Identifying Strongly Connected Components on Distributed Networks

DRP Report

Sudharshan Srinivasan
University of Oregon, Eugene, OR, USA
ssriniv2@cs.uoregon.edu

***Abstract*—Strongly connected components (SCC) are an essential property for understanding the structure of directed networks. Given that many real-world networks are significant, it is often computationally efficient to partition the network over many distributed systems and solve for SCC simultaneously over the partitioned network. In this paper, we present an algorithm for identifying SCC on distributed systems. Our algorithm comprises of three phases. In the first phase, we locally perform SCC over all partitions while in the second phase, we update the partial results across the partitions in a reduced graph form. Lastly, we reapply SCC over the reduced graph at all partitions. The computations within individual partitions are highly scalable using both shared-memory CPU and GPU threads. Due to its distributed nature, our algorithm allows for analysis of much larger networks than is previously possible. Comparisons with state-of-the-art baseline approaches are made to analyze the algorithm. Our results show performance speedups up to 3.1x and memory reductions up to 2.6x with respect to serial implementations.**

***Index Terms*—strongly connected components, distributed memory, GPU threading**

I. Introduction

Detecting Strongly Connected Components SCCs in a large directed graph is a fundamental graph analytics problem. A strongly Connected component can be defined as a subset of vertices in a directed graph where there is a path from any vertex to every other vertex in that subset. A single graph can have many SCCs, but vertices are mutually exclusive to these SCCs. Detecting SCCs has many advantages such as pattern matching [1], topological sort [2] and graph analytics [3]. The general idea for detecting SCCs is to perform a DFS (Depth-First- Search) on a directed graph. The traditional approach of using DFS works well for sequential networks. However, performing a DFS is expensive, and it is tough to compute DFS in parallel [4].

In order to overcome the challenges of parallelizing DFS on static networks, FW-BW (Forward-backward) approach was proposed [5]. To further improve the performance, trim techniques which fast reduce the large number of trivial SCCs (e.g., with one or two vertices, called trim-1 and trim-2, respectively) are introduced by [6]. To the best of our knowledge, there have only been one attempts to detect SCC on distributed networks [7]. For the most part, the state-of-the-art parallel approaches for detecting SCCs are optimized for shared-memory systems.

In this paper, we introduce a new hybrid algorithm called DistSYNC to detect SCCs on distributed networks. The approach is considered hybrid as it is enabled with distributed, multithreaded CPU and GPU parallelism. We also introduce a framework that efficiently implements our approach in a hybrid systems. The rest of the paper is organized into five sections. In section II, we introduce all 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 section finishes off with the conclusion and future works.

II. Background and related work

Detecting SCC's in a network is a well-studied problem. This section of the paper discusses all the related work on detecting SCC's in the large-scale network. We utilize some of the concepts in our approach, so this section also provides prerequisite background information. We have categorized the SCC implementations into the following categories: **Sequential**, **Shared Memory**, **Distributed**, and **GPU** implementations.

A. Sequential SCC

Tarjan's [8] implementation is the well-known sequential algorithm for detecting SCC. Both approaches use DFS (Depth First Search), and the complexity is 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 of them demonstrate poor scalability.

The Forward-Backward (FW-BW) algorithm uses a recursive approach, and the algorithm can be described as follows: Let V be the set of the vertices in the graph $G(V, E)$, O(V) be 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 that can reach from u and that set is called D. 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 are inserted to P. The intersection of two sets 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 O(n log n) to detect SCC. This approach was further improvised to remove those vertices which have zero in-degree and out-degree; this step is called trimming. By trimming these vertices, we can reduce the number of vertices in FW-BW sets and speed up the overall performance. The coloring approach is also similar to FW-BW with some modifications. Instead of just using one pivot, it uses multiple pivots.

B. Shared Memory SCC

Ji et al. [7] proposed a novel synchronization paradigm called Rsync to spanning tree-based detecting of SCC. The R-Sync approaches provide 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. [6] 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 [8] 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 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. [9] 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. In order to minimize synchronization, they avoid using locks. Their experiments on real-world graphs show better scalability on low-diameter networks. We use this approach to perform multi-threaded SCC within distributed processes.

C. GPU Implementation

Li et al. [10] 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 [11] extended the work by introducing an extended graph traversal implementations. Sthul ran experiments on the synthetic network 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. The emphLi et al. implementation [10] has a hybrid method that detects SCC in phases, wherein phase-1, the algorithm is only focussed on detecting a single large SCC, and in the second phase, the remaining small-sized subgraphs are processed. It is shown that identifying the small-sized SCC's takes more time than identifying the single large SCC.

