iSPAN: Parallel Identification of Strongly Connected Components with Spanning Trees

Yuede Ji
George Washington University
yuedeji@gwu.edu

Hang Liu*
University of Massachusetts, Lowell
Hang_Liu@uml.edu

H. Howie Huang
George Washington University
howie@gwu.edu

***Abstract*—Detecting strongly connected components (SCCs) in a directed graph is crucial for understanding the structure of graphs. Most real-world graphs have one large SCC that contains the majority of the vertices, as well as many small SCCs whose sizes are reversely proportional to the frequency of their occurrences. For both types of SCCs, current approaches that rely on depth or breadth first search (DFS and BFS) face the challenges of both strict synchronization requirement and high computation cost. In this paper, we advocate a new paradigm of identifying SCCs with simple spanning trees, since SCC detection requires only the knowledge of connectivity among the vertices. We have developed a prototype called iSPAN, which consists of parallel, relaxed synchronization construction of spanning trees for detecting the large and small SCCs, combined with fast trims for small SCCs. We further scale iSPAN to distributed memory system by applying different distribution strategies to the data and task parallel jobs. The evaluations show that iSPAN is able to significantly outperform current state-of-the-art DFS and BFS-based methods by average 18× and 4×, respectively.**

I. INTRODUCTION

In a directed graph, a strongly connected component (SCC) is a maximal subset of vertices such that every vertex has at least one directed path to all other vertices. Detecting all the SCCs in a graph is a fundamental problem for graph analytics [66]. A closely related problem is finding weakly connected component (WCC), which is a maximal subset of vertices such that every vertex can reach each other when changing all directed edges to undirected [58]. SCC has been used in many areas, including model verification [27], pattern matching [21], and graph understanding [65], [67]. In addition, SCC is a basic component for the widely used topological sort [52], [42] and reachability queries [18], [66], [65].

Traditional SCC algorithms are based on depth-first search (DFS) [60], [2]. However, DFS is hard to be parallelized [51]. New parallel algorithms, such as forward-backward (FW-BW) [23] and color propagation [49], are proposed. 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 [28].

State-of-the-art methods combine the power of trim and FW-BW to detect SCC [55], [28]. Particularly, this approach first eliminates trivial SCCs which contain one or two veritces. Afterwards, FW-BW performs BFS in both directions on the remaining graph, that is, starting from a selected pivot, it first performs a *forward* BFS to identify the vertex set that the pivot can reach, followed by a *backward* BFS to identify the set that can reach the pivot. The intersection between both sets is the SCC that contains the pivot [23].

This work is particularly interested in accelerating the FW-BW step of SCC detection stemming from the observation that Multistep [55] and FW-BW BFS [28], two state-of-the-art projects, spend on average 79% and 78%, respectively of the time on FW-BW step for fourteen graphs (Table II).

To accelerate FW-BW step, we adopt the idea that any spanning tree, not necessarily a BFS tree, is sufficient for FW-BW approach to detect SCC [66]. By definition, a spanning tree with the root vertex v is defined as a subgraph that uses the minimum edges to cover all the vertices that are connected with v. We admit BFS provides a satisfied spanning tree. However, BFS introduces extra overhead because spanning trees only need the connectivity information, while BFS also provides the correct levels. To make the levels correct, BFS has to satisfy the stringent requirements on which vertices shall be visited at each level. This leads to a significant, yet completely unnecessary synchronization bottleneck in existing SCC methods.

This paper introduces a new synchronization paradigm – relaxed synchronization (Rsync) – to take advantage of the spanning tree based SCC detection idea because neither synchronous (Sync) nor asynchronous (Async) traversal strategies can satisfy our requirements. Particularly, Sync, which is used in existing BFS methods, can provide better workload balance, but introduces the overhead of level synchronizations. Async can completely eliminate the synchronization overhead, but can easily cause workload imbalance. In contrast, Rsync is able to achieve not only reduced level synchronizations but also balanced workload. By judiciously applying Sync, Async, and Rsync strategies to direction-optimizing BFS, we build a novel spanning tree construction algorithm. We devise a fast SCC detection algorithm, iSPAN, by combining with the optimized usage on trim and an extended trim-3 technique. Further, we successfully scale iSPAN to distributed memory system with judiciously selected communication strategies towards data parallel and task parallel jobs.

Our main contributions are three fold:

First, we propose a *relaxed synchronization strategy, Rsync* (Section IV), which enables an earlier termination for conventional bottom-up traversal. Particularly, in lieu of only

*Work was, in part, done at the George Washington University.

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

terminating the neighbor checking after a parent neighboring vertex is found, Rsync terminates the inspection when a visited neighboring vertex is found. Rsync makes the termination earlier, potentially resulting in fewer neighbor checking and traversal iterations. Our evaluation demonstrates that Rsync achieves 2.7× speedup over Sync bottom-up on average.

Second, we introduce a *fast spanning tree construction algorithm* (Section V) by judiciously applying synchronous, asynchronous, and our novel relaxed synchronous strategies to direction-optimizing BFS, that is, starting with synchronous top-down, switching to relaxed synchronous bottom-up, and finishing with asynchronous top-down. Such a method is able to accelerate SCC detection by upto 6.1×.

Third, we have implemented both the multi-threaded (shared memory) and the distributed versions of iSPAN with the fast spanning tree algorithm, optimized usage on trim and our newly designed trim-3 technique for fast pruning size-3 SCCs. Our evaluation on twelve real-world and two synthetic graphs (Section VII) shows that iSPAN significantly outperforms current DFS and BFS-based methods, i.e., on average, 18× and 4×, respectively. Not limited there, we further evaluate iSPAN with billion-vertex graphs and demonstrate that iSPAN is able to achieve 1.7× speedup over the state-of-the-art. Our distributed version can achieve up to 10.7× speedup with 32 nodes.

The rest of this paper is organized as follows: Section II introduces the background. Section III overviews iSPAN. Section IV presents the relaxed synchronization strategy, Rsync. Section V presents the fast spanning tree construction method. Section VI describes the distributed design of iSPAN. Section VII describes the experimental setup and results. Section VIII presents the related work. Section IX concludes.

II. Background

In this paper, we use $G = (V, E)$ to denote a directed graph, where V is the set of vertices and E is the set of edges. $|V|$ and $|E|$ represent the number of vertices and edges in the original graph, $|V_r|$ and $|E_r|$ the vertex and edge count for the remaining graph after removing the large SCC. Existing parallel SCC works use BFS which has top-down and bottom-up methods [7]. Throughout this paper, we use the term *expand* to refer to loading the neighbors and *inspect* for checking the statuses of them.

A. Graph Property

Interestingly, real-world graphs demonstrate SCC features which resemble power-law property [28]. A single large SCC takes majority of the vertices which is in the same order of graph size. And the rest are small SCCs which are smaller in several orders of magnitude to the large SCC. For the Flickr graph [47] shown in Figure 1, the large SCC has 69.7% of the vertices, while half a million of small SCCs account for the remaining 30.3% of vertices. Interestingly, except the large SCC, this graph does not have other SCCs which has more than 1,000 vertices.

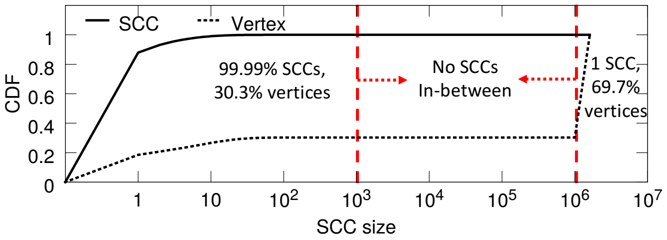


Fig. 1: Cumulative distribution function of SCC size in Flickr.

B. Trim and FW-BW SCC Detection

Trim aims to quickly identify trivial SCCs to greatly reduce the graph size. Trim-1 is for a vertex that is a SCC by itself [45]. The rule is simple: if a vertex has zero in-edges or out-edges, it is a size-1 SCC. In Figure 2(b), vertex 23 is trimmed due to 0 out-degree. Trim-1 will repeat since new size-1 SCCs may appear after trimming, e.g., 22. A recent work studies trim-2 [28]. The trim-2 pattern is that two vertices mutually point to each other, and except the two edges, may have other incoming or outgoing edges, but never both, which guarantees that they cannot belong to other SCCs. In this paper, we call this rule "single direction rule". Vertices 19, 20 are trimmed as size-2 SCC in Figure 2(b).

