Contour Algorithm for Connectivity

Zhihui Du, Oliver Alvarado Rodriguez, Fuhuan Li, Mohammad Dindoost and David A. Bader
Department of Data Science
New Jersey Institute of Technology
Newark, USA
{zhihui.du,oaa9,fl28,md724,bader}@njit.edu

***Abstract*—Finding connected components in a graph is a fundamental problem in graph analysis. In this work, we present a novel minimum-mapping based *Contour* algorithm to efficiently solve the connectivity problem. We prove that the *Contour* algorithm with two or higher order operators can identify all connected components of an undirected graph within $\mathcal{O}(\log d_{max})$ iterations, with each iteration involving $\mathcal{O}(m)$ work, where d_{max} represents the largest diameter among all components in the given graph, and m is the total number of edges in the graph. Importantly, each iteration is highly parallelizable, making use of the efficient minimum-mapping operator applied to all edges. To further enhance its practical performance, we optimize the *Contour* algorithm through asynchronous updates, early convergence checking, eliminating atomic operations, and choosing more efficient mapping operators. Our implementation of the *Contour* algorithm has been integrated into the open-source framework Arachne. Arachne extends Arkouda for large-scale interactive graph analytics, providing a Python API powered by the high-productivity parallel language Chapel. Experimental results on both real-world and synthetic graphs demonstrate the superior performance of our proposed *Contour* algorithm compared to state-of-the-art large-scale parallel algorithm FastSV and the fastest shared memory algorithm ConnectIt. On average, *Contour* achieves a speedup of 7.3x and 1.4x compared to FastSV and ConnectIt, respectively. All code for the *Contour* algorithm and the Arachne framework is publicly available on GitHub [1], ensuring transparency and reproducibility of our work.**

***Index Terms*—connected components, graph analytics, big data, parallel algorithm**

I. 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.

Finding connected components [4], [6], [13], [15] 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 [7].

[1]https://github.com/Bears-R-Us/arkouda-njit

There are three kinds of algorithms for identifying connected components of an undirected graph. The first is a graph traversal-based method. Breadth-First Search (*BFS*) [14] and label propagation [11], [19], [27] are two typical examples. *BFS* will search from a set of just visited vertices (current frontier, initially with one root vertex) and then extend to other unvisited vertices (next frontier) connected to visited vertices until all vertices are visited. The basic idea of label propagation is that each vertex is initially assigned a unique label. Then, each vertex subsequently compares its label with the labels of its neighbors and updates its label to be the smallest among them. This process is repeated until no label can be updated. There are many variants to improve the performance further. This method has high performance for low-diameter graphs. However, if a graph has a long diameter, a lot of time and iterations will be needed.

The second is the tree hooking-compressing-based method [2], [16], [22], [30]. This kind of method will start by initializing all vertices as singletons. Then, some tree hooking operations are employed to merge smaller components into larger components. Compressing operations will reduce the tree's height until all vertices are directly connected to a root vertex. The major feature of such a method is formulating the discovery of a big component as a forest building. Combining the tree hooking and compressing, a much smaller number of iterations will be needed to identify all the components, even if the given graph has a large diameter.

The third is the union-find or disjoint set-based method [10], [12], [18]. It models components as disjoint sets. The union operation will merge different sets and the find operation will return the representative member of a set. Unlike the previous two methods, union-find is not an iteration-based method. It can directly identify all connected components in one iteration of the tree-based method. However, for large-scale parallelism scenarios, union-find methods often lead to an unbalanced workload that can significantly affect their performance.

We abstract the connectivity as a contour lines discovery problem and develop simple and lightweight minimum-mapping operators to work on different edges to efficiently identify all the components in parallel. The minimum-mapping operator can map the connected vertices to the same contour line. Identifying one component is similar to identifying one contour line with the same minimum mapping label. Therefore, we name our algorithm "*Contour*" [8]. The minimum-mapping operator can be employed on different edges in parallel with high efficiency. Compared to tree hooking-compressing or union-find-based methods, this can significantly improve parallel performance and simplify implementation.

The major contributions of this work are as follows.

1) A novel *Contour* algorithm that formulates finding connected components as a contour lines discovery problem. Based on this perspective, simple and lightweight minimum-mapping operators are 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, the *Contour* algorithm can converge in $\mathcal{O}(log(d_{max}))$ 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 to state-of-the-art real-world and synthetic graphs methods.

II. Contour Algorithm

A. 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 $n-1$.

A label array $L[0..n-1]$ with size n can be used to store all the labels of different vertices. Initially, we assign each vertex's ID as its label. Identifying all connected components in G means that we will assign the vertices of the same components with the same vertex label. The label array is also regarded as a pointer graph [22]. $\forall v \in V, L[i]=v$ means that there is a direct edge from vertex i to v. The pointer graph will be updated 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. A star here is a unique type of graph characterized by a single root vertex connected to all other vertices, with no additional edges present.

B. 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 once iteration. 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 label. Here 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 1 (Conditional Vector Assignment)**.**

$$\begin{bmatrix} x_1 \\ ... \\ x_k \end{bmatrix} \xleftarrow{>} z. \tag{1}$$

It means that given a vector $X=\begin{bmatrix} x_1 \\ ... \\ x_k \end{bmatrix}$ *,* $\forall i \in \mathbb{N}, 1 \le i \le k$*,* $x_i = z$ *if* $x_i > z$.

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

Definition 2 (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.*

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

$MM^1(L_u, L, w, v)$ means that before the mapping operator, $L_u = L$. After employing the mapping operator, $L_u[w]$ and $L_u[v]$ will be updated if either of them is larger than z^1.

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

Definition 3 (h-Order Minimum-Mapping Operator)**.** *Given two connected vertices* $w, v \in V$*, let* $z^h = min(L^h[w], L^h[v])$*, where* $\forall x \in V, L^h[x] = L[L^{h-1}[x]], L^1[x] = L[x]$*. We define the h-order minimum-mapping operator as follows.*

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

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 quickly. 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 smaller number of vertices and operations. We will also show the different effects of its variants and combination patterns in Section IV.

