# BTS: Load-Balanced Distributed Union-Find for Finding Connected Components with Balanced Tree Structures

Chaeeun Kim[†]
*University of California, Santa Cruz*
CA, USA
ckim151@ucsc.edu

Changhun Han
*Kookmin University*
Seoul, Republic of Korea
codingnoye@kookmin.ac.kr

Ha-Myung Park[*]
*Kookmin University*
Seoul, Republic of Korea
hmpark@kookmin.ac.kr

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00089

***Abstract*—How can we efficiently find connected components with Union-Find in a distributed system? Union-Find is the most efficient sequential algorithm for finding connected components with low memory usage and high speed. Several studies have adapted Union-Find to distributed memory systems to process large graphs quickly; however, they all suffer from load balancing problems. We notice that the leading cause of the load balancing problems is the nature of Union-Find, which gathers more and more edges to a small number of vertices as it proceeds. In this paper, we propose BTS, a new fast and scalable distributed Union-Find algorithm for finding connected components in large graphs. BTS resolves the load balancing problems by proposing Balanced Union-Find, which allocates vertices to each processor and makes edges link to vertices in the same processor as much as possible. We further optimize BTS with edge refinement to minimize network traffic and memory usage. Experimental results show that BTS efficiently resolves the load balancing problems, processing 16-1024 times larger graphs with 3.1-261.9 times faster speeds than existing algorithms.**

***Index Terms*—Connected component, Graph algorithm, Graph mining, Distributed algorithm, MPI**

## I. Introduction

Given a large graph, how can we efficiently find connected components with Union-Find in a distributed system? A connected component in a graph is a maximal subset of vertices that are connected by paths. Finding all connected components is a crucial graph mining task and has been used in numerous applications such as graph compression [1], [2], pattern recognition [3], [4], reachability indexing [5], [6], and graph partitioning [7], [8]. Union-Find is the most efficient sequential algorithm for finding connected components with low memory usage and high speed. Union-Find gradually updates a forest from a stream of edges so that each connected component like in Fig. 1a is identified as a star graph like in Fig. 1d in the end.

For decades, many studies have proposed various algorithms for finding connected components with different approaches in different computational environments: graph traversal [9]–[14], label propagation & shortcutting [15]–[26], matrix-vector multiplication [27], [28], and Union-Find [29]–[36] in parallel, external memory, MapReduce, and distributed memory systems. Among them, Union-Find-based algorithms perform superior to other approaches on sequential or parallel systems in terms of running time and memory usage [30], [35]. For distributed systems, however, approaches that can be implemented as a series of sequential accesses, such as label propagation and matrix-vector multiplication, have been mainly studied rather than Union-Find, which requires many random accesses [11], [12], [18]–[22], [27], [28], [32]. Nevertheless, there have been a few studies to apply Union-Find to distributed systems [29]–[32]. They perform Union-Find independently in each processor and then integrate the results

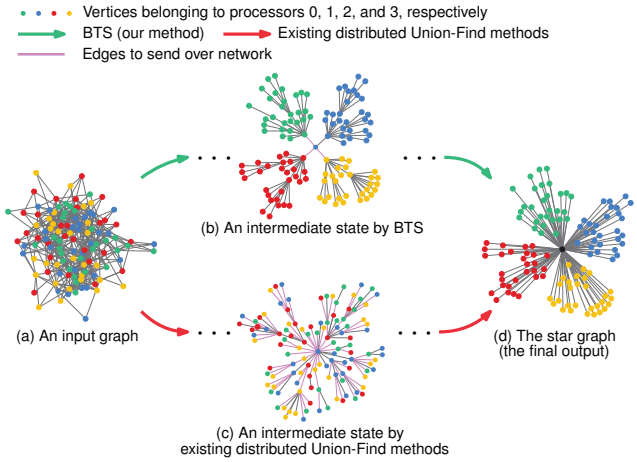



Fig. 1: Given an input graph as shown in (a), finding connected components is essentially the same as computing star graphs like in (d). Existing distributed Union-Find methods iteratively update the graph in a processor-agnostic way, resulting in imbalanced loads and extensive network communication, as shown in (c). Notably, edges that span vertices in different processors (highlighted in pink), necessitating network communication, largely emerge in intermediate states. BTS addresses these issues by computing balanced trees first as in (b), and transforming them into star graphs as in (d) without network communication. Vertices are color-coded by processors.

[†]Chaeeun Kim was at Kookmin University during this work.
[*]Ha-Myung Park is the corresponding author.