Existing methods [23], [28], [55] rely on a FW-BW algorithm that leverages BFS to detect SCCs. Starting from the pivot vertex v, the FW-BW algorithm first produces the forward vertex set, FW(v), that represents the vertices that can be traversed using the out-edges. As a result, this will yield a BFS tree shown in Figure 2(c). Next, it will create the backward vertex set, BW(v), that consists of vertices that can be traversed using the in-edges shown in Figure 2(d). Then, it calculates the intersection of FW(v) and BW(v), which is the detected SCC as shown in Figure 2(e).

Inspired by the graph property, state-of-the-art works apply different methodologies to large and small SCCs as shown in Figure 2(a) [28], [55]. Both of the works are using BFS-based FW-BW algorithm to detect the large SCC. For small SCCs, [28] uses trim-1 and new trim-2 to fast reduce graph size, followed by the same BFS-based FW-BW algorithm working on each weakly connected component (WCC). While [55] uses trim-1 for size-1 SCCs, and color propagation and serial Tarjan's algorithm for the remaining small SCCs.

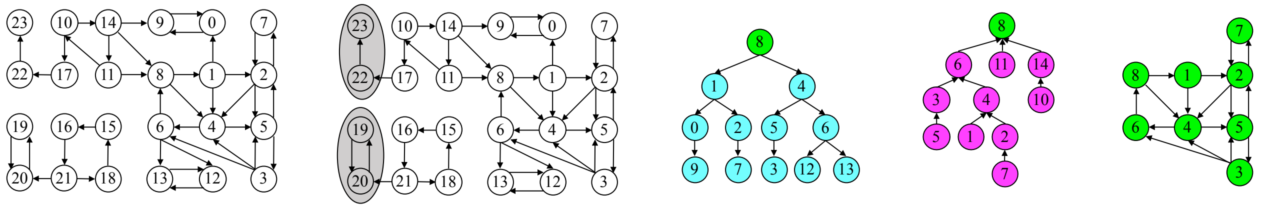
III. Overview

This section first overviews the framework of iSPAN, then shows the correctness of using spanning tree for SCC.

A. The Framework of iSPAN

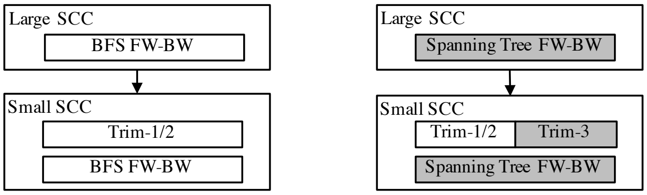
We will present our new SCC detection framework, iSPAN, following the flow charts in Figure 3(b).

Large SCC. iSPAN uses our newly proposed fast spanning tree construction method to accelerate the FW-BW algorithm for the large SCC. For pivot selection, iSPAN follows the same heuristic with [55], which selects the vertex that has the largest product of its in-degree and out-degree. Although this rule does not guarantee that the pivot is indeed from the large



(a) A Sample Graph (b) Trim-1 and Trim-2 (c) Forward BFS Tree (d) Backward BFS Tree (e) FW-BW SCC

Fig. 2: (a) A toy graph running through the paper, (b) Trim-1 and trim-2 (shaded vertices). For vertex 8 (pivot), (c) shows the forward BFS tree, (d) shows the backward one, (e) shows the detected SCC.



(a) State-of-the-art SCC detection (b) iSpan-based SCC detection

Fig. 3: SCC detection methods, (a) state-of-the-art [28], (b) iSpan (the differences are shaded).

SCC, it works well for most real-world graphs. We will leave other pivot selection rules to future works.

Small SCCs. iSpan uses trim techniques for fast detection of the SCCs and spanning tree based FW-BW to detect the remaining small SCCs. In particular, trim is used at two places, before and after large SCC detection. Before large SCC detection, iSpan only uses trim-1 due to the cost of other trims being higher than the benefits. After the large SCC is detected, iSpan trims again, including trim-1, trim-2, and our new extension of trim-3, before detecting small SCCs. For the remaining small SCCs, iSpan divides the graph into WCCs using color propagation algorithm [55], and runs the fast spanning tree FW-BW algorithm on each WCC. Since a SCC is a subset of a WCC, ideally one can select the number of pivots (equal to WCC count) to run FW-BW in parallel.

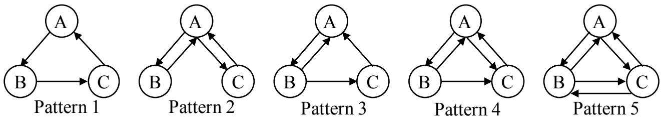


Fig. 4: The internal patterns of trim-3.

Trim-3 aims to quickly detect size-3 SCCs. In particular, we detect the five patterns of size-3 SCCs as shown in Figure 4. At the same time, the outside edges between a vertex from the size-3 SCC and the remaining graph must follow the single direction rule. It is possible to trim even larger SCCs beyond size-3, although at the risk of diminishing returns.

B. Using Spanning Tree for SCC Detection

We will show that any spanning tree is sufficient for FW-BW SCC detection, which serves as the theoretical guidance for our implementation.

The idea of using spanning tree to detect SCC is also investigated in [66]. That work mainly focuses on the I/O efficiency of semi-external SCC detection, which still uses sequential DFS to construct a spanning tree. Different from that, we improve the parallel FW-BW algorithm with spanning tree and devise a new framework for the fast construction of a spanning tree in parallel.

In an undirected graph, a *spanning tree* with the root vertex v is defined as a subgraph using the minimum number of edges to cover all the vertices that are connected with v. Given a root, one can generate many different valid spanning trees, including BFS and DFS trees. In a directed graph, for a root vertex v, there are two distinct types of spanning trees, forward and backward spanning trees. Covering all the vertices that root v reaches with outgoing edges forms the forward spanning tree, and incoming edges the backward spanning tree.

Any valid spanning tree construction method, including BFS and DFS, can produce valid forward or backward traversal. Which means, they can deliver the correct results for forward and backward traversal. Therefore, spanning tree based FW-BW algorithm can deliver the correct SCCs.

This observation can be summarized as follows:

Lemma 1: For vertex v in a graph G, the SCC containing v, SCC(v), can be obtained by the intersection of any pair of valid forward and backward ***spanning trees****. That is, SCC(v) = FST(v) ∩ BST(v).*

Proof 1: For vertex v, a valid forward spanning tree FST(v) contains all the vertices that can be reached from v. Similarly, a valid backward spanning tree BST(v) covers the vertices that can reach v. That is, FST(v) equals to FW(v) and so does BST(v) to BW(v). By definition, the SCC for vertex v will contain the vertices shared in both sets.

IV. Rsync: Relaxed Synchronization Strategy

This section will discuss the new relaxed synchronization strategy and its benefits.

A. Rsync: Relaxed Synchronization

Rsync relaxes the level-by-level inspection imposed by conventional synchronization (Sync) traversal but still synchronizes to avoid workload imbalance in Async. Algorithm 1 presents the pseudocode of bottom-up Rsync method.

Bottom-up Rsync can terminate as long as finding a visited parent or sibling. However, in conventional bottom-up BFS where a vertex can only be terminated by the visited parent vertex from the previous level. That is, for conventional BFS

Algorithm 1: rsyncBotUp(sa, beg_pos, adj_list)

```
1 foreach unvisited vertex $u$ in parallel do
2     foreach vertex $w \in InNeighbor(u)$ do
3         if sa[w] is visited then
4             sa[u] = visited;
5             break;
6 barrier();                    // synchronization point
```

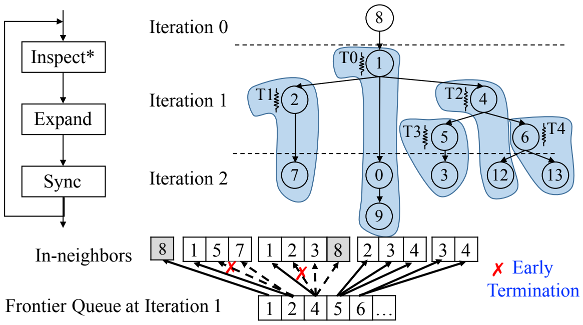


Fig. 5: Rsync bottom-up example (Shaded areas represent the workload of each thread).

at the level i, the inspection is limited to the neighbors of the vertices that belong to level $(i-1)$. In contrast, at the i-th level, Rsync allows to inspect and expand to the neighbors of all visited vertices, regardless of at which level the vertices have been visited. Rsync is also different from DFS because it only checks one hop of neighbor vertices.

The benefits of Rsync come from the *new early termination condition*, which allows further reduction of needed computation. That is, the flexibility increases the possibility of early termination, and reduces the amount of edges that need to be inspected in the bottom-up traversal, which can be seen in the two following cases.