III. Methodology

In this section, we will first explore the terms and notations that will be used throughout the rest of the paper. Then we will explain the DistSYNC algorithm with the help of an example workflow for a small graph. We will lastly look into the algorithmic complexities involved.

A. Terms and notations

Terms used across this paper are :

1) Original Graphs and subgraphs: Original graphs $G(V, E)$, where V and E represent the vertices and edges, are the full-sized input directed graphs before partitioning them and allocating them to different processes. After partitioning, each process holds a subset of the original graph G, which we refer to as subgraphs $G'(V', E')$. If an edge goes across partitions, they are allocated to both partitions by creating a *ghost vertex* for the vertex that doesn't belong to that partition. Usage of term *vertex A* is used to reference a vertex with label *A*. Vertex labels are numeric. Likewise, the term $\{X, Y\}$ is used to represent an edge between the vertices X and Y.

2) Local, external and global SCCs: SCCs are strongly connected components. We sometimes refer to them as just components. The term *SCC A* represents an SCC with the ID *A*. On the other hand, the term *SCC(x)* refers to the SCC ID to which vertex x belongs. *Computing SCCs* or *performing SCCs* refer to the act of finding the components and are not to be confused with *SCC(x)* or *SCC A*. We use the prefix local, external and global to mention the scope of the SCC ID across processes. Local SCC IDs are labels given to components within a process after computing SCC on the subgraph G'. Local IDs are unique within a process but not across processes. All local SCC IDs have a counterpart *external ID* which are labeled unique across processes. The

processes can reference the local IDs of other processes using its equivalent external ID and not have a clash in label with its own external ID. Lastly, global SCC IDs are labels given to components generated on the original graph G, which is the final output of DistSYNC. To clarify, external IDs aren't the same as global IDs and not the final answer. Across processes, a single vertex can have many external IDs, each unique to that specific process, but a vertex can only have one global ID regardless of which process looks at it.

3) Partial and complete meta-graphs: Meta-graphs $G''(MN, ME)$ are abstractions on top of existing graphs where the vertices, which is referred to as *meta-nodes(MN)*, are external SCC IDs. The edges of the meta-graph, which is referred to as *meta-edges(ME)*, are directed edges that connect two meta-nodes. The meta-graphs are much smaller than the original graph as many vertices belonging to an SCC can be represented using just a single label(external SCC ID/meta-node). Partial meta-graphs are locally created at each process using only the view of its own subgraph G'. Complete meta-graphs are globally created and shared by all processes. They are created by overlapping all partial meta-graphs. This overlapping happens because a single ghost vertex can exist in multiple processes and can belong to SCCs with different external IDs.

B. DistSYNC

This subsection will explain the workings of the DistSYNC algorithm and how it achieves reduced computation and space utilization. DistSync is a three-phase algorithm that identifies local SCC components and creates partial meta-graphs at each process on phase one, followed by a communication phase where the partial meta-graphs are exchanged across all processes. Lastly, we enter a recomputation phase that builds a complete meta-graph and recomputes SCC on the meta-graph to identify all the global components.

The motivation behind this algorithm stems from two lemmas on the structural property of graphs and strongly connected components.

Theorem

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.*

In Figure 1, there are two SCCs with labels A and B. Each SCC has three vertices, each labeled 1 through 6. We can see that there is a forward edge between vertex 1 and 2 in SCC A to vertex 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 that goes across two SCCs is redundant information when identifying the components. As we can see on the RHS of Figure 1, all the inter-SCC connections are replaced with one forward path between the two SCCs. By doing so, we are able to 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 in size 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.

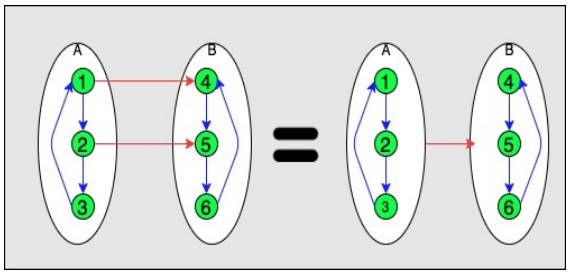


Figure 1: Example for lemma 1. The RHS has the same structural property as LHS with reduced connectivity

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 to a single component.*

On the LHS of Figure 2, 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 2.

