Minimum-Mapping based Connected Components Algorithm

ZHIHUI DU, OLIVER ALVARADO RODRIGUEZ, FUHUAN LI, MOHAMMAD DINDOOST, and DAVID A. BADER, New Jersey Institute of Technology, USA

Finding connected components is a fundamental problem in graph analysis. We develop a novel minimum-mapping based *Contour* algorithm to solve the connectivity problem. The *Contour* algorithm can identify all connected components of an undirected graph within $O(\log d_{max})$ iterations on m parallel processors, where d_{max} is the largest diameter of all components in a given graph and m is the total number of edges of the given graph. Furthermore, each iteration can easily be parallelized by employing the highly efficient minimum-mapping operator on all edges. To improve performance, the *Contour* algorithm is further optimized through asynchronous updates and simplified atomic operations. Our algorithm has been integrated into an open-source framework, Arachne, that extends Arkouda for large-scale interactive graph analytics with a Python API powered by the high-productivity parallel language Chapel. Experimental results on real-world and synthetic graphs show that the proposed *Contour* algorithm needs less number of iterations and can achieve 5.26 folds of speedup on average compared with the state-of-the-art connected component method *FastSV* implemented in Chapel. All code is publicly available on GitHub (https://github.com/Bears-R-Us/arkouda-njit).

Additional Key Words and Phrases: connected components, graph analytics, big data, parallel algorithm

ACM Reference Format:
Zhihui Du, Oliver Alvarado Rodriguez, Fuhuan Li, Mohammad Dindoost, and David A. Bader. 2023. Minimum-Mapping based Connected Components Algorithm. In . ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A graph is one of the fundamental mathematical structures used to model pairwise relations between abstract objects. Many problems in science, society, and economics can be modeled by graphs. The sizes of graph data collections continue to grow which makes the need for fast graph algorithms critical, especially under online and real-time scenarios [31].

Finding connected components [1, 4, 9, 10, 12, 16, 17, 21, 27] is a fundamental problem in graph analytics and an important first step for other graph algorithms. Many graph algorithms are based on the assumption that we already know a graph's connected components. In this work, we focus on the connectivity of undirected graphs. The connected components problem can be expressed as assigning each vertex with a label. If two vertices are in the same component or there is a path between them, they will be marked with the same label. Otherwise, the vertices will be marked with different labels [11]. We abstract the connectivity as a *Minimum-Mapping* problem and develop a simple and lightweight minimum-mapping operator to work on different edges to efficiently identify all the components in parallel. We develop an efficient minimum-mapping operator that maps the connected vertices to the same *contour*. Identifying one component is similar to identifying one *contour* with the same minimum label. Therefore, we name our algorithm "*Contour*". No tree

CHIUW'2023, June 1-2, 2023,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

hooking-compressing or set find-union operations are needed. At the same time, the minimum-mapping operator can be employed on different edges (or connected vertices) in parallel, whether their vertices are overlapping or not. This can significantly improve parallel performance and simplify implementation compared to tree/set merge-based methods.

The major contributions of this work are as follows.

(1) A novel *Contour* algorithm that formulates finding connected components as a minimum-mapping problem. Based on this perspective, a simple and lightweight minimum-mapping operator is developed to map the vertices in the same component to the same label in parallel. The proposed method is suitable for large graphs with different graph topologies.
(2) A proof is given to show that for a graph with d_{max} as its maximum diameter of any connected components, the *Contour* algorithm can converge in $O(log(d_{max}))$ iterations.
(3) The proposed method has been integrated into the graph package, Arachne. It is publicly available through the open-source Arkouda framework from GitHub to analyze large graphs using the popular Python interface.
(4) Extensive experimental results show that the proposed *Contour* algorithm can achieve significant speedup compared with the state-of-the-art traversal and tree-hooking-based methods on real-world and synthetical graphs.