C. Algorithm Description

Based on the proposed minimum-mapping operator in Section II-B, 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 convergence 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 races, we can use the atomic compare-and-swap (CAS) [2] operation to implement our conditional assignment as follows.

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

Algorithm 1: Minimum-Mapping based *Contour* Algorithm

```
   $Contour(G)$
   /* $G = \langle E, V \rangle$ is the input graph with edge
      set $E$ and vertex set $V$.                    */
1  forall i in 0..n-1 do
2      $L[i] = i$
3      $L_u[i] = i$
4  end
   /* Initialize the label array $L, L_u$            */
5  do
6      forall ($e = \langle w, v \rangle \in E$) do
7          $MM^2(L_u, L, w, v)$
8      end
9      $L = L_u$
10 while (There is any label change in L)
11 return $L$
```

Let $L_k^h[x]$ be the label of vertex x employing the h-order minimum-mapping operator after the k^{th} iteration, then $L_{u,k}[w] = min(L_{k-1}^2[w], L_{k-1}^2[v_1], L_{k-1}^2[v_2], ..., L_{k-1}^2[v_m]$, 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. We give the following definition to show how the vertices in the same component are mapped to the same minimum label step by step.

Definition 4 (Equal Minimum Set)**.** *Given label x, after the k^{th} iteration, its one-order equal minimum set $EMS(k)_x^1 = \{v | \forall v \in V, L_k[v] = x\}$. Its two-order equal minimum set $EMS(k)_x^2 = \{v | \forall v \in V, L_k^2[v] = x\}$.*

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

Definition 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 $MMS(k)^2 = \{v | \forall v \in V, EMS(k)_v^2 \neq \phi\}$.*

From the definition, we can see that for $k \geq 0, MMS(k)^1 \supseteq MMS(k)^2 \supseteq MMS(k+1)^1$. In other words, the merged minimum set's size will become smaller until it only contains the minimum vertices of different connected components.

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

Definition 6 (Rooted Tree and its Neighbor)**.** *After the k^{th} iteration, $k \geq 1$, the root vertices of different root trees in the pointer graph $R(t) = \{v | v \in MMS(k) \wedge L_k[v] = v\}$. $R(t)$ is also called the root tree set of the pointer graph. $\forall v_1, v_2 \in R(t)$, if $\exists \langle v_{1'}, v_{2'} \rangle \in E$, and $v_{1'}$ belongs to in root tree v_1, $v_{2'}$ belongs to root tree v_2, then we call v_1 the neighbor of v_2, vice versa.*

Our mapping operator has the following two effects on the rooted trees. (1) Compressing. If the original height of a rooted tree is x and we employ h order minimum mapping operator to it, its height will be reduced to no more than $\lfloor \frac{x+h-1}{h} \rfloor$. Every vertex in the rooted tree will point to its h order father or the root. (2) Minimum Merging. Any vertex v_m in one rooted tree may be merged into its neighbor rooted tree as the son of root or other vertices. At the same time, the subtree (if exists) with v_m as its root will be compressed and merged into its neighbor-rooted tree. Both compressing and merging can happen at the same time in one minimum-mapping operation. Minimum merging is very flexible and different from the existing tree-hooking or set union methods. One rooted tree can merge part of another rooted tree instead of the complete rooted tree. At the same time, it is simple and easy to implement.

The framework of Alg. 1 has some similarities to label propagation or tree hooking-compressing. However, the label propagation method can be regarded as a special case of our method when the mapping order is one. Compared with the existing tree hooking-compressing methods, they only allow merging two rooted trees. However, our method can merge any part of two rooted trees.

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

D. Time complexity analysis

Lemma 1 (Root Tree Constraint)**.** *Let $P = \langle s_0, \ldots, s_{n-1} \rangle, n \geq 2$, be a path with s_0 as the smallest vertex, and consider running Alg. 1 on P (here we assume the mapping operator can be employed up to twice in each iteration). After the k^{th} iteration, let the root tree set be $R(k)$, we have $(\frac{3}{2})^{k-1} \sum_{v \in R(k)} H_k(v) \leq n-1$, where $H_k(v)$ is the height of root tree v after the k^{th} iteration.*

Proof. Let's do induction on k.

For $k = 1$, if all vertices are in increasing order along the path P, then $MMS(1)^1 = s_0$ and $H_1(s_0) = n-1$, $\sum_{v \in R(1)} H_1(v) = n-1$. So, the inequation holds.

Otherwise, if there are multiple root trees $v_{m_1}, ..., v_{m_j}$ in the pointer graph and we let n_s be the number of vertices in root tree v_{m_s}, where $1 \leq s \leq j$. So, we have $\sum_{s=1}^{s=j} n_s = n$ and $H_1(v_{m_s}) \leq n_s - 1$. Therefore, $\sum_{s=1}^{s=j} H_1(v_{m_s}) \leq n - j < n-1$. So, the inequality holds for the base case.

Let $t \geq 1$; we assume that when $t = k$, the inequation holds. Now we prove when $k = t+1$, the inequation also holds. If $R(t) = \{s_0\}$ and $H_t(s_0) > 1$, then after the $(t+1)$ iteration, $H_{t+1}(s_0) \leq (\frac{2}{3}) H_t(s_0)$, so the inequation holds.

We discuss two cases if $|R(t)| > 1$.

(1) If $\forall v \in R(t)$, $H_t(v) > 1 \wedge v \in R(t+1)$, then $H_{t+1}(v) \leq (\frac{2}{3})H_t(v)$.

