Parallel Strong Connectivity Based on Faster Reachability

LETONG WANG, University of California, Riverside, USA
XIAOJUN DONG, University of California, Riverside, USA
YAN GU, University of California, Riverside, USA
YIHAN SUN, University of California, Riverside, USA

Computing strongly connected components (SCC) is among the most fundamental problems in graph analytics. Given the large size of today's real-world graphs, parallel SCC implementation is increasingly important. SCC is challenging in the parallel setting and is particularly hard on large-diameter graphs. Many existing parallel SCC implementations can be even slower than Tarjan's sequential algorithm on large-diameter graphs.

To tackle this challenge, we propose an efficient parallel SCC implementation using a new parallel reachability approach. Our solution is based on a novel idea referred to as vertical granularity control (VGC). It breaks the synchronization barriers to increase parallelism and hide scheduling overhead. To use VGC in our SCC algorithm, we also design an efficient data structure called the *parallel hash bag*. It uses parallel dynamic resizing to avoid redundant work in maintaining frontiers (vertices processed in a round).

We implement the parallel SCC algorithm by Blelloch et al. (J. ACM, 2020) using our new parallel reachability approach. We compare our implementation to the state-of-the-art systems, including GBBS, iSpan, Multi-step, and our highly optimized Tarjan's (sequential) algorithm, on 18 graphs, including social, web, k-NN, and lattice graphs. On a machine with 96 cores, our implementation is the fastest on 16 out of 18 graphs. On average (geometric means) over all graphs, our SCC is 6.0× faster than the best previous parallel code (GBBS), 12.8× faster than Tarjan's sequential algorithms, and 2.7× faster than the *best existing implementation on each graph*.

We believe that our techniques are of independent interest. We also apply our parallel hash bag and VGC scheme to other graph problems, including connectivity and least-element lists (LE-lists). Our implementations improve the performance of the state-of-the-art parallel implementations for these two problems.

CCS Concepts: • **Theory of computation → Shared memory algorithms**; **Graph algorithms analysis**.

Additional Key Words and Phrases: Parallel Algorithms, Graph Algorithms, Strong Connectivity, Reachability, Graph Analytics

ACM Reference Format:
Letong Wang, Xiaojun Dong, Yan Gu, and Yihan Sun. 2023. Parallel Strong Connectivity Based on Faster Reachability. *Proc. ACM Manag. Data* 1, 2, Article 114 (June 2023), 29 pages. https://doi.org/10.1145/3589259

1 INTRODUCTION

Computing strongly connected components (SCCs) is an important problem in graph analytics. Given a directed graph $G = (V, E)$, we denote $n = |V|$ and $m = |E|$. We use D as the diameter of G. For simplicity, we assume $m \geq n$, but all algorithms in this paper work for any n and m. For two vertices $v, u \in V$, we use $u \rightsquigarrow v$ to denote that a path exists from u to v. Two vertices v and u, are ***strongly connected*** if $u \rightsquigarrow v$ and $v \rightsquigarrow u$. An SCC is a maximal set of vertices on the graph that are strongly connected. The SCC problem is to compute a mapping from each vertex to a unique label for

Authors' addresses: Letong Wang, University of California, Riverside, Riverside, California, USA, lwang323@ucr.edu; Xiaojun Dong, University of California, Riverside, USA, xdong038@ucr.edu; Yan Gu, University of California, Riverside, USA, ygu@cs.ucr.edu; Yihan Sun, University of California, Riverside, USA, yihans@cs.ucr.edu.

2836-6573/2023/6-ART114
https://doi.org/10.1145/3589259

its strongly connected component. Computing SCC is useful in many applications inside and outside computer science, and is widely used in database and data management applications [5, 83, 104]. For example, analyzing SCC on social networks can identify communities [77] or estimate influence propagation [81]. Applying SCC on the k-NN graph (each point links to its k-nearest neighbors) for spatial data points is widely used in unsupervised learning [68, 91, 92]. Many other fundamental problems in CS use SCC as an important primitive, such as graph matching [42], topological sort [26, 87], graph contraction [31], and code analysis [86]. SCC is also widely used on various types of graphs in other disciplines, e.g., lattice graphs [32] in material science, on E. coli network [47] in computer biology [78], on food web analysis in ecology systems [7], and more [79].

Sequentially, Kosaraju's algorithm [6] and Tarjan's algorithm [101] find all SCCs in a graph in $O(m)$ work (number of operations). However, sequential algorithms can be slow to process today's large real-world graphs. For example, on the Hyperlink12 [75] graph with billions of edges, Tarjan's SCC algorithm takes more than half an hour (see Tab. 3) to finish. Hence, it is of great importance to seek *parallel SCC solutions*. Although SCC is also widely studied in parallel (see the literature review in Sec. 7), most existing algorithms are optimized based on two assumptions of the input graph: 1) with a low diameter and/or 2) with one large SCC. Many algorithms (e.g., iSpan [58] and Multi-step [98]) first use breadth-first searches (BFS) to identify the largest SCC and then run a subsequent *coloring* phase to find all other SCCs with $O(m'D)$ work. Here m' is the number of edges not in the largest SCC. When either of the assumptions is not satisfied, these algorithms will incur significantly extra work than the $O(m)$ sequential algorithms, which cannot be compensated by parallelism. Another solution in the GBBS library [35, 37] implements the BGSS algorithm [18]. BGSS uses $\log n$ rounds of multi-reachability searches, each using a subset of source vertices S to find all pairs of vertices (v, s), where $s \in S$ and $s \rightsquigarrow v$. BGSS has $O(m \log n)$ work that is independent of the two assumptions. However, the GBBS implementation uses parallel BFS for reachability searches, which processes vertices with the same distance (referred to as a ***frontier***) in each round. This approach incurs $O(D)$ rounds of global synchronization. As each synchronization is costly, this implementation incurs high scheduling overhead and can be slow when D is large.

Unfortunately, many applications of SCC have sparse input graphs with a large diameter D. For example, the k-NN graphs used in unsupervised learning [68, 91, 92] and lattice graphs used in computational chemistry [32] can have $D = \Theta(\sqrt{n})$ (see Tab. 2). We tested existing SCC algorithms on graphs with both small diameters (e.g., social networks) and large diameters (e.g., k-NN and lattice graphs). Fig. 1 shows that existing SCC algorithms work well for certain social networks but perform badly on other graphs. On average, for k-NN and lattice graphs, all existing parallel SCC algorithms on a 96-core machine are slower than the sequential Tarjan's algorithm.

In this paper, we propose ***an efficient SCC implementation with high parallelism on a wide range of graphs***. We also use the BGSS algorithm to bound the work. The core of our idea is to improve parallelism by avoiding $O(D)$ rounds of synchronization in reachability searches and thus reducing the scheduling overhead. To do this, we propose a novel idea referred to as the ***vertical granularity control*** (VGC) optimization.

The high-level idea of VGC is to break the synchronization barriers and increase parallelism. Particularly, for the reachability queries, unlike parallel BFS that only visits the neighbors of the vertices in the frontier, we want to visit a much larger set of vertices that can be multiple hops away. This is achieved by a "local search" scheme that allows each vertex in the frontier to visit more than direct neighbors in one round. We will discuss more details of VGC and local search in Sec. 3.1 and 3.2. This approach saves most synchronization rounds and improves the performance, especially on large-diameter graphs.

We note that the major technical difficulty in VGC and local search is to handle the non-determinism in generating the next frontier—all vertices in the frontier explore their proximity

	Social			KNN								
	LJ	TW	MEAN	HH5	CH5	GL2	GL5	GL10	GL15	GL20	COS5	MEAN
Ours	75.8	317	155	2.16	0.77	5.67	5.58	6.24	5.42	5.60	58.6	5.26
GBBS	24.5	185	67.3	0.11	0.05	1.13	0.46	0.76	0.83	0.92	15.8	0.63
iSpan	~~58.5~~	c	-	0.57	0.20	t	t	0.26	0.39	0.49	t	0.36
MultiStep	20.7	54.4	33.6	0.20	0.02	0.41	0.25	1.30	1.07	1.08	3.29	0.46
Seq	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

	Web				Lattice					Overall
	SD	CW	HL14	HL12	MEAN	SQR	REC	SQR'	REC'	MEAN	MEAN
Ours	52.7	33.4	30.1	19.1	31.7	26.8	13.5	5.00	3.75	9.08	12.9
GBBS	19.7	14.6	9.22	5.05	10.8	1.39	0.41	1.45	0.60	0.84	2.13
iSpan	~~21.7~~	n	n	n	-	~~3.47~~	~~1.32~~	0.26	~~0.70~~	0.96	1.18
MultiStep	55.8	n	n	n	-	1.23	0.30	2.16	0.92	0.93	1.35
Seq	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

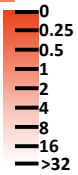

Fig. 1. The heatmap of relative speedup for parallel SCC algorithms over the sequential algorithm using 96 cores (192 hyperthreads). Larger/white background means better. "Seq"= Tarjan's algorithm [101]. The numbers indicate how many times a parallel algorithm is faster than Tarjan's sequential algorithm (< 1 means slower). The baseline algorithms [35, 58, 98] are introduced in Sec. 6. "t"= timeout. "c"= crash. "n"= no support. Strikethrough numbers mean a wrong number of SCCs reported. Complete results are in Tab. 3.

in parallel, in a random order decided by the runtime scheduling. This disables the "edge-revisit" approach in existing BFS algorithms [35], which first process the frontier to count the size of the next frontier and pre-allocate memory and then revisit the frontier to output vertices to the next frontier. We present more details about this challenge in Sec. 2 and 3.3. To maintain the frontier more efficiently (and correctly) in VGC, we propose a new data structure called the ***parallel hash bag***. It supports efficient insertion and extract-all operations for an unordered set structure and dynamic resizing in an efficient manner. Our parallel hash bag is similar to a resizable hash table but avoids copying on resizing. We maintain a multiple-level structure in exponentially growing sizes. We use atomic operations to enable concurrent insertions and a sampling scheme to support resizing. Our hash bag is theoretically-efficient and fast in practice. We describe the details of the hash bag in Sec. 3.3.

Using these techniques, our new SCC approach achieves good performance on various graphs (see Fig. 1), all widely used in real-world, including the *largest publicly available graph (HL12)* with 3.6 billion vertices and 128 billion edges. Among all implementations, our SCC achieves the best performance on 16 out of 18 graphs. As expected, our SCC performs particularly well on large-diameter graphs (k-NN and lattice graphs). Compared to *the fastest running time among existing parallel algorithms on each graph*, our implementation is 2.3–12× faster on large-diameter graphs, and up to 3.7× faster on low-diameter graphs. Our results show that the good performance of our SCC mainly comes from VGC (using 3–200× fewer rounds than BFS, see Fig. 10) and hash bags (see Fig. 9). Our code is publicly available at GitHub [106].

We believe that our proposed techniques are general and can be applied to many graph algorithms. As proofs-of-concept, we apply the proposed techniques to two more algorithms: connected components (CC) and least-element lists (LE-lists). Using our new techniques, our solutions outperform the state-of-the-art algorithms by up to 3.2× on CC and 10× on LE-lists. We overview these problems in Sec. 5 and present the experimental results in Sec. 6.4.

The full version of this paper is available online [107]. We summarize our contributions as follows.

(1) Two general techniques (vertical granularity control and parallel hash bag) to optimize the performance of graph traversal.

(2) Fast implementations on SCC, CC, and LE-lists using the proposed techniques. Our SCC algorithm greatly outperforms existing implementations.

(3) In-depth experimental evaluation of the problems and the optimizations in this paper across multiple types of graphs.

2 PRELIMINARIES

Notations and Computational Model. We use $\tilde{O}(f(n))$ to denote $O(f(n) \cdot polylog(n))$. We use $O(f(n))$ ***with high probability*** (*whp*) in n to demote $O(kf(n))$ with probability at least $1 - n^{-k}$ for $k \geq 1$. We omit "in n" with clear context. For a graph $G = (V, E)$, we denote $n = |V|$, $m = |E|$, and D as the diameter of the graph.

This paper uses work-span (or work-depth) model for nested parallelism based on fork-join and binary forking [17, 31] for theoretical analysis. We assume a set of threads that share the memory. Each thread acts like a sequential RAM, plus a `fork` instruction to create two new child threads running in parallel. When both children finish, the parent thread continues, implying a synchronization. The computation can be executed by a randomized work-stealing scheduler in practice. We assume unit-cost atomic operation `compare_and_swap`(p, v_{old}, v_{new}) (or `CAS`), which atomically reads the memory location pointed to by p, and write value v_{new} to it if the current value is v_{old}. It returns *true* if successful and *false* otherwise. An algorithm's ***work*** is the total number of instructions and the ***span*** (depth) is the length of the longest sequence of dependent instructions in the computation. Our algorithm uses the scheduler in Parlaylib [14] to support fork-join parallelism.

Reachability Search. Given a directed graph $G = (V, E)$ and a set of source vertices $S \subseteq V$, a (forward) reachability search finds the set of reachable pairs from any source $\{(s, t) : s \in S, t \in V, s \rightsquigarrow t\}$. We say a vertex v is ***reachable from*** s if $s \rightsquigarrow v$, and ***reachable to*** s if $v \rightsquigarrow s$. We define *backward reachability search* as searching in the reverse direction of the edges (i.e., on the transpose graph G^T). We use ***single-reachability search*** to refer to the case where $|S| = 1$, and use ***multi-reachability search*** otherwise. Fig. 2 illustrates single- and multi-reachability searches.

