempt to develop a parallel algorithm to detect SCC and outperform the sequential Tarjan [18] implementation. Based on the small-world property, they have identified that wiring a few edges in the diameter of a real-world graph can shrink its size. Their



(a) Lemma 1 (b) Lemma 2

Fig. 2: Example for Lemmas 1 and 2. Both the right-hand sides have the same structural properties as the left-hand sides.

main idea is to expand trimming operation and decomposing after the initial SCC is found based on partitioning on weakly connected components.

Slota et al. [19] proposed a shared memory multistep approach that uses a parallel BFS and graph coloring. They have used variants of FW-BW and applied Orzan's coloring method. To minimize synchronization, they avoid using locks. Their experiments on real-world graphs show better scalability on low-diameter networks. The coloring approach is also similar to FW-BW with some modifications. Instead of just using one pivot, it uses multiple pivots. We use this approach to perform multi-threaded SCC within distributed processes.

C. GPU Implementation

Li et al. [20] proposed a GPU implementation of detecting SCC using the FB-BW-Trim algorithm. They present a hybrid method that allows the adoption of different parallelism strategies for various graph properties. Barnet et al. were the first to implement the FB-Trim algorithm using CUDA. Stuhl [21] extended the work by introducing an extended graph traversal implementation. Sthul ran experiments on the synthetic network and demonstrated good performance on synthetic networks and when running on real-world networks, except for one. The reason for poor scalability was due to the nature of the real-world graphs and skewed component sizes. Li et al. [20] implement a hybrid method that detects SCC in two phases. In the first phase, the algorithm is only focused on detecting a single large SCC. In the second phase, the remaining small-sized subgraphs are processed. It is shown that identifying small-sized SCCs takes more time than identifying a single large SCC.

D. Dynamic Graphs

Attempts at getting batched/snapshot-based frameworks for graph algorithms are explored in [22] and [23]. STINGER [24] is a shared memory solution that can ingest structural changes at a rate of 10 million events per second with an updating kernel peak rate of around 1 million events per second. A Shared memory parallel algorithm for weakly connected components on dynamic graphs is given in [25] while a distributed implementation is given by [26]. There have been no parallel implementations for strongly connected components on graphs with edge insertions, let alone distributed algorithms.

III. Methodology

Before discussing the details of the DistSYNC algorithm, we must first explain the motivation behind this algorithm, which stems from two lemmas on the structural property of graphs and strongly connected components.

Lemma 1. *Given two SCCs A and B, if there is a forward edge between any vertex in SCC A to a vertex in SCC B, we could say there is a forward path between all vertices in SCC A to all vertices in SCC B.*

Proof. If there exists a forward edge $e(a_k, b_k) \in E$ such that $\text{SCC}(a_k) = A$ and $\text{SCC}(b_k) = B$, then there exists a direct forward path $a_k \xrightarrow{Fwd} b_k$. Let, $SCC(A) = \{a_0, a_1, \ldots, a_k, \ldots, a_i\}$ and, $SCC(B) = \{b_0, b_1, \ldots, b_k, \ldots, b_j\}$. By definition of SCC, $a_x \xrightarrow{Fwd} a_k$ and $b_k \xrightarrow{Fwd} b_y \; \forall x, y$ where $x \in [0, i]$ and $y \in [0, j]$ where " $\xrightarrow{Fwd}$ " *denotes the existence of forward path.* By transitive property, $a_x \xrightarrow{Fwd} a_k \xrightarrow{Fwd} b_k \xrightarrow{Fwd} b_y$. Thus, $a_x \xrightarrow{Fwd} b_y : \forall x, y \; SCC(a_x) = A; SCC(b_y) = B$ □

In Figure 2a, there are two SCCs with labels A and B. Each SCC has three vertices, each labeled 1 through 6. There exists a forward edge between vertices 1 and 2 in SCC A to vertices 4 and 5 in SCC B. Since there is a forward path between all vertices within an SCC, we can say that vertex 1 in SCC A is reachable from vertex 3. Similarly, there is a forward path between vertex 4 and vertex 6. This added to the fact that there is a forward path between vertex 1 and 4 means there is a forward path between vertex 3 and vertex 6 through vertices 1 and 4 even though there is no direct edge between them.

Because of this structural property, maintaining any other edge across two SCCs is redundant information when identifying the components. As we can see on the RHS of Figure 2a, all the inter-SCC connections are replaced with one forward path between the two SCCs. By doing so, we can reduce the number of inter-SCC edges while representing the same structural information of the graph. This lemma could be extrapolated to any SCCs that are bigger than the provided example as long as there is one forward path that connects both the SCCs acting as a one-way bridge between the SCCs.