(2) If $\exists v \in R(t) \wedge H_t(v) = 1 \wedge L_{k+1}[v] = v' \wedge v \neq v'$, then root tree v will be merged into the root tree v_m that contains vertex v' after the $(t+1)^{th}$ iteration. If $H_k(v_m) = 1$, then $H_{k+1}(v_m) = 1 < \frac{2}{3}(H_k(v) + H_k(v_m)) = \frac{2}{3} \times 2 = \frac{4}{3}$. If $H_k(v_m) > 1$, we know that $H_{t+1}(v_m) \leq \frac{2}{3}H_t(v_m) < \frac{2}{3}(H_t(v_m) + H_t(v))$.

If $\exists v \in R(t) \wedge H_t(v) = 1 \wedge L_{k+1}[v] = v$, it means that v is less than its neighbour vertex $v_x \in R(t)$. So, v will merge its neighbor root tree or partial vertices of its neighbor root tree. Since merging the complete root tree is the same as in the above case, we only consider the case when only partial vertices are merged into v. In this case, the neighbor root tree v_n must have $H_t(v_n) > 1$. Otherwise, the neighbor root tree will be merged into the v root tree. Here, we can employ the mapping operator twice. If $H_{t+1}(v_n) = 1$, then $H_{t+1}(v_n) + H_{t+1}(v) = 2$. We have $\frac{2}{3}(H_t(v_n) + H_t(v_n)) \geq \frac{2}{3}(1+2) = 2$. If $H_{t+1}(v_n) > 1$, then we have $H_{t+1}(v_n) \leq \frac{1}{3}H_t(v_n)$ so $H_{t+1}(v_n) + 1 \leq \frac{2}{3}(H_t(v_n) + 1)$ when $H_{t+1}(v_n) > 1$.

Hence, considering all the cases, we also have the same conclusion. □

Lemma 2 (Path Convergence)**.** *Let $P = \langle s_0, \ldots, s_{n-1} \rangle, n \geq 2$, be a path with s_0 as the smallest vertex, and consider running Alg. 1 on P. Marking all vertices as s_0 will need at most $\lceil \log_{\frac{3}{2}}(n-1) \rceil + 1$ iterations.*

Proof. Based on Lemma 1, $k \leq \lceil \log_{\frac{3}{2}} \frac{(n-1)}{\sum_{v \in R(k)} H_k(v)} \rceil + 1$, when $\sum_{v \in R(k)} H_k(v) = 1$, the maximum value of k should be $\lceil \log_{\frac{3}{2}}(n-1) \rceil + 1$. So, after at most $\lceil \log_{\frac{3}{2}}(n-1) \rceil + 1$ iterations, all vertices on P will be marked as s_0. □

Lemma 3 (Diameter Convergence)**.** *For a connected graph G with diameter d, Alg. 1 will take at most $(\lceil \log_{\frac{3}{2}}(d) \rceil + 1)$ iterations to spread the minimum vertex label to all the other vertices.*

Proof. Let the smallest vertex in G be s_0, then all shortest paths from s_0 to other vertices cannot be larger than d. Based on Lemma 2, the vertices on any shortest path from s_0 to other vertices can be mapped to s_0 within $(\lceil \log_{\frac{3}{2}}(d) \rceil + 1)$ iterations. So the conclusion holds. □

Theorem 1 (Graph Convergence)**.** *For any graph G, let d_{max} be the maximum diameter of all graph G's components. Alg. 1 will take at most $(\lceil \log_{\frac{3}{2}}(d_{max}) \rceil + 1)$ iterations to identify all the components.*

Proof. Let G_c be any connected component of G and d be its diameter with $d \leq d_{max}$. Based on Lemma 3, we know that after $(\lceil \log_{\frac{3}{2}}(d) \rceil + 1)$ iterations, all vertices in G_c will be mapped to their minimum vertices. Since d_{max} is the maximum diameter of all graph G's components, after $\lceil \log_{\frac{3}{2}}(d_{max}) \rceil + 1)$ iterations, all connected components of G must have been mapped to their minimum vertices. So, Alg. 1 will take at most $\lceil \log_{\frac{3}{2}}(d_{max}) \rceil + 1)$ iterations to identify all the components. □

III. Integration with Arachne and Performance Optimization

A. Integration Method

Our method is integrated into Arachne [21], a large-scale graph analytics package on top of Arkouda [17], [20]. Arkouda is an open-source framework in Python created to be a NumPy replacement at scale. It replaces the *ndarray* abstraction with the *pdarray*. 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 [3]. 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.

B. Algorithm Optimization

Alg. 1 presents the fundamental concept of our method. However, we can further optimize it to enhance its practical performance when we integrate the method into Arachne.

1) Asynchronous Update: The essence of the asynchronous *Contour* algorithm is to update the label array L immediately, eliminating the need for maintaining an update label array L_u. An asynchronous update will not affect the correctness or final convergence of the algorithm. However, the practical performance will be very different. This approach offers several advantages:

(1) Faster convergence speed: Vertices can be mapped to lower labels more rapidly.

(2) Reduction of unnecessary operations: The step $L = L_u$ in Alg. 1 becomes unnecessary and can be removed.

(3) Memory usage reduction: The L_u array is no longer required and can be eliminated.

Experimental results in Section IV demonstrate that asynchronous updates significantly improve the performance of the algorithm.

2) Early Convergence Check: With the definition of our minimum mapping operator, for any edge $e = (v, w) \in E$, if $(L[v] \neq L^2[v] || L[w] \neq L^2[w] || L[v] \neq L[w])$, we need to continue to the next iteration. However, if these conditions are not met, even if there are updates in the current iteration, we can confidently conclude that the algorithm has converged, and we can exit the iteration directly. This early convergence check allows us to save additional iterations.

By performing this convergence check, we can efficiently terminate the algorithm once the convergence condition is met, reducing unnecessary computations and improving the overall efficiency of the algorithm.

[3]https://zeromq.org/

3) Eliminating Atomic Operations: In union-find algorithms, atomic operations are essential to ensure correctness. However, in iteration-based methods, atomic operations can impact the number of iterations but not the correctness of the algorithm. These atomic operations can be computationally expensive compared to simple assignments.