Many SCC algorithms are based on reachability searches [10, 18, 30, 58, 89, 98, 103]. The key idea is that 1) all vertices reachable both from and to a vertex v are in the same SCC, and 2) two vertices are in the same SCC iff. they have the same reachability information from *any source*.

Therefore, an edge (u, v) is not in any SCC (and can be removed) if u and v have different reachability information from *any source*. We call such edges ***cross edges***. We present an example in Fig. 2. We use a 2×2 lattice as the reachability information of a vertex (called the ***signature*** of it). An edge connecting two vertices with different signatures is a cross edge, and can be removed.

Some theoretical results show that a single-reachability search can be done in $\tilde{O}(m)$ work and $o(n)$ span [43, 56]. However, the hidden terms in the bounds are large, and these algorithms are unlikely to be practical. In practice, BFS with $O(m)$ work is used for reachability searches [35, 98]. It works well on low-diameter graphs but performs poorly on large-diameter graphs. We review parallel BFS algorithms for reachability search later in this section.

The BGSS Algorithm. Our parallel SCC solution uses the BGSS SCC algorithm [18] based on reachability searches, as shown in Alg. 1. To achieve good parallelism while bounding the work, the BGSS algorithm uses $\log n$ batches of reachability searches. The algorithm first randomly permutes the vertices and groups them into batches of sizes 1, 2, 4, 8, ... in a prefix-doubling manner (the multiplier is not necessary to be 2, but can be any constant $\beta > 1$). In the i-th round, the algorithm uses batch i with 2^{i-1} vertices as the sources to run (forward and backward) multi-reachability searches, marks SCCs, and removes cross edges. In this way, the BGSS algorithm takes $O(W_R(n, m) \log n)$ expected work and $O((D_R(n, m) + \log n) \log n)$ span *whp*, where $W_R(n, m)$ and $D_R(n, m)$ are the work, and span for a reachability search on a graph with n vertices and m edges. The BGSS algorithm was implemented by Dhulipala et al. as part of the GBBS library [35, 37], which

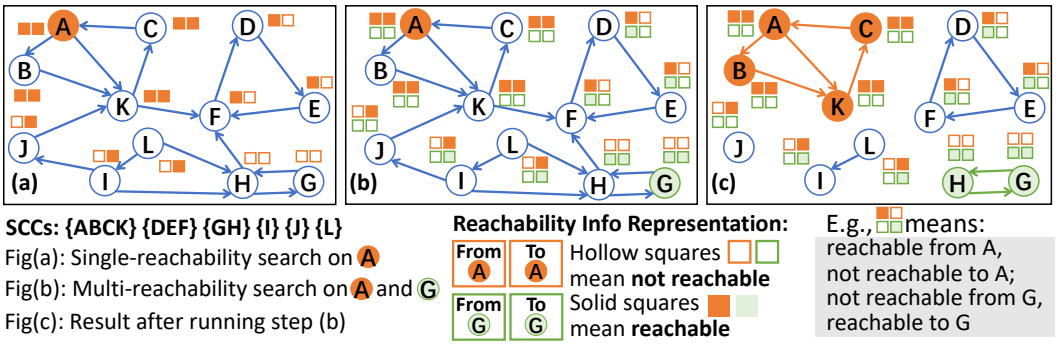

Fig. 2. Example of single- and multi-reachability searches and reachability-based SCC algorithms. The SCCs are $\{A, B, C, K\}$, $\{D, E, F\}$, $\{G, H\}$, $\{I\}$, $\{J\}$, $\{L\}$. (a). Single-reachability search results on A. (b). Multi-reachability search results on A and G. The reachability information (or the *signature*) of a vertex is shown as a 2×2 lattice. (c). After running multi-reachability searches from A and G, we can identify two SCCs: all orange vertices (reachable from and to A) and all green vertices (reachable from and to G). Two vertices are in the same SCC only if they have the same signature. Thus, all cross edges (endpoints with different signatures) are removed. Later steps in the algorithm only need to process the blue vertices and edges in (c).

Algorithm 1: The BGSS algorithm for parallel SCC [18]

Input: A directed graph $G = (V, E)$
Output: The component label $L[\cdot]$ of each vertex.

$L \leftarrow \{-1, \ldots, -1\}$
Partition V into $\log n$ batches $P_{1..\log n}$, where $|P_i| = 2^{i-1}$
$V' \leftarrow V$
for $i \leftarrow 1, \ldots, \log n$ **do**
  $\mathcal{F} \leftarrow \{v \in P_i \cap V'\}$ ▷ Initial frontier
  // MultiReach skips an edge (u, v) if $L(u) \neq L(v)$
  $L_{out} \leftarrow \textsc{MultiReach}(G, L, \mathcal{F})$ ▷ Forward reachable pairs
  $L_{in} \leftarrow \textsc{MultiReach}(G^T, L, \mathcal{F})$ ▷ Backward reachable pairs
  parallel_for_each $i \in V'$ **do**
    $R_1 \leftarrow \{v \mid (v, i) \in L_{in}\}$ ▷ Reachable to i
    $R_2 \leftarrow \{v \mid (v, i) \in L_{out}\}$ ▷ Reachable from i
    if $R_1 \cap R_2 \neq \emptyset$ **then** $L[i] \leftarrow \max_{v \in R_1 \cap R_2} v$ **else** $L[i] \leftarrow hash(L[i], R_1, R_2)$
  $V' \leftarrow V' \setminus \{i \mid \exists (v, i) \in L_{in} \cap L_{out}\}$
return L

uses (multi-)BFS for reachability searches (see more details later). There are two SCC algorithms in GBBS, and we refer to the *RandomGreedy* version, as it is faster in most of our tests.

Parallel BFS. We briefly review parallel BFS, because it is used in previous work for reachability search, and some of the concepts are also used in our algorithm. There are many BFS algorithms (e.g., [13, 80, 113]). We review the version in Ligra [95], as it is widely-used in other graph libraries [34–36], and more importantly, later extended to ***multi-BFS*** that can be used in multi-reachabilty searches needed by our SCC algorithm. We start with BFS from a *single* source $s \in V$ (high-level idea in Alg. 2). The algorithm maintains a ***frontier*** of vertices to explore in each round, starting from the source, and finishes in D rounds. In round i, the algorithm *processes* (visits their neighbors) the current frontier $\mathcal{F}_i$, and puts all their (unvisited) neighbors to the next frontier $\mathcal{F}_{i+1}$. If multiple vertices in $\mathcal{F}_i$ attempt to add the same vertex to $\mathcal{F}_{i+1}$, a `compare_and_swap` is used

Algorithm 2: Framework of Parallel BFS

Input: A directed graph $G = (V, E)$ and a source $s \in V$

1 $\mathcal{F}_0 = \{s\}$
2 $i \leftarrow 0$
3 **while** $\mathcal{F}_i \neq \emptyset$ **do**
4 Process all $v \in \mathcal{F}_i$ and their edges in parallel; put all their unvisited neighbors (but avoid duplicates) to $\mathcal{F}_{i+1}$
5 $i \leftarrow i + 1$

to guarantee that only one will *succeed*. In existing libraries [35, 95], processing $\mathcal{F}_i$ involves visiting all incident edges of $\mathcal{F}_i$ twice. The first visit decides the *successfully* visited neighbors for each $v \in \mathcal{F}_i$, and assigns the right size of memory in $\mathcal{F}_{i+1}$ for each of them. The second visit lets each $v \in \mathcal{F}_i$ write these neighbors to $\mathcal{F}_{i+1}$. We call this the ***edge-revisit*** scheme, and we will show how our *hash bag* avoids the second visit and improve the performance.

This idea has been extended to *multiple* sources $S \subseteq V$ with two changes [35]. First, a parallel hash table T [96] is used to maintain the ***reachability pairs*** (v, s) where $v \in V$ and $s \in S$. Second, for each $v \in \mathcal{F}_i$ and its successfully visited neighbor u, we find all pairs (v, s) from the hash table T, and add (u, s) to T if $(u, s) \notin T$. The number of reachability pairs generated by the BGSS algorithm is proved to be $O(n \log n)$ *whp* [15].

One challenge of using parallel BFS for reachability queries is the large cost to create and synchronize threads between rounds, which is especially expensive for large-diameter graphs (more rounds needed). In this paper, we will show how our new techniques reduces the scheduling overhead to achieve better parallelism.

3 FAST PARALLEL ALGORITHM FOR REACHABILITY

To implement an efficient parallel SCC algorithm, we use the BGSS algorithm to bound the work, and present novel ideas for fast reachability search to enable high parallelism. In this section, we present two main techniques in this paper: the ***vertical granularity control (VGC)*** with the ***hash bag*** data structure. Our VGC optimization is designed to address the challenge of low parallelism in computing SCC on sparse and large-diameter graphs. The goal is to enable a proper size for each parallel task to hide the scheduling overhead. Our idea is to let each vertex search out multiple hops in each parallel task, and thus the number of needed rounds in reachability searches is reduced. The details of the local search is in Sec. 3.1. While the high-level idea sounds simple, this brings up the challenge of non-determinism—each vertex may explore multiple hops and the explored neighborhood depend on runtime scheduling, which results in some complication in generating the frontier by the edge-revisit scheme (see Line 13). Therefore, we propose a data structure called the ***parallel hash bag***, to maintain the frontier more efficiently. Our hash bag is theoretically-efficient and fast in practice, and more details are given in Sec. 3.3.

Combining both techniques, we achieve fast single- and multi-reachability searches. Plugging them into Lines 6 and 7 in Alg. 1 gives a high-performance parallel SCC algorithm. We believe that these techniques are general and useful in many graph algorithms. As proofs-of-concept, in Sec. 5, we apply the proposed ideas to two more algorithms: connected components (CC) and least-element lists (LE-lists), and show new algorithms with better performance. We present notation and parameters used in this paper in Tab. 1.

3.1 Vertical Granularity Control

In this section, we present our vertical granularity control (VGC) optimization. As mentioned, previous work [35] uses parallel BFS for reachability searches, where the number of rounds is

General notations and vertical granularity control:	
n	The number of vertices in a graph.
m	The number of edges in a graph.
P	The number of processors available.
β	The multiplier of prefix-doubling for SCC, LDD, and LE-List algorithms. Usually $\beta \in (1, 2]$. We use $\beta = 1.5$ in our system.
τ	The threshold for vertical granularity control, which is the upper bound of visited neighborhood size per node. We use $\tau = 512$ as the default value.
Hash Bag:	
λ	The first chunk size of hash bag. Theoretically, $\lambda = \Omega((P + \log n) \log n)$. We set $\lambda = 2^{10}$.
σ	The threshold of the number of samples to trigger hash bag resizing. Theoretically, $\sigma = \Omega(\log n)$. We use $\sigma = 50$ in our system.

Table 1. Notations used in this paper.

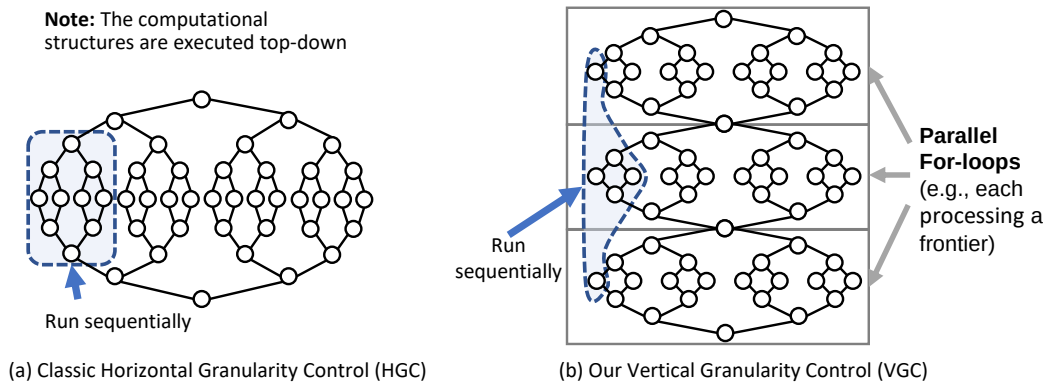

Fig. 3. An illustration of (a) the classic horizontal granularity control (HGC) and (b) our new vertical granularity control (VGC). We consider work-stealing schedulers. (a): The computation structure of parallel-for or divide-and-conquer algorithms. HGC groups computation in the same level and run sequentially to reduce scheduling overhead. (b): The computation structure of several rounds of parallel-for, each starting from one thread and forking parallel tasks in a nested fashion. Each of them represents a round (processing a frontier) in a parallel graph algorithm. VGC groups computations in different rounds, and run sequentially to reduce scheduling overhead. It breaks the synchronization points and reduces the number of rounds.

proportional to the diameter of the graph. On many real-world sparse graphs with large diameters, both the frontier size and the average degree are small, which leads to two challenges. First, every parallel task (roughly processing one vertex in the frontier) is small, and the cost of distributing the tasks to the processor can be much more than the actual computation. Second, the number of rounds is large, resulting in many rounds of distributing and synchronizing threads.

Audiences familiar with parallel programming must know the concept of *granularity control* (aka. coarsening), aiming to avoid the overhead caused by generating unnecessary parallel tasks. For computations with sufficient parallelism, e.g., a parallel for-loop of size $n \gg P$ where P is the number of processors, most existing parallel software (e.g., [2, 14, 65]) will automatically stop recursively creating parallel tasks at a certain subproblem size and switch to a sequential execution (see Fig. 3(a)) to hide the scheduling overhead. We refer to this approach as the horizontal granularity control (HCG) since it merges the computation on the same level (sibling leaf nodes in Fig. 3(a)).

Unfortunately, this idea does not directly apply to reachability searches or similar problems on sparse graphs. HGC is used when there is excessive parallelism to saturate all processors.