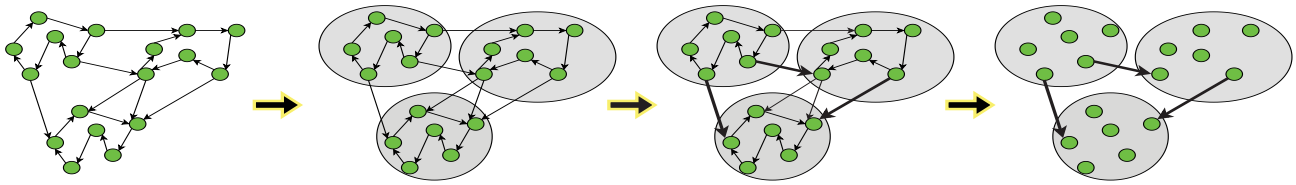


Fig. 3: Converting initial graph to meta-graph. The initial graph is shown in the leftmost part. They are then segregated into different SCCs, which we refer to as meta-vertices. Only one representational edge that traverses two meta-vertices is converted as a meta-edge while the rest is discarded. We finally get a meta-graph with three meta-vertices and three meta-edges

Lemma 2. *Given two SCCs, A and B, if there is a forward and backward path between them, then the vertices in both the SCCs could be merged into a single component.*

Proof. Let $S_1 = \{a_0, a_1, \ldots a_f, \ldots a_b, \ldots a_m\} \in SCC(A)$, $S_2 = \{b_0, b_1, \ldots b_f, \ldots b_b, \ldots b_n\} \in SCC(B) : \forall a, b \in V$.

Given $e_f(a_f, b_b) \in E : a_f \xrightarrow{Fwd} b_b$*(direct forward path)*

also, $e_b(b_f, a_b) \in E : a_b \xleftarrow{Bwd} b_f$*(direct backward path)*

Now, $\forall x \in [0, m]$

$a_x \xrightarrow{Fwd} S_1 \cup S_2$ *S1 by definition of SCC; S2 through e_f and Lemma 1*

$a_x \xleftarrow{Bwd} S_1 \cup S_2$ *S1 by definition of SCC; S2 through e_b and Lemma 1*

Similarly,$\forall y \in [0, n]$

$b_y \xleftarrow[Bwd]{Fwd} S1 \cup S2$ *" $\xleftarrow[Bwd]{Fwd}$ " denotes existence of path from both direction*

Therefore, $a_x \xleftarrow[Bwd]{Fwd} b_y$

Hence, we can merge A and B into single SCC C such that C consists of all vertices in $S_1 \cup S_2$ □

On the LHS of Figure 2b, there are two SCCs with three vertices, each labeled 1 through 6. We can see that there is a forward path from vertex 1 to vertex 4 while there is a backward path to vertex 3 from vertex 6. Once again, from both the fact that there is a forward path between any two vertices within the same SCC and there is a forward and backward path between the two SCCs, we can say there is a forward path between any two vertices from both the SCCs and hence all the vertices belong to the same SCC. This is represented on the right side of Figure 2b.

By applying Lemma 2, we can represent two SCCs residing in different processes as a single component, thereby reducing the unique components we need to represent the same structural information of the graph. Like Lemma 1, this can be extrapolated to any number of SCCs of any size as long as a forward and backward path exists among them.

A. Meta-graphs

We introduce the meta-graphs $G''(MN, ME)$ as an abstraction on top of the existing graph where the vertices (referred to as meta-nodes(MN) or meta-vertices), are the SCCs. The edges of G'', which are referred to as meta-edges(ME), are the directed edges that connect two meta-vertices. They are created by first identifying the various SCCs in the original graph. These SCCs act as the meta-vertex, with each meta-vertex comprising all the vertices that belong to that SCC. From lemma 1, we know that for computing SCC, when multiple redundant edges traverse two different SCCs, then they could be represented using a single edge while still holding on to the same structural property. Thus a single representational edge that traverses the two SCCs/meta-vertex is chosen as the meta-edge while the rest is discarded.

We take into account the direction of the edge so that only edges that traverse in the same direction are considered redundant, and a representational edge is picked from them. Meta-graphs are much smaller than the original graph because all the vertices belonging to an SCC can be represented using a single label (color). For instance, the original graph with 18 vertices and 23 edges in Figure 3 is converted to a meta-graph with three meta-vertices and three meta-edges. It is also to be noted that initially, a meta-graph is directed and acyclic (DAG), as multiple SCCs that form a cycle cannot exist without being absorbed into a single SCC. We will be leveraging this fact to overlay dynamic edge additions on top of this meta-graph to see if new cycles are formed to identify updated SCC. Storing graphs in a meta-graph format enables us to recompute the SCC on a reduced-size meta-graph rather than the larger original graph. Also, a lot of real-world graphs consist of many large SCCs, which in turn could be represented using a single label. Hence, meta-graph abstraction drastically reduces the size of these graphs. This is one of the key factors for the improved performance of our algorithm.

B. Forward color propagation and backward confirmation messages

Forward color propagation sends the color of a pivot meta-vertex to its next neighbors, along with the accumulated size of all the meta-vertices in that chain originating from the pivot. These are implemented as functions that can be executed by the neighbor vertex using a remote procedure call (RPC). The neighbors recursively keep calling that function and pass it further to their neighbors until the base condition is reached. This is a mechanism we will use in DistSYNC to identify whether a chain is a cycle. Likewise, backward confirmations are recursive messages sent to the previous sender of a forward color propagation message notifying the presence of a cycle for that chain along with its total size. Meta-vertices asynchronously fire these messages to destination meta-vertices when they are asked to and go on to wait for further messages.

C. DistSYNC

The critical steps in our distributed algorithm for SCC are bookkeeping the forward and backward messages sent from different pivots. Figure 4 explains the workflow of our algorithm using a sample graph with four initial SCCs, denoted by four colors, which we partitioned across two distributed ranks. The four components are denoted using the starting letters of their respective colors, namely, R, G, B, Y. It is to be assumed that the vertices within the same color are interconnected, but for convenience, we show only the edges that go across different colors denoted by a single arrow. Newly updated edges that arrive in the first time step are denoted by dotted arrows. We can see that in the initial phase, there are two existing edges, RG and GB, along with two updated edges, BR and BY.