2 CONTOUR ALGORITHM

2.1 Problem Description

Given an undirected graph $G =< V, E >$, where V is the set of vertices, and E is the set of edges. Let $m = |E|$ be the total number of edges and $n = |V|$ be the total number of vertices in G. Without loss of generality, here we assume that vertex IDs are from 0 to $n - 1$. Identifying all connected components in G means we will assign the vertices of the same components with the same label.

We can use a label array $L[0..n-1]$ with size n to store all the label values of different vertices. Initially, we assign each vertex's ID as its label. The label array can also be regarded as a point graph [33]. $\forall v \in V, L[i] = v$ means that there is a direct edge from vertex i to v. The point graph will change after each iteration. It is a forest of rooted trees plus self-loops that occur only in the root. Finally, if graph G has S components, L will represent S stars after all components are found.

2.2 Minimum-Mapping Operator

$\forall v \in V$, $L[v]$ is the mapped vertex or label of v. $L_u[0..n-1]$ is used to store the updated value of different vertices after one update. If there is a path between w and v or w and v are connected, and the values of their labels are different, we should assign them the same minimum label. We use the minimum value among $L[w]$ and $L[v]$ to update the old label values in L_u array.

First, we define the conditional vector assignment operator as follows.

Definition 2.1 (Conditional Vector Assignment).

$$(x_1, ..., x_k) \xleftarrow{>} z \tag{1}$$

It means that given a vector $X = (x_1, ..., x_k), \forall i \in \mathbb{N}, 1 \leq i \leq k, x_i = z$ if $x_i > z$.

Based on the definition of conditional vector assignment, we will further define our minimum-mapping operators.

Definition 2.2 (One-Order Minimum-Mapping Operator). Given two connected vertices $w, v \in V$, let $z^1 = min(L[w], L[v])$. We define the one-order minimum-mapping operator as follows.

$$MM^1(L_u, L, w, v) : (L_u[w], L_u[v]) \xleftarrow{>} z^1 \tag{2}$$

$MM^1(L_u, L, w, v)$ means that given two vertices w and v and the original label value array L, the updated results after the one-order minimum-mapping operation will be kept in the updated array L_u.

Higher $h>1$ order minimum-mapping operators $MM^h(L_u, L, w, v)$ can also be defined similarly.

Definition 2.3 (h-Order Minimum-Mapping Operator). Given two connected vertices $w, v \in V$, if we let $z^h = min(L^h[w], L^h[v])$, we can define the h-order minimum-mapping operator as follows.

$$MM^h(L_u, L, w, v) : (L_u[w], L_u[v], ..., L_u[L^{h-1}[w]], L_u[L^{h-1}[v]]) \xleftarrow{>} z^h. \tag{3}$$

where $\forall v \in V, L^h[v] = L[L^{h-1}[v]], L^1[v] = L[v], z^h = min(L^h[w], L^h[v])$.

A higher-order minimum-mapping operator may include more mapped vertices based on the two given vertices. So it may find the final minimum contour quicker. However, it will also perform many more operations. In this paper, we take the two-order minimum-mapping operator as the default operator because it can achieve a quick convergence (logarithmic time complexity) with a minimum-mapping operator involving a much small number of vertices and operations.

2.3 Algorithm Description

Based on the proposed minimum-mapping operator, our *Contour* algorithm is given in Alg. 1. The complete algorithm is straightforward and easy to parallelize.

For lines from 1 to 4, we initialize the label array L and the corresponding update array L_u with each vertex's ID. From line 5 to line 10, we update the label array L until it is converged or there are no changes in the array. From lines 6 to 8, for each edge $e = \langle w, v \rangle \in E$, we will execute the two-order minimum-mapping $MM^2(w, v)$ in parallel. $MM^2(w, v)$ may update the value of $L_u[w], L_u[v], L_u[L[w]], L_u[L[v]]$ if they are larger than the minimum value z^2. In line 9, all the old values in L will be updated with the new values in L_u.