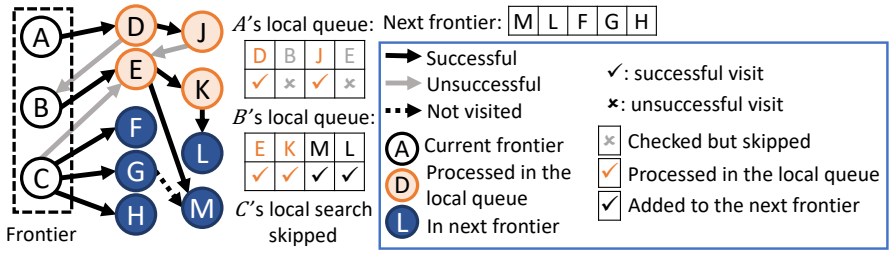

Fig. 4. A possible execution of a local search. The initial frontier is $\{A, B, C\}$ and $\tau = 4$. A successfully visits one neighbor D. As its local queue is not full, it then visits D's neighbor B (skipped) and J. J visits E but fails. No more vertex are left in A's local queue. B visits E, and E visits K and M. Then, K visits L and add it to the queue. Now B's queue is full. The unfinished vertices M and L will be flushed to the next frontier. C has four ($\geq \tau$) neighbors, so we directly check all neighbors and add successful ones (F, G, and H) to the next frontier. We process 2-hop neighbors from the frontier in one round.

However, when processing sparse graphs, the issue becomes that we have insufficient computation (frontiers with small sizes) to saturate all processors for good parallelism, and grouping sibling (horizontal) computation in the same round only makes it worse. To tackle this challenge, we propose a novel and very different approach, referred to as *vertical granularity control* (VGC). The high-level idea is still to increase each task size to hide the scheduling overhead, but we merge the computation across *different levels* to acquire more work and saturate all processors in each round (an example in Fig. 3(b)). In this way, we break the synchronization points and reduce scheduling overhead. Note that this also means that VGC is unlikely to be automatic (unlike HGC)—breaking the synchronization structures may significantly change the computation and needs a careful redesign of the algorithm (in our case, we need the new data structure *hash bag* to deal with non-determinism, see Sec. 3.3). In the rest of this section, we will show how to apply VGC to reachability searches and achieve good parallelism. Our motivation is from some theory work for parallel reachability algorithms.

Motivations from the Theory Work and Our Solution. To reduce the number of rounds in BFS-like algorithms, many theoretical results use *shortcuts* [20, 28, 43, 56, 62, 93, 105] to reduce the diameter of the graph. Unfortunately, these approaches can be impractical because they incur high overhead for storing the shortcuts, increased memory footprint, and a significant preprocessing cost (e.g., Fineman's algorithm [43] has $\Omega(m \log^6 n)$ preprocessing work). Hence, these algorithms are unlikely to beat the $O(m)$ Tarjan's sequential algorithm using modern computers with tens to hundreds of processors.

To perform VGC without much overhead, we wish to add shortcuts without explicitly generating them. Instead of shortcutting *every* vertex [20, 93], which can be costly, we only shortcut those in the current frontier, so that a vertex in the frontier can perform some work in the next several rounds "in advance". Without shortcutting, visiting the same vertices may take multiple rounds to finish. We wish to process the shortcuts locally and sequentially to avoid space overhead. For each vertex v being processed, we shortcut it to some (nearest) vertices reachable from v on the fly by a ***local search*** from v, such that we do not need to store the shortcuts. Our local search is similar to the sequential BFS algorithm. We maintain a ***local queue*** starting from v and visit each of its neighbor u. We will set all sources reachable to v as reachable to u. If any of these sources s is new to u, we add (u, s) as a reachable pair, and add u to the tail of the local queue. We then move to process the next vertex in the local queue. This process terminates when we have performed "sufficient work" in the local search, and we discuss how to control the granularity in Sec. 3.2.

We call it the "local" queue since we allocate it in the stack memory that is not visible to other processors. This avoids allocating arrays from the global memory (the heap space), which can be costly (and complicated) in parallel. The local searches from different vertices in the frontier are independent and in parallel with each other. The only information that one search needs is whether a vertex has been visited or not, which can be maintained using a boolean array (for single-reachability) or a parallel hash table (for multi-reachability) that support atomic updates. As such, all searches on the same processor can reuse the same stack space for different local queues.

Using VGC, we can reduce the number of rounds and thus the scheduling overhead since vertices in multiple hops can be visited in one round. Fig. 10 shows that VGC reduces the number of rounds in reachability searches by 3–200× and greatly improves performance.

Although our idea of using (on-the-fly) shortcuts for VGC is intuitive, two technical challenges remain. The first is load balancing—vertices can have various degrees and neighborhood patterns, and one vertex may explore a large neighborhood region sequentially. We discuss the control of granularity (the local search size) in Sec. 3.2. The second is non-determinism. In VGC, each vertex can search out several hops, and the explored region depends on runtime scheduling. Hence, we cannot use the edge-revisit scheme in previous work (Line 13) since the second visit may not perform the same computation as the first one. To tackle this, we propose the *hash bag* data structure to efficiently maintain the frontier.

3.2 Control of Granularity

The goal to control granularity is to make each task large enough and hide the cost of scheduling it. However, we cannot let them be arbitrarily large since they are executed sequentially and may cause load-imbalance. We wish to let all threads perform (roughly) a similar amount of work. In our implementation, we control the number of visited vertices in each local search by a parameter τ, including both successful and unsuccessful ones. This number provide an estimation of the workload for each local search.

In particular, when processing a vertex v in the frontier, we first check the number of v's outgoing neighbors. If it is more than τ, we process all its neighbors in parallel as in the standard way, since we have sufficient work to do and no more shortcuts are needed. Otherwise, we start the local search and maintain a counter t starting from zero. When processing a vertex v in the local queue, we increment t for every neighbor visited, successfully or unsuccessfully. Note that since the local search is performed sequentially, there is no race condition in maintaining the counters. We stop the local search either when the queue becomes empty (all possible vertices have been visited), or when the counter reaches τ (this task is reasonably large). For all remaining vertices in the local queue, we directly add them to the next frontier. Conceptually, we shortcut v to the τ nearest vertices that otherwise may need multiple rounds to reach. We present an illustration in Fig. 4. The desired granularity can be controlled by the parameter τ.

Intuitively, we can choose the parameter τ empirically based on traditional HGC base-case size: usually using base-case size around 1000 operations is sufficient to hide scheduling overheads. We also experimentally study the value of τ in Sec. 6.3. Compared to plain BFS (no VGC used), we noticed that the performance on most graphs (both large- and low-diameter graphs) except three graphs improved using any $1 < \tau \leq 2^{16}$. Overall, the performance is not sensitive in a large parameter space $2^6 \leq \tau \leq 2^{12}$ on almost all graphs. We simply set $\tau = 2^9$ as the default value, which is similar to the typical HGC threshold. One can control granularity using other measures such as the number of generated reachability pairs or successfully visited vertices. We believe that the measure of granularity is independent with the idea of VGC. We plan to explore more criteria to control granularity for VGC in future work.

```
1  struct Hashbag {
2    double α;                                                          // desired load factor
3    data* bag;
4    int *tail, *sample;                                  // tail: tail of chunk i; sample: #samples of chunk i
5    int r = 0;                                                         // current chunk id
6    Hashbag(int n) {                                                   // constructor
7      allocate tail[] and sample[] of size ⌈log (n+λ)/α⌉; initialize sample[] and tail[] to 0;
8      tail[0] = λ;
9      for (i = 1; i< ⌈log (n+λ)/α⌉; i++) tail[i] = tail[i-1]*2;
10     bag = new data[tail[i-1]];}
11   void insert(data k) {
12     r' = r;
13     if (sampled successfully) {
14       // Lines 15–16: fetch_and_add using compare_and_swap
15       t = sample[r'];
16       while (!compare_and_swap(&sample[r'], t, t+1)) {
17       if (sample[r'] too large) {                                    // current chunk is full
18         try_resize(r');
19         return insert(k); } } }
20     i = random position in tail[r'-1]..tail[r']
21     while (!compare_and_swap(&bag[i],∅,k)) {
22       i = next(i);                                                   // linear probe to find the next slot
23       if (has probed more than κ times) {
24         try_resize(r');
25         return insert(k);  }  }  }
26   void try_resize(int r') {compare_and_swap(&r, r',r'+1);}}
```

Fig. 5. Pseudocode for the hash bag.

3.3 Parallel Hash Bag

As mentioned, VGC brings up the challenge of maintaining the frontier efficiently. Recall that in parallel BFS, the "edge-revisit" scheme first visits all edges incident the frontier to decide the successfully visited vertices, and then revisits all edges to output them to a consecutive array as the next frontier. Since the candidate of the next frontier $\mathcal{F}_{i+1}$ are all neighbors of $\mathcal{F}_i$, we can use a boolean flag to record the success information and let the second visit do the same computation as the first time. However, with VGC, each vertex can search out several hops, and the order of the searches is decided by the runtime scheduling. Note that the local queue is stored in the stack space and discarded after the search. If we want to borrow the "edge-revisit" scheme in BFS, we need to explicitly store the information of the local queues, which can be very costly. To tackle this challenge, we propose a new data structure called ***parallel hash bag*** to maintain the frontier efficiently, such that the next frontier can be generated by visiting the edges only once. Our hash bag supports Insert, ExtractAll, and ForAll efficiently both in theory (work, span, and I/O) and in practice. We start with defining the interface of hash bags:

- Insert(v): add the element v into the bag (resize if needed). It can be called concurrently by different threads.
- ExtractAll(): extract all elements in the bag into an array and remove them from the bag.
- ForAll(): apply a function to all elements in the bag in parallel.

In hash bags, we require to know an upper bound n of the total size, which is true for most applications of hash bags (e.g., $n = |V|$ for maintaining frontiers). We pre-allocate $O(n)$ number of slots as an array *bag* to hold elements to be inserted. However, instead of directly using all the slots, we only use a prefix of them with $O(s)$ in expectation, where s is the current number of elements in the bag. This guarantees the efficiency for ExtractAll and ForAll since we only need to touch

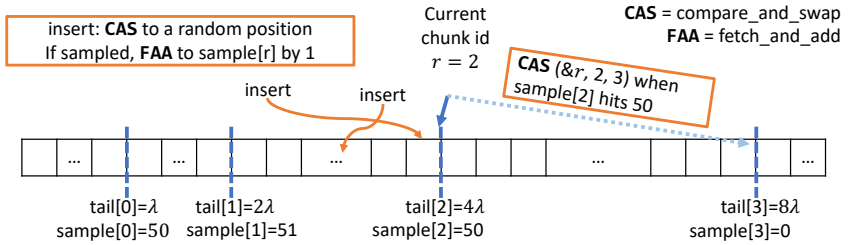

Fig. 6. Parallel Hash Bag. The hash bag is a preallocated array of size $O(n)$, split into chunks of exponentially grown sizes, starting from λ. $tail[i]$ is the last index to use for chunk i. The current chunk id is r. An Insert puts the element to a random position in the current chunk (linear probe for conflict/collision). Each element is sampled at a certain rate. $sample[i]$ is the number of samples in chunk i. When the $sample[r]$ reaches a threshold ($\sigma = 50$ in this example), we resize it by CAS r to $r + 1$.

$O(s)$ space to process s elements. The problem then boils down to maintaining the right size of the used prefix and how to "resize" efficiently.

We show (part of) the pseudocode of hash bags in Fig. 5 and an illustration in Fig. 6. The size of the *bag* is preset as $\Theta(n/\alpha)$, where α is the desired load factor and n is the upper bound of total size as mentioned before. We conceptually divide the *bag* into chunks and use them one by one. A resizing means moving to the next chunk for use. The chunks have doubling sizes of $\lambda, 2\lambda, 4\lambda$ ..., where λ is a parameter for the first chunk size. At initialization, we set up an array $tail[\cdot]$, where $tail[i]$ is the end index of the i-th chunk. We use a variable r to indicate the current in-use chunk id, starting from 0. Elements are always inserted into the r-th chunk (indices from $tail[r-1]$ to $tail[r]$ for $r \geq 1$).

An Insert randomly selects an empty slot in this chunk (Line 20), attempts to put the element in this slot using CAS, and linear probes if the CAS fails (Lines 21–22). Note that different from the hash table, the Insert on hash bags does not check duplicates, but all applications in our paper (maintaining frontiers) can ensure that no duplicates will be added to the bag. For example, duplicates can be checked before calling Insert, e.g., using a boolean flag for each vertex to indicate if it is in the frontier (the array *visit* in Alg. 3, details explained below).

To efficiently decide when resizing is needed, we use a sampling strategy to estimate the size of the hash bag. We use $sample[\cdot]$ to count the number of samples in chunk i, and resize when the number of samples hits σ. We fix σ for all chunks, but set sample rate accordingly for each chunk as σ/α divided by the chunk size. Conceptually, this means to trigger a resize once the load factor goes beyond α. The larger the chunk is, the smaller the sampling rate is. Theoretically, getting accurate estimations requires $\sigma = \Omega(\log n)$.

In Insert, we sample the element with the current rate. If sampled successfully, we increment $sample[r]$ (r is the current chunk) by 1 atomically by CAS (conceptually this is an atomic `fetch_and_add` operation). When $sample[i]$ hits σ, a constant fraction of this chunk is full *whp*, so a resizing attempt is triggered (`try_resize`). Also, when we linear probe for more than a certain number of times, we also trigger a resizing (although this should be rare). In both cases, we resize by increasing r by 1 using a CAS, and call Insert again to add this element to the new chunk.