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. Similar to Lemma 1, this can be extrapolated to any number of SCCs of any size as long as there is a forward and backward path between the both of them.

We apply these two lemmas before the communication phase of the DistSYNC algorithm to create a reduced

size meta-graph that retains the same structural information required to compute the global SCC of a graph.

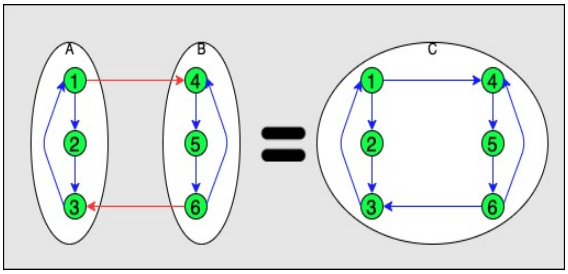


Figure 2: Example for lemma 2. The RHS has the same structural property as LHS but with reduced number of SCCs

Figure 3 outlines the overall workflow of the three-phase DistSYNC algorithm. To the left, we have our input original graph, for which we need to find all the strongly connected components. First, the input graph is partitioned N-ways based, where N is the number of processes. When we say partitioned, we allocate every vertex in that graph with a partition ID. This partition ID is directly linked with the process ID. Each process reads and stores only edges from the graph that has vertices with the same partition ID. If both the vertices of an edge have the same partition ID, it represents an intra-process edge, and only the process with the same process ID reads that edge.

On the other hand, edges that have vertices allocated to two different partition IDs represent an inter-process edge. The processes with the same ID as the source vertex reads that edge. The destination vertex in that edge that doesn't belong to the respective process is marked as a ghost vertex. By doing so, we can allocate smaller subgraphs to different processes while all processes put together still maintain the overall connectivity of the original graph.

In Figure 3, we can see that each process P1, P2, and P3 reads only the edges which are allocated to the partition. For instance, process P1 reads the intra-process edges {1,2} , {2,3} and {3,1} along with inter-process edge {2,8} as vertices 1, 2 and 3 belong to partition 1. Similarly, P2 reads edges {4,5} , {5,6} , {6,4} and {4,1} while P3 reads {7,8} , {8,9} , {9,1} and {7,5}. Vertices 8, 1, 5 are marked as ghost vertices(yellow) in P1, P2, and P3, respectively.

With each process owning a unique subgraph, each of them proceeds to perform the three phases in parallel. The three phases are executed in sequence from left to right within a process, as shown in Figure 3.

1) Phase 1 : Initial SCC and partial meta-graph: During phase 1, each process takes in the allocated subgraph as input and performs SCC on the subgraph to allocate a local SCC ID to each vertex in the subgraph. It is to be noted that this local ID is not the global SCC ID with respect to the entire original graph but rather just the partial SCCs with respect to the allocated subgraph. This is represented in Figure 3 where during phase 1 at P1, vertices 1, 2, 3 are allocated to external SCC A while vertex 8 is allocated to external SCC B. To recap from section II, external SCC IDs are another set IDs given to every local SCC ID that are unique across processes unlike local SCC ID. The same happens at P2 and P3 with their own allocated subgraph to produce components C, D, E and F during phase 1.

Algorithm 1 Phase 1

Input: Subgraph G'(V',E')
Output: list of meta-edges ME[] and meta-nodes MN[]

```
1: Perform initial SCC on G'
2: for each edge e{x, y} ∈ E' do
3:     if e{SCC(x), SCC(y)} ∉ ME then
4:         ME[ ] ← e{SCC(x), SCC(y)}
5:         MN[ ] ← SCC(x)
6:         MN[ ] ← SCC(y)
7:     end if
8: end for
```

After mapping vertices with their respective local SCC IDs, we move on to creating a partial meta-graph. As explained in section III-A, a meta-graph is an abstraction on top of an original graph where the vertices(meta-nodes) are external SCC IDs and edges(meta-edges) are connections between any two SCCs. These meta-edges are created by applying lemma 1 to reduce multiple edges that connect the vertices from two different SCCs to one representational meta-edge that connects both the SCCs. This can be visualized in P1 of Figure 3 where edge {2,8} is transformed to meta-edge {A,B} after phase 1. The vertices are contained within the two SCCs represented as a green circles in Figure 3. The term **partial** comes from the fact that each process creates its own meta-graph only with the view of its allocated subgraph rather than the original graph.

