Parallel Algorithms Can Be Provably Fast and Scalable

Xiaojun Dong
xdong038@ucr.edu
University of California, Riverside
Riverside, CA, USA
Supervised by Yan Gu and Yihan Sun

Abstract

As multi-core processors become more widely available, parallel computing has entered its prime era. Despite significant advances in hardware and extensive theoretical research, there remains a noticeable gap between theory and practice. Many theoretically efficient parallel algorithms, although optimal in theory, are often outperformed by less theoretically rigorous alternatives in practical applications. Conversely, algorithms that excel in real-world scenarios frequently lack a solid theoretical foundation. Our research aims to bridge this divide by redesigning existing algorithms to achieve both theoretical efficiency and practical performance. Our new algorithms demonstrate not only strong theoretical guarantees but also excellent scalability across diverse input sizes, dataset types, and number of processors, making them robust and versatile for real-world applications.

Keywords

Parallel Algorithms, Graph Algorithms, Sorting Algorithms, Edit Distance

VLDB Workshop Reference Format:
Xiaojun Dong. Parallel Algorithms Can Be Provably Fast and Scalable. VLDB 2024 Workshop: VLDB Ph.D. Workshop.

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ucrparlay.

1 Introduction

In this paper, we will discuss our recent work [4–10, 18, 19] in two key areas: large-scale graph processing and fundamental building blocks. As real-world graphs continue to expand, efficient processing and the ability to address a wide range of queries have become increasingly critical. Our research tackles these challenges by developing advanced techniques and algorithms that enhance both the theoretical foundations and practical performance of graph processing. However, many parallel primitives are not as optimized as their sequential counterparts available in standard libraries, which limits the overall performance of many parallel applications. To address this issue, we also focus on improving the efficiency of these building blocks. Our work is summarized in Fig. 1. Due to space limit, we illustrate our techniques with a key example algorithm from each category to convey the high-level ideas of our approach.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment. ISSN 2150-8097.

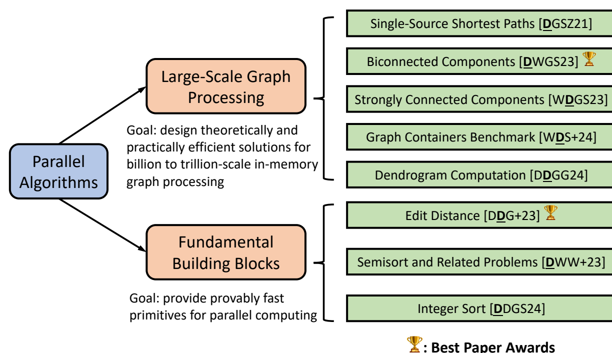

Figure 1: Overview of Our Work.

2 Large-Scale Parallel Graph Processing

Parallel graph processing has great significance, given its capability to handle massive datasets (up to trillion-edge graphs) efficiently. Our research in this domain is dedicated to providing state-of-the-art solutions for connectivity-related problems. Notably, our work has *closed the longstanding performance gap between small- and large-diameter graphs across many problems.* Despite large-diameter graphs are much harder to be processed due to low parallelism and long dependency chains, we overcome these challenges by trading off work and parallelism. Our techniques use a small constant factor more work to reduce the length of dependency chains. Due to low parallelism in the previous design, the increase of work will only saturate all processors but not increasing the overall running time.

In this section, we will elaborate our work on parallel biconnectivity [9], which is the first theoretically-efficient algorithm with optimal work, span and extra space.

2.1 Problem Definition

Given an *undirected* graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, a ***connected component (CC)*** is a maximal subset in V such that every two vertices in it are connected by a path. A ***biconnected component (BCC)*** (or blocks) is a maximal subset $C \subseteq V$ such that C is connected and remains connected after removing any vertex $v \in C$.

Many existing BCC algorithms use the ***skeleton-connectivity*** framework, which first generates a skeleton as an auxiliary graph G' from G, and then finds the CCs on G' that reflect BCCs of the input graph G. The sequential Hopcroft-Tarjan algorithm [11] computes BCC in $O(n+m)$ work by maintaining an implicit depth-first search (DFS) skeleton, but DFS is considered hard to be parallelized. The Tarjan-Vishkin [17] algorithm has $O(n + m)$ optimal work and polylogarithmic span by using an ***arbitrary spanning tree (AST)***. Although Tarjan-Vishkin is optimal in work and span, it is space-inefficient ($O(m)$ extra space instead of $O(n)$). In practice, existing implementations [2, 3, 16] overcome the space issue by