ExtractAll and ForAll are applied to all elements in $bag[\cdot]$ up to the current chunk r (indices from 0 to $tail[r]$). ExtractAll uses a standard parallel pack [55] to output all (non-empty) elements in an array and remove them from the bag in parallel. ForAll calls a parallel for-loop to apply the function on all elements (skip the empty slots).

We present the framework on using a hash bag H for reachability query in Alg. 3. Given the current frontier $\mathcal{F}_i$, we will visit all vertices in $\mathcal{F}_i$ in parallel and perform local searches from them.

Algorithm 3: Parallel Single-Reachability Using Hash Bags

```
   Input: A directed graph G = (V, E) and a set sources S ∈ V
1  F_0 = S
2  visit[v] ← false for all v ∈ V except visit[s] ← true
3  i ← 0
4  H ← Hashbag()                                   ▷ initial H as an empty hash bag
5  while F_i ≠ ∅ do
6  |  parallel_for_each v ∈ F_i do
7  |  |  Visit v's neighborhood, use local search if applicable
8  |  |  foreach u visited by v do                 ▷ Processing a reachability pair (v, u)
9  |  |  |  if compare_and_swap(&visit[u], false, true) then H.Insert(u)   (*)
10 |  F_{i+1} ← H.ExtractAll()                     ▷ pack elements and clean the bag
11 |  i ← i + 1
12 (*): Note: vertices visited within local searches will not be added
```

We use an array of boolean flags $visit[\cdot]$ to record whether each vertex has been visited. When a vertex $v \in \mathcal{F}_i$ visits a vertex u, we will use CAS to set $visit[u]$ as *true* (Line 9). As mentioned, CAS guarantees that only one concurrent visit to u will succeed. Note that if $visit[u]$ is already *true*, this if-condition will also fail, which guarantees no duplicates in the hash bag. If the CAS succeeds, we call Insert(u) to add u to the hash bag. Note that if local search is enabled, vertices visited within the local search are not added to the next frontier (see details in Sec. 3.1). We omit such cases in the pseudocode for simplicity. Finally, when we finish exploring all the vertices in $\mathcal{F}_i$, we extract (emit and clean) all vertices from the hash bag to form the next frontier (Line 10).

Theoretical Analysis of Hash Bags. We now show the cost bounds of the hash bag.

Theorem 3.1. *For a parallel hash bag of total size n and first chunk size $\lambda = \Omega((P + \log n)\log n)$, inserting s elements using P processors costs $O(s)$ expected work and $O(\log s \log n)$ span* whp*, and listing or packing s elements uses $O(s + \lambda)$ work and $O(\log s)$ span, both* whp*, with mild assumptions (see below).*

We will provide the formal proof in the full version of this paper [107]. In the analysis, we assume the threads are loosely synchronized, where between two consecutive executions of Line 16, other processors can execute at most a constant number of instructions. This assumption is reasonable in practice and is used in analyzing other parallel algorithms such as the analysis of the work-stealing scheduler [3, 48]. Note that the value P is usually a small number (up to hundreds) in practice, and can generally be considered as polylogarithmic of input size n. In practice, we set $\lambda = 2^{10}$ and $\sigma = 50$. We pick $\sigma = 50$ since it is close to $\log n$. We use $\lambda = 2^{10}$ since our analysis indicates λ should roughly be $\log^2 n$. We tested λ for a large range and it affected the running time minimally for $2^8 \le \lambda \le 2^{16}$, so we simply use a single value for all tests.

Our experiments show that hash bags are fast in practice due to the space efficiency and fewer memory accesses. Although we design hash bags for VGC, our experiments show that hash bag itself also improves the algorithms' performance because it avoids scanning the frontier twice. When applying it to LE-lists (see Sec. 5.2), where we can use hash bags but not VGC, we also achieve up to 10× speedup over existing implementations.

4 IMPLEMENTATION DETAILS

We use the techniques in Sec. 3 (VGC with hash bags) to implement reachability searches in the BGSS algorithm for SCC (Alg. 1). This section further presents some details in the implementation. Many of these ideas are also adopted in other recent parallel SCC implementations or graph

libraries [35, 51, 58, 74, 98]. We summarize the cost of our implementation in five categories: ***trimming***, ***first SCC***, ***multi-search***, ***labeling***, and ***hash table resizing***. In Sec. 6.2, we show a running time breakdown based on these five categories.

4.1 Trimming. The algorithm first filters all vertices with zero in- or out-degrees, since they must be in isolated SCCs. It is used in almost all existing SCC implementations.

4.2 Finding the first SCC. As the first reachability search in BGSS only contains one source, we use single-reachability to find the first SCC, and use the standard *dense-backward* [12, 95] optimization. This optimization is designed for single-BFS when the frontier is large. Instead of checking all the *out*-edges from $\mathcal{F}_i$, the dense mode checks each unvisited vertex u and its *in*-neighbors. If any of u's in-neighbor is in the previous frontier $\mathcal{F}_i$, u must be reachable from the source, and we can skip the rest of the edges incident u to save work. We refer to this optimization as ***dense mode***, and the aforementioned approach as ***sparse mode***. We note that dense mode does not work in multi-reachability searches—even if we find a neighbor of u in $\mathcal{F}_i$, we cannot skip the rest of the neighbors since they may come from different sources than v. Therefore, we only use dense mode in single-reachability searches.

4.3 Multi-reachability search. Next, we start $(\log n) - 1$ batches of multi-reachability searches in both forward and backward directions, where round i uses 2^i sources (Lines 6–7). During the multi-reachability search, we need a hash table to identify the duplicate reachability pairs. We use the phase-concurrent hash table [96]. To avoid high overhead in hash table resizing, we use a heuristic to estimate the hash table size, which is discussed below in Sec. 4.5.

4.4 Labeling. After finding all reachability pairs, we mark all vertices strongly connected with any source as *finished*, and label them using the largest vertex id in this SCC (Line 11). For the other vertices, we need to compute their "signatures" of reachability to determine cross edges. We do this also by setting a label for them (Line 11), which is a hash value of the set of vertices reachable from and to v (combining R_1, R_2 with its current label). In this way, two vertices with different labels are in different SCCs. We set the hash value also as the largest vertex id among all vertex reachable from or to v. To avoid the cost of explicitly removing the cross edges, we just skip cross edges in later reachability searches if the endpoints have different labels.

4.5 Heuristic for hash table resizing. The phase-concurrent hash table [96] requires knowing the upper bound of the size before concurrent insertions. With VGC, we do not know a tight upper bound of the number of reachability pairs (v, s), since v can be several hops away from s, and the number of possible pairs can be large. Instead, we compute the number of pairs a in the previous frontier and the number of unfinished vertices b, and use $\max(0.3b, 1.5a)$ and round it up to the next power of 2 as the next hash table size. We resize our hash table once an insertion probes too many times. This heuristic is inspired by recent analyses of the BGSS algorithm in [15]. As shown in Fig. 9, on many graphs, resizing hash tables is costly. Our heuristic effectively reduces this cost.

5 OTHER RELEVANT ALGORITHMS

The two general techniques (VGC and parallel hash bag) introduced in Sec. 3 are general. In this section, we use them to accelerate two other graph algorithms. In particular, in Sec. 5.1 we show how to apply these techniques in a parallel graph connectivity algorithm, and in Sec. 5.2 we show that using hash bags in the algorithm for least-element lists can lead to significantly faster performance.

5.1 Connected Components (CC)

Graph connectivity is one of the most widely-studied graph problems. A recent framework ConnectIt [36] implemented over 232 shared-memory parallel algorithms, based on numerous previous studies both theoretically [8, 16, 23, 35, 54, 94, 95, 97, 100] and practically [13, 16, 35, 36, 95, 97].

Algorithm 4: LDD-UF-JTB Algorithm for Connectivity [97]

Input: A graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$
Output: The connectivity labels $L(\cdot)$ of V

```
1  L ← LDD(G)
2  parallel_for_each (v, u) ∈ E do
3      if FIND(L(v)) ≠ FIND(L(u)) then UNION(L(v), L(u))
4  return L(·)
5  Function LDD(G = (V, E))
6      Set visit[v] ← false and L(v) ← v for all v ∈ V
7      B ← Permute V and group vertices into O(log n) batches in exponentially increasing sizes
8      F ← B_1
9      Set visit[v] ← true for all v ∈ B_1
10     for i ← 2, ..., |B| do
11         F' ← ∅
12         parallel_for_each v ∈ F do
13             parallel_for_each u : (u, v) ∈ E, visit[u] = false do
14                 visit[u] ← true
15                 L(u) ← L(v)
16                 Add u to F'
17         F = F' ∪ {v | v ∈ B_i and visit[v] = false}
18         Set visit[v] ← true for all v ∈ B_i
19     return L(·)
```

Since connectivity is not the main focus of this paper, we picked one of the algorithms from ConnectIt, referred to as "LDD-UF-JTB", as a proof-of-concept to show the effectiveness and generality of the new techniques in this paper. We note that none of the algorithms in ConnectIt has overwhelming advantages on all graphs. LDD-UF-JTB is one of the fastest algorithms, and our new version accelerates it by up to 3.2× compared to the original version in ConnectIt.

LDD-UF-JTB has two major components: the first step uses ***low-diameter decomposition*** (LDD) [76], and the finishing step uses the ***union-find structure*** by Jayanti et al. [57]. We apply our new techniques to the LDD step. An LDD of a graph means to find a decomposition (partition of vertices) of the graph where each component has a low diameter and the number of edges crossing components is small. This LDD step first randomly permutes all vertices, and then starts with a single source and searches out using BFS. Then in later rounds, new sources are added to the frontier (Line 17) in exponentially increasing batches (Line 7) along with the execution of BFS (Line 12 to Line 16). In our implementation, we increase the batch sizes by 1.2× in each round.

Our implementation replaces the BFS in ConnectIt with the more efficient reachability algorithm with VGC optimization and the parallel hash bag. Similar to SCC, we do not need the BFS ordering in computing connectivity, so replacing BFS with (undirected) reachability searches is still correct. In this case, our algorithm can explore more vertices in one round, which leads to fewer rounds and better parallelism. LDD has only $O(\log n)$ rounds (Line 10), so it is already reasonably fast. By using local search and parallel hash bag, we further improve its performance by 1.67× (geometric mean on all graphs). We present the experimental details in Sec. 6.4.

5.2 Algorithm on Least-Element Lists (LE-Lists)

Given an undirected graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$ in a given random total order, a vertex u is in vertex v's ***least-element list (LE-list)*** if and only if there is no earlier vertex than u in V that is closer to v [27]. More formally, for $d(u, v)$ being the shortest distance between u and v, the

Algorithm 5: BGSS Algorithm for LE-Lists [18]

Input: A graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$
Output: The LE-lists $L(\cdot)$ of G

1 Set $\delta(v) \leftarrow +\infty$ and $L(v) \leftarrow \emptyset$ for all $v \in V$
2 Partition V into $\log n$ batches $P_{1..\log n}$, where $|P_i| = 2^{i-1}$
3 **for** $i \leftarrow 1, \ldots, \log n$ **do**
4 Apply multi-BFS from vertices in P_i, and let $S = \{\langle u, v, d(v,u)\rangle \mid v \in P_i, d(v,u) < \delta(u)\}$
5 **parallel_for_each** $\langle u, v, d\rangle \in S$ **do** $\delta(u) \leftarrow \min\{\delta(u), d\}$
6 $L'(u) \leftarrow \{(u, v, i) \in S\}$
7 Sort $L'(u)$ based on the distances in decreasing order, filter out triples that violate constraints, and append v (the second element) to $L(u)$
8 **return** $L(\cdot)$

LE-lists of v_i is:

$$L(v_i) = \left\{v_j \in V \mid d(v_i, v_j) < \min_{1 \le k < j} d(v_i, v_k)\right\} \tag{1}$$

sorted by $d(v_i, v_j)$.

LE-lists have applications in estimating the influence of vertices in a network [25, 29, 40], estimating reachability set size [61, 85], and probabilistic tree embeddings of a graph [19, 60], which further have numerous applications. In this paper, we focus on the unweighted LE-lists algorithm, so the distances can be computed by BFS.

The state-of-the-art parallel algorithm to compute LE-list is the BGSS algorithm (in the same paper as the BGSS SCC algorithm [18]). A pseudocode is given in Alg. 5. It first permutes the vertices V, divides V into $\log_2 n$ batches of size 1, 2, 4, 8, ..., and processes one batch at a time. A tentative distance $\delta(\cdot)$ is maintained for each vertex, initialized as $+\infty$. In each batch, it runs multiple BFS from all vertices in the batch simultaneously, based on $\delta(\cdot)$ from the previous batch. For a vertex $v \in V$, if its search reaches u using a distance smaller than $\delta(u)$, the algorithm add $\langle u, v, d(v,u)\rangle$ to a set S. Finally, we use S to update the tentative distance $\delta(\cdot)$ in this round (Line 5), and the LE-list of each vertex (Lines 6 and 7). Blelloch et al. showed that running multi-BFS in $\log_2 n$ batches enables parallelism, while the work is asymptotically the same. Each LE-list has size $O(\log n)$ *whp*, and the entire algorithm runs $O(m \log n)$ time. A preliminary implementation is given in ParlayLib [14], using multi-BFS discussed in Sec. 2.

We note that we can use the parallel hash bag introduced in Sec. 3.3 to maintain the frontier in the multi-BFS, which avoids the second visit in multi-BFS. VGC is not directly applicable here because the BFS order is required. In addition, we use a parallel hash table [96] to check if a source-target pair is already visited. In the round i, if a source vertex v in the current batch reaches u by a distance shorter than $\delta[u]$, and if (v, u) are not in the hash table, we insert (v, u) to the hash table and hash bag, and pack the hash bag as the next BFS frontier. We insert all such triples (u, v, i) to a set S (Line 4), where v and u are described as above, and i is the current round (which is also the distance between u and v). Our parallel LE-lists implementation outperforms the existing implementation from ParlayLib (from the authors of the BGSS LE-lists algorithm) by 4.34× on average.