Algorithm 1 defines the pseudo-code for phase 1. At line 1, we perform the initial SCC on the subgraph. Line 2 to 8 explains how we create the partial meta-graph. The partial meta-graph is represented as a list of meta-edges(ME[ ]) and meta-nodes(MN[ ]).

2) Phase 2 : Communication and complete meta-graph: After each process creates its partial meta-graph, they enter phase 2. Internally, phase 2 is a collection of three sub-phases, namely, **initial communication**, **complete meta-edge creation**, and **final communication**. During the initial communication sub-phase, the partial meta-graph in the form of meta-nodes and meta-edges from every process is exchanged with every other

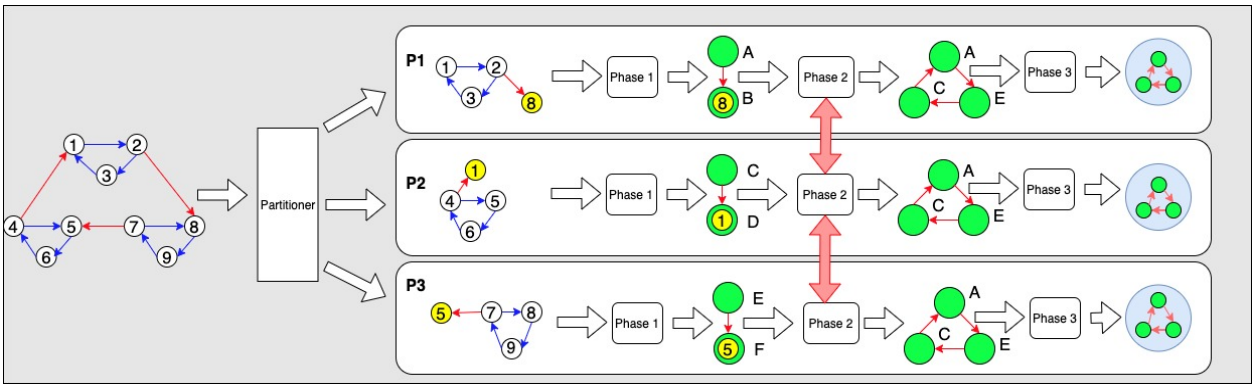


Figure 3: Example workflow of DistSYNC. The original graph on the left is partitioned across processes P1, P2 and P3. The yellow vertices represent ghost vertices. The green circles represents SCCs containing vertices. Thin red arrows indicate inter-process edges. The solid white arrows represent the flow of execution. The subgraph at each process is transformed to a partial meta-graph in phase 1 and goes through the communication phase 2 where it is transformed to a complete meta-graph. This in-turn goes through phase 3 to give the global SCC as shown in the last blue circles.

process using an *AllToAll* collective operation. It is to be noted that only the meta-graph is exchanged and not the original graph, so the data communicated is much smaller.

Algorithm 2 Phase 2

Input: list of partial meta-edges ME and meta-nodes MN

Output: list of complete meta-edges ME' and meta-nodes MN'

1: Gather ME and MN at all procs
2: **for** each node $x \in MN$ **do**
3: **for** each node $y \in MN$ **do**
4: **if** $e\{x, y\} \in ME$ **then**
5: $ME'[\,] \leftarrow e\{x, y\}$
6: $MN'[\,] \leftarrow x$
7: $MN'[\,] \leftarrow y$
8: **end if**
9: **end for**
10: **end for**
11: Gather ME' at all procs

Once they finish initial communication, every process overlaps the collected partial meta-graph from the other processes to create a complete meta-graph. For instance, in Figure 3, process P1 already has its partial meta-graph, which is the meta-edge {A, B}. After initial communication, P3 establishes that vertex 8 in its external SCC E is also a ghost vertex that belongs to external SCC B at P1. As a result, SCC B is renamed as SCC E, and the meta-edge {A, E} is created at P3. Likewise, P2 creates meta-edge {C, A} and P1 creates meta-edge {E,C}. Now, these new meta-edges are once again exchanged with every other process. This is the final communication in the second phase.

Since we aren't creating any new meta-nodes but rather just new meta-edges, every process already holds the list of all possible meta-nodes after initial communication. So for the final communication, we perform an *AllReduce* operation that fills these missing meta-edges rather than an *AllToAll* collective operation as implemented during the initial communication. After phase 2, every process will maintain the complete meta-graph, which is the same for all processes. As we can see in Figure 3, all processes have the same meta-graph with three meta edges {C,A}, {A,E} and {E,C} after phase 2. Phase 2 uses bulk synchronous communication, so every process needs to reach this phase before executing it.