Utilizing asynchronous updates can accelerate the convergence speed and reduce the total number of iterations. Consequently, we have the opportunity to replace costly atomic updates with simple assignments, further enhancing the practical performance of the algorithm.

By removing atomic operations and employing simple assignments, we can achieve better computational efficiency without compromising the correctness of the algorithm. It is similar to the effect of replacing synchronization updates with asynchronous updates. These optimizations contribute to the overall improvement in practical performance.

4) Selecting Suitable Minimum Mapping Operators: The choice of minimum mapping operators and their combination patterns can also significantly impact the performance of the algorithm for a given graph. We will provide recommendations based on experimental results in subsection IV-E. Here, we introduce six different variants of our *Contour* algorithm:

C-Syn: This is the synchronous method described in Alg. 1 without employing any other optimization methods. Except for the minimum mapping operator, it is almost the same as the *FastSV* algorithm. It can only achieve limited speedup compared with *FastSV*.

C-1: This variant employs the one-order minimum mapping operator.

C-2: This variant employs the two-order minimum mapping operator.

C-m: For large-diameter graphs, we may use a higher-order minimum mapping operator greater than two to reduce the total number of iterations.

C-11mm: This variant combines operators. It starts with the one-order mapping operator for a few iterations and then switches to a higher-order operator until convergence.

C-1m1m: This variant alternates between the one-order and higher-order operators until convergence.

In subsection IV-E, we will provide specific analysis and guidance on selecting the most suitable variant based on the characteristics of the graph to achieve optimal performance.

C. State-of-the-Art Algorithms

In addition to our *Contour* algorithm, we have incorporated two state-of-the-art algorithms, namely *FastSV* [30] and *ConnectIt* [10], into Arachne. This integration allows us to expand our algorithm repository and conduct performance comparisons.

The seminal Shiloach-Vishkin (SV) algorithm [22] is capable of identifying graph components in $\mathcal{O}(\log(n))$ time on a CRCW PRAM machine with $(m+n)$ parallel processors. Various variants and improvements [3] based on the original SV algorithm exist, and *FastSV* represents the latest and most advanced version for large-scale parallel systems. However, the synchronization introduced between different hooking and compressing steps, along with the need to assign the current label array L with the updated label array L_u before the next iteration, significantly affects its performance compared to our simple and flexible minimum mapping operators.

Union-find algorithms were developed to handle disjoint set data structures and can achieve almost linear time complexity [29]. Patwary *et al.*'s [18] experimental results reveal that Rem's simple union-find algorithm delivers superior practical performance. Dhulipala *et al.* developed the *ConnectIt* framework, which incorporates hundreds of different connected components algorithms, and their experimental results on large graphs demonstrate that Rem's algorithm is the best within their shared memory system. In Arachne, we have integrated the optimal union-find algorithm from the ConnectIt framework for comparison. Our experimental results (see subsection IV-F) demonstrate that our *Contour* algorithm can effectively exploit parallel resources to achieve improved performance.

IV. Experiments

A. Dataset Description

Our dataset comprises a selection of publicly available synthetic and real-world datasets, sourced from reputable repositories. We have drawn graphs from the SuiteSparse Matrix Collection[4], Stanford Large Network Dataset Collection[5], and the MIT GraphChallenge graph datasets[6].

To ensure comprehensive testing and performance comparison, we have carefully chosen a combination of real-world and synthetic graphs, as outlined in Table I. Including both types of graphs allows us to highlight and evaluate the efficacy of our *Contour* algorithm compared with state-of-the-art algorithms.

The real-world graphs in our dataset may vary in characteristics. They typically exhibit degree distributions that follow a power-law distribution. These features provide a diverse set of challenges and scenarios to thoroughly assess the algorithms' performance.

Additionally, we have included a set of synthetic graphs known as *Delaunay*. These graphs are constructed based on Delaunay triangulations of randomly generated points in the plane. Unlike graphs with power law distribution, *Delaunay* graphs have vertices with degrees that are relatively close to each other. Including synthetic graphs allows us to observe how the algorithm's performance varies with graph size.

By incorporating a diverse range of datasets, our evaluation encompasses various real-world scenarios and provides valuable insights into the scalability and effectiveness of our *Contour* algorithm, as well as its comparison with state-of-the-art methods.

B. Experimental Platform

Experiments were done on a 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 @

[4]https://sparse.tamu.edu/
[5]https://snap.stanford.edu/data/
[6]https://graphchallenge.mit.edu/data-sets

TABLE I: Real World and Synthetic graphs

Graph Name	Graph ID	Number of Edges	Number of Vertices
ca-GrQc	0	28980	5242
ca-HepTh	1	51971	9877
facebook_combined	2	88234	4039
wiki	3	103689	8277
as-caida20071105	4	106762	26475
ca-CondMat	5	186936	23133
ca-HepPh	6	237010	12008
email-Enron	7	367662	36692
ca-AstroPh	8	396160	18772
loc-brightkite_edges	9	428156	58228
soc-Epinions1	10	508837	75879
com-dblp	11	1049866	317080
com-youtube	12	2987624	1134890
amazon0601	13	2443408	403394
soc-LiveJournal1	14	68993773	4847571
higgs-social_network	15	14855842	456626
com-orkut	16	117185083	3072441
road_usa	17	28854312	23947347
kmer_A2a	18	180292586	170728175
kmer_V1r	19	232705452	214005017
uk_2002	20	298113762	18520486
delaunay_n10	21	3056	1024
delaunay_n11	22	6127	2048
delaunay_n12	23	12264	4096
delaunay_n13	24	24547	8192
delaunay_n14	25	49122	16384
delaunay_n15	26	98274	32768
delaunay_n16	27	196575	65536
delaunay_n17	28	393176	131072
delaunay_n18	29	786396	262144
delaunay_n19	30	1572823	524288
delaunay_n20	31	3145686	1048576
delaunay_n21	32	6291408	2097152
delaunay_n22	33	12582869	4194304
delaunay_n23	34	25165784	8388608
delaunay_n24	35	50331601	16777216

2.30GHz CPUs with ten cores per CPU. Each server has 512GB of RAM. A high-performance Infiniband network system connects all nodes.