In *phase one*, we build a meta-graph with four meta-vertices denoting the four colors and four meta-edges denoting the new and existing edges between them. Then to trigger the start of forward checks in *phase two*, all the meta-vertices that are a source to a newly updated edge that traverses to another meta-vertex are considered pivots. They then forward propagate their colors and size to the respective destination meta-vertex. In Fig 4, the solid yellow connectors denote the forward propagation through BR and BY as they are newly updated meta-edges. R, upon receiving a propagated message from B, forwards that same color downstream to G along with the combined size of both of them. This is shown in fig 4 with a solid yellow connection RG. Throughout the forward chain, everyone propagates the pivot color along with the updated size in the chain. When B receives a forward message from G with itself as the pivot, it recognizes that it is a cycle. When a cycle is detected, it checks if its current size is less than the size of the chain and triggers a backward confirmation to the sender of that message. In this case, the size of the chain is 16, which is greater than that of B. So B updates its colors and triggers a backward confirmation with the new size to G, which in turn does the same to R. So R and G are now essentially subsumed by B, with 16 being their new size.

A case where the own size of the meta-vertex is greater than the size taken from the backward confirmation message could only happen if that vertex has been updated to a new color after it sent out the previous forward message. In that case, it doesn't propagate backward confirmation but instead asks the successor who sent the backward confirmation to revert to its previous state. By that point, it would already have sent forward propagation messages with the new chain sizes and pivot.

The algorithm terminates when there are no more forward, backward, or revert messages to send through the entire network, which would mean every meta-node is correctly updated to reflect the new component it belongs to and the new size. Individual forward, backward, and revert messages with different pivots would flow back and forth through the network until everyone hits a stable state and there are no more new messages. Since this is completely asynchronous, none of the meta-vertices wait in anticipation of a backward confirmation after a forward check. The meta-vertices process these messages as they are received and go back to listening for further messages until the algorithm terminates.

In essence, DistSYNC is an extended version of cycle detection but over a meta-graph. Detecting cycles are expensive, but we highlight two key advantages that ensure that DistSYNC is efficient. Since it operates over a meta-graph, the number of edges it traverses to detect a cycle is dramatically reduced. Also, the number of pivot points that trigger forward checks is limited to only the meta-nodes that have newly updated meta-edges. This number is dependent on the size of the update batch; a larger batch means more pivot points that slow down the entire process, as we discuss in Section V, but we can cleverly partition the update batch to maximize efficiency. In the worst case, every meta-edge can be traversed to find the cycle giving it an amortized time and communication complexity of $\mathcal{O}(MV + ME)$. Due to the usage of distributed hash tables from YGM, each process only stores the meta-vertices that belong to it. Hence the space complexity is $\mathcal{O}((V + E)/N)$ where N is the number of processes.

Algorithm 1 Forward and backward propagation

```
1:  procedure CHECKFORWARD(pivot, sz)
2:      if self == pivot then
3:          if sz > size then
4:              size = sz
5:              Forwardsender
        ⟶ConfirmBackward(pivot, sz)
6:          end if
7:      end if
8:      if self != pivot then
9:          sz= sz + size
10:         for Each forward neighbor y do
11:             y ⟶CheckForward(pivot, sz)
12:         end for
13:     end if
14: end procedure
15: procedure CONFIRMBACKWARD(pivot, sz)
16:     if sz > size then
17:         color(self) = color(pivot)
18:         size = sz
19:         Forwardsender
        ⟶ConfirmBackward(pivot, sz)
20:     end if
21:     if sz < size then
22:         confirmationsender ⟶Revert(pivot)
23:     end if
24: end procedure
25: procedure REVERT(pivot)
26:     revert to the previous state
27:     confirmationsender ⟶Revert(pivot)
28: end procedure
```

Algorithm 1 explains the procedures required for forward and backward propagation, respectively, while algorithm 2

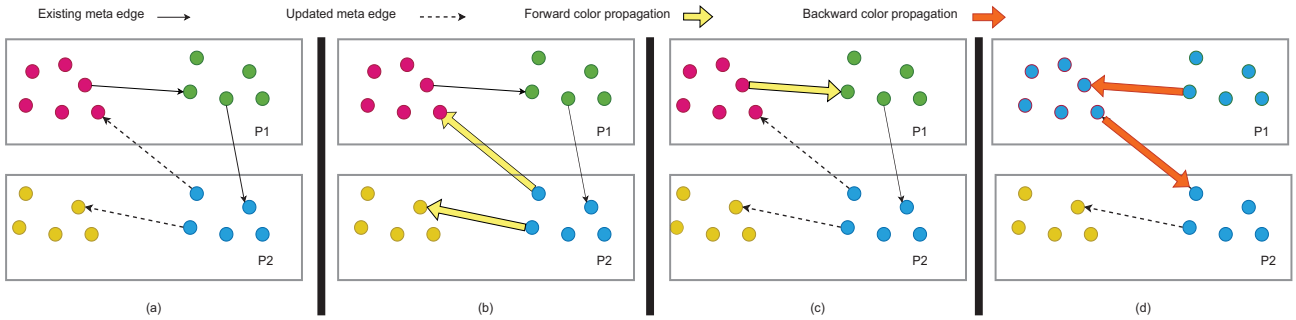


Fig. 4: Example workflow of DistSYNC. The initial meta-graph is shown in (a) with four different SCCs/meta-vertices split across two ranks p1 and p2. Blue forward propagates its color to yellow and red based on the updated edge in (b). In (c), red, in turn, propagates blue forward to its neighbor green. Green changes its color to blue and confirms it backward to red in (d). Red does the same and propagates it backward to blue. Now everyone in the cycle is blue.