Since all the conditional assignments can be executed in parallel, to avoid write race, we use the atomic compare-and-swap (CAS) [1] operation to implement our conditional assignment as follows.

$$\begin{aligned} &while\ (oldx_i = atomic\_read(x_i) > z)\ \{ \\ &\quad CAS(x_i, oldx_i, z) \\ &\} \end{aligned} \tag{4}$$

This is different from the CRCW PRAM model-based method [33]. CRCW will generate arbitrary write results. Our CAS-based method ensures that only the minimum value is assigned to the L array.

After k^{th} iteration, the label of w should be $L_{u,k}[w] = min(L^2_{k-1}[w], L^2_{k-1}[v_1], L^2_{k-1}[v_2], ..., L^2_{k-1}[v_m]$, where $v_1, v_2, ..., v_m$ are the vertices that directly connect with w, or directly connect with the vertices that are mapped to w; $L^h_k[x]$ is the h order label of x after the k^{th} iteration. We can ignore h when $h = 1$. We first give the following definition to show how the vertices in the same component are mapped to the same label step by step.

Definition 2.4 (Equal Minimum Set). Given label a, after the k^{th} iteration, its one-order equal minimum set $EMS(k)^1_a = \{v | \forall v \in V, L_k[v] = a\}$. Its two-order equal minimum set $EMS(k)^2_a = \{v | \forall v \in V, L^2_k[v] = a\}$.

We use the equal minimum set to indicate the vertices mapped to the same vertex label.

[1] https://chapel-lang.org/docs/primers/atomics.html

Algorithm 1: Minimum-Mapping based *Contour* Algorithm

```
1  Contour(G)
   /* G = ⟨E, V⟩ is the input graph with edge set E and vertex set V. m = |E| is the total number of edges and n = |V| is the
      total number of vertices.                                                                                          */
2  forall i in 0..n-1 do
3  |   L[i] = i
4  |   L_u[i] = i
5  end
   /* Initialize the label array L, L_u                                                                                  */
6  while (There is any label change in L) do
7  |   forall (e = ⟨w, v⟩ ∈ E) do
8  |   |   MM^2(L_u, L, w, v)
9  |   end
10 |   L = L_u
11 end
12 return L
```

Definition 2.5 (Merged Minimum Set). Let $MMS(0) = V$. After the k^{th} iteration, $k \geq 1$, the one-order merged minimum set is defined as $MMS(k)^1 = \{v | \forall v \in V, EMS(k)_v^1 \neq \phi\}$. Similarly, the two-order merged minimum set $MMS(k)^2 = \{v | \forall v \in V, EMS(k)_v^2 \neq \phi\}$.

Compared to other label propagation algorithms, Alg. 1 has two major differences. (1) We use edge-based parallel method that is easy to achieve load balancing for degree-skewed real-world graphs. However, label propagation typically uses vertex-based parallel methods. (2) The kernel of existing label propagation methods is similar to our one-order minimum mapping operator, which cannot achieve high performance for high-diameter graphs. However, our algorithm can employ high-order minimum-mapping operators to reduce the total number of iterations. Compared to existing tree-hooking-based vertices merging methods, both are edge-based parallel methods. However, our minimum-mapping operator can significantly simplify the tree operations.

The following section will prove that our Alg.1 can converge in logarithmic iterations.

We can prove that Alg. 1 will take $O(\log(d_{max}))$ iterations to identify all the components. The detailed proof is ignored here.