C. Number of Iterations

In Fig. 1, we observe that for different graphs, the *C-1* operator consistently requires the largest total number of iterations. Notably, Graph 17 *road_usa* exhibits the highest iteration count at 2369 iterations. This behavior is expected as *C-1* represents the lowest-order minimum mapping operator, only considering directly connected vertices or those within a search distance of 1.

Comparatively, *C-2* performs significantly better than *C-1* in iteration numbers, involving all vertices that are at a distance of 2 from each edge $e = (v, w)$. Consequently, even a minimal increase in the order of the minimum mapping operator leads to a significant reduction in the total number of iterations for long-diameter graphs. For instance, Graph 17 *road_usa* only requires 5 iterations when using *C-2*.

Further increasing the minimum mapping order to *C-m* (here $m = 1024$) yields additional reductions in the total number of iterations, but the improvement is not as significant. Across all graphs, *C-m* achieves a maximum reduction of 3 iterations compared to *C-2*. Therefore, we observe the following relationship for the total number of iterations: *Number of Iterations (C-m)* $\leq$ *Number of Iterations (C-2)* $\leq$ *Number of Iterations (C-1)*.

Next, we analyze the behaviors of the combined minimum mapping operators *C-11mm* and *1m1m*. Among the 38 graphs, the majority (21) exhibit the same number of iterations for both operators. For the remaining 13 graphs, *1m1m* shows a slightly higher number of iterations than *C-11mm*. Thus, *C-11mm* generally demonstrates a slightly better performance than *1m1m* in terms of iteration count. Additionally, *C-11mm* exhibits a total iteration count that is close to *C-2*.

Comparing *C-Syn* with *FastSV*, we find them to be quite similar in terms of the total number of iterations. However, *C-Syn* possesses a more efficient and simplified minimum mapping operator, contributing to the slight advantage in iteration count over *FastSV*. The optimized *C-2* operator significantly reduces the number of iterations compared to *C-Syn* in most cases, validating the effectiveness of our optimization in reducing iterations, as also reflected in subsection IV-D.

ConnectIt, as a non-iteration-based method, requires one union operation on all edges and one compression operation on all vertices. Consequently, we assign the iteration count for *ConnectIt* as 1 for all graphs.

In summary, the average number of iterations, from low to high, are as follows: *C-m*=2.19, *C-2*=3.19, *C-11mm*=3.89, *C-1m1m*=4.31, *C-Syn*=6.83, *FastSV*=6.97, *C-1*=83.86.

D. Execution Time

In Fig. 2, we observe the execution times of different methods. Notably, there is a general trend that as the size of graphs increases (measured by the total number of edges and vertices), the execution time also increases. This pattern is expected since our server has a fixed number of 20 cores. As the graph size grows, each core has to handle a larger number of edges, leading to increased execution times. However, the execution times may vary due to differences in graph topology.

Analyzing the Delaunay graphs, which share similar topology, we find that as the graph size grows from *delaunay_n10* to *delaunay_n24* (both the number of edges and vertices increase about 16000 times), the execution time of *C-2* increases by 895 times, *C-1m1m* increases by 1072 times, *C-m* increases by 1268 times, *ConnectIt* increases by 1303 times, *C-11mm* increases by 1329 times, *C-Syn* increases by 2705 times, and *FastSV* increases by 4096 times.

Additionally, we observe that, in most cases, *FastSV* exhibits longer execution times compared to all other methods. Only when the diameters of some graphs are particularly large does the execution time of *C-1* surpass *FastSV*. Moreover, *C-Syn* consistently shows longer execution times compared to other *Contour* variants. As mentioned previously, this is due to *C-Syn* employing synchronous updates instead of immediate asynchronous updates, which hinders the quick spreading of small labels to other vertices, thereby reducing its convergence speed.

In summary, the execution times of the algorithms generally follow the trend of increasing with graph size. However, specific algorithm characteristics, such as synchronous vs. asynchronous updates, also play a significant role in determining execution times. The overall performance of our *Contour* algorithm outperforms *FastSV*, highlighting the effectiveness of our optimization strategies in reducing execution times.

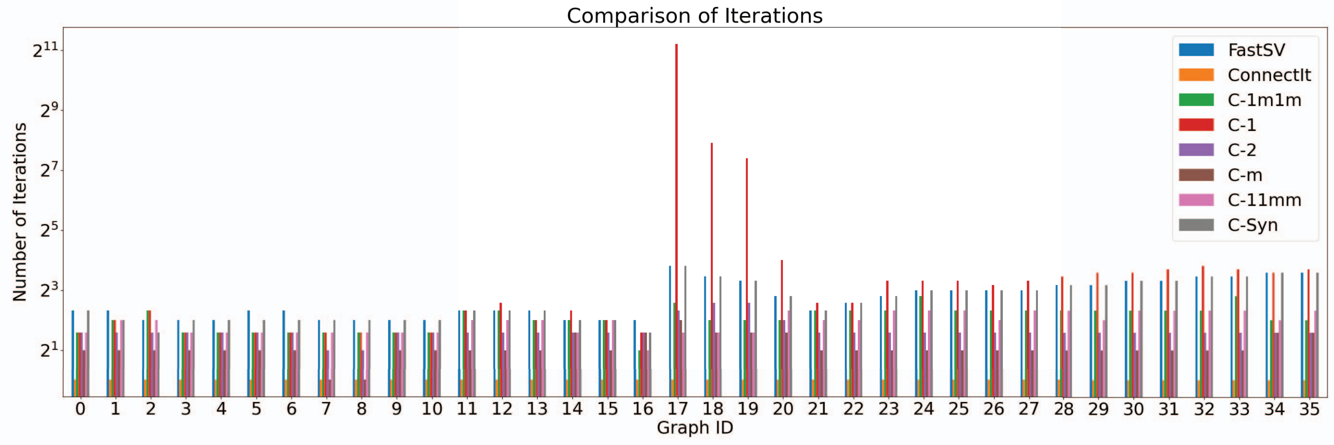

Fig. 1: Number of Iterations of *FastSV*, *ConnectIt*, and Different *Contour* Variants.