puts everything together and explains the full DistSYNC algorithm. An arrow denotes a remote-procedural call where the sender asks the LHS of the arrow to execute the function on the RHS with the given arguments. After constructing the initial meta-graph, every process spawns a non-blocking listener thread to receive forward and backward messages from other processes. This is shown in lines 1-3 in algorithm 2. Meanwhile, the application thread skips to line 4 and initiates a forward check for all newly updated meta-edges. The forward check is done in lines 1-14 of algorithm 1.

Algorithm 2 DistSYNC

Require: list of meta-edges ME and meta-nodes MN
1: **for** All meta-nodes $\in MN$ **do**
2: Listen to forward or backward message
3: **end for**
4: **for** each edge $x, y \in ME$ **do**
5: **if** Meta-edge x, y is new **then**
6: $y \longrightarrow$CheckForward($x, size$) ▷ x asks y to execute CheckForward
7: **end if**
8: **end for**
9: Barrier()

When a destination meta-vertex receives a forward check message, it propagates that color and updated size to all its immediate neighboring meta-vertices. In lines 1 and 2 of Algorithm 1, a meta-vertex checks if it is the received pivot and if the size is bigger than itself, which determines that a cycle is detected. If so, it triggers a backward confirmation in line 5. If its size is greater than the received size, it means that it has since been updated, so it doesn't send a backward confirmation. If the meta-vertex is not a pivot, it just routes the forward message to its neighbors with the updated size shown on line 11.

IV. Implementation

This section will go over the details of implementing DistSYNC algorithm with the design choices and selection of data structures for the most efficient approach. The distributed algorithm was implemented in C++ using YGM, an asynchronous communication framework with a built-in suite of distributed data structures. YGM uses MVAPICH under the hood for scaling the application across distributed processes.

A. Partitioning

This step may not be a part of the two phases of DistSYNC, but it is essential that the graph is partitioned efficiently by avoiding load imbalance and minimizing inter-process communication. For partitioning the input graph, we use ParMetis [27], a multi-way partitioning library. ParMetis internally uses Kernighan-Lin algorithm [28] which allocates partitions to vertices such that the number of inter-partition edges is minimized, i.e., the inter-process communication load is reduced.

B. YGM

The presence of non-uniform communication patterns in large-scale graph algorithms makes regular MPI an inefficient framework for this algorithm. Also, the absence of distributed data structures like hash maps and sets makes it hard to code complicated distributed algorithms. For these reasons, we chose to use YGM as our base framework. YGM uses RPC-style fire-and-forget semantics for its communication interface. Messages in YGM have three basic components: a function to execute, arguments to pass to the function, and an MPI rank at which to evaluate the function. The procedures for forward and backward checks are encapsulated into asynchronous messages and fired to a destination rank that holds the receiving meta-vertex. The destination rank receives that message and executes that procedure with its arguments. In this particular case, the CheckForward() and ConfirmBackward() procedures from algorithm 1 are sent as asynchronous RPC messages.

The fact is that DistSYNC, like any graph algorithm, generates large numbers of small messages. Hence, YGM provides message buffering capabilities that bundle together multiple small messages between a sender and receiver to reduce the total number of remote MPI messages underneath and thus improve bandwidth. Lastly, to communicate non-fixed

width data structures, YGM serializes the structured messages to variable length byte arrays. This gives us the flexibility of communicating complicated structures without having to take a performance penalty.

C. Initial SCC computation

Initially, each process needs to compute the local SCCs of its allocated subgraphs. For computing the initial SCCs, we use Multistep [19], a shared memory implementation that uses a combination FW-BW-Trim and their novel BFS coloring algorithm. It is implemented in C++ with OpenMP directives, and we created a wrapper module to interface Multistep within our framework. It takes in an edge list representation of the allocated subgraph and produces a vector with indices representing the vertex IDs and SCC IDs as values.

D. Data stuctures

To perform DistSYNC algorithm for dynamic graphs, we need to accurately track forward and backward meta-edges from every meta-vertex. But these vertices would reside across different ranks, so traditional data structures would not be sufficient. For this purpose, we make use of the suite of distributed data structures provided by YGM. In specific, we use distributed hash maps and hash sets. YGM internally stores these tables across many ranks and provides constant time lookup for every entry into the map. If a process tries to look up a particular key that isn't stored in that process, it queries the other process that holds that key and returns its value. This seamlessly happens underneath while the interface provides a global view of that hash table and hence enables every process to look up every key-value pair of the distributed map.

The color associated with each vertex is stored in a YGM hash map accessible by every process. The list of neighbors for every meta-vertex is also stored in YGM hash map. This lets each process forward and backward propagate colors to its neighbors by looking them up in constant time with at most two hops. Lastly, every process builds a set of all forward propagated colors it received. Hence, when it receives a backward propagated color, it checks this set for an equivalent match and updates its color if it finds one. This is a way of checking forward and backward paths before updating colors.