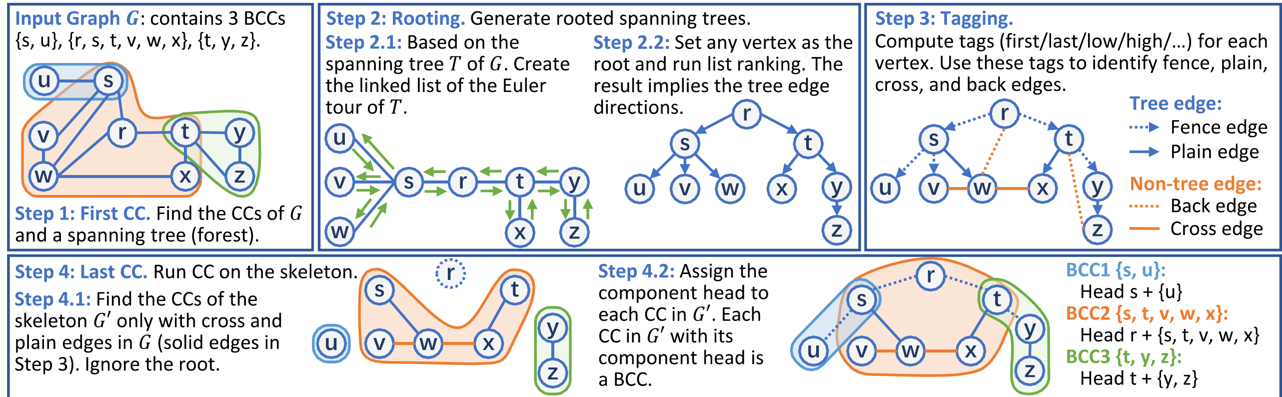

Figure 2: The outline of the FAST-BCC algorithm and a running example. The four steps are explained in detail in Sec. 2.2.

		Ours	GBBS	SM'14	SEQ
Social	YT	5.88	4.36	3.15	1.00
	OK	30.51	19.91	5.66	1.00
	LJ	17.92	11.77	n	1.00
	TW	34.21	17.42	2.40	1.00
	FT	39.26	18.93	10.22	1.00
	MEAN	21.23	12.75	4.57	1.00
Web	GG	8.92	5.65	n	1.00
	SD	29.74	16.46	n	1.00
	CW	30.37	17.52	n	1.00
	HL14	32.46	19.96	n	1.00
	HL12	33.99	29.15	n	1.00
	MEAN	24.53	15.68	-	1.00
Road	CA	5.15	0.55	n	1.00
	USA	6.69	0.49	0.60	1.00
	GE	10.77	1.43	2.44	1.00
	MEAN	7.18	0.73	1.21	1.00

		Ours	GBBS	SM'14	SEQ
K-NN	HH5	7.01	1.14	n	1.00
	CH5	4.11	0.37	n	1.00
	GL2	6.24	1.64	n	1.00
	GL5	8.53	1.44	n	1.00
	GL10	10.59	4.31	n	1.00
	GL15	11.88	5.91	n	1.00
	GL20	11.84	6.88	n	1.00
	COS5	14.16	6.86	n	1.00
	MEAN	8.68	2.42	-	1.00
Synthetic	SQR	18.50	1.59	10.56	1.00
	REC	12.48	0.36	3.02	1.00
	SQR'	8.06	0.85	n	1.00
	REC'	7.81	0.48	n	1.00
	Chn7	11.97	0.04	0.08	1.00
	Chn8	11.97	0.04	0.06	1.00
	MEAN	11.30	0.27	0.18	1.00
	TOTAL MEAN	12.89	2.50	0.96	1.00

0 .5 1 2 4 8 16 32 >32

MEAN = geometric mean
n = no support

Figure 3: The heatmap of relative speedup for parallel BCC algorithms over the sequential Hopcroft-Tarjan algorithm [11] using 96 cores (192 hyper-threads). Larger/green means better. The numbers indicate how many times a parallel algorithm is faster than sequential Hopcroft-Tarjan (< 1 means slower). The two baseline algorithms are from [3, 16].

using breadth-first search (BFS) skeletons. These algorithms are fast on low-diameter graphs, but can be even slower than the sequential implementation (see Fig. 3) on large-diameter graphs because BFS has span proportional to the graph diameters.