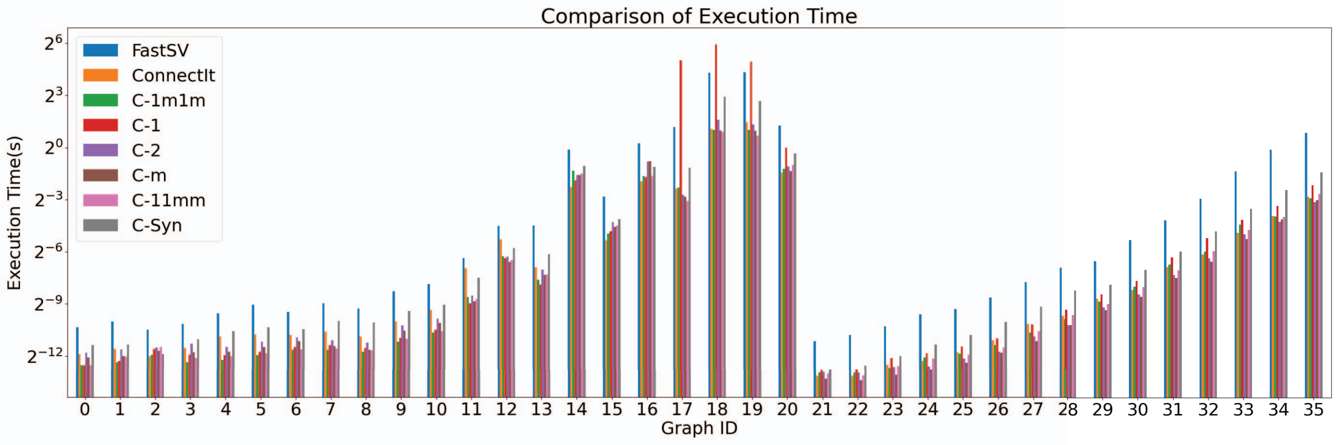

Fig. 2: Execution Time of *FastSV*, *ConnectIt*, and Different *Contour* Variants.

E. Speedup compared with FastSV

In Fig. 3, we observe the speedups of all the methods compared to *FastSV*. The average speedups, from highest to lowest, are as follows: *C-m* with a speedup of 7.3, *C-11mm* with 6.6, *ConnectIt* with 6.49, *C-1m1m* with 6.33, *C-2* with 6.33, *C-1* with 4.62, and *C-Syn* with 2.87. This indicates that high-order minimum mapping operators often perform significantly better than *FastSV*.

However, *C-1* shows particularly good speedup when the size and diameter of the graphs are small. The reason behind this behavior is that when the diameter of a graph is small, *C-1* can converge quickly within a few iterations. Additionally, for each iteration, the total workload for each core is very low because of the small graph size and *C-1*'s focus on only checking one-path neighbors. This operation exhibits excellent locality, which can be explained clearly by the work-depth model [5] well. Consequently, under these conditions, *C-1* achieves better performance. However, for larger graphs or graphs with higher diameters, *C-1* cannot maintain this better speedup compared to other variants due to a higher overall workload or larger number of iterations.

C-m achieves the best average speedup, but it may not be suitable for all cases because each iteration will have a higher cost. As mentioned above, *C-1* excels in scenarios with small diameters and sizes, while *C-m* is most effective for large-diameter or large-size graphs. It reduces the total number of iterations to minimize the overall cost.

C-2, on the other hand, exhibits a relatively small cost in each iteration as it only checks reachable vertices within two steps. Simultaneously, it can significantly reduce the total number of iterations for graphs with large diameters. Thus, *C-2* stands as a stable and simple operator that fits well in most cases.

C-1m1m is also a stable operator, but its policy differs from *C-2*. It alternates between two extreme operators, *C-1* and *C-m*. *C-1* reduces the cost of each iteration, while *C-m* focuses on minimizing the total number of iterations. Combining these two operators optimizes the overall performance.

The strategy behind *C-11mm* is different. It attempts to

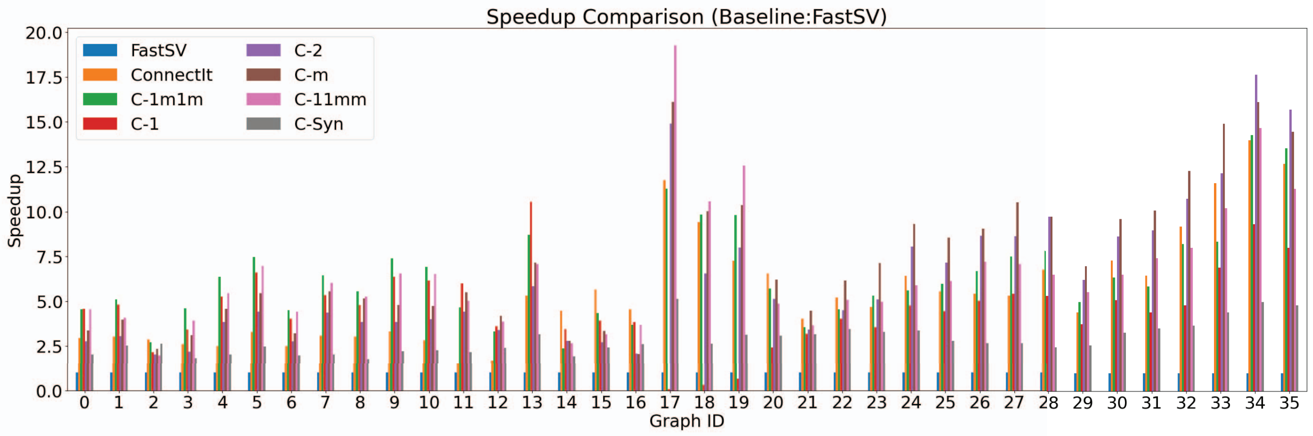

Fig. 3: Speedups of *ConnectIt* and Different *Contour* Variants compared with *FastSV*.