V. Experimental evaluation

In this section, we will discuss the results from running distributed experiments starting with the strong scaling results in comparison with the iSpan [10] as the baseline. Then we compare the speedups of DistSYNC with varying batch sizes of dynamic updates. Then lastly, we discuss the performance metrics of DistSYNC using YGM, including average memory utilized and interprocess communication bandwidth.

A. Experimental setup

We ran the distributed experiments on Intel Xeon dual E5-2690v4 processors with 28 cores per node. The timing for these experiments is recorded after the creation of initial meta-graphs to recompute the SCC for a dynamic batch of edge additions. For our benchmark datasets, we used four real-world graphs; Flicker (Fl), Facebook (Fb), Orkut (Or), and Roadnet-USA (Rusa), along with two synthetic graphs; RMAT26 (R26) and RMAT27 (R27). The RMAT was generated with the probabilities (a=0.45, b=0.15, c=0.15, d=0.25) and scale-free degree distribution. Table I gives more details about the datasets.

The baseline iSpan is not a dynamic model that can handle batches of updates; hence, in the iSpan experiments, we recompute the SCC over the entire graph, i.e., the initial graph plus the update batch. This is currently the only available way to compute distributed SCC on incremental graphs. The graph properties give us an idea of how big the meta-graphs will be. For example, larger SCCs would mean more vertices can be grouped under a single meta-vertex and, in turn, reduce the total number of meta-vertices. This reduction in the total number of SCCs enables us to create a meta-graph that is much smaller than the original graph and hence significantly reduces the cost of recomputing SCC.

B. Performance

The first and primary metric for analyzing performance is strong parallel scaling. Figure 5 shows the strong scaling results of DistSYNC and iSpan for a single-threaded implementation of Tarjan's algorithm. DistSYNC scales well, with max speedups ranging from 8x to 28x for all graphs on up to 64 processes. In comparison, iSpan scales with max speedups ranging from 6x to 9x. DistSYNC is also able to outperform iSpan in all but one dataset, namely, Roadnet-USA(Rusa), with its speedups flatlining at 8x. This is because the size of the largest SCC is significantly smaller, and the number of SCCs is large. This dataset essentially has a lot of small SCCs. This means the meta-graph also has a lot of meta-vertices, thereby reducing the impact of constructing a meta-graph.

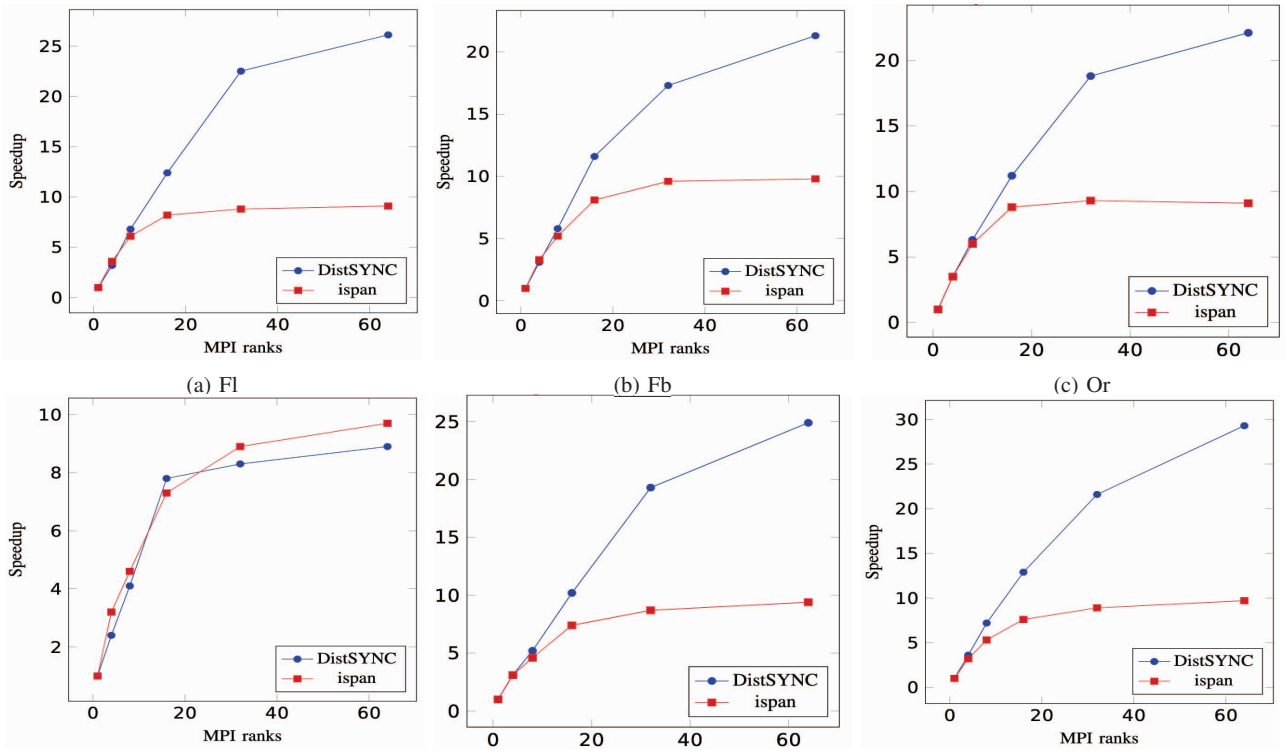
DistSYNC performs at its worst when the original graph consists of many small SCCs, but those types of graphs are much rarer because of the small-world property of graphs, as explained in [7], where vertices of real-world graphs usually cluster together to form few large SCCs followed by several medium to small SCCs.