Algorithm 2 explains the working of phase 2 where at step 1, we gather the meta-nodes and meta-edges at all processes. Steps 2-8 is a nested for loop that checks every 1to1 combination of meta-nodes to see if that process has that meta-edge. At step 8, we once again gather the new meta-edges

3) Phase 3: Final SCC: This is the last phase of the DistSYNC algorithm. With each process holding the complete meta-graph after phase 2, we once again compute SCC using the complete meta-graph as the input. This produces output mapping of meta-nodes to their allocated Global component. This is visualized in Figure 3 where phase 3 at all processes take in the three-node meta-graph and produce just a single component which is the final result. We need to remember that this is not a vertex to SCC ID mapping but rather a meta-node to SCC ID mapping. To get the SCC ID for a specific vertex, we first need to lookup the vertex to meta-node mapping that is maintained after phase 1

and then lookup the appropriate global SCC ID for that meta-node. The execution of phase 3 is the same for all processes as all of them have the same complete meta-graph as input and produce output. By performing this phase redundantly at all processes, we make sure that all of them maintain the final view of their vertex to global SCC mapping. Since the size of the meta-graph is much smaller than the original graph, this is not a relatively intensive computation and can be redundantly done at all processes without too much overhead rather than them separately and performing one more round of bulk synchronous communication.

Algorithm 3 Phase 3

Input: Meta-graph G"(MN',ME')
Output: Global SCC result
1: Perform final SCC on G"

Algorithm 3 explains the working of phase 3 with only a single step that performs SCC on the complete meta-graph that we obtain from phase 2.

IV. Implementation

This section will go step by step through three phases of the DistSYNC algorithm and explain how we implemented it along with its design choices and choice of data structures for the most efficient approach. We also include subsections for analyzing the computation, communication, and space complexities based on the design choices. The scaling of the application across distributed processes is done using MPI 3.2. Each process is enabled with multi-threaded execution using shared memory OpenMP directives.

A. Partitioning

This step may not be a part of the three 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 [12] , a multi-way partitioning library. ParMetis internally uses the Kernighan-Lin algorithm [13] to allocate partitions to vertices based on minimizing the number of inter-process edges, thereby reducing the communication load.

B. Initial SCC computation

Initially, each process needs to compute the local SCCs of its allocated subgraphs. For computing the initial SCCs, we use Multistep [9], 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. It is to be noted that Multistep only accepts continuous vertex IDs, but that isn't possible in our implementation as the original graph is partitioned based on communication overhead. To overcome this, we map the actual vertex IDs to continuous counterparts and re-map them when we query for results.

C. Partial meta-graph creation

As explained in subsection III-A, meta-graphs are graph abstractions with global SCC IDs as the meta-nodes and the representative edges between pairs of components, if any, as the meta-edges. A partial meta-graph is one where the SCC IDs are allocated to vertices with only the view of the allocated subgraph and not the full-sized original graph. The meta-nodes are stored using a vector representation with their size equal to the number of local components.

For implementing lemma 1 to create a representative edge from a list of all existing edges between any two meta-nodes, we use unordered sets. As we scan through the list of edges, we insert it into unordered sets in {SCC(v1),SCC(v2)} format where v1 and v2 are the source and destination vertices of an edge. Inserting to unordered sets ensures uniqueness by hashing elements so the first edge scanned between a given pair of SCCs is the representative meta-edge between the pair.

D. Initial and final communications

The initial communication uses an *AllToAll* collective operation to exchange the partial meta-graphs. This is done in MPI using *MPI_AllGather* and *MPI_AllGatherV* functions. First, the vector of all the partial meta-nodes along with its respective number of outgoing meta-edges is gathered from each process. Using this information, we can allocate appropriate buffer sizes based on the number of all meta-edges. Then, we gather the list of meta-edges for every meta-node from each process. We send the number of meta-edges and meta-edge values in two separate operations rather than clubbing them together as it enables us to access meta-edges of each meta-node in parallel. A single thread can be in charge of reading all the meta-edges of a single meta-node based on offsets in receiving buffer derived from the sizes.

By reading the initial messages, each process creates a bit vector where the indices represent a 1to1 combination between all meta-nodes and its values being a single bit indicating the presence or absence of a meta-edge between the two meta-nodes. Only the processes that own the two meta-nodes will know if there is a meta-edge between them and will have a value of 1 for that position. Every other process will mark that position as 0. In order to communicate the presence of this meta-edge with all other processes, we perform a final round

of communication within phase 2 using *MPI_Allreduce* function with **AND** operation to flip all the 0 bits to 1 if any of the processes has a 1 for that position.