handle graphs with the smallest cost first. If, after several iterations, the graph does not converge, *C-11mm* employs the *C-m* operator to reduce the total number of iterations rapidly. When a graph contains both very small and very large diameter components, *C-11mm* quickly converges the small diameter components with minimal cost before efficiently handling the large diameter components using the *C-m* operator.

In summary, the speedup of the algorithms compared to *FastSV* exhibits variations based on the size and characteristics of the graphs. Different operators are more suitable for different scenarios, depending on graph size, and diameter. The overall performance of our *Contour* algorithm outperforms *FastSV* in many cases, particularly when utilizing high-order minimum mapping operators, validating the effectiveness of our approach.

F. Speedup compared with ConnectIt

In Fig. 4, we examine the speedups of our *Contour* algorithm compared to another state-of-the-art algorithm, *ConnectIt*. We will expose another perspective that can significantly affect the performance of different algorithms.

Across the 36 graphs, *C-m* outperforms *ConnectIt* on 31 graphs, with an average speedup of 1.41. Similarly, *C-2* achieves better performance on 26 graphs, with an average speedup of 1.2. Both *C-1m1m* and *C-11mm* outperform *ConnectIt* on 23 graphs, with average speedups of 1.37 and 1.35, respectively. All of these *Contour* variants achieve better performance on more than half of the graphs. *C-1* shows better performance on 14 graphs, with an average speedup of 1.11. *C-Syn*, on the other hand, only outperforms *ConnectIt* on 2 graphs, with an average speedup of 0.62.

The experimental results provide valuable insights into when to use *Contour* algorithms and when to use *ConnectIt* to achieve better performance. In general, when we have a sufficient number of parallel cores to significantly reduce the cost of one iteration, employing our *Contour* algorithm will lead to better performance. The *Contour* algorithm's efficiency lies in its ability to reduce the total number of iterations and workload per iteration, resulting in overall speedup. However, if the graph size is very large, and the number of parallel cores is relatively small, each core will have to handle a considerable number of edges in each iteration, limiting the parallel effect. This is very similar to sequential instead of parallel computing. In such scenarios, the performance improvement is driven by high efficiency instead of high scalability because the system cannot provide sufficient parallel resources, where *ConnectIt* excels with almost linear time complexity, approaching optimality. Thus, *ConnectIt* can achieve better performance when the workload per core is significantly high or when the system lacks parallel resources.

The work-depth model can clarify these results. When the work per iteration is high and parallel resources are limited, *ConnectIt* stands as an ideal choice since it requires only one iteration. Conversely, when parallel resources can significantly reduce the work per iteration, *Contour* algorithms achieve better overall performance with their ability to tolerate more iterations.

In conclusion, the choice between *Contour* algorithms and *ConnectIt* depends on the available parallel resources or the size of different graphs. Our *Contour* algorithm demonstrates superior performance when enough parallel resources are available, but *ConnectIt* remains a suitable choice for scenarios with high workloads and limited parallel resources.

G. Distributed Memory Results

The previous sections' results were based on shared memory parallel execution. However, when we consider distributed memory parallel executions involving multiple computing nodes, the absolute execution times become much longer. In practical scenarios, using multiple distributed memory computing nodes to solve a problem with a much longer time is not reasonable if it can be handled by a single shared memory parallel node with much less time. Therefore, we just give a brief summary instead of the detailed experimental results as follows.