Apart from the number of SCCs, the other factor influencing the performance of DistSYNC is the size of the update batches. Figure 6 shows us the speedups of DistSYNC and iSpan at 32 nodes when varying the size of the update batch. The X-axis denotes the percentage of the initial batch size. For instance, 20% of the batch of edges are kept for updates while the remaining 80% are added to the initial graph to keep the overall size constant. We perform these experiments on 20, 40, 60, and 80% batch sizes.

We observe that DistSYNC is at its fastest for smaller update batch sizes, and the speedup gradually decreases as we increase the batch size. This is because at smaller batch sizes, the number of new edges to be added is less, and hence fewer forward and backward messages are triggered by each process. We note that the overall size of both batches combined remains

TABLE I: Details of benchmark datasets

Dataset	Initial edges	# of edges in update	# of SCCs	Largest SCC size	Best iSpan time(MS)	Best DistSYNC time(MS)
Flicker(Fl)	1,151,463	1,158,925	487,659	9,752	21.3	6.24
Facebook(Fb)	67,255,691	67,255,782	1,576,432	963,487	230.4	98.2
Orkut(Or)	122,346,784	122,346,157	2,975,565	1,865,468	349.11	146.52
Roadnet-USA(Rusa)	93,568,872	93,568,112	3,501,682	443,923	248.6	279.4
RMAT26(R26)	67,108,864	67,107,927	1,136,282	1,082,223	215.6	75.4
RMAT27(R27)	734,217,728	734,279,598	9,987,245	92,742,613	2472.7	733.4



(a) Fl (b) Fb (c) Or

(d) Rusa (e) R26 (f) R27

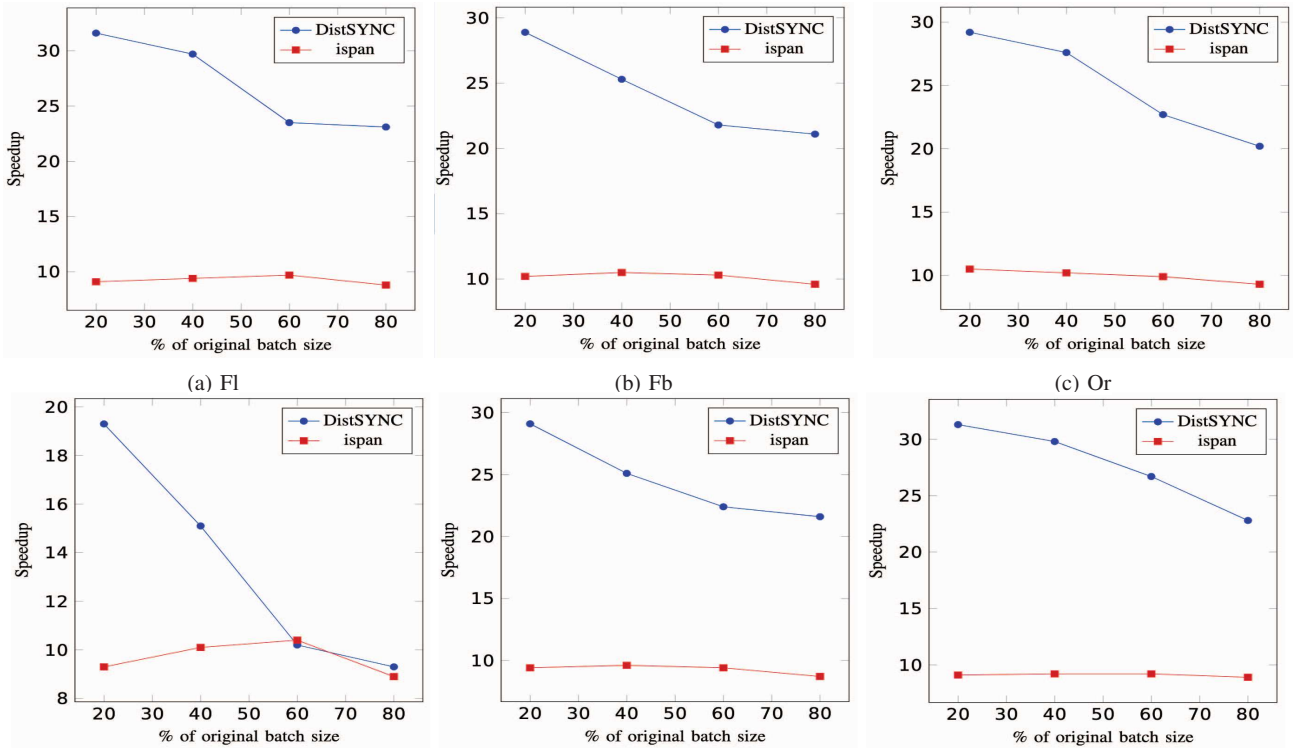
Fig. 5: Speedups for DistSYNC(blue) and baseline iSpan(red) with respect to single-threaded Tarjan's implementation.

constant. Only the allocation of edges between the initial batch and the updated batch varies.