3 INTEGRATION WITH ARACHNE AND PERFORMANCE OPTIMIZATION

3.1 Integration Method

Our method is integrated into Arachne [30], a large-scale graph analytics package on top of Arkouda [24, 29]. Arkouda is an open-source framework in Python created to be a NumPy replacement at scale. Our work aims to extend Arkouda for graph analytics, where we use the underlying *pdarray* to implement and execute our algorithms. Through this, we create an end-to-end response system from Chapel to Arkouda. In Python, our calling method is called *graph_cc(graph)* where the user passes to a function a graph. We added this method to Arkouda's front-end file called *graph.py*. The calling messages are added into *arkouda_server.chpl*. The Chapel method is invoked when the function is called in Python, and the messages are passed from Python to Chapel through ZMQ [2]. The messages are recognized at the back-end by *arkouda_server.chpl*, and the proper functions are invoked and executed in the chapel back-end.

3.2 Algorithm Optimization

Alg.1 shows the basic idea of our method. However, we can continue to optimize it to improve practical performance. So, we propose the asynchronous version of Alg.1 to improve the convergence speed.

[2]https://zeromq.org/

The basic idea of the asynchronous *Contour* algorithm is that we will immediately update the label array L instead of keeping the old value and updating it in the update label array L_u. So, the update label array L_u can be removed entirely. The asynchronous algorithm has the following advantages: (1) accelerating the convergence speed (vertices can be mapped to lower label quickly); (2) reducing unnecessary operations ($L = L_u$ in Alg.1 is removed); (3) reducing unnecessary memory (L_u array is removed).

Another aspect is optimizing the implementation of the conditional vector assignment in Eq. 4. We design a new compare-and-assignment atomic operation $CAA(old, new)$. It means that if $old > new$, then $old = new$. Suppose the low-level system can support such a compare-and-assignment atomic operation with high performance. In that case, we can use such a new atomic operation to implement our *Contour* algorithm. We currently replace the conditional assignment in Eq. 4 with Eq. 5 to partially simulate its effect.

$$\begin{aligned} oldx_i &= atomic\_read(x_i) \\ CAS(x_i, oldx_i, z) \end{aligned} \tag{5}$$

We also implement the state-of-the-art *FastSV* [43] in Chapel for comparison.

4 EXPERIMENTS

4.1 Dataset Description

Our datasets are chosen from publicly available synthetic and real-world datasets. The SuiteSparse Matrix Collection[3] and the MIT GraphChallenge graph datasets[4] are used in selecting our test graphs. A combination of real-world and synthetic graphs is selected to highlight the performance of our optimized *Contour* algorithm in Table 1. The real-world graphs can be either weighted or unweighted. Real-world graphs have degree distributions that follow a power-law distribution, while sparse synthetic graphs tend to follow a normal distribution.

The synthetic graphs used are random geometric graphs (Rgg) and kron-g500 graphs from the DIMACS10 Implementation Challenge [5].

Table 1. Dataset 1 - Real-World and Synthetic Graphs

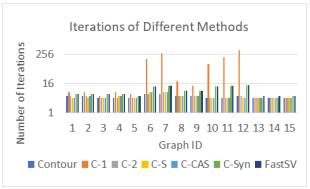
Graph Type	Graph ID	Graph Name	m	n
Real-World Graph	1	loc-brightkite_edges	214078	58228
	2	soc-Epinions1	405740	75879
	3	amazon0601	2443408	403394
	4	com-youtube.ungraph	2987624	1134890
	5	soc-LiveJournal1	68993773	4847570
	6	kmer_V1r	232705452	214005017
	7	kmer_A2a	180292586	170728175
	8	uk-2002	261787258	18484117
	9	uk-2005	783027125	39454746
Synthetic Graph	10	rgg_n_2_21_s0	14487995	2097148
	11	rgg_n_2_22_s0	30359198	4194301
	12	rgg_n_2_24_s0	132557200	16777215
	13	kron_g500-logn16	2456071	55321
	14	kron_g500-logn18	10582686	210155
	15	kron_g500-logn20	44619402	795241

4.2 Experimental Platform

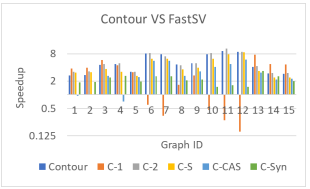
Experiments were done on an 32-node cluster system. Each node is a CentOS Linux release 7.9.2009 (Core) high-performance server with 2 x Intel Xeon E5-2650 v3 @ 2.30GHz CPUs with ten cores per CPU. Each server has an amount of 512GB of RAM. All nodes are connected by a high-performance Infiniband network system.