We give the first space-efficient (O(n) auxiliary space) parallel BCC algorithm that has efficient $O(m+n)$ work and polylogarithmic span. Unlike Tarjan-Vishkin, our G' is a subgraph of G and can be maintained *implicitly* in $O(n)$ auxiliary space. We implement FAST-BCC and compare it to existing parallel BCC implementations [3, 16] and our sequential Hopcroft-Tarjan implementation. On all graphs, FAST-BCC is faster than all baselines.

2.2 The FAST-BCC Algorithm

Our FAST-BCC algorithm has four steps: *First-CC* (generate spanning trees), *Rooting* (root the spanning trees using ETT), *Tagging* (compute tags for each vertex), and *Last-CC* (run CC on the skeleton and post-processing).

First-CC (Step 1 in Fig. 2, line 1 in Alg. 1). This step finds all CCs in G and generates a spanning forest F of G. For simplicity, in the following, we focus on one CC and its spanning tree T. If G contains multiple CCs, they are simply processed in parallel.

Rooting (Step 2 in Fig. 2, line 2 in Alg. 1). We use the Euler tour technique (ETT) in [11] to root T, which implies the tree edge directions (Fig. 2, Step 2).

Tagging (Step 3 in Fig. 2, line 3 in Alg. 1). This step generates the tags used in the algorithm, including $w_1[\cdot]$, $w_2[\cdot]$, $low[\cdot]$, $high[\cdot]$, $first[\cdot]$, $last[\cdot]$, and the parent array $p[\cdot]$. $low[\cdot]$ and $high[\cdot]$ values are computed by looping over all edges and getting arrays w_1 and w_2, and applying n 1D range-minimum queries (RMQ). These tags will help to decide the four edge types (see details below).

Last-CC (Step 4 in Fig. 2, line 4–6 in Alg. 1). Our skeleton graph G' contains plain tree edges and cross edges. To achieve space efficiency, we do not explicitly store G'. Since G' is a subgraph of G, we can directly use G but skip the fence edges and back edges, which can be determined using the tags generated in Step 3 (line 7–14). Then we compute the CCs on the skeleton G' (line 4), which assigns a label $l[v]$ to each vertex (Fig. 2, Step 4.1). We then assign the head to each CC (lines 5 and 6) by looping over all fence edges (Fig. 2, Step 4.2). For a fence edge u–$p(u)$, if u and $p(u)$ have different labels (line 5), we assign $p(u)$ as the component head of u's CC in G'. This step also only requires $O(n)$ auxiliary space, which is needed by running CC on G but skip certain edges.

Algorithm 1: The FAST-BCC algorithm

Input: An undirected graph $G = (V, E)$
Output: The labels $l[\cdot]$ for vertices, and the component head for each BCC

1. Compute the spanning forest F of G ▹ First CC
2. Root all trees in F using the Euler tour technique ▹ Rooting
3. Compute tags (e.g., *low*, *high*) of each vertex based on the Euler tour ▹ Tagging
4. Compute the vertex label $l[\cdot]$ using connectivity on G with edges satisfying InSkeleton(u, v) = *true* ▹ Last CC
5. **ParallelForEach** $u \in V$ *with* $l[u] \neq l[p(u)]$
6. Set the component head of $l[u]$ as $p(u)$
7. **Function** InSkeleton(u, v) ▹ Decide if u–v is in skeleton G'
8. **if** (u, v) *is a tree edge* **then**
9. **return** ¬ Fence(u, v) *and* ¬ Fence(v, u)
10. **else return** ¬ Back(u, v) *and* ¬ Back(v, u)
11. **Function** Fence(u, v) ▹ Decide if tree edge is fence edge
12. **return** $first[u] \leq low[v]$ *and* $last[u] \geq high[v]$
13. **Function** Back(u, v) ▹ Decide if non-tree edge is back edge
14. **return** $first[u] \leq first[v]$ *and* $last[u] \geq first[v]$

Correctness and Cost Bounds. For space limit, we only present Thm. 2.1. The full analysis is available in the full version.

THEOREM 2.1. *Alg. 1 computes the BCCs of a graph G with n vertices and m edges using $O(n+m)$ expected work, $O(\log^3 n)$ span* whp, *and $O(n)$ auxiliary space (other than the input).*

3 Fundamental Building Blocks

Despite modern code libraries (e.g., Boost and Abseil) are usually integrated with fundamental algorithms, their parallel counterparts do not exist or are often not as well-optimized as the sequential ones. Our work in this domain *aims to provide efficient building blocks for parallel computing, both in theory and practice*. Many of my works are *fundamental primitives that can be used in by a wide range of algorithms or applications*.