This bit-vector is created in parallel at each process using shared memory OpenMP directives. The indices of the bit-vector are the same across all processes as they are created in the order of meta-nodes received during initial communication.

E. Complete meta-graphs and recomputing SCC

These are the final two operations where we first create the complete meta-graph by scanning the bit-vector and creating a collection of meta-edges of the form {meta-node1, meta-node2} for the respective indices that have a 1-bit value. The process of creating complete meta-graphs are embarrassingly parallel due to the way we structured the MPI buffers. The output edge list from phase 2 acts as the input for our SCC recomputation. The recomputation is once again done using the Multistep library, producing a vector of meta-nodes as indices and global SCC IDs as the value. This is the final output of the entire application.

Phase	Time	Space	Comm
1	V+E	V	-
2	MN^2	$MN^2 + ME$	$MN^2 + ME$
3	MN+ME	MN	-

Table I: Big O notations of time, space and communication complexities of each phase at a single process. V and E denotes the number of vertices and edges of the allocated subgraph while MN and ME denotes the number of meta-nodes and meta-edges

F. Complexities

In this subsection, we analyze the time, space, and communication complexities of each phase of DistSYNC given in table I. Communication only happens in phase 2 where each process exchanges the meta-edges, which is O(ME), and a bit-vector with 1to1 combinations of meta-nodes which is $O(MN^2)$.

The time complexity of phases 1 and 3 are to the order of the number of vertices + edges and number of meta-nodes + meta-edges, respectively. The time complexity for phase 2 is $O(MN^2)$ as we are creating the 1to1 bit vector. Lastly, the space complexity of phases 1 and 3 is O(V) and O(MN), respectively, as computing SCC results in storage of vertex to SCC ID map while phase 2 is $O(MN^2 + ME)$ to store the bit-vector and meta-edges.

V. Experimental evaluation

In this section, we will discuss the type of experiments we ran and analyze the experimental results. We will look into speedups and memory utilization along with properties of the benchmark graphs and improvements made by enabling GPU threads. We ran the distributed experiments on Intel Xeon dual E5-2690v4 processors with 28 cores per node. The GPU computations were offloaded to dual NVIDIA Tesla K80 GPU nodes with 24GB memory.

A. Performance

We used three real-world graphs; Flicker(FL), LiveJournal(LJ), Wikipedia(WE), along with one synthetic RMAT graph as our benchmark. The RMAT was generated with the probability (a=0.45, b=0.15,c=0.15, d=0.25) and scale-free degree distribution. Figure II describes the properties of our benchmark graphs, including the number of SCCs and the size of the largest SCC. It also shows the total time taken for running those benchmarks on iSpan for baseline comparison and our hybrid DistSYNC implementations. The graph properties give us an idea of how big the meta-graphs are going to be. Larger SCCs would mean more vertices can be clubbed under a single meta-node and, in turn, reduce the total number of meta-nodes.