6 EXPERIMENTS

Setup. We run our experiments on a 96-core (192 hyperthreads) machine with four Intel Xeon Gold 6252 CPUs and 1.5 TB of main memory. We implemented all algorithms in C++ using ParlayLib [14] for fork-join parallelism and parallel primitives (e.g., sorting). We use `numactl -i all` for parallel tests to interleave the memory pages across CPUs in a round-robin fashion. All reported numbers are the average running time of the last five out of six runs.

		n	m	D	$\|SCC_1\|$	$\|SCC_1\|$%	#SCC	Notes
Social	**LJ**	4.85M	69.0M	16	3,828,682	78.98%	971,232	soc-LiveJournal1 [9]
	TW	41.7M	1.47B	65	33,479,734	80.38%	8,044,729	Twitter [63]
Web	**SD**	89M	2.04B	241	47,965,727	53.74%	39,205,039	sd_arc [75]
	CW	978M	42.6B	666	774,373,029	79.15%	135,223,661	ClueWeb [75]
	HL14	1.72B	64.4B	793	320,754,363	18.60%	1,290,550,195	Hyperlink14 [75]
	HL12	3.56B	129B	5275	1,827,543,757	51.28%	1,279,696,892	Hyperlink12 [75]
KNN	**HH5**	2.05M	10.2M	980	257,914	12.59%	94,010	Household [41, 108], k=5
	CH5	4.21M	21.0M	4550	497,331	11.82%	248,227	CHEM [45, 108], k=5
	GL2	24.9M	49.8M	4142	5,368	0.02%	9,705,931	GeoLife [108, 114], k=2
	GL5	24.9M	124M	12059	860,403	3.46%	3,198,626	GeoLife [108, 114], k=5
	GL10	24.9M	249M	4531	3,042,330	12.23%	326,811	GeoLife [108, 114], k=10
	GL15	24.9M	373M	5491	3,239,156	13.02%	187,646	GeoLife [108, 114], k=15
	GL20	24.9M	498M	5275	3,336,963	13.41%	128,021	GeoLife [108, 114], k=20
	COS5	321M	1.61B	1148	301,413,787	93.88%	2,273,690	Cosmo50 [64, 108], k=5
Lattice	**SQR**	100M	300M	10002	99,101,606	99.10%	829,495	2D grid $10^4 \times 10^4$
	REC	10M	30M	5946	9,890,647	98.91%	101,059	2D grid $10^3 \times 10^4$
	SQR'	100M	120M	51	58	0.00%	78,052,793	sampled SQR
	REC'	10M	12M	80	42	0.00%	7,819,050	sampled REC

Table 2. Graph information. n = number of vertices. m = number of edges. D = estimated diameter (a lower bound of the actual value). $|SCC_1|$ = largest strongly connected component (SCC) size. $|SCC_1|\% = |SCC_1|/n$, ratio of the largest SCC. #SCC = number of SCCs.

We use $\tau = 2^9$ in all tests except for those in Fig. 11 which studies the choice of τ. We tested 18 directed graphs, including social networks, web graphs, k-NN graphs, and lattice graphs. All social, web, and k-NN graphs are real-world graphs, with up to 3.6 billion vertices and up to 128 billion edges. The lattice graphs are generated by a similar model in [32], which uses SCC to study the percolation on isotropically directed lattices. Basic information on the graphs is given in Tab. 2. For social graphs, we use *LiveJournal* (LJ) [9] and *Twitter* (TW) [63]. For web graphs [75], we use *sd-arc* (SD), *ClueWeb* (CW), *Hyperlink12* (HL12) and *Hyperlink14* (HL14). k-NN graphs are widely used in machine learning algorithms [24, 46, 50, 59, 71, 72, 82, 90, 102]. In k-NN graphs, each vertex is a multi-dimensional data point and has k edges pointing to its k-nearest neighbors (excluding itself). We use *Household* with $k = 5$ (HH5) [41, 108], *Chemical* with $k = 5$ (CH5) [45, 108], *GeoLife* with $k = 2, 5, 10, 15, 20$ (GL2, GL5, GL10, GL15, GL20) [108, 114], and *Cosmo50* with $k = 5$ (COS5) [64, 108]. We also created four lattice graphs [32], including two $10^4 \times 10^4$ 2D-lattices (SQR and SQR'), and two $10^3 \times 10^4$ 2D-lattices (REC and REC'). Each row and column in the lattice graphs are circular. In SQR and REC, for each vertex u and its adjacent vertex v, we add a directed edge from u to v with probability 0.5, and from v to u otherwise, then remove duplicate edges. In SQR' and REC', for each vertex u and each of its adjacent vertex v, we create an edge from u to v with probability 0.3, and from v to u with probability 0.3, and create no edge with probability 0.4, then remove duplicate edges. To test our connectivity and LE-lists algorithms, we symmetrize all 18 directed graphs and use 4 more real-world undirected graphs, com-orkut (OK) [112], Friendster (FT) [112], RoadUSA (USA) [1], and Germany (GE) [1]. Graph details are given in Tab. 4.

We call the social and web graphs *low-diameter graphs* as they usually have low diameters (roughly polylogarithmic in n). We call the k-NN and lattice graphs *large-diameter graphs* as their diameters are large (roughly $\Theta(\sqrt{n})$). When comparing the *average* running times across multiple graphs, we always use the ***geometric mean***.

		Ours			GBBS			Other Benchmarks			T_{best}
		par.	seq.	spd.	par.	seq.	spd.	iSpan	MS	SEQ	/ ours
Social	LJ	0.038	1.06	27.7	0.118	1.44	12.1	0.05[?]	0.14	2.90	1.30
	TW	0.226	14.3	63.2	0.387	19.7	50.9	c	1.32	71.7	1.71
Web	SD	1.96	104	53	5.2	110	21.0	4.78[?]	1.86	104	0.95
	CW	17.6	1189	67.4	40.4	1,166	28.9	n	n	589	2.29
	HL14	20.6	1622	78.8	67.3	2,041	30.3	n	n	620	3.27
	HL12	95.5	8528	89.3	361	7,022	19.5	n	n	1,822	3.78
KNN	HH5	0.208	3.1	14.9	3.95	3.51	0.89	0.79	2.21	0.45	2.16
	CH5	0.557	5.83	10.5	8.39	5.84	0.70	2.15	17.6	0.43	0.77
	GL2	0.598	39.1	65.3	3.00	82.4	27.5	t	8.36	3.39	5.01
	GL5	0.865	45.8	53	10.5	91.0	8.69	t	19.1	4.83	5.58
	GL10	1.49	61.6	41.4	12.3	76.5	6.24	35.2	7.14	9.3	4.79
	GL15	2.09	75.5	36.1	13.7	84.5	6.15	29.4	10.6	11.3	5.06
	GL20	2.38	86	36.1	14.5	96.6	6.68	27.3	12.3	13.3	5.18
	COS5	3.22	284	88.2	12.0	367	30.7	t	57.4	189	3.72
Synthetic	SQR	0.577	24.7	42.8	11.1	28.5	2.57	4.45[?]	12.6	15.5	7.72
	REC	0.117	2.08	17.8	3.82	2.14	0.56	1.19[?]	5.24	1.57	10.2
	SQR'	1.38	105	76.3	4.76	243	51.0	26.4[?]	3.19	6.90	2.31
	REC'	0.159	9.38	59	1.00	18.7	18.8	0.85[?]	0.64	0.60	3.75

Table 3. The running times (in seconds) of all tested algorithms on SCC. "iSpan" = iSpan algorithm [58]. "MS" = Multi-step algorithm [98]. "SEQ" = classic sequential SCC [101]. "par." = parallel running time (on 192 hyperthreads). "seq."= sequential running time. "spd."= self-relative speedup. "?"= number of SCCs is different from other implementations. "t"= timeout (more than 5 hours). "c"= crash. "n"= no support. We set $\tau = 2^9$. The fastest runtime for each graph is underlined. Red numbers are parallel running times *slower than SEQ*.

Baseline Algorithms. We call all existing algorithms that we compare to the ***baselines***. We compare the number of SCCs and the largest SCC size reported by each algorithm with SEQ to verify correctness. For SCC, we compare to GBBS [35, 37], iSpan [58], and Multi-step [98]. GBBS also implements the BGSS algorithm, so we also compare our breakdown and sequential running times with GBBS. We also implemented and compared to Tarjan's sequential SCC algorithm [101]. We call it SEQ. On six graphs, iSpan's results are off by 1, noted with "?" in Tab. 3. (We communicated with the authors but could not correct it.) Multi-step and iSpan do not support CW, HL12, and HL14 because they have more than 2^{32} edges. For connectivity, we apply our two techniques to the LDD-UF-JTB algorithm in ConnectIt [36, 36] and compare it to the original implementation in ConnectIt. For LE-lists, we compare to ParlayLib, which is the only public parallel LE-lists implementation to the best of our knowledge.

We first summarize the overall performance of the algorithms and scalability tests in Sec. 6.1. Next, we show some experimental studies on performance breakdown in Sec. 6.2, and an in-depth study of VGC in Sec. 6.3. Finally, we provide a brief summary of our experimental results for connectivity and LE-lists in Sec. 6.4.

6.1 Overall Performance

We show the running times in Tab. 3 and a heatmap in Fig. 1. We mark the parallel running times *slower than the sequential algorithms* in red in Tab. 3. Our implementation is almost always the fastest except on SD and CH5. On CH5, we are 23% slower than SEQ. CH5 has a very large diameter (4000+) compared to its small size (4M vertices), and none of the parallel implementations

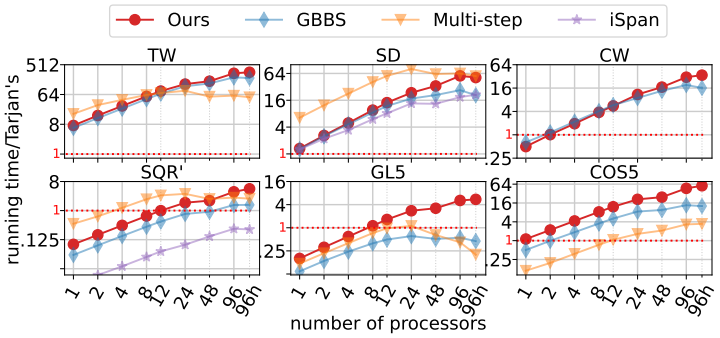

Fig. 7. Speedup over Tarjan's sequential algorithm for different algorithms on different numbers of processors. Larger is better. The red horizontal line shows the running time of Tarjan's algorithm. We omit an algorithm on a graph in cases of crash/timeout/no support.

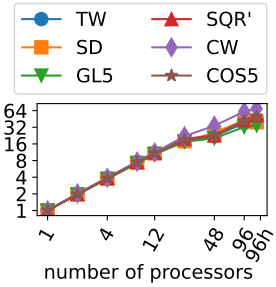

Fig. 8. Self-relative speedup on different processor counts, with $\tau = 2^9$ on six graphs. y-axis the self speedup.

outperform SEQ. On SD, we are only 4% slower than Multi-step with $\tau = 2^9$. SD is one of the graphs that are dense and potentially has good parallelism, and thus may prefer smaller τ. As we will show in Fig. 11, using $\tau \leq 2^8$ will achieve a better performance than all existing implementations on SD, but we keep the results in Tab. 3 all using $\tau = 512$ for simplicity. The highlighted columns in Tab. 3 show the speedup of our algorithms to the *best* baseline (including SEQ) on each graph. Compared to the best baseline, we are up to 10.2× faster and 3.1× faster on average. All the implementations perform favorably on all low-diameter graphs (5–317× faster than SEQ). Conceptually, all parallel implementations first use BFS-like algorithms to find the largest SCC. On all but one low-diameter graph, the largest SCC contains more than 50% vertices. Therefore, using a parallel BFS (with optimizations such as dense modes) gives decent performance. Even so, using hash bags and VGC still gives good performance on low-diameter graphs, and we are faster (up to 3.8×) than the best baseline on all but one graphs. One interesting finding is that on TW, our implementation, GBBS, and Multi-step are faster than SEQ even running sequentially. Similar trends (running the parallel algorithm sequentially is faster than the classic sequential algorithm) have been observed in other BFS-like graph algorithms [95]. This is mostly due to the dense-mode optimization as described in Sec. 4.2. When the frontier size is large, triggering the dense mode can skip many edges, so the number of visited edges can be fewer than $\Theta(m)$ as in the standard sequential solution. Another reason is that our implementation (and GBBS's) using BFS is more I/O-friendly than Tarjan's DFS-based algorithm.

Our algorithm has dominating advantages on large-diameter graphs. On k-NN and lattice graphs, existing parallel implementations are slower than the sequential algorithms in 24 out of 36 tests. If we take the average time of the baseline parallel algorithms for k-NN and lattice graphs, all of them are slower than SEQ (see the "MEAN" columns in Fig. 1). In comparison, our implementation is 5.3× better than SEQ on k-NN graphs and 9.1× better on lattice graphs. We believe the high performance is from good parallelism. We study the scalability of our algorithm in the next paragraph.

Scalability Tests. We show the speedup of four algorithms (ours, GBBS, iSpan, Multi-step) over the sequential Tajan's algorithm on six representative graphs in Fig. 7. We vary the number of processors from 1 to 96h (192 hyperthreads). The red horizontal dot lines represent the running time of Tarjan's algorithm (SEQ), above which means faster than Tarjan's algorithm.