In this section, we will elaborate my work on parallel edit distance [5]. Our implementations can process billion-scale strings with small edits in a few seconds.

3.1 Problem Definition

Given two strings $A[1..n]$ and $B[1..m]$ over an alphabet Σ and a set of operations allowed to edit the strings, the ***edit distance*** between A and B is the minimum number of operations required to transform A into B. WLOG, we assume $m \le n$. We use k to denote the edit distance for strings A and B throughout this paper. One useful observation is that, in real-world applications, the strings to be compared are usually *reasonably similar*, resulting in a relatively small edit distance. We say an edit distance algorithm is *output-sensitive* if the work is $o(nm)$ when $k = o(n)$.

3.2 Our Algorithms

The classic dynamic programming (DP) algorithm solves edit distance by using the states $G[i, j]$ as the edit distance of transforming $A[1..i]$ to $B[1..j]$. $G[i, j]$ can be computed as:

$$G[i,j] = \begin{cases} \max(i,j), & \text{if } i = 0 \text{ or } j = 0 \\ G[i-1, j-1], & \text{if } A[i] = B[j] \text{ and } i > 0, j > 0 \\ 1 + \min(G[i-1,j], G[i-1,j-1], G[i,j-1]), & \text{otherwise} \end{cases}$$

We propose four algorithms (BFS-SA, BFS-HASH, BFS-B-HASH, and DAC-SD) to efficiently compute edit distance in parallel.

Algorithm	Work	Span	Space*
BFS-SA	$O(n+k^2)$	$\tilde{O}(k)$	$O(n)$
BFS-HASH*	$O(n+k^2\log n)$	$\tilde{O}(k)$	$O(n)$
BFS-B-HASH*	$O(n+k^2 b\log n)$	$\tilde{O}(kb)$	$O(n/b+k)$
DAC-SD	$O(nk\log k)$	$\tilde{O}(1)$	$O(nk)$

Table 1: Algorithms in this paper. k is the edit distance. b is the block size. *: Monte Carlo algorithms due to the use of hashing. "Space*" means auxiliary space used in addition to the input. Here we assume constant alphabet size for BFS-SA.

BFS-based Algorithms Many existing output-sensitive algorithms [13, 14] are based on breadth-first search (BFS). These algorithms view the DP matrix for edit distance as a DAG, as shown in Fig. 4. We use x and y to denote the row and column ids of the cells in the DP matrix, respectively. Each state (cell) (x, y) has three incoming edges from $(x-1, y)$, $(x, y-1)$, and $(x-1, y-1)$ (if they exist). The edge weight is 0 from $(x-1, y-1)$ to (x, y) when $A[x] = B[y]$, and 1 otherwise. Then edit distance is equivalent to the shortest distance from $(0, 0)$ to (n, m). Since the edge weights

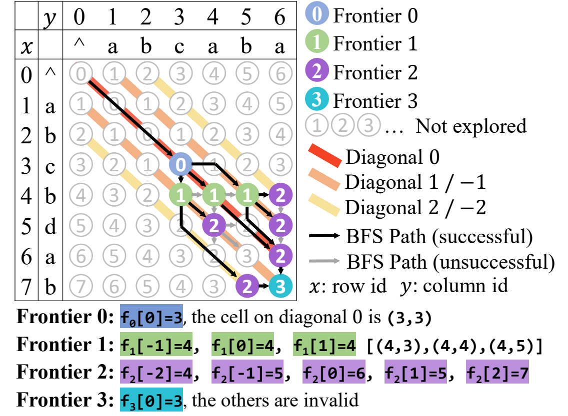

Figure 4: BFS-based edit distance on $A[1..n]$ and $B[1..m]$. $f_t[i]$ is the row-id of the last cell on diagonal i with edit distance t (frontier t), representing cell $(f_t[i], f_t[i]-i)$.

are 0 or 1, we can use a special BFS to compute the shortest distance. In round t, we process states with edit distance t. The algorithm terminates when we reach cell (n, m). Note that all states with $|x-y| > k$ will not be reached. Thus, this BFS will touch at most $O(kn)$ cells, leading to $O(kn)$ work.