In contrast, the performance of iSpan remains fairly constant across all batch sizes as the size of updates will not influence it, considering it is not a truly dynamic model and recomputes the SCC for the entire graph every time there is a new batch. We perform this set of experiments to elucidate the benefits of using a dynamic algorithm for graphs with incremental updates. These benefits are amplified further when there is more than one iteration of updates because, for every iteration, only the new batch of meta-edges will be considered for pivots while the previous iterations will be baked into the meta-graph. This is unlike any of the currently available state-of-the-art parallel SCC algorithms where the entire graph, along with incoming updates, needs to be recomputed for every update.

C. Memory Utilization

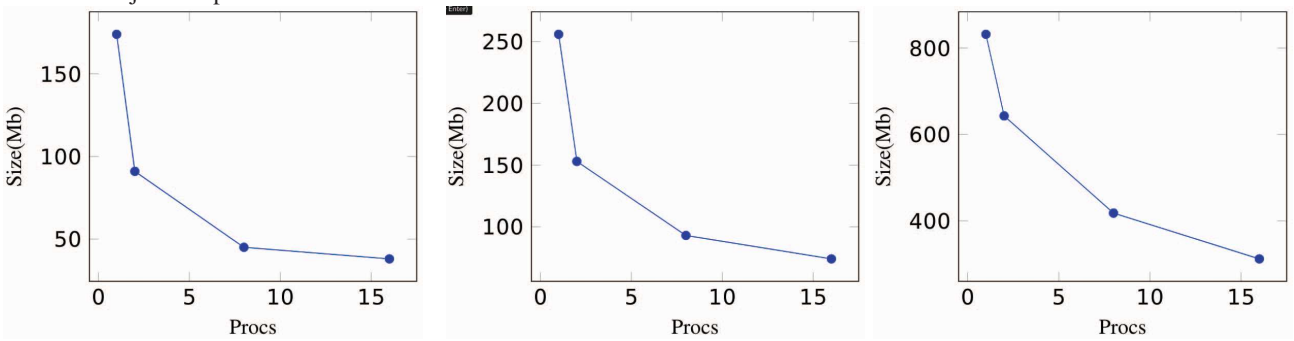
Distributed implementations always have the added advantage over shared memory implementations of reducing memory utilized per process to perform massive-scale computations. In this section, we highlight memory utilization to demonstrate the benefits of using YGM as the distributed substrate for DistSYNC. Figure 7 gives us the average memory utilized per process as we increase the number of processes. We can see that for all the datasets, the average memory utilized decreases fairly consistently as we scale up. A significant contributor to this decrease is the use of distributed data structures provided by YGM. Considering DistSYNC uses hash tables to keep track of forward and backward connectivity for each vertex while using hash sets to keep track of all vertices in a meta-vertex, the number of hash entries can be significant if we use a traditional hash table. The YGM hash table stores only entries of vertices that belong to that process, while looking up external entries by exchanging messages between the processes. This is seamlessly handled under the hood by YGM, while all the entries appear as one unified hash table for the user. To the best of our knowledge, the state-of-the-art iSpan replicates the entire graph on all the ranks, so memory utilization doesn't scale with increasing ranks.



(a) Fl (b) Fb (c) Or

(d) Rusa (e) R26 (f) R27

Fig. 6: Speedup on 32 cores with varying batch sizes for DistSYNC(blue) and baseline iSpan(red) with respect to single-threaded Tarjan's implementation.



(a) Fl (b) Or (c) R27

Fig. 7: Average memory utilized per process as the number of processes increases.

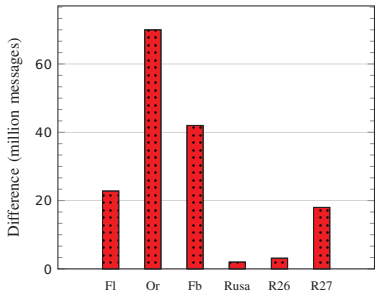


Fig. 8: Reduction in the number of MPI messages by coalescing in YGM.

D. Message coalescing

YGM also enables us to coalesce multiple messages between two processes and send them as a single message. These messages are stored in a fixed-size 512 Kb buffer in each process. When the buffer is filled up or when it is forced to flush, it is sent to the destination process, which handles all the incoming messages. These messages are usually forward or backward check messages that contain individual lookups of vertices in hash tables. Since these are extremely small messages, sending them individually in MPI would significantly increase the total number of remote communi-

cations and result in poor bandwidth utilization. The order in which these messages are handled wouldn't matter, as this is a completely asynchronous algorithm. At worst, a process would start checking the backward path while still waiting for confirmations on forward paths from other processes, in which case, it would simply have to trigger a new forward check once it is updated.

Figure 8 shows the reduction in the number of MPI messages by coalescing small messages in YGM. In the Orkut graph, the difference is amplified as it consists of a relatively small number of densely packed meta-vertices that frequently communicate with a small subset of processes. Coalescing these messages ensures the buffers are filled before sending it as an MPI message. Likewise, Roadnet-USA has the least difference due to its relatively large number of meta-vertices communicating with a larger subset of processes and hence sending sparse buffers before filling them fully. As a result, the total number of messages is increased. This is another reason why DistSYNC is less performant for many small SCCs.