[3]https://sparse.tamu.edu/

[4]https://graphchallenge.mit.edu/data-sets



(a) Number of iterations of our different variants of *Contour* algorithm and *FastSV* algorithm for both real-world and synthetic graphs (one locale).



(b) The performance of our *Contour* algorithm and its variants compared with the *FastSV* algorithm for both real-world and synthetic graphs (one locale).

4.3 Experimental Results

In the following sections, we use *Contour* to indicate our highly optimized algorithm. *C-1* means we just use the fixed one-order minimum-mapping operator. *C-2* means that we just use the fixed two-order minimum-mapping operator. *C-S* means we employ the simplified Eq. 5 to replace the atomic operation in a loop. *C-CAS* means we employ a loop-based compare-and-swap atomic operation according to Eq. 4. *C-Syn* means the synchronized *Contour* algorithm. *FastSV* is the result of the *FastSV* algorithm. We execute the tests in two ways, shared memory and distributed memory. For shared memory, we only use one node. For distributed memory, we will use four nodes. Both are executed in parallel. We will first discuss the shared memory results. For the Chapel parallel program, it means that we only use one locale[5] to explore the parallelism in one shared memory computing device.

Comparison of Number of Iterations: First, we analyze how different methods can affect the number of iterations. Fig. 1a shows the total number of iterations for different graphs with our optimized *Contour* algorithm, its different variants, and the *FastSV* method. We can see our *Contour* method has the smallest number of iterations for almost all the different graphs. The number of iterations of *C-1* is very high for high-diameter graphs. This is reasonable because the number of iterations of our one-order minimum-mapping operator will be linear with the diameter of a graph. The *C-S* and *C-CAS* variants are very close to each other and also have a very small number of iterations. For some real-world graphs, the numbers of iterations of *C-S* and *C-CAS* are slightly higher than the *Contour* method. *C-Syn* and *FastSV* have almost the same number of iterations for most graphs and they are always the worst ones. This is because they have a similar iteration mechanism. Both will be synchronized after each iteration.

Speedup Compared with FastSV: Fig. 1b shows the speedup of our *Contour* algorithm and its variants compared with the state-of-the-art *FastSV* method. For all the graphs, our *Contour* algorithm can achieve 5.26 folds of speedup on average. The maximum speedup is 9.17 for the synthetic graph rgg_n_2_22_s0. The minimum speedup is 2.68 for the real-world graph loc-brightkite_edges. We can see the performance of *C-1* can be very high for low-diameter graphs. But it can also be very low for high-diameter graphs. So, the performance of *C-1* is not stable. *C-2* is very good for most of cases because it can balance the low-diameter and high-diameter graphs. *C-S* is always better than *C-CAS* because removing the loop operation will save time. This also means that our suggested hardware-supported atomic *CAA* can perform better than *C-CAS*. So, the performance trend is *Contour* > *C-S* > *C-CAS* > *C-Syn* > *FastSV*, where $a > b$ means that the performance of a is better than that of b.

The trend of the results on 4 locales is similar to that of the one locale results, so, we do not include the results here.

[5]https://chapel-lang.org/docs/primers/locales.html

4.4 More Results and Discussions

We also implement the *BFS*-based connected components method in Arachne and compare our *Contour* algorithm with the widely used *BFS* method. Here both *Contour* and *BFS* are running on one locale using data set 2. The total number of iterations of *BFS* is much larger than that of our *Contour* algorithm, especially for the large diameter graphs. For most of the graphs, *Contour* can converge in 4 iterations. However, BFS will need much more iterations (the largest number of iterations is 1510 for *BFS*). The speedup will increase with graph size at first and then become flat and then drop. The average speedup is 196. The maximum speedup is 320 for delaunay_n19. The minimum speedup is 30 for delaunay_n10.