Another key observation is that starting from any cell (x, y), if there are diagonal edges with weight 0, we should always follow the edges until a unit-weight edge is encountered. Namely, we should always find the longest common prefix (LCP) from $A[x+1]$ and $B[y+1]$, and skip to the cell at $(x+p, y+p)$ with no edit, where p is the LCP length. This idea is used in Landau and Vishkin [14] on parallel approximate string matching, and we adapt this idea to edit distance here. Using the modified parallel BFS algorithm by Landau-Vishkin [14], only $O(k^2)$ states need to be processed—on each diagonal and for each edit distance t, only the last cell with t edits needs to be processed. Hence, the BFS runs for k rounds on $2k+1$ diagonals, which gives the $O(k^2)$ bound above.

Algorithm Based on Suffix Array (BFS-SA). Using the SA algorithm in [12] and the LCP algorithm in [15] for Landau-Vishkin gives $O(n+k^2)$ work and $\tilde{O}(k)$ span, where k is the edit distance.

Algorithm Based on String Hashing (BFS-HASH). Although BFS-SA is theoretically efficient with $O(n)$ preprocessing work to construct the SA, the hidden constant is large. For better performance, we consider string hashing as an alternative for SA. A hash function $h(\cdot)$ maps any substring $A[l..r]$ to a unique hash value, which provides a fingerprint for this substring in the LCP query. The high-level idea is to binary search the query length, using the hash value as validation. We precompute the hash values for all prefixes, i.e., $T_A[x] = h(A[1..x])$ for the prefix substring $A[1..x]$ (similarly for B). We can compute $h(A[l..r])$ by $T_A[r] \ominus T_A[l-1]$. With the preprocessed hash values, we dual binary search the LCP of $A[x..n]$ and $B[y..m]$. This indicates $O(\log n)$ work in total per LCP query. Combining the preprocessing and query costs, BFS-HASH computes the edit distance between two sequences of length n and $m \le n$ in $O(n+k^2\log n)$ work, $\tilde{O}(k)$ span, and $O(n)$ auxiliary space, where k is the edit distance.

BFS-HASH is simple and easy to implement. However, BFS-SA and BFS-HASH use $O(n)$ extra space, and such space overhead may be a concern in practice. Below we discuss how to make our edit

Algorithm 2: BFS-based parallel edit distance [14]

Input: Two sequences $A[1..n]$ and $B[1..m]$.
Output: The edit distance between A and B.

```
1  f_0[0] ← LCP(A[1..n], B[1..m])                    // Starting point
2  t ← 0
3  while f_t[n − m] ≠ n do
4      t ← t + 1
       // Find new frontier for diagonal i
5      ParallelForEach −t ≤ i ≤ t do
6          f_t[i] ← f_{t−1}[i]                          // Start from the last cell
7          foreach ⟨dx, dy⟩ ∈ {⟨0,1⟩, ⟨1,0⟩, ⟨1,1⟩} do
               // The previous cell is from diagonal j
               // j = (x − dx) − (y − dy) = i − dx + dy
8              j ← i − dx + dy
9              if |j| ≤ t − 1 then
10                 The row id x ← f_{t−1}[j] + dx
11                 The column id y ← x − i
                   // Skip the common prefix and keep the largest row id
12                 f_t[i] ← max(f_t[i], x+LCP(A[x+1..n], B[y+1..m]))
13 return t
```

distance algorithms more space efficient.

Algorithm Based on Blocked-Hashing (BFS-B-Hash). Our BFS-B-Hash algorithm can provide a more space-efficient solution by trading off worst-case time. To achieve better space usage, we divide the strings into blocks of size b, and we only store the hash values for prefixes of the entire blocks $h(A[1..ib])$. Using this approach, we only need auxiliary space to store $O(n/b)$ hash values, and thus we can control the space usage using the parameter b. To compute these hash values, we will first compute the hash value for each block, and run a parallel scan (prefix sum on $\oplus$) on the hash values for all the blocks. Similarly, we refer to these arrays as $T_A[i] = h(A[1..ib])$ (and $T_B[i]$ accordingly), and call them ***prefix tables***. In this way, we can plug the block hash values into the dual binary search in BFS-Hash. In each step of dual binary search, the concatenation of hash value can have at most b steps. Thus, BFS-B-Hash computes the edit distance between two sequences of length n and $m \le n$ in $O(n + k^2 \cdot b \log n)$ work and $\tilde{O}(kb)$ span, using $O(n/b + k)$ auxiliary space, where k is the edit distance.

The term k in space usage is from the BFS (each frontier is at most size $O(k)$). $O(b \log n)$ is the work for each LCP query. Note that this is an upper bound—if the LCP length L is small, the cost can be significantly smaller (a tighter bound is $O(\min(L, b \log L))$).