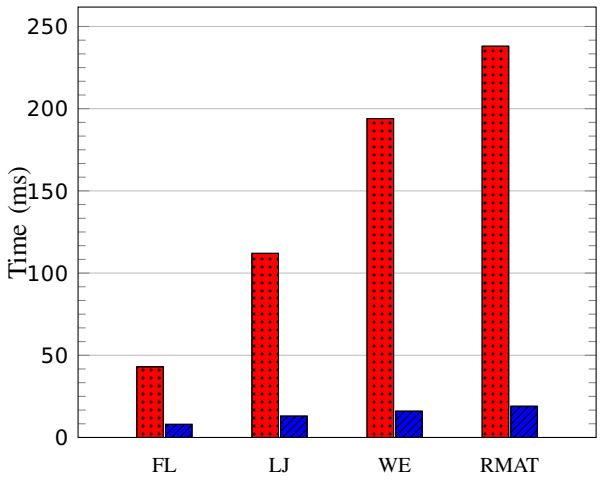
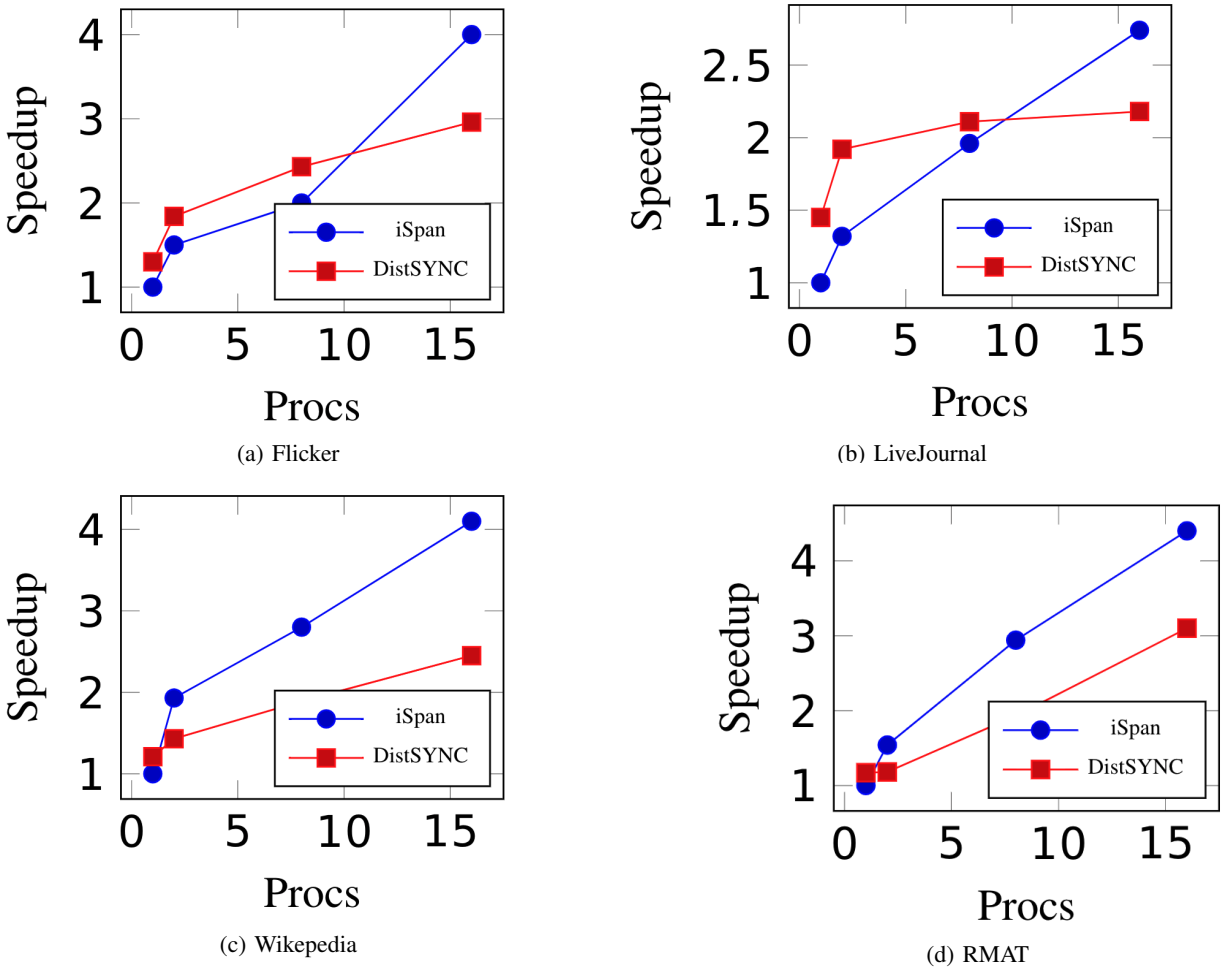


Figure 6: Runtime for creating complete meta-graphs with GPU(blue) and serial execution(red)

Figure 4 shows the comparison of speedups for hybrid DistSYNC and iSpan with respect to single-threaded Tarjan's implementation. The only available distributed memory implementation is iSpan, so we use it as a baseline. The speedups for DistSYNC range from 1.3x to 3.1x times over sequential, while speedups for iSpan range from 1.2x to 4.1x over sequential. Using the timing data from Figure II and the speedup plots from Figure 4, we can see that the performance of DistSYNC is heavily dependent on the number of SCCs. For instance, the number of SCCs in LiveJournal is around 972 thousand,

Table II: Properties of benchmark graphs and execution time for iSpan and DistSYNC with GPU parallelism

Graph	#Nodes	#Edges	#SCC	Largest SCC size	iSpan(ms)	DistSYNC(ms)
Fl	2,302,926	33,140,017	485,572	1,605,184	137	196
LJ	4,875,572	68,475,391	971,233	3,828,682	450	518
WE	16,777,215	134,511,383	12,381,433	3,045,754	540	1084
RMAT24	18,268,993	172,183,984	14,459,547	3,796,073	760	1226