Case 1: A vertex can be early terminated by newly inspected vertices. The conventional BFS tree of Figure 2(b) is shown in Figure 2(c). Vertex 4 needs to traverse 4 edges (increasing order, 1, 2, 3, 8) so that it can be visited. However, in Rsync bottom-up, assume the workload is distributed as shaded in Figure 5, the (fast-running) thread 0 has already visited vertex 1 earlier at iteration 1. The (slow-running) thread 2 is able to visit vertex 4 by checking only 1 edge. In this example, Rsync reduces the traversed edge number by 3.

Case 2: The vertex that should be inspected at later level can be inspected earlier. Rsync also allows the inspection of the vertices which would not be allowed in conventional BFS, e.g., vertices 2, 5, 6 in Figure 5. Rsync can work on all unvisited vertices, unlike conventional BFS that only works on the unvisited ones belong to the current depth. This method introduces more parallelisms and better workload balance.

Rsync essentially eliminates the inter-level constraint in BFS, and synchronizes when the threads complete processing (at the end of an iteration) as shown in line 6 of Algorithm 1. It can generate the correct spanning tree because a vertex which can be visited in conventional BFS will be guaranteed to be visited in Rsync bottom-up. Therefore, Rsync bottom-up can be used to detect SCC.

B. Benefits of Rsync

Bottom-up BFS has been shown to be faster to traverse the levels in the middle [7]. We will show that Rsync bottom-up is faster than Sync under the same conditions.

Sync Bottom-Up. Let N denote the number of vertices in a graph, $\bar{d}$ the average in-degree, $N_v(k)$ the number of vertices visited in the k-th level, and $N_u(k)$ the number of vertices remaining unvisited in the k-th level. As a result, the probability of an unvisited vertex will be visited at k-th level is:

$$p = \frac{N_v(k-1)}{N} \tag{1}$$

At level k, for each unvisited vertex i, assuming it has d_i in-neighbors, this vertex can either find no parent, which has the probability of $(1-p)^{d_i}$, or a parent of the j-th neighbor, which has a probability of $(1-p)^{j-1} \cdot p$. Thus, for vertex i, the expected number of edges traversed at level k is:

$$\mathbb{E}_k^i(p) = d_i \cdot (1-p)^{d_i} + p \cdot \sum_{j=1}^{d_i} j \cdot (1-p)^{j-1} \tag{2}$$

There is a geometric series in Equation 2. We can get Equation 3 by multiplying $(1-p)$ on both parts.

$$(1-p)\mathbb{E}_k^i(p) = d_i \cdot (1-p)^{d_i+1} + p \cdot \sum_{j=1}^{d_i} j \cdot (1-p)^{j} \tag{3}$$

By doing subtraction of Equation 2 and 3, we can get the final expectation as Equation 4.

$$\mathbb{E}_k^i(p) = \frac{1-(1-p)^{d_i}}{p} \tag{4}$$

Assuming d_i is an integer constant in range $[0, \infty)$ the first derivative of $\mathbb{E}_k^i(p)$ against p is:

$$\frac{\partial \mathbb{E}_k^i(p)}{\partial p} = \frac{d_i p(1-p)^{d_i-1} + (1-p)^{d_i} - 1}{p^2} \tag{5}$$

Assuming p is in the range $(0, 1)$, we can transform Equation 5 to

$$\frac{\partial \mathbb{E}_k^i(p)}{\partial p} = \frac{d_i p + 1 - p - (1-p)^{1-d_i}}{p^2(1-p)^{1-d_i}} \tag{6}$$

The denominator is greater than 0 due to $p \in (0, 1)$. Let $g(p)$ denote the numerator and its first derivative is:

$$\frac{\partial g(p)}{\partial p} = (d_i - 1)\frac{(1-p)^{d_i} - 1}{(1-p)^{d_i}} \tag{7}$$

$\frac{\partial g(p)}{\partial p}$ is smaller than 0 when $d_i \in (1, \infty)$, is 0 when d_i is 0 or 1. When $d_i \in (1, \infty)$, $g(p)$ is a nonincreasing function and $g(0)$ equals 0, thus $g(p)$ is smaller than 0 when $p \in (0, 1)$. That means, $\frac{\partial \mathbb{E}_k^i(p)}{\partial p}$ is smaller than 0, which denotes that $\mathbb{E}_k^i(p)$ is a nonincreasing function when $d_i \in (1, \infty)$. When d_i is 0 or 1, $\mathbb{E}_k^i(p)$ is 0 or 1, respectively. **In conclusion, $\mathbb{E}_k^i(p)$ is either a nonincreasing function ($d_i \in (1, \infty)$) or a fixed number (d_i is 0 or 1).**

Rsync Bottom-Up.

Lemma 2: Assuming the same switching condition is applied for both Sync and Rsync bottom-up, Rsync will check less edges than Sync.

Proof 2: As shown in Algorithm 1, Rsync relaxes the early termination condition by allowing the termination as long as one neighbor is visited. As a result, the probability of visiting a vertex for Rsync, previously shown in Equation 1, is changed to:

$$p_r = \frac{\lambda N_v(k) + N_v(k-1)}{N} = \frac{\lambda N_v(k)}{N} + p \tag{8}$$

where λ is the portion of vertices that terminate based on vertices just visited at the k-th iteration. This equation means Rsync not only terminates the inspection when it finds a visited neighbor, as $N_v(k-1)$, but also when it meets a vertex that is visited by level k as only a portion – λ of vertex can terminate based on vertices belonging to level k. At level k, for all unvisited vertices (total as $N_u(k)$), bottom-up BFS needs to check the edges of

$$T_k = \sum_{i=1}^{N_u(k)} \mathbb{E}_k^i(p) \tag{9}$$

As a result, the difference between Sync and Rsync of checked number of edges is

$$\Delta = \sum_{i=1}^{N_u(k)} (\mathbb{E}_k^i(p) - \mathbb{E}_k^i(p_r)) \tag{10}$$

Since $\mathbb{E}_k^i(p)$ is nonincreasing and $p_r \geq p$, we can conclude that $\mathbb{E}_k^i(p) - \mathbb{E}_k^i(p_r) \leq 0$. Therefore, the accumulated $\Delta \leq 0$, which means, Rsync checks less edges than Sync bottom-up.

C. Actual Rsync Behaviors

To illustrate the performance benefit, we compare the visited edges of Rsync and Sync bottom-up for detecting the large SCC, under the same switch condition and configurations (graphs are shown in Table II). The ratio is calculated by the edge number of Sync over Rsync. On average, Sync traversed 2.66× and 3.17× more edges than Rsync for forward and backward as shown in Figure 6.

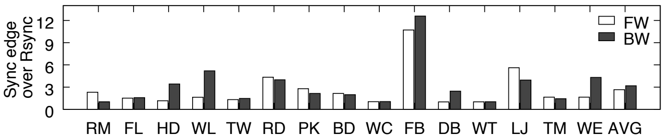
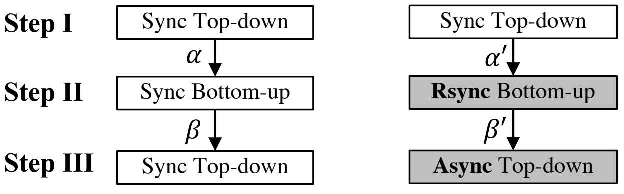


Fig. 6: Traversed edge of Sync over Rsync bottom-up.

V. Fast Spanning Tree Construction Method

The novelty of our fast spanning tree construction dwells in our judicious choice of applying the most suitable synchronization mechanisms for various traversal steps, despite, similar to existing projects [28], [55], we adopt direction optimizing BFS [7] for iSpan. Figure 7 compares our iSpan traversal (Figure 7(b)) against state-of-the-art direction-optimizing approach (Figure 7(a)). Particularly, iSpan starts with a conventional synchronous top-down (Step I), switches to our novel Rsync bottom-up (Step II), and finishes with Async top-down (Step III). The pseudocode of our forward and backward traversal is shown in Algorithm 2. This algorithm is called twice, one for forward and another for backward traversal. The graph is represented in compressed sparse row (CSR) format [15], which is widely used in contemporary graph systems [39], [11], [46]. The forward CSR is represented by fw_beg_pos[$|V|$+1] and fw_adj_list[$|E|$] and the backward CSR uses bw_beg_pos[$|V| + 1$] and bw_adj_list[$|E|$].