Divide-and-Conquer-based Algorithm. Our parallel output-sensitive algorithm DaC-SD is inspired by the AALM algorithm [1], and also uses it as a subroutine. Due to space limit, we only present the theoretical analysis of our DaC-SD algorithm. More details are provided in the full paper.

Theorem 3.1. *The DaC-SD algorithm computes the edit distance between two sequences of length n and $m \le n$ in $O(nk \log k)$ work and $O(\log n \log^3 k)$ span, where k is the edit distance.*

4 Conclusion

In this paper, we discuss the theory and practice of many parallel algorithms, including graph algorithms (e.g., single-source shortest paths, connectivity, biconnectivity, strongly connected components, and dendrogram) and some fundamental parallel primitives (e.g., semisort, integer sort, and edit distance). We developed new parallel algorithms that not only improve theoretical bounds but also enhance practical performance. Additionally, we demonstrated that some theoretically optimal algorithms can be implemented in a manner that is practical and efficient.

For future work, we aim to develop a shared-memory parallel graph database capable of handling trillion-scale input graphs and supporting a wide range of queries. To achieve this, we plan to design a dynamic container for maintaining input graphs and expand support for additional query types, such as k-core, k-truss, and triangle counting, as found in existing graph databases.

References

[1] Alberto Apostolico, Mikhail J Atallah, Lawrence L Larmore, and Scott McFaddin. 1990. Efficient parallel algorithms for string editing and related problems. *SIAM J. on Computing* 19, 5 (1990), 968–988.

[2] Guojing Cong and David Bader. 2005. An experimental study of parallel biconnected components algorithms on symmetric multiprocessors (SMPs). In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE.

[3] 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.

[4] Laxman Dhulipala, Xiaojun Dong, Kishen Gowda, and Yan Gu. 2024. Optimal Parallel Algorithms for Dendrogram Computation and Single-Linkage Clustering. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[5] Xiangyun Ding, Xiaojun Dong, Yan Gu, Yihan Sun, and Youzhe Liu. 2023. Efficient Parallel Output-Sensitive Edit Distance. In *European Symposium on Algorithms (ESA)*.

[6] Xiaojun Dong, Laxman Dhulipala, Yan Gu, and Yihan Sun. 2024. Parallel Integer Sort: Theory and Practice. In *ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*.

[7] Xiaojun Dong, Yan Gu, Yihan Sun, and Letong Wang. 2024. Brief Announcement: PASGAL: Parallel And Scalable Graph Algorithm Library. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[8] 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.

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

[10] Xiaojun Dong, Yunshu Wu, Zhongqi Wang, Laxman Dhulipala, Yan Gu, and Yihan Sun. 2023. High-Performance and Flexible Parallel Algorithms for Semisort and Related Problems. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[11] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM* 16, 6 (1973), 372–378.

[12] Juha Kärkkäinen and Peter Sanders. 2003. Simple linear work suffix array construction. In *Intl. Colloq. on Automata, Languages and Programming (ICALP)*. Springer, 943–955.

[13] Gad M Landau and Uzi Vishkin. 1988. Fast string matching with k differences. *J. Computer and System Sciences* 37, 1 (1988), 63–78.

[14] Gad M Landau and Uzi Vishkin. 1989. Fast parallel and serial approximate string matching. *J. Algorithms* 10, 2 (1989), 157–169.

[15] Julian Shun. 2014. Fast parallel computation of longest common prefixes. In *International Conference for High Performance Computing, Networking, Storage, and Analysis (SC)*. IEEE, 387–398.

[16] George Slota and Kamesh Madduri. 2014. Simple parallel biconnectivity algorithms for multicore platforms. In *IEEE International Conference on High Performance Computing (HiPC)*. IEEE, 1–10.

[17] Robert E Tarjan and Uzi Vishkin. 1985. An efficient parallel biconnectivity algorithm. *SIAM J. on Computing* 14, 4 (1985), 862–874.

[18] Letong Wang, Xiaojun Dong, Yan Gu, and Yihan Sun. 2023. Parallel Strong Connectivity Based on Faster Reachability. *ACM SIGMOD International Conference on Management of Data (SIGMOD)* 1, 2 (2023), 1–29.

[19] Brian Wheatman, Xiaojun Dong, Zheqi Shen, Laxman Dhulipala, Jakub Łącki, Prashant Pandey, and Helen Xu. 2024. BYO: A Unified Framework for Benchmarking Large-Scale Graph Containers. In *Proceedings of the VLDB Endowment (PVLDB)*.