(a) Flicker

(b) LiveJournal

(c) Wikepedia

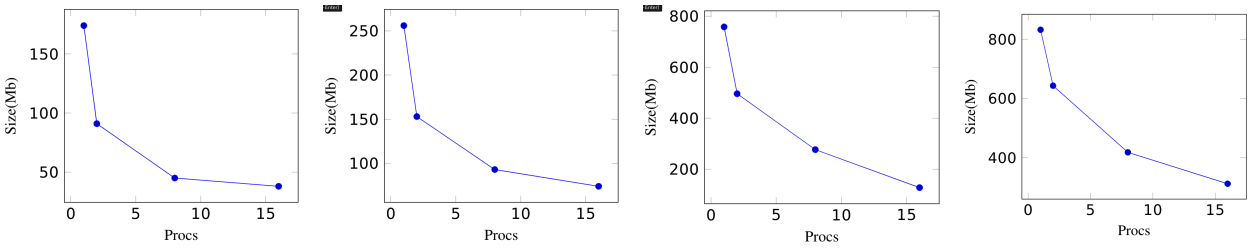
(d) RMAT

Figure 4: Speedups for DistSYNC(red) and baseline iSpan(blue) with respect to single threaded Tarjan's implementation.

while Wiki has over 14 million even though the Wiki graph is only a little over twice its size in terms of the number of edges. And the size of the largest SCC is almost the same. With bigger SCCs, an entire collection of vertices can be represented with just one integer meta-node ID, proving both beneficial for performance and memory utilization. This explains why the speedup of Wiki is significantly less compared to LiveJournal.

Although the number of meta-nodes is only proportional to the number of SCCs and not exact as that would depend on how we partition the graph and if vertices of one SCC are partitioned across multiple processes, making many local SCCs and meta-nodes. Computing phase 1 initial SCCs aren't impacted too much by the number of meta-nodes as they internally apply the trim step explained in section II taking care of small-sized SCCs.

Figure 6 shows us the time taken to fully build the complete meta-graph at phase 2, comparing execution with GPU enabled and sequential. Building the complete meta-graph is a computationally-intensive process as we have to build a vector with all 1to1 pairs of all meta-nodes, but because of how we structured the buffers that receive the meta-nodes and edges during communication,



(a) Flicker (b) LiveJournal (c) Wikepedia (d) RMAT

Figure 5: Average memory utilized per process as number of processes increase

this is a pretty parallelizable operation. We can observe speedups ranging from 5.3x for Flicker up to 12.5x for RMAT.

B. Memory utilization

A significant reason for using distributed memory systems apart from performance is to reduce the memory utilized when processing large graphs. Real-world graphs can get so big that shared memory implementations become unfeasible. Figure 5 shows us the average memory utilized per process as we scale the number of processes. These measurements were taken using memcheck from Valgrind. It should be noted that we use memcheck to track only the heap allocations made by the application and not stack allocations.

We can see that the memory utilized per process increases with more partitions as we reduce the original graph into smaller subgraphs. The reduction in size is initially much greater, but as we add more partitions, this reduction tapers off. This is because creating more partitions tends to crack large SCCs and distribute those vertices across processes making multiple meta-nodes eventually increasing the size of the meta-graph.

VI. Conclusion and Future Work

This paper introduces DistSYNC, a hybrid algorithm leveraging distributed, multithreaded CPU and GPU parallelism for identifying Strongly Connected Components(SCC) in distributed networks. We have also supported the algorithm with implementation details of a hybrid framework. We show that our approach can offer performance speedups up to 3.1x and memory reductions up to 2.6x over a sequential single process implementation. In the future, we aim to extend our approach with point-to-point communication rather than bulk synchronous communication, which we are using now. This would enable us to perform asynchronous execution. We also aim to extend this approach for dynamic networks that change over time. Doing so would enable us to compute SCCs over just the changeset rather than recomputing the entire network for every batch of updates.

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*. IEEE, 2015, pp. 17–27.

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

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

[6] 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*, 2013, pp. 1–11.

[7] 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*, ser. SC '18. IEEE Press, 2018. [Online]. Available: https://doi.org/10.1109/SC.2018.00061

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

[9] 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*. IEEE, 2014, pp. 550–559.

[10] 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.

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

[12] 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.

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