On low-diameter graphs (TW, SD, and CW), all algorithms show reasonably good speedup. On large-diameter graphs (SQR', GL5, and COS5), our algorithm achieves significantly better scalability than the baselines. Our algorithm is the only one that achieves almost linear speedup on all the

six graphs. For all the other algorithms, their performance stops increasing (dropped or flattened) with more than 24 threads on one or more graphs. Multi-step shows good performance on SD, and has better performance than our algorithm especially on a small number of threads. However, Multi-step does not scale well to more processors on most of the graphs.

We also show the self-speedup of our algorithms on six graphs in Fig. 8. We vary the number of processors from 1 to 96h (192 hyperthreads). Due to page limitation, we do not show the curves for all graphs, but the self-speedup on all graphs on 96h (192 hyperthreads) is given in Tab. 3. Our self-speedup is more than 35 except for some very small graphs. This indicates that high parallelism is a crucial factor contributing to the high performance of our code. Compared to GBBS, the fastest previous parallel SCC implementation, our self-speedup is 1.2–32× better. With limited parallelism, GBBS can be slower than SEQ on 8 out of 14 large-diameter graphs—the BGSS SCC algorithm has $O(m \log n)$ work compared to $O(m)$ of Tarjan's sequential algorithm, so with poor self-speedup, the parallelism cannot make up the factor of $O(\log n)$ loss in the total work.

We believe that our good performance comes from using hash bags (saving work on processing sparse frontiers) and VGC (reducing the number of rounds in reachability searches and improving parallelism). We will discuss more details by comparing the performance breakdown with GBBS in Sec. 6.2, and studying the benefit brought up by VGC in Sec. 6.3.

6.2 Performance Breakdown

To better understand the performance of our algorithm, we compare the performance breakdown with GBBS in Fig. 9 since GBBS is also based on the BGSS algorithm and we have similar framework. We compare the running time in six parts (see Sec. 4): 1) ***Trimming***: trimming vertices with no in- or out-degrees; 2) ***First SCC***: finding the first SCC using two single-reachability searches; 3) ***Multi-search***: all multi-reachability searches; 4) ***Hash Table Resizing***: resizing the hash table storing reachability pairs; 5) ***Labeling and Others***: assigning labels to vertices and other costs. We show the breakdown figure for all graphs in Fig. 9. We tested three versions of our algorithm: the ***plain*** version uses parallel hash bags without VGC, the "***VGC1***" version enables VGC in single-reachability to find the first SCC, and the "***final***" version fully enables VGC in both single- and multi-reachability search. We note that some graphs requires more time on *First-SCC* while the others spent more time on *Multi-search* because of different graph patterns, which is indicated by the value of $|SCC_1|$% as shown in Tab. 2.

One straightforward improvement of our algorithm is from our better heuristic to estimate the hash table size (see Sec. 4) to avoid frequent size predicting and hash table resizing. This can be seen by comparing the time of "hash table resizing" (green bars) for GBBS and our versions. This optimization saves much time on almost all graphs. In the following, we use the breakdown results to illustrate the performance improvement from our two main techniques: the hash bag and VGC.

Evaluating hash bags. Parallel hash bags improve the performance by maintaining the frontiers without the edge-revisiting scheme. Note that both our algorithm and GBBS use the BGSS algorithm and perform the same computation in each round, but GBBS uses edge-revisiting and our algorithm avoids that by using the hash bag. Therefore, we compare our *plain* version (i.e., disabling VGC) with GBBS to evaluate the improvement from hash bags, because the major difference between them is the use of hash bags. We also exclude the hash table resizing time (the green bars) for fair comparison. On all but one graphs, using hash bags greatly improve the performance in single- and/or multi-reachabilty searches. Comparing the total running time of reachability searches (red and blue bars), our algorithm is up to 4× faster than GBBS (2× faster on average), and the major improvement is from hash bags.

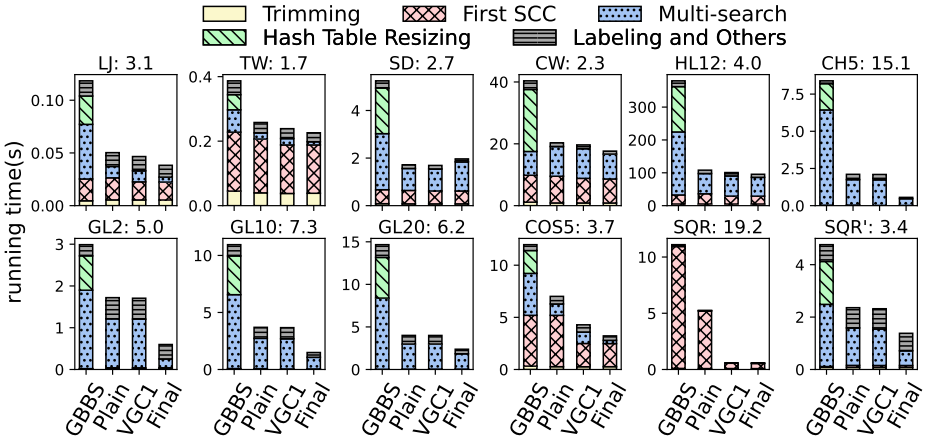

Fig. 9. SCC breakdown time (in seconds). y-axis is the running time. All settings are the same as Tab. 3. "Plain"= our implementation with hash bags but not local search. "VGC1"= adding VGC to the single-reachability search. "Final"= our final version with VGC enabled on both single- and multi-reachability searches. The numbers on top show the speedup of our implementations over GBBS (the first bar).

Evaluating VGC. On top of our *plain* version, VGC improves the performance on almost all graphs. Note that for low-diameter graphs, since the number of needed rounds is small, there is sufficient parallelism to explore. Therefore, using VGC does not improve the performance too much. As mentioned, on SD, the performance drops slightly using local search with $\tau = 2^9$, but using smaller values of τ can still improve the performance (see Sec. 6.3). To keep the parameter setting simple, we still report the numbers with $\tau = 2^9$ in Tab. 3 and Fig. 9. The large-diameter graphs with the largest SCC as 50% of the graph (e.g., COS5, REC, and SQR) greatly benefit from *VGC1* (using VGC in the single-reachability search to find the first SCC). Comparing "first-SCC" of *plain* and *VGC1*, VGC makes the performance 2.2–17× faster in the single-reachability search on COS5, SQR, and REC. All the other large-diameter graphs get significant improvement from *VGC1* to *final* (using VGC also in multi-reachability searches). For all large-diameter graphs, the "multi-search" time in *final* is smaller than that in *VGC1* (1.43–14.7× improvement). As we will show in Sec. 6.3, this is because VGC reduces the number of rounds in reachability searches by 3–200×.

In summary, comparing our *plain* version with GBBS, we can see that hash bag and our heuristic on hash table resizing improves the performance over GBBS by about 1.5–4.3×. Comparing *plain* with *VGC1* and *final*, we can see that VGC improves the performance in both single- and multi-reachability queries by up to 14.7×.

6.3 In-depth Performance Study of VGC

Reduced Number of Rounds. We study the improvement of VGC by reporting the number of rounds in the reachability searches with or without VGC (see Fig. 10). In a given graph, for all single- and multi-reachability searches in the SCC algorithm, we record the number of rounds y needed in plain BFS and the number of rounds x with VGC enabled. We then plot all such points (x, y) on a 2D plane to illustrate the effectiveness of local search, shown in Fig. 10. We also report the average ratio of y/x on the top of each figure. The conceptual "slope" indicated by the points illustrates the factor in the reduction of the number of rounds by using local search. For most of the graphs, especially the k-NN graphs, thousands of rounds were needed in each multi-reachability search using BFS. With VGC, the number of rounds is mostly within 100. Even for the cases where

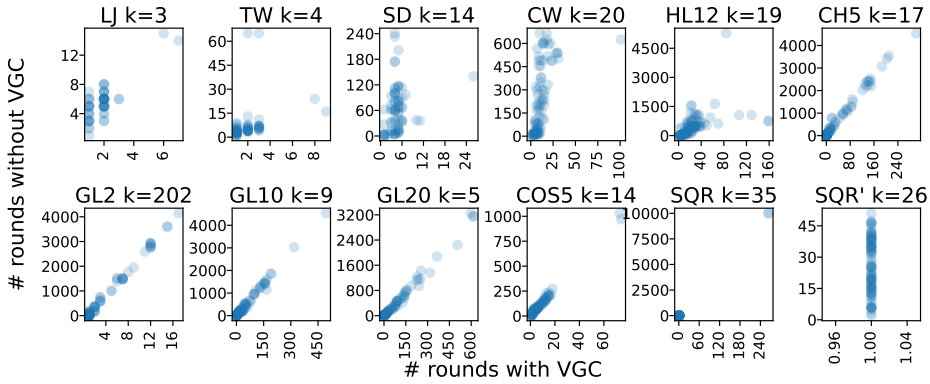

Fig. 10. Number of rounds with and without VGC for each batch. All settings are the same as Tab. 3. Each data point (x, y) means that in one reachability search, y rounds are needed using local search, and x rounds are needed without local search. k for each graph is the average number of y/x for all data points, which means, on average, using local search reduces the number of rounds needed by a factor of k.

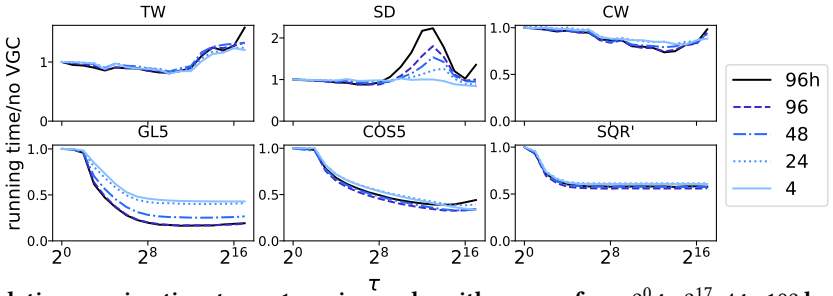

Fig. 11. Relative running time to $\tau = 1$ on six graphs with τ range from 2^0 to 2^{17}, 4 to 192 hyperthreads (96h). LJ has similar trends as TW. HL12 and HL14 show similar trends as CW. All k-NN and lattice graphs show similar trends as GL5, COS5 and SQR'.

BFS only needs a few (10–100) rounds, VGC still reduces the number of rounds to be within 10 rounds (e.g., LJ, TW, COS5, SQR', REC'). On all graphs, the number of rounds is reduced by 3–200×. As a result, the scheduling and synchronization overhead is greatly reduced.

Choice of Parameter τ. To understand the impact of τ values on performance, we record the speedup over our *plain* version (i.e., no VGC) with different values of $\tau = 1$ to 2^{17}, and under different number of processors from 96h (192 hyperthreads) to 4. For page limitation, we show six graphs in Fig. 11 (at least one in each graph type). All the other graphs showed similar trends as one of the six examples. We start from the curves of 192 hyperthreads on different graphs. On all graphs except for LJ, TW and SD, we observe improvement as long as VGC is used (compared to plain BFS where $\tau = 1$) for any $1 < \tau \leq 2^{16}$. Overall, the performance is not sensitive (and is always better than $\tau = 1$) in a large parameter space $2^6 \leq \tau \leq 2^{12}$ on almost all graphs. Based on the results, we set $\tau = 2^9$ as it gives the best overall performance across all graphs. Using $\tau = 2^9$, SD is the only graph that has worse performance than $\tau = 1$. We note that SD still benefits from VGC with $\tau \leq 2^8$. Note that using larger τ suppresses parallelism, and for dense graphs with sufficient parallelism, a smaller τ may perform better.

Although we choose the best parameter by using experiments on 96h, we also test how different numbers of processors P affect the choices of τ. Interestingly, the trends are usually similar regardless of the number of threads used. With a smaller value of P, the performance is less sensitive to the τ

value. This is because τ trades off between scheduling overhead and load balancing, and both affect the performance more when P is large.

We believe that an interesting future work is to set τ dynamically to achieve the best benefit from VGC, possibly based on the sparseness of the graph and the potential parallelism, e.g., the edge-vertex ratio m/n, the number of processors P, or the frontier size.

6.4 Experiments on Connectivity and LE-Lists

Experiments on Connectivity. We implement the LDD-UF-JTB algorithm for graph connectivity in ConnectIt using our parallel hash bags (Sec. 3.3) to maintain the frontiers and the local search optimization (Sec. 3.1). Both optimizations are applied to the sparse rounds in LDD. In Tab. 4, we compare our algorithm to the same algorithm in ConnectIt.

On social networks with low diameters, our algorithm is slightly slower than ConnectIt, but is generally comparable. This is because most of the vertices are visited in the dense mode, which is implemented similarly in both algorithms. The slowdown in our algorithm on social networks seems to be from that VGC brings more work to the first several sparse rounds, which reduces the benefit of using dense modes. For other graphs where dense modes do not significantly dominate the cost, our algorithm generally performs well. On web graphs, our code is 1.21× faster than ConnectIt on average. On the large-diameter graphs, our implementation is 1.98× faster than ConnectIt on average. Since parallel hash bags and VGC only apply to sparse rounds, our speedup over ConnectIt has a correlation with the diameter of the graph. Note that LDD is guaranteed to finish in $O(\log n)$ rounds, as opposed to $O(D)$ for diameter D in SCC. Therefore, the improvement of our implementation over ConnectIt is not as significant as the improvement of our SCC over existing work. However, our implementation still outperforms ConnectIt on 16 instances out of 20, and is 1.67× faster than ConnectIt on average. We believe that the experiments on connectivity provide additional evidence to show that our hash bags and VGC are general and practical.