VI. Conclusion

This paper introduces DistSYNC, an asynchronous algorithm leveraging distributed, multithreaded CPU parallelism for identifying Strongly Connected Components(SCC) in incremental networks with edge additions. We have also supported the algorithm with the implementation details of the distributed framework. We show that our approach can offer performance speedups of up to 30x over single-threaded Tarjan's implementation and up to 2.8x over the state-of-the-art. In the future, we plan to extend DistSYNC with edge deletions. We are also working on caching meta-graphs in persistent memory like NVMe ssd. This would enable us to read and write meta-graphs much more quickly in between each timestep for efficient recomputation.

Acknowledgments: This work is supported by NSF OAC-SPX grants 1725755, 1725566, and 1725585 for the collaborative SANDY project, and OAC-CSSI grants 2104076, 2104078, 2104115 for the collaborative CANDY project.

References

[1] L. Zhang and J. Gao, "Incremental graph pattern matching algorithm for big graph data," *Scientific Programming*, vol. 2018, 2018.

[2] S. Allesina, A. Bodini, and C. Bondavalli, "Ecological subsystems via graph theory: the role of strongly connected components," *Oikos*, vol. 110, no. 1, pp. 164–176, 2005.

[3] G. M. Slota, S. Rajamanickam, and K. Madduri, "High-performance graph analytics on manycore processors," in *2015 IEEE International Parallel and Distributed Processing Symposium*, pp. 17–27, IEEE, 2015.

[4] J. H. Reif, "Depth-first search is inherently sequential," *Information Processing Letters*, vol. 20, no. 5, pp. 229–234, 1985.

[5] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of streaming processing of dynamic graphs: Concepts, models, and systems," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[6] L. K. Fleischer, B. Hendrickson, and A. Pınar, "On identifying strongly connected components in parallel," in *International Parallel and Distributed Processing Symposium*, pp. 505–511, Springer, 2000.

[7] S. Hong, N. C. Rodia, and K. Olukotun, "On fast parallel detection of strongly connected components (scc) in small-world graphs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, 2013.

[8] R. Mazumder and T. Hastie, "Exact covariance thresholding into connected components for large-scale graphical lasso," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 781–794, 2012.

[9] S. D. Pollard, S. Srinivasan, and B. Norris, "A performance and recommendation system for parallel graph processing implementations: Work-in-progress," in *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, (New York, NY, USA), p. 25–28, Association for Computing Machinery, 2019.

[10] Y. Ji, H. Liu, and H. H. Huang, "ISpan: Parallel identification of strongly connected components with spanning trees," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, IEEE Press, 2018.

[11] S. Srinivasan, S. D. Pollard, B. Norris, S. K. Das, and S. Bhowmick, "A shared-memory algorithm for updating tree-based properties of large dynamic networks," *IEEE Transactions on Big Data*, vol. 8, no. 2, pp. 302–317, 2022.

[12] A. Khanda, S. Srinivasan, S. Bhowmick, B. Norris, and S. K. Das, "A parallel algorithm template for updating single-source shortest paths in large-scale dynamic networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 929–940, 2022.

[13] S. Srinivasan, S. Riazi, B. Norris, S. K. Das, and S. Bhowmick, "A shared-memory parallel algorithm for updating single-source shortest paths in large dynamic networks," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, pp. 245–254, 2018.

[14] A. Khanda, S. Bhowmick, X. Liang, and S. K. Das, "Parallel vertex color update on large dynamic networks," in *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 115–124, 2022.

[15] A. Bernstein, A. Dudeja, and S. Pettie, "Incremental scc maintenance in sparse graphs," in *29th Annual European Symposium on Algorithms (ESA 2021)*, 2021.

[16] W. Fan and C. Tian, "Incremental graph computations: Doable and undoable," *ACM Transactions on Database Systems (TODS)*, vol. 47, no. 2, pp. 1–44, 2022.

[17] T. Steil, T. Reza, K. Iwabuchi, B. W. Priest, G. Sanders, and R. Pearce, "Tripoll: computing surveys of triangles in massive-scale temporal graphs with metadata," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2021.

[18] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[19] G. M. Slota, S. Rajamanickam, and K. Madduri, "Bfs and coloring-based parallel algorithms for strongly connected components and related problems," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 550–559, IEEE, 2014.

[20] G. Li, Z. Zhu, Z. Cong, and F. Yang, "Efficient decomposition of strongly connected components on gpus," *Journal of Systems Architecture*, vol. 60, no. 1, pp. 1–10, 2014.

[21] M. Stuhl, "Computing strongly connected components with cuda," *Master's thesis, Masaryk University*, 2013.

[22] A. P. Iyer, L. E. Li, T. Das, and I. Stoica, "Time-evolving graph processing at scale," in *Proceedings of the fourth international workshop on graph data management experiences and systems*, pp. 1–6, 2016.

[23] 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*, EuroSys '12, (New York, NY, USA), p. 85–98, Association for Computing Machinery, 2012.

[24] 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*, pp. 1–5, 2012.

[25] R. McColl, O. Green, and D. A. Bader, "A new parallel algorithm for connected components in dynamic graphs," in *20th Annual International Conference on High Performance Computing*, pp. 246–255, IEEE, 2013.

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

[27] G. Karypis and V. Kumar, "Parallel multilevel series k-way partitioning scheme for irregular graphs," *Siam Review*, vol. 41, no. 2, pp. 278–300, 1999.

[28] B. Hendrickson and R. W. Leland, "A multi-level algorithm for partitioning graphs.," *SC*, vol. 95, no. 28, pp. 1–14, 1995.