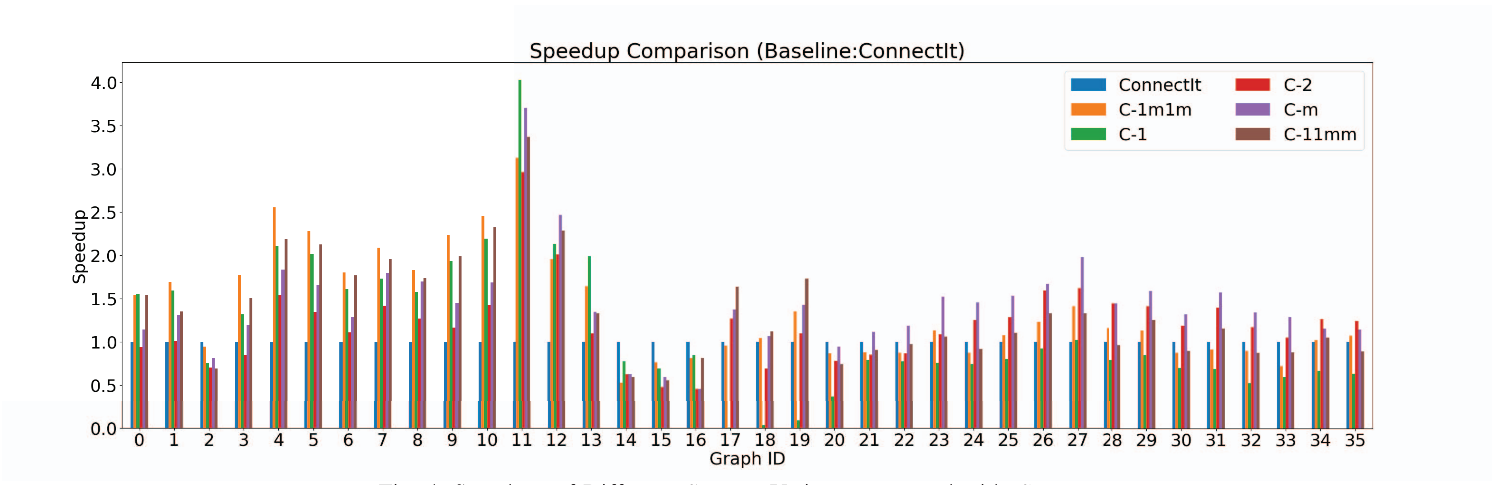

Fig. 4: Speedups of Different *Contour* Variants compared with *ConnectIt*

When comparing with *FastSV*, our *Contour* algorithm demonstrates significantly better speedup than that in the shared memory parallel node setting. Among all the variants of our *Contour* algorithm, *C-1* achieves much better speedup when the total number of iterations is relatively low. The reason for this lies in *C-1*'s ability to achieve high locality and reduce additional communication. Communication becomes a major performance bottleneck in distributed system scenarios, overshadowing computation.

Taking advantage of high-level parallel language Chapel, the shared memory *ConnectIt* algorithm can be run on distributed systems. Similarly, *ConnectIt* exhibits better relative performance compared to *Contour* when dealing with large-sized graphs. Due to relatively less communication overhead, *ConnectIt* even achieves good performance for middle-sized graphs. For small and low-diameter graphs, *C-1*, *C-11mm*, and *C-1m1m* are more efficient and offer better performance.

V. Related Work

For connected component problems, graph traversal methods [9], [23]–[25] have a major problem where they cannot achieve high performance when graph diameters are large or a graph has many small components. Label propagation methods [11], [19], [23], [25], [27] cannot converge fast when the graph diameter is large.

The Shiloach-Vishkin (SV) algorithm [22] is the pioneering tree-based hooking-compressing method to reduce the total number of iterations efficiently. There are different kinds of improvements to the SV algorithm. Awerbuch and Shiloach (AS) [1] 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 [28] is an extension of the SV algorithm that approaches optimal work efficiency by processing subgraphs in each iteration. The LACC [2] algorithm uses linear algebraic primitives to implement connected components and is based on the PRAM AS algorithm. FastSV [30] further simplifies and optimizes LACC's tree hooking and compressing method to improve the performance. Iteration-based tree hooking-compressing methods exploit large-scale parallel resources to reduce the cost of each iteration and the total number of iterations.

Union-find-based algorithms [10], [12], [18] take advantage of the disjoint set data structure to reduce the total operations in one iteration. Tree-based methods try to reduce the number of iterations, but disjoint set-based methods focus on reducing the total number of operations. So, tree-based methods are suitable for large-scale parallel execution but disjoint set-based methods are good for parallel resources limited scenarios.

There are some works combining different methods together to optimize the performance further. Slota *et al.* [26] developed a distributed memory multi-step method that combines parallel *BFS* and label propagation technique. The ParConnect algorithm [14] is based on both the SV algorithm and parallel breadth-first search (*BFS*). ConnectIt [10] provides a framework to provide different sampling strategies and tree hooking and compression schemes.

Recently, different optimization methods for connected component problems have been proposed. Thrifty Label Propagation (TLP) algorithm [11] uses the skewed degree distribution of real-world graphs to develop their optimized label propagation algorithm. Sutton *et al.* [28] 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.

We formulate the connected components as a contour line discover problem and develop different minimum mapping operators for different scenarios. Our method is flexible and simple. It can achieve high performance in different scenarios.

VI. Conclusion

In this study, we addressed the fundamental graph problem of finding connected components using a novel method called "minimum mapping." Our approach is characterized by its simplicity, flexibility, and efficient implementation, setting it apart from existing state-of-the-art methods.

We proved that our method achieves convergence in $\mathcal{O}(\log_2(d_{max}))$ time, where d_{max} represents the largest diameter among all components in a graph. Experimental results also show that our algorithm can converge in a small number of iterations for different graphs.

Experimental results showed that our *Contour* method significantly outperforms the state-of-the-art large-scale parallel *FastSV* method. Additionally, our method complements the state-of-the-art shared memory parallel *ConnectIt* method. Notably, we have successfully integrated our method and the state-of-the-art methods into an open-source graph package, Arachne. Arachne extends an open-source framework for Python users, enabling efficient large-scale graph analytics on supercomputers. This integration empowers high-level Python users to conduct large graph analytics efficiently, regardless of their familiarity with supercomputing and large data processing intricacies.

Acknowledgment

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

VII. References

[1] Baruch Awerbuch and Yossi Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, 36(10):1258–1263, 1987.

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

[3] D.A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 38–, 2004.

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

[5] Guy E Blelloch and Bruce M Maggs. Parallel algorithms. In *Algorithms and theory of computation handbook: special topics and techniques*, pages 25–25. 2010.

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

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

[8] Richard Courant and Herbert Robbins. *What is Mathematics?: an elementary approach to ideas and methods*. Oxford University Press, USA, 1996.

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

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

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

[12] Zvi Galil and Giuseppe F Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR)*, 23(3):319–344, 1991.

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

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

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

[16] Arvind Krishnamurthy, Steven Lumetta, David E Culler, and Katherine Yelick. Connected components on distributed memory machines. *Third DIMACS Implementation Challenge*, 30:1–21, 1997.

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

[18] Md Mostofa Ali Patwary, Jean Blair, and Fredrik Manne. Experiments on union-find algorithms for the disjoint-set data structure. In *Experimental Algorithms: 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings 9*, pages 411–423. Springer, 2010.

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

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

[21] Oliver Alvardo Rodriguez, Zhihui Du, Joseph T. Patchett, Fuhuan Li, and David A. Bader. 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*, 2022.

[22] Yossi Shiloach and Uzi Vishkin. An O(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

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

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

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

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

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

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

[29] Robert E Tarjan and Jan Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, 31(2):245–281, 1984.

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