(a) Direction-optimizing BFS (b) iSpan traversal

Fig. 7: The spanning tree construction method in (a) Direction-optimizing BFS (a.k.a., state-of-the-art [28], [55] approach) and our (b) iSpan traversal.

Algorithm 2: iSpanTraversal(pivot, sa, fw_beg_pos, fw_adj_list, bw_beg_pos, bw_adj_list)

```
isSyncTopDown = true;
isBottomUp = false;
isAsyncTopDown = false;
sa[pivot] = 1;
level = 1;
while frontier queue changes do
    foreach thread $t \in T$ in parallel do
        if isSyncTopDown then
            syncTopDown(sa, fw_beg_pos, fw_adj_list);
        else if isBottomUp then
            rsyncBotUp(sa, bw_beg_pos, bw_adj_list);
        else if isAsyncTopDown then
            asyncTopDown(sa, fw_beg_pos, fw_adj_list);
        // Switch condition
        if isSyncTopDown and $M_f > (M_r/\alpha)$ then
            isSyncTopDown = false;
            isBottomUp = true;
        else if isBottomUp and $N_f < (|V|/\beta)$ then
            isBottomUp = false;
            isAsyncTopDown = true;
    barrier();                          // synchronization point
    level++;
```

A. Synchronization Strategy

Synchronous (Sync) method requires synchronization across different threads at the end of every level. Later, the workload will be redistributed to each thread to balance the workloads. Applying this philosophy to top-down traversal, as shown in Figure 8, each thread at each level identifies the frontiers that will be expanded at the next level and stores them in private frontier queues. For synchronization, all threads need to combine all the frontiers into one global frontier queue. At the next level, the threads will get equally distributed work from the shared global queue. Thus, the workload of each thread is balanced.

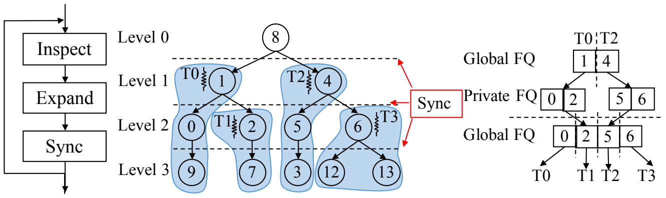


Fig. 8: Sync top-down example (shaded areas represent the workload of each thread).

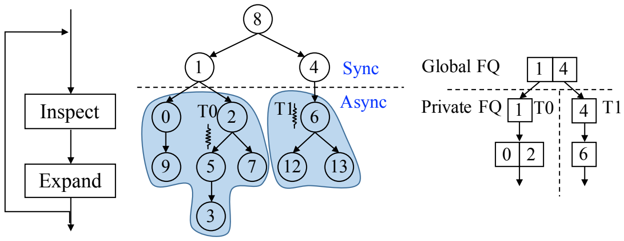


Fig. 9: Async top-down example (shaded areas represent the possible workload of each thread).

Asynchronous (Async) approach, in contrast, allows every thread to work on its private frontier queue and does not impose any synchronization. Still using top-down as an example, as shown in Figure 9, after returning from bottom-up to top-down, the global queue $\{1, 4\}$ is divided into two private ones, thread 0 has $\{1\}$ and thread 1 has $\{4\}$. Then, each thread will work on this private queue and stops when it becomes empty without synchronizing with other threads. Thus, the workloads of all the threads are easy to be imbalanced.

Async also faces race conditions when two threads access the same vertex at the same time. Both threads will put the vertex into their private queues. However, when the two threads expand from this vertex, they will inspect the status before expanding the neighbors. Since the probability of expanding the same vertex at the same time again is rather low, it only wastes the status inspection time of one thread for one vertex, but the spanning tree is still correct.

Relaxed-Synchronization (Rsync). To fill the gap between Sync and Async, we leverage our relaxed synchronization strategey, Rsync. We should note that Rsync can reduce the number of level synchronizations but cannot fully avoid, and it cannot be used in top-down traversal.

In summary, Sync gains better workload balance but limited by thread synchronization, Async avoids thread synchronization but may run into workload imbalance, Rsync provides better workload balance and reduces synchronization levels. This comparison is summarized in Table I.

B. Direction-Aware Fast Spanning Tree Construction Method

Applying Sync Top-down to Step I: Sync instead of Async is selected for this step for two reasons. First, this step needs to switch to bottom-up at a certain level. Such a decision can only be made when we know the global amount of workload across all participating threads, which contradicts the design of Async that is complete asynchronous traversal. Second, often, step I only requires very few iterations before switching, which makes level synchronization overhead negligible comparing to its benefit of balancing the workload. Therefore, we use Sync top-down to initialize our spanning tree construction.

α′: Our spanning tree method follows the same switch condition as in [7], that is, when $M_f > (M_u/\alpha)$, where M_f denotes the number of edges in the frontier, M_u the number of unvisited edges, and α is a pre-defined threshold. Similarly, we approximate $M_f = N_f * d$, $M_u = N_r * d$, where N_f denotes the number of visited vertices, d denotes the average degree, N_r denotes the remaining unvisited vertices [55]. A larger α value will lead to an earlier switch, and as a result, Rsync bottom-up can be leveraged to decrease the number of traversed edges and provide better performance than Sync as we will show later. Our current implementation leaves α as a runtime parameter which can be tuned based on the application need. In our evaluation, we set α to a fixed value of 30.

TABLE I: Comparison of different traversal methods.

	Top-down			Bottom-up		
	Sync	Rsync	Async	Sync	Rsync	Async
Reduced synchronization	✗	–	✓	✗	✓	–
Workload balance	✓	–	✗	✓	✓	–

Applying Rsync Bottom-up to Step II: As the traversal continues, the amount of edges that need to be expanded and inspected climbs rapidly, leading to the switch from top-down to bottom-up. Async bottom-up is not selected because direction switching requires the collective information across all threads which is not supported by Async. On the other hand, we select Rsync instead of Sync bottom-up because Rsync is proved to be faster under the same condition.

β′: As the frontier size becomes smaller, iSpan needs to switch back to top-down. For this, iSpan uses another condition $N_f < (|V|/\beta)$, where N_f denotes the number of vertices in the frontier, and β is a pre-defined parameter. The larger the β value, the later the switching happens. We set the β value much larger than [7] to fully utilize the power of $Rsync$. In particular, we set β to 200 in our experiments instead of 24 in [7].

Applying Async Top-down to Step III: We select Async instead of Sync top-down to mainly cope with the long-tail phenomena that is commonly presented in real-world graphs [48]. Formally, long-tail is the situation that the traversal lasts for large number of iterations with few vertices in a frontier. Figure 10 demonstrates such a scenario in Wikipedia graph (WL). At first, there exists a large number of frontiers, more than millions at certain levels. However, after the 30-th level, the frontier size becomes smaller than 10, and reduces to 1 at the 425-th level till its termination at $1,361$ level.

In this case, the workload is extremely small from level 30 - 1,361 which suggests that even synchronizing the traversal at each level cannot affect workload distribution. However, the overhead of synchronization stays. Actually, synchronization becomes the major time consumer during 30 - 1,361 levels which motivates our design of Async. It is also important to mention that Async top-down can provide comparable to, if not better than, Sync top-down even without long-tail.

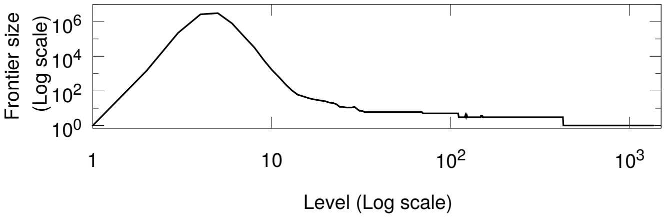


Fig. 10: Long tail in the Wikipedia graph (WL).

VI. Distributed iSpan

This section scales iSpan to distributed memory system with OpenMPI. We partition the graph using row-wise 1-d partitioning method [56], [16], [36]. This simple method can produce graph partitions that are communication friendly because the vertices in each partition are consecutive, and also beneficial to bottom-up approach as shown in [8].

A. Data Parallel for the Large SCC

The large SCC detection is data parallel because all the workers are working together to resolve one task. The challenge in such a data parallel job is the high communication cost – all the workers have to communicate vertex statuses at the end of each iteration [7]. A naive communication strategy will communicate the status array of size $|V|$ at the end of each iteration. To reduce the high communication cost, we design a hybrid communication strategy that adaptively uses bitwise compression and frontier queue only mechanisms.