Experiments on LE-lists. We compare our LE-lists implementation with ParlayLib [14] in Tab. 4. Their implementation is the state-of-the-art and released in 2022. Note that, unlike CC and SCC, here we can only use parallel hash bags for LE-lists but not VGC since the BFS traverse orders need to be preserved.

On low-diameter graphs, our LE-list algorithm is 1.20–3.91× faster (2.73× on average) than ParlayLib's implementation. On large-diameter graphs, the speedup increases to 2.49–10.1× (5.36× on average). We believe this is because hash bags maintains frontier more efficiently, and processing large-diameter graphs involves more rounds (frontiers). Our and ParlayLib's implementation are unable to compute the LE-lists of the three largest graphs CW, HL14, and HL12, because the output size of LE-lists is $O(n \log n)$, which is larger than the memory of our machine. We also report the size of the LE-lists on each graph, and compare it to both ParlayLib's implementation and Cohen's sequential algorithm [27]. ParlayLib's implementation does not report the correct numbers on REC, SQR', and REC', and this is probably why they have poor performance on these graphs.

Overall, our algorithm is faster than ParlayLib's implementation on all graphs. On average, our version is 4.34× faster than ParlayLib's implementation on graphs with correct answers. We note that it remains an interesting question on how to apply a similar local search to LE-lists. We plan to study it in future work.

7 RELATED WORK

Parallel SCC has been widely studied. Prior to the BGSS algorithm based on (multi-)reachability searches, there had been other approaches. The first type of approach is based on parallelizing DFS [21, 22, 70]. However, since DFS is inherently sequential [84] and hard to be parallelized,

	Connectivity			LE-Lists				Connectivity			LE-Lists		
	Ours	DHS'21	Spd.	Ours	Parlay	Spd.		Ours	DHS'21	Spd.	Ours	Parlay	Spd.
OK	0.01	0.01	0.76	0.58	1.52	2.6	**TW**	0.09	0.10	1.05	4.88	17	3.41
LJ	0.01	0.01	0.91	0.5	1.96	3.9	**FT**	0.20	0.15	0.76	24.9	30	1.2
SD	0.22	0.27	1.22	13.9	49.3	3.6	**HL14**	3.69	4.46	1.21	out of memory		
CW	2.43	2.84	1.17	out of memory			**HL12**	8.45	10.4	1.23	out of memory		
USA	0.05	0.09	2.07	14.9	101	6.7	**GE**	0.04	0.13	3.23	5.98	32	5.42
HH5	0.02	0.04	2.4	2.06	15.5	7.6	**GL10**	0.12	0.21	1.71	11	57	5.15
CH5	0.04	0.03	0.76	5.38	54.2	10	**GL15**	0.15	0.24	1.61	11.7	59	5.06
GL2	0.05	0.13	2.97	2.89	14.1	4.9	**GL20**	0.17	0.24	1.46	11.9	64	5.37
GL5	0.07	0.18	2.4	12.4	68.1	5.5	**COS5**	1.31	2.70	2.06	132	329	2.49
SQR	0.16	0.29	1.8	45.4	184	4.1	**SQR'**	0.13	0.28	2.06	46.8	202?	4.32
REC	0.02	0.04	1.66	7.28	520?	71	**REC'**	0.02	0.04	2.08	8.57	648?	75.7

Graph information about the four undirected graphs

	n	m	Notes		n	m	Notes
OK	3M	234M	com-orkut [112]	**FT**	65M	3.6B	Friendster [112]
USA	24M	58M	RoadUSA [1]	**GE**	12M	32M	Germany [1]

Table 4. Running time (in seconds) of connectivity and LE-Lists implementations. DHS'21=the LDD-UF-JTB connectivity implementation in ConnectIt [36]. Parlay=the LE-lists implementation in ParlayLib [14]. Spd.=Baseline_time / our_time. "?"=results different from our parallel and sequential version, and the running time may not be accurate. OK and FT are social networks. USA and GE are road networks.

these algorithms are shown to be slower than existing reachability-based solutions [58]. Another widely-adopted approach is based on single-reachability search (aka. the *forward-backward search*, or *Fw-Bw*) [30, 44, 51, 58, 74, 98, 110]. However, *Fw-Bw* does not provide sufficient parallelism to find all SCCs. Hence, these systems only use *Fw-Bw* to find large SCCs and use other techniques such as coloring and trimming to find small SCCs, which do not have good theoretical guarantees. For this type of approach, we compared the two newest ones with the released code: Multi-step [98] and iSpan [58]. They perform well on graphs with a small diameter and a large SCC_1 (SCC_1 is the largest SCC in the graph), but do not work well on graphs with a large diameter or a small SCC_1 (e.g., the k-NN and lattice graphs in our tests).

Parallel SCC has also been studied on other platforms such as GPUs [10, 33, 52, 53, 67, 69, 99, 109] and distributed systems [11, 73, 74, 88, 111]. Comparing the wall-clock running times reported in the papers, it seems that shared-memory algorithms are much faster, but we note that different platforms have their own use cases.

Related Work of Parallel Hash Bag. There exist other variants of hash tables designed for parallel algorithms [38, 49, 66]. The *parallel bag* [66] supports similar interfaces as our hash bag, but uses a very different design. Parallel bags are organized using pointers, causing additional cache misses in practice. Our hash bag uses flat arrays and is practical and I/O-friendly. The k-level hash table designed for NVRAMs [49] requires allocating memory when resizing, while one of the goals of hash bags is to avoid explicit resizing. Our work is also the first to formalize the interface of maintaining frontiers in parallel reachability search and proposes a practical data structure (the hash bag) with theoretical analysis.

Parallel BFS and DFS. There exist other parallel BFS implementations. Some of them also consider reducing synchronization costs [13, 80, 113]. However, these implementations only consider a

single source. We are unaware of how to directly apply to multi-reachability searches needed in SCC. A recent paper [4] proposed an asynchronous DFS-like searching approach for reachability. However, the approach is not under the fork-join paradigm, and it is unclear if it can generalize to multi-reachability used in BGSS. Our BFS-based approach enjoys better locality and is more general (such as connectivity and LE-lists as shown in Sec. 5).

8 DISCUSSIONS AND FUTURE WORK

In this paper, we show that using faster algorithms on reachability queries can significantly accelerate the performance of SCC and related algorithms, especially for large-diameter graphs. We tested our SCC algorithm on large-scale graphs with up to hundreds of billions of edges. On average, our SCC algorithm is 6.0× faster than the best previous parallel implementation (GBBS), 8.1× faster than Multi-step, and 12.9× faster than Tarjan's sequential algorithms.

We believe that the two key techniques in this paper, the hash bag and vertical granularity control, are general and of independent interest. In this paper, we apply them to graph connectivity and LE-lists. The experimental results show that they lead to improved performance than prior work. We believe that they also apply to many other applications.

Hash bags are used to maintain frontiers (a subset of vertices) in graph algorithms. Many state-of-the-art graph libraries (e.g., GBBS [35] and Ligra [95]) use the abstract data type (ADT) called VertexSubset to maintain frontiers on many graph algorithms. Hash bags can be used to implement this ADT by replacing the current data structure (fixed-size array). With careful engineering, we believe hash bags can potentially improve the performance of these implementations. We leave this as future work.

The high-level idea of VGC applies to traversal-based graph algorithms, such as BFS, algorithms for connectivity, biconnectity, single source shortest paths (SSSP), and some others in [34, 35, 95]. VGC can potentially accelerate them on large-diameter graphs. Our specific "local-search" idea does not directly apply as is. When the traversing order does not matter (e.g., reachability-based algorithms), local search can be applied directly. In a recent paper, we apply local search to graph biconnectivity [39], which improved the overall performance by up to 4× on a variety of graphs. For some distance-based algorithms, we need additional designs on top of local-search, such as supporting revisiting certain vertices (e.g., in BFS, SSSP, LE-lists) for relaxation, or some wake-up strategies to find the next frontier (e.g., in k-core). We believe that this is an interesting research direction, and plan to explore it in the future.

ACKNOWLEDGMENTS

This work is supported by NSF grants CCF-2103483, IIS-2227669, NSF CAREER award CCF-2238358, and UCR Regents Faculty Fellowships. We thank the anonymous reviewers for the useful feedback.

REFERENCES

[1] 2010. OpenStreetMap © OpenStreetMap contributors. https://www.openstreetmap.org/.
[2] Umut A Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. 2019. Provably and practically efficient granularity control. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 214–228.
[3] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2002. The Data Locality of Work Stealing. *Theoretical Computer Science (TCS)* 35, 3 (2002).
[4] Umut A Acar, Arthur Charguéraud, and Mike Rainey. 2015. A work-efficient algorithm for parallel unordered depth-first search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.
[5] Rakesh Agrawal and HV Jagadish. 1990. Hybrid Transitive Closure Algorithms.. In *Proceedings of the VLDB Endowment (PVLDB)*. 326–334.
[6] V Aho Alfred, E Hopcroft John, D Ullman Jeffrey, V Aho Alfred, H Bracht Glenn, D Hopkin Kenneth, C Stanley Julian, Brachu Jean-Pierre, Brown A Samler, Brown A Peter, et al. 1983. *Data structures and algorithms*. USA: Addison-Wesley.

[7] Stefano Allesina, Antonio Bodini, and Cristina Bondavalli. 2005. Ecological subsystems via graph theory: the role of strongly connected components. *Oikos* 110, 1 (2005), 164–176.

[8] Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. 2018. Parallel graph connectivity in log diameter rounds. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE, 674–685.

[9] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 44–54.

[10] Jiří Barnat, Petr Bauch, Luboš Brim, and Milan Ceska. 2011. Computing strongly connected components in parallel on CUDA. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 544–555.

[11] Jiří Barnat, Jakub Chaloupka, and Jaco Van De Pol. 2011. Distributed algorithms for SCC decomposition. *Journal of Logic and Computation* 21, 1 (2011), 23–44.

[12] Scott Beamer, Krste Asanović, and David Patterson. 2012. Direction-optimizing breadth-first search. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. 1–10.

[13] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[14] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib — a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 507–509.

[15] Guy E Blelloch, Laxman Dhulipala, Phillip B Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2021. The read-only semi-external model. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 70–84.

[16] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 181–192.

[17] Guy E. Blelloch, Jeremy T. Fineman, Yan Gu, and Yihan Sun. 2020. Optimal parallel algorithms in the binary-forking model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 89–102.

[18] Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2020. Parallelism in Randomized Incremental Algorithms. *J. ACM* 67, 5 (2020), 1–27.

[19] Guy E. Blelloch, Yan Gu, and Yihan Sun. 2017. Efficient Construction of Probabilistic Tree Embeddings. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*.

[20] Guy E. Blelloch, Yan Gu, Yihan Sun, and Kanat Tangwongsan. 2016. Parallel Shortest Paths Using Radius Stepping. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 443–454.

[21] Vincent Bloemen. 2015. *On-the-fly parallel decomposition of strongly connected components*. Master's thesis. University of Twente.

[22] Vincent Bloemen, Alfons Laarman, and Jaco van de Pol. 2016. Multi-core on-the-fly SCC decomposition. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 1–12.

[23] Nairen Cao, Jeremy T. Fineman, and Katina Russell. 2020. Improved work span tradeoff for single source reachability and approximate shortest paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 511–513.

[24] Edgar Chávez and Eric Sadit Tellez. 2010. Navigating K-Nearest Neighbor Graphs to Solve Nearest Neighbor Searches. In *Advances in Pattern Recognition*. 270–280.

[25] Wei Chen, Yajun Wang, and Siyu Yang. 2009. Efficient influence maximization in social networks. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 199–208.

[26] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. 2013. Tf-label: a topological-folding labeling scheme for reachability querying in a large graph. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 193–204.

[27] Edith Cohen. 1997. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. System Sci.* 55, 3 (1997), 441–453.

[28] Edith Cohen. 1997. Using selective path-doubling for parallel shortest-path computations. *Journal of Algorithms* 22, 1 (1997), 30–56.

[29] Edith Cohen and Haim Kaplan. 2004. Efficient estimation algorithms for neighborhood variance and other moments. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 157–166.

[30] Don Coppersmith, Lisa Fleischer, Bruce Hendrickson, and Ali Pinar. 2003. *A divide-and-conquer algorithm for identifying strongly connected components*. Technical Report. IBM Research.

[31] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms (3rd edition)*. MIT Press.

[32] Aurelio WT De Noronha, André A Moreira, André P Vieira, Hans J Herrmann, José S Andrade Jr, and Humberto A Carmona. 2018. Percolation on an isotropically directed lattice. *Physical Review E* 98, 6 (2018), 062116.

[33] Shrinivas Devshatwar, Madhur Amilkanthwar, and Rupesh Nasre. 2016. GPU centric extensions for parallel strongly connected components computation. In *Workshop on General Purpose Processing using Graphics Processing Unit*. 2–11.

[34] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2017. Julienne: A Framework for Parallel Graph Algorithms using Work-efficient Bucketing. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 293–304.

[35] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2021. Theoretically efficient parallel graph algorithms can be fast and scalable. *ACM Transactions on Parallel Computing (TOPC)* 8, 1 (2021), 1–70.

[36] Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. ConnectIt: a framework for static and incremental parallel graph connectivity algorithms. *Proceedings of the VLDB Endowment (PVLDB)* 14, 4 (2020), 653–667.

[37] Laxman Dhulipala, Jessica Shi, Tom Tseng, Guy E Blelloch, and Julian Shun. 2020. The graph based benchmark suite (GBBS). In *International Workshop on Graph Data Management Experiences & Systems (GRADES)*. 1–8.