Multi-locale is employed to test the performance of a distributed system. The trend of the results on 4 locales is similar to that of the one locale results. However, the performance is very different for different methods. For the *C-1* method, its number of iterations is the largest. However, its performance is very good for graphs whose diameters (graph sizes) are not very large, even better than the *Contour* version. Only when the graph diameter (graph size) becomes very large will its performance drop. The reason is that the major cost for a distributed memory graph algorithm is communication instead of computation. The *C-1* method can exploit the locality well. However, all the other methods will involve more communication relatively. This is why *C-1* can achieve much better performance when the graph diameter (graph size) is not large. Our experimental results show that the absolute performance of distributed memory version is much slower than the corresponding shared memory version.

For delaunay_n20, *FastSV* in Chapel needs 0.290952s. *Contour* needs 0.038154s. We implement our method in two graph packages, LAGraph and Graph Based Benchmark Suite (GBBS). In LAGraph (in C and GraphBLAS), *FastSV* needs 0.0226186s. *Contour* cannot be faster than *FastSV* because GRaphBLAS interface cannot support the flexible minimum-mapping operator well. In GBBS, we compare our algorithm with the package's simplified *SV* algorithm. Our *Contour* will need 0.012859s, and the simplified SV algorithm takes 0.022097s.

Based on the above experiments and comparison, we can see that compared with some C or C++ based graph packages, our package in Chapel has some overhead. However, we also find that the Chapel is much more flexible to express different parallel methods.

5 RELATED WORK

Connected components problem can be formulated as graph traversal or tree-hooking/set-union problems. Graph traversal [13, 34–36] or label propagation methods [15, 28, 34, 36, 39], have a major problem where they cannot achieve high performance when graph diameters are large or a graph has many small components. The tree-hooking based methods [2, 4, 6, 7, 18, 22, 23, 25, 33, 38, 41, 43] or union-find based methods [8, 19, 26] can overcome this problem and work well on varying graph topologies.

The Shiloach-Vishkin (SV) algorithm [33] is the pioneering tree-hooking method. It is a widely known PRAM algorithm and can achieve $O(\log n)$ time on $O(n+m)$ processors. There are different kinds of improvements to the SV algorithm. Awerbuch and Shiloach (AS) [2] use a very efficient parallelization using proper computational primitives and sparse data structures. The AS algorithm only keeps the information of the current forest and the convergence criterion for AS is to check whether each tree is a star. Afforest [40] is an extension of the SV algorithm that approaches optimal work efficiency by processing subgraphs in each iteration. The LACC [3] algorithm uses linear algebraic primitives to implement connected components and is based on the PRAM AS algorithm. FastSV [43] further simplifies and optimizes LACC's tree hooking and compressing method to improve the performance.

Thrifty Label Propagation (TLP) algorithm [15] uses the skewed degree distribution of real-world graphs to develop their optimized label propagation algorithm. Sutton *et al.* [40] uses sampling to find the connected components on a subset of the edges, which can be used to reduce the number of edge inspections when running connectivity on the remaining edges.

There are some works combining different methods together to further optimize the performance. Slota *et al.* [37] developed a distributed memory multi-step method that combines parallel *BFS* and label propagation technique. The ParConnect algorithm [20] is based on both the SV algorithm and parallel breadth-first search (*BFS*). ConnectIt [14] provides a framework to provide different sampling strategies and tree hooking and compression schemes.

Deng *et al.* works on exploiting the existence of high-degree nodes to create hubs to extract connected component sizes and numbers [12]. Wu and Du use label propagation to compute the connected components of graphs built on top of MapReduce [42]. Another similar work by Sharma and Misra also utilizes MapReduce to compute connected components by reducing the upper bound of MapReduce rounds down from linear to logarithmic [32].