Bitwise status compression. Bitwise status array has been explored in distributed memory systems [53]. In particular, [53] compresses the 4-byte status array into bitwise status array in order to exchange the newly visited vertices. Note that this method cannot directly use bitwise status array for traversal because it has to differentiate the unvisited, and the levels of visited vertices (more than two statuses). During traversal, this approach has to update both the original and bitwise status arrays. After communication, it also has to use received bitwise status array to update the original status array. However, in our case, because iSpan eliminates the needs of recording the level information, it simply uses a bitwise array to record the status of each vertex. Therefore, our communication is largely simplified. Although bottom-up traversal only needs visited or unvisited information, it will need the last level information when switching to top-down. Thus, we need three statuses (2 bits) for a vertex, i.e., unvisited (00), previously visited (01), and newly visited (10). If bottom-up is used, every worker will scan the vertices in its partition and change the status from newly visited to previously visited. Then, it will traverse in the normal bottom-up manner. When it switches to top-down, one can get the frontier queue by extracting vertices with newly visited status.

Frontier queue. Chances are, at the beginning and end iterations of large SCC detection, only a very small portion of the status array will be updated [7]. This implies that communicating the entire status array, albeit compressed, is wasteful. We thus only communicate the frontiers. After receiving the frontiers, we update the bitwise status array correspondingly. This approach has been used in [16].

Hybrid. Clearly, the aforementioned two communication mechanisms excel at complementary scenarios, that is, bitwise status compression prefers large volume of updates while the other is opposite. Our hybrid design chooses the best communication strategy at each iteration based upon the number of frontiers. We use node-to-node traffic to quantify the communication cost. In particular, at each iteration, bitwise status compression communicates $\frac{|V|}{8}$ bytes for top-down traversal and $\frac{|V|}{4w}$ bytes for bottom-up because each worker can update the entire status array in top-down while only touches its own partition in bottom-up, where w denotes the number of workers. For the second design, assuming f_i^j denotes the frontier queue size of the j-th worker at the i-th iteration, and we use 4-byte integer to represent each frontier, this method will exchange the maximum frontier queue size among all the workers, which is $\max_{j\in w}(4f_i^j)$ bytes. Therefore, we choose the frontier queue approach if $\max_{j\in w}(4f_i^j)$ is smaller and the bitwise status compression method otherwise. In summary, the forward traversal needs to communicate $\sum_{i=1}^{t_T} \min(\max_{j\in w}(4f_i^j), \frac{|V|}{8}) + \sum_{i=1}^{t_B} \min(\max_{j\in w}(4f_i^j) + \frac{|V|}{4w})$ bytes of data, where t_T, t_B denote the number of iterations in top-down and bottom-up traversals, respectively. Assuming the frontier queues are equally distributed among the workers and the forward traversal shares the same frontier size and iteration number with the backward, the size of communication packet will be $2 \cdot \sum_{i=1}^{t_T} \min(\frac{4f_i}{w}, \frac{|V|}{8}) + 2 \cdot \sum_{i=1}^{t_B} \min(\frac{4f_i}{w}, \frac{|V|}{4w})$ bytes since $\max_{j\in w}(4f_i^j)$ is simplified to $\frac{4f_i}{w}$, where f_i denotes the frontier queue size in the i-th iteration.

B. Task Parallel for the Small SCCs

The small SCC detection, which is comprised of thousands of tasks, is clearly task parallel because each task is fulfilled by one worker exclusively. In trim-1, each worker only needs to check the vertices in its local partition, which is communication free. However, for the trim-2/3 and non-trivial small SCCs, each worker may access the vertices that are not in its local partition. If exploiting our designed hybrid communication strategy, frequent communications will introduce high overhead.

Instead, we compact the remaining graph into a smaller subgraph and distribute one copy across all workers to avoid communications stemming from the following two reasons: First, we observe that the remaining subgraph only contains, on average, 2.1% vertices and 0.5% edges of the original graphs for the fourteen tested graphs. The largest percentages are 11.8% and 3.2% for the vertex and edge, respectively. Such a small subgraph can be easily generated and stored across all workers. Second, we can reorder the vertices during graph compaction which can potentially bring better cache locality [4], [62], [32].

Furthermore, we introduce **graph compaction** technique in this distributed setting. Initially, every machine reads the status array and builds the mapping from the original vertex IDs to the new ones. Then, every machine scans its local partition and gets the size of the remaining vertices and edges in that partition. Subsequently, all the machines communicate to get the size array of each partition and calculate their global addresses in CSR. Afterwards, every node will rescan its local partition and update their CSR. Finally, we rely on MPI_Allgatherv to construct the full view of the graph across all machines. Let $|Vr_j|$, $|Er_j|$ denote the number of vertices and edges in the j-th worker for the remaining graph, the node-to-node communication consumption is $8 + \max_{j\in w}(4 \cdot$

$|Vr_j|) + \max_{j\in w}(4 \cdot |Er_j|)$ bytes, where the 8 bytes is for the two size arrays. Thus, the graph compaction operation for both the forward and backward CSR will communicate the data of $16 + 2 \cdot \max_{j\in w}(4 \cdot |Vr_j|) + 2 \cdot \max_{j\in w}(4 \cdot |Er_j|)$ bytes. Assuming the vertices are equally distributed among the workers, the communication can be further simplified to $8 \cdot \frac{|Vr|+|Er|}{w} + 16$, where $|Vr|$ and $|Er|$ denote the number of vertices and edges for the remaining graph r, with the analogous simplification process from Section VI-A.

C. *Communication Complexity*

In addition to the communications in the large SCC and graph compaction, there are several other communications. In particular, iSpan needs to communicate the status array before detecting the large SCC, which communicates $\frac{4|V|}{w}$ bytes of data. With the compacted graph, the coloring-based WCC computation communicates $\sum_{i=1}^{t_r} \frac{4f_i}{w}$ data because iSpan uses bottom-up for color propagation which cannot use the bitwise status compression, where t_r denotes the number of iterations of traversing the remaining graph. The later trim-2/3 and small SCC detection will only incur two synchronizations, which will exchange $\frac{8|V_r|}{w}$ data. Thus, the total amount node-to-node communication data will be $\frac{4|V|+16|Vr|+8|Er|}{w} + 2 \cdot \sum_{i=1}^{t_T} \min(\frac{4f_i}{w}, \frac{|V|}{8}) + 2 \cdot \sum_{i=1}^{t_B} \min(\frac{4f_i}{w}, \frac{|V|}{4w}) + \sum_{i=1}^{t_R} \frac{4f_i}{w} + 16$, where t_T, t_B denote the number of iterations of top-down and bottom-up traversal for the original graph.

VII. Experiments

The experiments are performed on a server with two Intel Xeon E5-2683 (2.00 GHz) CPUs, each of which has 14 cores and 28 hardware threads with 35 MB of last-level cache and 512 GB of main memory. The server runs CentOS Linux (7.2) operating system. iSpan is implemented in about 4,900 lines of C++ codes and compiled using g++ version 4.8.5 with the -$O3$ option. We use OpenMP version 3.1 as the multithread library. The results are reported with an average of ten runs.

A. *Graph Benchmarks*

We evaluate the performance of iSpan on 12 real-world graphs and 2 synthetic graphs shown in Table II. The real-world graphs are collected from University of Koblenz-Landau [35] and Stanford University [37]. They are classified into three categories: social networks, web graphs, and communication networks as summarized in Table II.

B. *Comparison of State-of-the-Art*

This section compares the performance of iSpan with state-of-the-art approaches. Specifically, *Tarjan* stands for the classical serial Tarjan's algorithm [60], *UFSCC* is a DFS-based on-the-fly SCC detection approach [13], *BFS FW-BW* stands for a three-step FW-BW SCC detection approach [28], and *Multistep* [55] is another three-step detection project. We get the source codes of *UFSCC*, Hong's *BFS FW-BW* and *Multistep* from the authors, and Tarjan algorithm from Hong's implementation. We run their source codes on our server with the same configurations. Note that *BFS FW-BW* sometimes cannot select a pivot from the large SCC for graphs, like DB and HD, which dramatically lowers its performance. For these graphs, we choose to report their best performance from multiple tests. Also, Multistep does not work for several graphs including DB, WC, WT and HD, even after our attempts to adjust configurations such as stack size in the code.

TABLE II: Graph benchmarks specification.