[38] Xiaojun Dong, Yan Gu, Yihan Sun, and Yunming Zhang. 2021. Efficient Stepping Algorithms and Implementations for Parallel Shortest Paths. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 184–197.

[39] Xiaojun Dong, Letong Wang, Yan Gu, and Yihan Sun. 2023. Provably Fast and Space-Efficient Parallel Biconnectivity. *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)* (2023), 52–65.

[40] Nan Du, Le Song, Manuel Gomez-Rodriguez, and Hongyuan Zha. 2013. Scalable influence estimation in continuous-time diffusion networks. In *Advances in Neural Information Processing Systems (NIPS)*. 3147–3155.

[41] Dheeru Dua and Casey Graf. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/.

[42] Wenfei Fan, Jianzhong Li, Shuai Ma, Hongzhi Wang, and Yinghui Wu. 2010. Graph homomorphism revisited for graph matching. *Proceedings of the VLDB Endowment (PVLDB)* 3, 1-2 (2010), 1161–1172.

[43] Jeremy T. Fineman. 2018. Nearly Work-Efficient Parallel Algorithm for Digraph Reachability. In *ACM Symposium on Theory of Computing (STOC)*. 457–470.

[44] Lisa K Fleischer, Bruce Hendrickson, and Ali Pınar. 2000. On identifying strongly connected components in parallel. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Springer, 505–511.

[45] Jordi Fonollosa, Sadique Sheik, Ramón Huerta, and Santiago Marco. 2015. Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring. *Sensors and Actuators B: Chemical* 215 (2015), 618–629.

[46] Pasi Franti, Olli Virmajoki, and Ville Hautamaki. 2006. Fast Agglomerative Clustering Using a k-Nearest Neighbor Graph. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28, 11 (2006), 1875–1881.

[47] Socorro Gama-Castro, Heladia Salgado, Alberto Santos-Zavaleta, Daniela Ledezma-Tejeida, Luis Muñiz-Rascado, Jair Santiago García-Sotelo, Kevin Alquicira-Hernández, Irma Martínez-Flores, Lucia Pannier, Jaime Abraham Castro-Mondragón, et al. 2016. RegulonDB version 9.0: high-level integration of gene regulation, coexpression, motif clustering and beyond. *Nucleic acids research* 44, D1 (2016), D133–D143.

[48] Yan Gu, Zachary Napier, and Yihan Sun. 2022. Analysis of Work-Stealing and Parallel Cache Complexity. In *SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*. SIAM, 46–60.

[49] Yan Gu, Yihan Sun, and Guy E. Blelloch. 2018. Algorithmic Building Blocks for Asymmetric Memories. In *European Symposium on Algorithms (ESA)*.

[50] Ville Hautamaki, Ismo Karkkainen, and Pasi Franti. 2004. Outlier detection using k-nearest neighbour graph. In *International Conference on Pattern Recognition*, Vol. 3. 430–433.

[51] Sungpack Hong, Nicole C Rodia, and Kunle Olukotun. 2013. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. 1–11.

[52] Junteng Hou, Shupeng Wang, Guangjun Wu, Ge Fu, Siyu Jia, Yong Wang, Binbin Li, and Lei Zhang. 2019. Parallel strongly connected components detection with multi-partition on gpus. In *International Conference on Computational Science*. Springer, 16–30.

[53] Junteng Hou, Shupeng Wang, Guangjun Wu, Bingnan Ma, and Lei Zhang. 2020. Parallel SCC Detection Based on Reusing Warps and Coloring Partitions on GPUs. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 31–46.

[54] Chirag Jain, Patrick Flick, Tony Pan, Oded Green, and Srinivas Aluru. 2017. An adaptive parallel algorithm for computing connected components. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2428–2439.

[55] Joseph JáJá. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.

[56] Arun Jambulapati, Yang P Liu, and Aaron Sidford. 2019. Parallel reachability in almost linear work and square root depth. In *IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE, 1664–1686.

[57] Siddhartha Jayanti, Robert E Tarjan, and Enric Boix-Adserà. 2019. Randomized concurrent set union and generalized wake-up. In *ACM Symposium on Principles of Distributed Computing (PODC)*. 187–196.

[58] Yuede Ji, Hang Liu, and H Howie Huang. 2018. ispan: Parallel identification of strongly connected components with spanning trees. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE, 731–742.

[59] George Karypis, Eui-Hong Han, and Vipin Kumar. 1999. Chameleon: Hierarchical clustering using dynamic modeling. *Computer* 32, 8 (1999), 68–75.

[60] Maleq Khan, Fabian Kuhn, Dahlia Malkhi, Gopal Pandurangan, and Kunal Talwar. 2012. Efficient distributed approximation algorithms via probabilistic tree embeddings. *Distributed Computing* 25, 3 (2012), 189–205.
[61] Valerie King and Garry Sagert. 2002. A fully dynamic algorithm for maintaining the transitive closure. *J. Computer and System Sciences* 65, 1 (2002), 150–167.
[62] Philip N. Klein and Sairam Subramanian. 1997. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms* 25, 2 (1997), 205–220.
[63] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *International World Wide Web Conference (WWW)*. 591–600.
[64] YongChul Kwon, Dylan Nunley, Jeffrey P Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. 2010. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *International Conference on Scientific and Statistical Database Management*. Springer, 132–150.
[65] Charles E. Leiserson. 2010. The Cilk++ concurrency platform. *The Journal of Supercomputing* 51, 3 (2010), 244–257.
[66] Charles E. Leiserson and Tao B. Schardl. 2010. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 303–314.
[67] Guohui Li, Zhe Zhu, Zhang Cong, and Fumin Yang. 2014. Efficient decomposition of strongly connected components on GPUs. *Journal of Systems Architecture* 60, 1 (2014), 1–10.
[68] Lin Li, Xiang Chen, and Chengyun Song. 2022. A robust clustering method with noise identification based on directed K-nearest neighbor graph. *Neurocomputing* 508 (2022), 19–35.
[69] Pingfan Li, Xuhao Chen, Jie Shen, Jianbin Fang, Tao Tang, and Canqun Yang. 2017. High performance detection of strongly connected components in sparse graphs on GPUs. In *Proceedings of the 8th International Workshop on Programming Models and Applications for Multicores and Manycores*. 48–57.
[70] Gavin Lowe. 2016. Concurrent depth-first search algorithms based on Tarjan's algorithm. *International Journal on Software Tools for Technology Transfer* 18, 2 (2016), 129–147.
[71] Małgorzata Lucińska and Sławomir T. Wierzchoń. 2012. Spectral Clustering Based on k-Nearest Neighbor Graph. In *Computer Information Systems and Industrial Management*. 254–265.
[72] Markus Maier, Matthias Hein, and Ulrike Von Luxburg. 2009. Optimal construction of k-nearest-neighbor graphs for identifying noisy clusters. *Theoretical Computer Science* 410, 19 (2009), 1749–1764.
[73] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 135–146.
[74] William Mclendon Iii, Bruce Hendrickson, Steven J Plimpton, and Lawrence Rauchwerger. 2005. Finding strongly connected components in distributed graphs. *J. Parallel Distrib. Comput.* 65, 8 (2005), 901–910.
[75] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. 2014. Web Data Commons — Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph.
[76] Gary L Miller, Richard Peng, and Shen Chen Xu. 2013. Parallel graph decompositions using random shifts. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 196–203.
[77] Alan Mislove, Massimiliano Marcon, Krishna P Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2007. Measurement and analysis of online social networks. In *ACM SIGCOMM conference on Internet measurement*. 29–42.
[78] Flaviano Morone, Ian Leifer, and Hernán A Makse. 2020. Fibration symmetries uncover the building blocks of biological networks. *Proceedings of the National Academy of Sciences* 117, 15 (2020), 8306–8314.
[79] Matthieu Nadini, Laura Alessandretti, Flavio Di Giacinto, Mauro Martino, Luca Maria Aiello, and Andrea Baronchelli. 2021. Mapping the NFT revolution: market trends, trade networks, and visual features. *Scientific reports* 11, 1 (2021), 1–11.
[80] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 456–471.
[81] Naoto Ohsaka, Takuya Akiba, Yuichi Yoshida, and Ken-ichi Kawarabayashi. 2014. Fast and accurate influence maximization on large networks with pruned monte-carlo simulations. In *AAAI Conference on Artificial Intelligence*, Vol. 28.
[82] Rodrigo Paredes and Edgar Chávez. 2005. Using the k-Nearest Neighbor Graph for Proximity Searching in Metric Spaces. In *String Processing and Information Retrieval*. 127–138.
[83] Keith H Randall, Raymie Stata, Rajiv G Wickremesinghe, and Janet L Wiener. 2002. The link database: Fast access to graphs of the web. In *IEEE Data Compression Conference (DCC)*. IEEE, 122–131.
[84] John H Reif. 1985. Depth-first search is inherently sequential. *Inform. Process. Lett.* 20, 5 (1985), 229–234.
[85] Liam Roditty and Uri Zwick. 2008. Improved dynamic reachability algorithms for directed graphs. *SIAM J. on Computing* 37, 5 (2008), 1455–1471.

[86] Radu Rugina and Martin Rinard. 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Sigplan Notices* 35, 5 (2000), 182–195.

[87] Dimitris Sacharidis, Stavros Papadopoulos, and Dimitris Papadias. 2009. Topologically sorted skylines for partially ordered domains. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, 1072–1083.

[88] Semih Salihoglu and Jennifer Widom. 2014. Optimizing Graph Algorithms on Pregel-like Systems. *Proceedings of the VLDB Endowment (PVLDB)* 7, 7 (2014), 577–588.

[89] Warren Schudy. 2008. Finding strongly connected components in parallel using $O(\log^2 n)$ reachability queries. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 146–151.

[90] Thomas B. Sebastian and Benjamin B. Kimia. 2002. Metric-Based Shape Retrieval in Large Databases. In *International Conference on Pattern Recognition (ICPR)*. 291–296.

[91] Ankush Sharma and Amit Sharma. 2017. KNN-DBSCAN: Using k-nearest neighbor information for parameter-free density based clustering. In *2017 International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*. IEEE, 787–792.

[92] Mihir Shekhar, Lini Thomas, and Kamalakar Karlapalem. 2018. High Dimensional Clustering: A Strongly Connected Component Clustering Solution (SCCC). In *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 1104–1111.

[93] Hanmao Shi and Thomas H. Spencer. 1999. Time-Work Tradeoffs of the Single-Source Shortest Paths Problem. *Journal of Algorithms* 30, 1 (1999), 19–32.

[94] Yossi Shiloach and Uzi Vishkin. 1982. An $O(\log n)$ Parallel Connectivity Algorithm. *J. Algorithms* 3, 1 (1982), 57–67. https://doi.org/10.1016/0196-6774(82)90008-6

[95] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 135–146.

[96] Julian Shun and Guy E Blelloch. 2014. Phase-concurrent hash tables for determinism. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 96–107.

[97] Julian Shun, Laxman Dhulipala, and Guy Blelloch. 2014. A Simple and Practical Linear-work Parallel Algorithm for Connectivity. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 143–153.

[98] George M. Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 550–559.

[99] Miroslav Stuhl. 2013. Computing Strongly Connected Components with CUDA. *Master's thesis, Masaryk University* (2013).

[100] Michael Sutton, Tal Ben-Nun, and Amnon Barak. 2018. Optimizing parallel graph connectivity computation via subgraph sampling. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 12–21.

[101] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. on Computing* 1, 2 (1972), 146–160.

[102] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. 2000. A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science* 290, 5500 (2000), 2319–2323.

[103] Daniel Tomkins, Timmie Smith, Nancy M Amato, and Lawrence Rauchwerger. 2014. SCCMulti: an improved parallel strongly connected components algorithm. *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)* 49, 8 (2014), 393–394.

[104] Silke Trißl and Ulf Leser. 2007. Fast and practical indexing and querying of very large graphs. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 845–856.

[105] Jeffrey D Ullman and Mihalis Yannakakis. 1991. High-probability parallel transitive-closure algorithms. *SIAM J. Comput.* 20, 1 (1991), 100–125.

[106] Letong Wang, Xiaojun Dong, Yan Gu, and Yihan Sun. 2023. Parallel Strong Connectivity. https://github.com/ucrparlay/Parallel-Strong-Connectivity.git.

[107] Letong Wang, Xiaojun Dong, Yan Gu, and Yihan Sun. 2023. Parallel Strong Connectivity Based on Faster Reachability. *arXiv preprint arXiv:2303.04934* (2023).

[108] Yiqiu Wang, Shangdi Yu, Laxman Dhulipala, Yan Gu, and Julian Shun. 2021. GeoGraph: A Framework for Graph Processing on Geometric Data. *ACM SIGOPS Operating Systems Review* 55, 1 (2021), 38–46.

[109] Anton Wijs, Joost-Pieter Katoen, and Dragan Bošnački. 2014. GPU-based graph decomposition into strongly connected and maximal end components. In *International Conference on Computer Aided Verification*. Springer, 310–326.

[110] Taihua Xu and Guoyin Wang. 2018. Finding strongly connected components of simple digraphs based on generalized rough sets theory. *Knowledge-Based Systems* 149 (2018), 88–98.

[111] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of the VLDB Endowment (PVLDB)* 7, 14 (2014), 1821–1832.

[112] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

[113] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.
[114] Yu Zheng, Like Liu, Longhao Wang, and Xing Xie. 2008. Learning transportation mode from raw gps data for geographic applications on the web. In *International World Wide Web Conference (WWW)*. 247–256.

Received October 2022; revised January 2023; accepted February 2023