We formulate the connected components as a minimum-mapping problem instead of a tree-hooking/set union problem. Our method is flexible and simple, so it can significantly simplify the existing tree-hooking-based methods to improve practical performance.

6 CONCLUSION

Finding connected components is a fundamental graph problem, and we present a novel problem formulation method, minimum mapping, to solve the problem. Our method is very flexible and simple. Therefore, it can be implemented efficiently compared with the existing tree-hooking/set-union methods. At the same time, it can avoid the weakness of graph travel-based methods.

We have proved that our method can converge in $O(\log(d_{max}))$ time, where d_{max} is the largest diameter among all the components in a graph. The result is better than the existing $O(\log(n))$ time because d_{max} can be much smaller than n, where n is the total number of vertices.

Most noteworthy, we have integrated our method into an open-source graph package Arachne, which is an extension of an open-source framework for Python users to handle large data easily on supercomputers. Through Arachne, high level Python users can conduct large graph analytics efficiently without knowing the details of supercomputing and large data processing.

ACKNOWLEDGMENTS

We appreciate the help from the Arkouda and Chapel community when we integrated the algorithms into Arkouda. This research was funded in part by NSF grant number CCF-2109988.

REFERENCES

[1] Reyhaneh Abdolazimi, Maryam Heidari, Armin Esmaeilzadeh, and Hassan Naderi. 2022. Mapreduce preprocess of big graphs for rapid connected components detection. In *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 0112–0118.

[2] Baruch Awerbuch and Yossi Shiloach. 1987. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Comput.* 36, 10 (1987), 1258–1263.

[3] Ariful Azad and Aydın Buluç. 2019. LACC: A linear-algebraic algorithm for finding connected components in distributed memory. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2–12.

[4] David A Bader, Guojing Cong, and John Feo. 2005. On the architectural requirements for efficient execution of graph algorithms. In *2005 International Conference on Parallel Processing (ICPP'05)*. IEEE, 547–556.

[5] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 2013. *Graph Partitioning and Graph Clustering*. Contemporary Mathematics, Vol. 588. American Mathematical Society, New York. https://doi.org/10.1090/conm/588

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

[7] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An asynchronous multi-GPU programming model for irregular computations. *ACM SIGPLAN Notices* 52, 8 (2017), 235–248.

[8] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 181–192.

[9] Guy E Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 2020. Parallelism in randomized incremental algorithms. *Journal of the ACM (JACM)* 67, 5 (2020), 1–27.

[10] Ka Wong Chong and Tak Wah Lam. 1995. Finding connected components in O(log n log log n) time on the EREW PRAM. *Journal of Algorithms* 18, 3 (1995), 378–402.

[11] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms* (fourth ed.). MIT press.

[12] Ye Deng, Jun Wu, and Yue-jin Tan. 2016. A fast connected component algorithm based on hub contraction. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 000066–000069. https://doi.org/10.1109/SMC.2016.7844221

[13] 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.

[14] Laxman Dhulipala, Changwan Hong, and Julian Shun. 2020. Connectit: A framework for static and incremental parallel graph connectivity algorithms. *arXiv preprint arXiv:2008.03909* (2020).

[15] Mohsen Koohi Esfahani, Peter Kilpatrick, and Hans Vandierendonck. 2021. Thrifty label propagation: Fast connected components for skewed-degree graphs. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 226–237.

[16] Andrew Gamble. [n. d.]. Survey of Strongly Connected Components Algorithms. ([n. d.]).

[17] Costantino Grana, Daniele Borghesani, and Rita Cucchiara. 2010. Optimized block-based connected components labeling with decision trees. *IEEE Transactions on Image Processing* 19, 6 (2010), 1596–1609.