Graph (Abbr.)	# Nodes	# Edges	# SCC	Large SCC size	# Small SCC	Size-1 SCC	Size-2 SCC	Size-3 SCC
Baidu (BD)	2,141,301	17,794,839	1,503,004	609,905	1,503,003	1,480,722	18,688	2,473
Dbpedia (DB)	3,966,925	13,820,853	3,636,316	178,593	3,636,315	3,587,274	24,933	8,868
Facebook (FB)	96,079,682	679,728,426	93,892,292	2,186,877	93,892,291	93,891,890	322	54
Flickr (FL)	2,302,926	33,140,017	485,572	1,605,184	485,571	426,936	25,620	10,567
Hudong (HD)	2,452,716	18,854,882	2,189,120	185,668	2,189,119	2,153,858	24,832	4,786
LiveJournal (LJ)	4,847,572	68,475,391	971,233	3,828,682	971,232	947,777	16,875	3,280
Pokec (PK)	1,632,804	30,622,564	325,893	1,304,537	325,892	323,799	1,904	151
Twitter (TW)	41,652,231	1,468,365,182	8,044,729	33,479,734	8,044,728	7,947,098	80,112	12,198
Wiki-com (WC)	2,394,386	5,021,410	2,281,880	111,881	2,281,879	2,281,312	529	29
Wiki-en (WE)	18,268,993	172,183,984	14,459,547	3,796,073	14,459,546	14,450,686	7,201	1,107
Wiki-link (WL)	11,196,008	340,309,824	4,266,559	6,916,926	4,266,558	4,260,669	3,014	1,128
Wiki-talk (WT)	2,987,536	24,981,161	2,736,716	249,610	2,736,715	2,735,641	992	58
Random (RD)	4,000,001	256,000,000	2	4,000,000	1	1	0	0
R-MAT (RM)	3,999,984	256,000,000	2,105,950	1,894,035	2,105,949	2,105,948	0	0

The total runtime is composed of the three steps (trim, large SCC, small SCCs) and the pivot selection. Figure 11 shows the speedup achieved by iSpan over these methods. Correspondingly, Table III presents the detailed time consumption of each method on various graphs. For all the graphs, on average, iSpan can get 67.3×, 20.9×, 4.1×, and 3.6× speedup over Tarjan, UFSCC, BFS FW-BW, and Multistep, respectively. For DFS-based Tarjan and UFSCC, iSpan obtains the largest speedup from graph FB by 271.7× and 150.3×. The reason is that for graph FB, the vertices in size-1 SCCs take the major of about 97.72%. For this kind of graph, trim benefits a lot for three-step detection approaches (e.g., BFS FW-BW, Multistep, and iSpan).

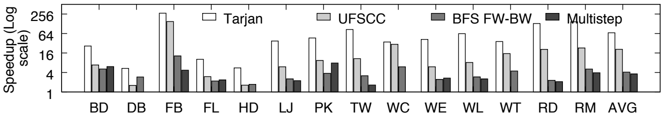


Fig. 11: Speedup of iSpan over state of the art (56 threads). The x-axis shows graphs and the last one is average.

Compared to BFS FW-BW method, iSpan achieves the maximum and minimum speedup from FB and HD of 12.9× and 1.7×. And with regard to the Multistep method, iSpan gets the maximum speedup by 7.8× on PK and minimum speedup from TW by 1.6×.

C. *Shared Memory Scalability*

In this section, we will present three experiments, large graphs, the speedup over Tarjan's algorithm, and the scalability over itself.

We test iSpan on several larger graphs as shown in Table IV, specifically TM and FR graphs from [35], and a synthetic graph KR generated from Graph500 generator. Note that while iSpan is able to run all three graphs, current implementations of BFS FW-BW [28] and Multistep [55] crash on billion vertex graphs due to segmentation faults. We successfully modify the codes in Multistep to support large graphs but fail for BFS FW-BW. On the average of ten runs, iSpan achieves 1.7× and

TABLE III: Runtime (ms) (The speedup of Rsync over current *best approach* is shown in parentheses).

Graph	BD	DB	FB	FL	HD	LJ	PK	TW	WC	WE	WL	WT	RD	RM	Avg
Tarjan	414	352	21,262	551	292	1,710	628	38,999	171	3,801	6,591	507	5,281	3,578	6,010
UFSCC	107	*104*	11,759	160	*85*	274	127	4,887	145	544	846	214	838	554	1,475
BFS FW-BW	*81*	191	1011	*117*	91	116	*51*	1450	*29*	*220*	*302*	*92*	124	*62*	281
Multistep	167	-	*406*	152	-	*109*	144	*816*	-	314	312	-	115	*104*	*264*
ISPAN	16 (5.1)	66 (1.6)	78 (5.2)	54 (2.2)	53 (1.6)	46 (2.4)	13 (3.9)	457(1.8)	5 (5.8)	91 (2.4)	104 (2.9)	14 (6.6)	40 (2.9)	24 (2.6)	76 (3.5)

TABLE IV: Runtime (seconds) on large graphs (- denotes program crash caused by segmentation fault).

Graph	$\|V\|$	$\|E\|$	BFS FW-BW	Multistep	iSpan (speedup)
Twitter_MPI (TM)	52M	2.0B	1.8	1.2	0.7 (1.7×)
Friendster (FR)	68M	2.6B	-	1.3	1.1 (1.2×)
Kron_30 (KR)	1.07B	17.2B	-	62.7	46.7 (1.3×)

1.2× speedup on TM and FR. For a graph with billions of vertices, iSpan takes tens of seconds to compute, specifically 46.7 seconds on KR, 1.3× speedup over Multstep.

ISPAN is a parallel solution that can scale to a large number of threads. We compared the performance of ISPAN against other approaches under different number of threads. We select seven representative graphs covering social network graphs (LJ, FL, TM, TM), web graph (WE, WL) and synthetic graph (RD). Figure 12 presents the speedup over the serial Tarjan algorithm as [28], [55] did in their experiments. For all the graphs from Table II and TM from Table IV, Figure 12(h) presents the average speedup. The other two graphs (FR and KR) from Table IV are not included stemming from the failure of the baseline method (Tarjan's implementation from [28]). One can see that ISPAN achieves the best performance with the increase of threads on both real-world and synthetic graphs. Taking LJ as an example, for 1 thread, ISPAN, Multistep, and BFS FW-BW get 5.9×, 5.1×, 1.5× speedup, while UFSCC is worse than Tarjan. As the thread increases, the speedups of the four approaches also improve. When it reaches to 56 threads, ISPAN can get upto 37× speedup, while Multistep, BFS FW-BW, and UFSCC get at most 16× speedup.

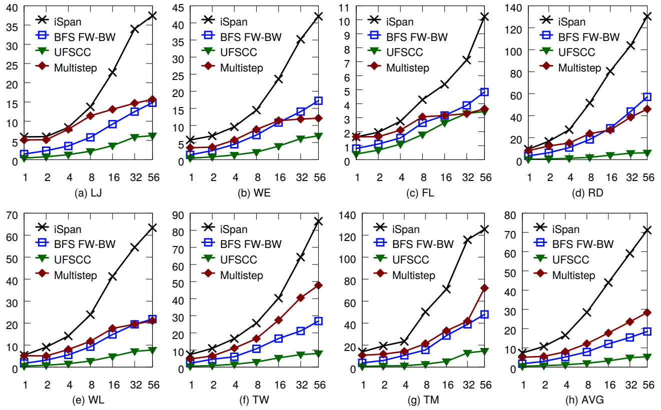


Fig. 12: The speedup over Tarjan's serial algorithm (x-axis shows the number of threads, y-axis shows the speedup).

Further, we show the scalability with regarding to the increase of threads. Figure 13 presents the scalability of the three largest graphs and the average on all the 17 graphs. While ISPAN is able to run all the graphs, two related projects fail on some graphs[1]. One can see that for all the graphs, ISPAN is able to achieve 11× speedup on average. Particularly, ISPAN can scale upto to 19× speedup on the largest graph KR, which almost doubles the scalability of Multistep.

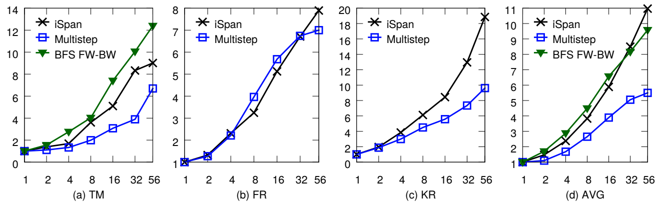


Fig. 13: The scalability on the large graphs and average (x-axis shows the number of threads, y-axis shows the speedup).