[18] John Greiner. 1994. A comparison of parallel algorithms for connected components. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*. 16–25.

[19] Jayadharini Jaiganesh and Martin Burtscher. 2018. A high-performance connected components implementation for GPUs. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 92–104.

[20] 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.

[21] David R Karger, Noam Nisan, and Michal Parnas. 1992. Fast connected components algorithms for the EREW PRAM. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. 373–381.

[22] S Cliff Liu and Robert E Tarjan. 2018. Simple concurrent labeling algorithms for connected components. *arXiv preprint arXiv:1812.06177* (2018).

[23] Ke Meng, Jiajia Li, Guangming Tan, and Ninghui Sun. 2019. A pattern based algorithmic autotuner for graph processing on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 201–213.

[24] Michael Merrill, William Reus, and Timothy Neumann. 2019. Arkouda: interactive data exploration backed by Chapel. In *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*. 28–28.

[25] Sreepathi Pai and Keshav Pingali. 2016. A compiler for throughput optimization of graph algorithms on GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 1–19.

[26] Md Mostofa Ali Patwary, Peder Refsnes, and Fredrik Manne. 2012. Multi-core spanning forest algorithms using the disjoint-set data structure. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 827–835.

[27] Michael J Quinn and Narsingh Deo. 1984. Parallel graph algorithms. *ACM Computing Surveys (CSUR)* 16, 3 (1984), 319–348.

[28] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. 2007. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E* 76, 3 (2007), 036106.

[29] William Reus. 2020. CHIUW 2020 Keynote Arkouda: Chapel-Powered, Interactive Supercomputing for Data Science. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 650–650.

[30] Oliver Alvardo Rodriguez, Zhihui Du, Joseph T. Patchett, Fuhuan Li, and David A. Bader. 2022. Arachne: An Arkouda Package for Large-Scale Graph Analytics. In *The 26th Annual IEEE High Performance Extreme Computing Conference (HPEC), Virtual, September 19-23, 2022*.

[31] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2020. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB journal* 29, 2 (2020), 595–618.

[32] Amar Sharma and Rajiv Misra. 2016. Computing Large Connected Components Using Map Reduce in Logarithmic Rounds. In *2016 IEEE 2nd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International*

Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS). 1–6. https://doi.org/10.1109/BigDataSecurity-HPSC-IDS.2016.18

[33] Yossi Shiloach and Uzi Vishkin. 1982. An O(logn) parallel connectivity algorithm. *Journal of Algorithms* 3, 1 (1982), 57–67. https://doi.org/10.1016/0196-6774(82)90008-6

[34] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[35] Julian Shun and Guy E Blelloch. 2014. A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction. *ACM Transactions on Parallel Computing (TOPC)* 1, 1 (2014), 1–20.

[36] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2014. BFS and coloring-based parallel algorithms for strongly connected components and related problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 550–559.

[37] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2016. A case study of complex graph analysis in distributed memory: Implementation and optimization. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 293–302.

[38] Jyothish Soman, Kothapalli Kishore, and PJ Narayanan. 2010. A fast GPU algorithm for graph connectivity. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, 1–8.

[39] Stergios Stergiou, Dipen Rughwani, and Kostas Tsioutsiouliklis. 2018. Shortcutting label propagation for distributed connected components. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*. 540–546.

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

[41] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T Riffel, et al. 2017. Gunrock: GPU graph analytics. *ACM Transactions on Parallel Computing (TOPC)* 4, 1 (2017), 1–49.

[42] Bin Wu and YaHong Du. 2010. Cloud-based Connected Component Algorithm. In *2010 International Conference on Artificial Intelligence and Computational Intelligence*, Vol. 3. 122–126. https://doi.org/10.1109/AICI.2010.360

[43] Yongzhe Zhang, Ariful Azad, and Zhenjiang Hu. 2020. FastSV: A distributed-memory connected component algorithm with fast convergence. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 46–57.