D. Distributed Scalability

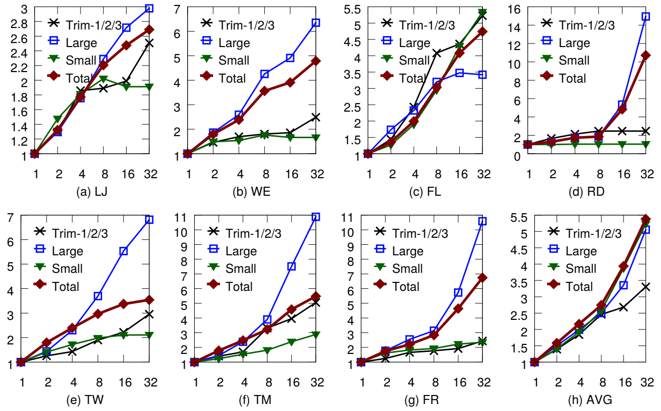
We test the scalability of distributed ISPAN on a cluster. We scale ISPAN to 32 nodes and use seven representative graphs covering social network graphs (LJ, FL, TW, TM, FR), web graph (WE), and synthetic graph (RD). Later, we will show in Figure 16 the performance of ISPAN on additional 9 graphs[2].

Figure 14 presents the scalability of each step and total runtime for the seven graphs and the average. The time of small SCC includes CSR compaction time, WCC computation time, and remaining FW-BW computation time. For the total runtime, ISPAN achieves 10.7×, 6.7×, 5.5×, 4.8×, 4.7×, 3.7×, and 3.5× speedups on RD, FR, TM, WE, FL, LJ, and TW graphs, respectively. In particular, trim technique enjoys good scalability for the 7 graphs and achieves upto 4.4× speedup on FL, because it is a pure task parallel job which is communication free. Large SCC scales well and reaches upto 14.9× for RD. It is a computation intensive job in which communication overhead is canceled out by computation time. Large SCC detection dominates the distributed scalability which is consistent with the results from the shared memory tests. Small SCC can scale for FL, but does not scale for RD because it has zero small SCCs. For other graphs, small SCC do not scale well when the nodes are more than 8 due to the large communication overhead. Overall, both the data and task parallel jobs can scale well. The data parallel jobs can enjoy the benefit of our hybrid communication strategy especially when it dominates the runtime.

Furthermore, we compare to the state-of-the-art distributed SCC implementation, named HPCGraph [56] as shown in Figure 15. When it scales to 32 nodes, ISPAN achieves better scalability for graph WE, FL, and RD. Particularly, ISPAN

[1] BFS FW-BW fails on the two largest graphs (FR, KR), and Multistep fails on four graphs (DB, HD, WC, and WT)

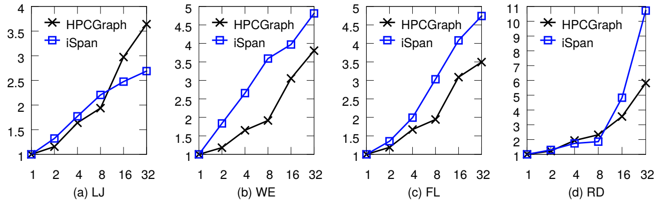
[2] HPCGraph [56] fails on several graphs, e.g., FR, TW, TM, DB, HD, WC, and WT.



(a) LJ (b) WE (c) FL (d) RD

(e) TW (f) TM (g) FR (h) AVG

Fig. 14: The scalability of iSPAN in distributed systems (x-axis shows the number of nodes, y-axis shows the speedup).



(a) LJ (b) WE (c) FL (d) RD

Fig. 15: The distributed scalability comparison (x-axis shows the number of nodes, y-axis shows the speedup).

can achieve 4.7×, 4.8×, and 10.7× compared to HPCGraph's 3.8×, 3.5×, and 5.8×, respectively. iSPAN has lower scalability for graph LJ 2.7× compared to HPCGraph's 3.6×. However, the runtime of our baseline (i.e, 1 node) is much faster than the baseline of HPCGraph. Ours is able to achieve 8×, 1.9×, 3.3×, and 32.4× speedup over HPCGraph for graph LJ, WE, FL, and RD, respectively. Therefore, iSPAN achieves significant improvement for the distributed SCC detection.

We present the details of the execution and communication time breakdown of the distributed iSPAN. Figure 16 presents the breakdown of running with 32 nodes on the 16 graphs. One can see that, the large SCC computation dominates most graphs with, on average, 48.5% of the total time across all the datasets. The communication time during computing the large SCC takes the largest communication cost with 12.5% on average.

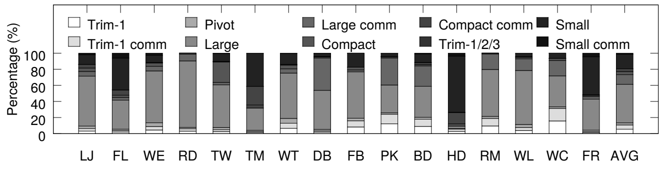


Fig. 16: Execution time breakdown of the distributed iSPAN.

VIII. Related Work

This section discusses the related work landscape of iSPAN from three categories, namely, DFS-based, FW-BW-based and the other remaining endeavors.

DFS-based SCC detection. SCC detection originates from a DFS-based work [60]. Another work [2] conducts two DFS computations, the first on the original graph, the other on the transposed graph to improve the parallelism. A recent DFS work devises an on-the-fly SCC detection method [13]. DFS can be parallelized, but with a number of drawbacks [20], [1].

A closely related DFS work is a serial semi-external SCC detection method [66], which uses spanning tree with weak order DFS to reduce the edge number for I/O efficiency. In contrast, iSPAN completely removes the constraint of the order, delivering very fast construction of spanning trees. Specifically, iSPAN devises a BFS-based parallel method, which can be orders of magnitude faster. For example, [66] takes about 20s to process a graph with 34M edges, while iSPAN needs only 54ms for a similar size graph (FL).

FW-BW-based SCC detection. The FW-BW algorithm paves the road for parallel SCC detection. Fleischer et al. [23] first introduces FW-BW algorithm, divide-and-conquer strong components method, to improve the parallelism. Later, Mclendon et al. [45] extends FW-BW algorithm by adding trim. Recently, a BFS-based FW-BW algorithm [28] designs a three step FW-BW-Trim approach for small-world graphs. Multistep [55] goes further by combining the power of FW-BW, color propagation, and Tarjan's DFS to detect SCC.

Both [28] and [55] follow the original FW-BW algorithm to detect the large SCC. For detecting small SCCs, [28] introduces trim-2 and WCC-based FW-BW algorithm, while [55] uses color propagation algorithm. Different from them, iSPAN improves the FW-BW algorithm by using the spanning trees, and design a new relaxed synchronization technique. Combined with trim-3, iSPAN is able to deliver about 4× speedup.

Others. Color propagation algorithm is also proposed to detect SCC in parallel [49], while it suffers from load imbalance caused by large components. iSPAN is also related to the graph traversal and connected component detection works [6], [63], [19], [61], [54], [59], [24], [22], [26], [25], [33], [34], [40]. We will explore iSPAN in future works from three directions, better distributed scalability [14], [5], [29], [57], [44], [50], [3], [12], Graphics Processing Units (GPUs) [9], [41], and more applications [30], [17], [31], [58], [43], [10], [38], [64].

IX. Conclusion

This work designs iSPAN, a new spanning tree-based SCC detection method that leverages a novel fast spanning tree construction method by judiciously applying synchronous, asynchronous, and relaxed synchronization strategy to direction-optimizing BFS to achieve better workload balance and reduced level synchronization. As a result, iSPAN can significantly outperform state-of-the-art DFS and BFS-based methods by average 18× and 4×, respectively. iSPAN is able to achieve 1.7× speedup on three large graphs (upto billion vertex) and upto 10.7× speedup when scaling to 32 nodes.

Acknowledgment

We thank the anonymous reviewers for their helpful suggestions. This work was supported in part by National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774.

References

[1] U. A. Acar, A. Charguéraud, and M. Rainey. A work-efficient algorithm for parallel unordered -first search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 67. ACM, 2015.

[2] A. V. Aho, J. D. Ullman, and J. E. Hopcroft. Data structures and algorithms. 1983.

[3] M. J. Anderson, N. Sundaram, N. Satish, M. M. A. Patwary, T. L. Willke, and P. Dubey. Graphpad: Optimized graph primitives for parallel and distributed platforms. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 313–322. IEEE, 2016.

[4] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 22–31. IEEE, 2016.

[5] A. Azad and A. Buluc. Towards a graphblas library in chapel. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 1095–1104. IEEE, 2017.

[6] L. Barrière and et al. Connected graph searching. *Information and Computation*, 2012.

[7] S. Beamer, K. Asanovic, and D. Patterson. Direction-optimizing breadth-first search. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10. IEEE, 2012.

[8] S. Beamer, A. Buluc, K. Asanovic, and D. Patterson. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1618–1627. IEEE, 2013.

[9] O. Beaumont, B. Becker, A. Deflumere, L. Eyraud-Dubois, T. Lambert, and A. Lastovetsky. Recent advances in matrix partitioning for parallel computing on heterogeneous platforms. 2017.

[10] J. Berry, M. Oster, C. A. Phillips, S. Plimpton, and T. M. Shead. Maintaining connected components for infinite graph streams. In *Proceedings of the 2nd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 95–102. ACM, 2013.

[11] M. Besta, F. Marending, E. Solomonik, and T. Hoefler. Slimsell: A vectorizable graph representation for breadth-first search. In *IPDPS'17*.

[12] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 93–104. ACM, 2017.

[13] V. Bloemen, A. Laarman, and J. van de Pol. Multi-core on-the-fly scc decomposition. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 8. ACM, 2016.

[14] A. Buluc, S. Beamer, K. Madduri, K. Asanovic, and D. Patterson. Distributed-memory breadth-first search on massive graphs. *arXiv preprint arXiv:1705.04590*, 2017.

[15] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures (SPAA)*, pages 233–244. ACM, 2009.

[16] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 65. ACM, 2011.

[17] J. Cao, Q. Li, Y. Ji, Y. He, and D. Guo. Detection of forwarding-based malicious urls in online social networks. *International Journal of Parallel Programming*, 44(1):163–180, 2016.

[18] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 193–204. ACM, 2013.

[19] G. Cong and K. Makarychev. Optimizing large-scale graph analysis on multithreaded, multicore platforms. In *Proc. of IPDPS'12*, 2012.

[20] J. A. Edwards and U. Vishkin. Better speedups using simpler parallel programming for graph connectivity and biconnectivity. In *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 103–114. ACM, 2012.

[21] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. *Proceedings of the VLDB Endowment*, 3(1-2):1161–1172, 2010.

[22] J. S. Firoz, M. Zalewski, and A. Lumsdaine. A scalable distance-1 vertex coloring algorithm for power-law graphs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 391–392. ACM, 2018.

[23] L. K. Fleischer, B. Hendrickson, and A. Pınar. On identifying strongly connected components in parallel. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 505–511. Springer, 2000.

[24] P. Flick, C. Jain, T. Pan, and S. Aluru. A parallel connectivity algorithm for de bruijn graphs in metagenomic applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 15. ACM, 2015.

[25] O. Green, M. Dukhan, and R. Vuduc. Branch-avoiding graph algorithms. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, pages 212–223. ACM, 2015.

[26] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*, pages 166–177. ACM, 2014.

[27] R. Hojati, R. K. Brayton, and R. P. Kurshan. Bdd-based debugging of designs using language containment and fair ctl. In *International Conference on Computer Aided Verification (CAV)*, pages 41–58. Springer, 1993.

[28] S. Hong, N. C. Rodia, and K. Olukotun. On fast parallel detection of strongly connected components (scc) in small-world graphs.

[29] J. Iverson, C. Kamath, and G. Karypis. Evaluation of connected-component labeling algorithms for distributed-memory systems. *Parallel Computing*, 44:53–68, 2015.

[30] Y. Ji, Y. He, X. Jiang, J. Cao, and Q. Li. Combating the evasion mechanisms of social bots. *computers & security*, 58:230–249, 2016.

[31] Y. Ji, Y. He, D. Zhu, Q. Li, and D. Guo. A mulitiprocess mechanism of evading behavior-based bot detection approaches. In *ISPEC*.

[32] K. I. Karantasis, A. Lenharth, D. Nguyen, M. J. Garzarán, and K. Pingali. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 921–932. IEEE Press, 2014.

[33] P. Kumar and H. H. Huang. G-store: high-performance graph store for trillion-edge processing. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*, pages 830–841. IEEE, 2016.

[34] P. Kumar and H. H. Huang. Falcon: Scaling io performance in multissd volumes. In *Usenix ATC*, 2017.

[35] J. Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web (WWW)*, pages 1343–1350. ACM, 2013.

[36] D. LaSalle, M. M. A. Patwary, N. Satish, N. Sundaram, P. Dubey, and G. Karypis. Improving graph partitioning for modern graphs and architectures. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, page 14. ACM, 2015.

[37] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[38] C. Liu, M. Xu, and S. Subramaniam. A reconfigurable high-performance optical data center architecture. In *Global Communications Conference (GLOBECOM), 2016 IEEE*, pages 1–6. IEEE, 2016.

[39] H. Liu and H. H. Huang. Enterprise: breadth-first graph traversal on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 68. ACM, 2015.

[40] H. Liu and H. H. Huang. Graphene: Fine-grained io management for graph computing. In *FAST*, 2017.

[41] H. Liu, H. H. Huang, and Y. Hu. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 403–416. ACM, 2016.

[42] X. Luo, J. Gao, C. Zhou, and J. X. Yu. Uniwalk: Unidirectional random walk based scalable simrank computation over large graph. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 325–336. IEEE, 2017.

[43] A. Magner, A. Grama, J. Sreedharan, and W. Szpankowski. Recovery of vertex orderings in dynamic graphs. In *Information Theory (ISIT), 2017 IEEE International Symposium on*, pages 1563–1567. IEEE, 2017.

[44] S. Maleki, D. Nguyen, A. Lenharth, M. Garzarán, D. Padua, and K. Pingali. Dsmr: a shared and distributed memory algorithm for single-source shortest path problem. *ACM SIGPLAN Notices*, 51(8):39, 2016.

[45] W. Mclendon Iii, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger. Finding strongly connected components in distributed graphs. *Journal of Parallel and Distributed Computing*, 65(8):901–910, 2005.

[46] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *ACM SIGPLAN Notices*, volume 47, pages 117–128. ACM, 2012.

[47] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Growth of the Flickr social network. In *Proc. Workshop on Online Social Networks*, pages 25–30, 2008.

[48] M. E. Newman. The structure and function of complex networks. *SIAM review*, 2003.

[49] S. M. Orzan. On distributed verification and verified distribution. *Ph.D. dissertation*, 2004.

[50] K. Raffenetti, A. Amer, L. Oden, C. Archer, W. Bland, H. Fujita, Y. Guo, T. Janjusic, D. Durnov, M. Blocksome, et al. Why is mpi so slow?: analyzing the fundamental limits in implementing mpi-3.1. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 62. ACM, 2017.

[51] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[52] D. Sacharidis, S. Papadopoulos, and D. Papadias. Topologically sorted skylines for partially ordered domains. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 1072–1083. IEEE, 2009.

[53] N. Satish, C. Kim, J. Chhugani, and P. Dubey. Large-scale energy-efficient graph traversal: a path to efficient data-intensive supercomputing. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11. IEEE, 2012.

[54] Y. Shiloach and U. Vishkin. An o (logn) parallel connectivity algorithm. *Journal of Algorithms*, 1982.

[55] G. M. Slota, S. Rajamanickam, and K. Madduri. Bfs and coloring-based parallel algorithms for strongly connected components and related problems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 550–559. IEEE, 2014.

[56] G. M. Slota, S. Rajamanickam, and K. Madduri. A case study of complex graph analysis in distributed memory: Implementation and optimization. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 293–302. IEEE, 2016.

[57] F. Song, H. Ltaief, B. Hadri, and J. Dongarra. Scalable tile communication-avoiding qr factorization on multicore cluster systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11. IEEE, 2010.

[58] S. Srinivasan and et al. Application of graph sparsification in developing parallel algorithms for updating connected components. In *IPDPSW'16*.

[59] G. Tan, D. Fan, J. Zhang, A. Russo, and G. R. Gao. Experience on optimizing irregular computation for memory hierarchy in manycore architecture. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 279–280. ACM, 2008.

[60] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[61] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 1985.

[62] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, pages 1813–1828. ACM, 2016.

[63] B. West and et al. A hybrid approach to processing big data graphs on memory-restricted systems. In *Proc. of IPDPS'15*, 2015.

[64] M. Xu, C. Liu, and S. Subramaniam. Podca: A passive optical data center architecture. In *Communications (ICC), 2016 IEEE International Conference on*, pages 1–6. IEEE, 2016.

[65] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment*, 7(14):1821–1832, 2014.

[66] Z. Zhang, J. X. Yu, L. Qin, L. Chang, and X. Lin. I/o efficient: computing sccs in massive graphs. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 181–192. ACM, 2013.

[67] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: a total order approach. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 1323–1334. ACM, 2014.