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00089

to compute the global connectivity in processor-agnostic ways. However, we observe that this processor-agnostic Union-Find approach leads to substantial network communication and significant load balancing issues. Fig. 1c illustrates these issues with an example of the intermediate state of a processor-agnostic distributed Union-Find algorithm. It links vertices by edges regardless of the processors to which they belong, resulting in significant network traffic; edges that need to be sent over the network are highlighted in pink. Moreover, as the operation progresses, edges become increasingly concentrated on a few vertices, such as the central blue vertex. This concentration leads to increased network communication, computational cost, and memory consumption on these few specific vertices, exacerbating the overall load balancing challenges.

In this paper, we propose BTS, a new fast and scalable distributed Union-Find algorithm for finding connected components in large graphs. BTS resolves the aforementioned problems by employing a *balanced tree structure* that constrains edges to remain within each processor as much as possible during the process. This approach aims to minimize network communication and balance the workloads. BTS computes balanced trees like in Fig. 1b first and then converts them to star graphs while each subtree is handled by a processor independently without network communication. Our research includes further optimizing BTS to minimize network traffic and memory usage by refining edges. We summarize the contributions of this paper as follows:

- **Algorithm.** We propose BTS, a fast and scalable distributed Union-Find algorithm that balances the workload with balanced tree structures and minimizes network traffic and memory consumption with edge refinement.
- **Theory.** We theoretically show that BTS resolves the load balancing problems by limiting the number of children a vertex can have to $O(\rho^2+|V|/\rho)$, reduces the network traffic down to $O(C \times (\rho-1))$, and bounds the memory usage of each processor to $O((|V|+|E|)/\rho)$ where $|V|$, $|E|$, and $C$ are the numbers of vertices, edges and connected components in the graph, and $\rho$ is the number of processors.
- **Experiment.** Experimentally evaluated on large real-world and synthetic graphs, BTS efficiently resolves the load balancing problems, processing 16-1024 times larger graphs with 3.1-261.9 times faster speed than existing algorithms.

The source codes and datasets used in this paper are available at https://github.com/cekim10/BTS.

## II. Related Work

In this section, we introduce existing algorithms for finding connected components in large graphs. We first present the concept of Union-Find algorithms. Then, we concentrate on existing distributed Union-Find algorithms since the proposed algorithm BTS is one of that kind. After that, we concisely outline other approaches to finding connected components in large graphs. Methods introduced in this section are summarized in Table I.

TABLE I: Table of connected components algorithms

| | Union-Find | Graph Traversal | Label Propagation & Shortcutting | Matrix-Vector Multiplication |
|---|---|---|---|---|
| Parallel, in-memory | Cybenko et al. [29], Anderson & Woll [33], Simsiri et al. [34], **ConnectIt** [35] | Bader & Cong [9], Pearce et al. [10] | Shiloach & Vishkin [15], Liu el al [23], [24], PPA-assembler [25] | LACC [40], FastSV [27], [41] |
| Parallel, External | Agarwal et al. [36] | Pearce et al. [10] | FlashGraph [16], **Mosaic** [17] | - |
| Distributed, in-memory | UFM [29], DUF [30], **D-Rem** [31], ALBUF [32], **BTS (our)** | GD-CC [13], [14], Jain et al. [11] | Yan et al. [26], **D-Galois** [18] | LACC [40], **FastSV** [27], [41] |
| Distributed, external | - | Asokan [12] | Hash-to-Min [22], Kiveris et al. [19], **PACC** [20], [21] | Pegasus [28] |

### A. Union-Find Algorithms

Union-Find is a well-known algorithm for finding connected components in a graph. A typical data structure used for Union-Find is a forest array, an array of parent pointers representing each connected component as a rooted tree. Union-Find initially sets each vertex's pointer to the vertex itself; that is, each vertex constitutes a connected component alone. Union-Find uses two operations, *union* and *find*. Given an edge spanning two connected components, the *union* operation merges them by setting the pointer of the root of one tree to a vertex in the other tree. Given a vertex, the *find* operation returns the root of the tree containing the vertex; when two vertices $u$ and $v$ are in the same connected component, $find(u)$ and $find(v)$ return the same root. Union-Find usually uses union-by-rank or union-by-size technique for the union operation and path compression, path splitting, or path halving technique for the find operation [37]. The combination of them is proved to take $O(|E| \cdot \alpha(|E|,|V|))$ time where $|V|$ and $|E|$ are the numbers of vertices and edges in the graph, respectively, and $\alpha$ is the very slowly growing inverse Ackermann's function [38]. *Rem* [39] improves the union operation to enable early termination by alternately updating the parent pointers of two end vertices of each edge. The time complexity of Rem is $O(|E| \cdot log_{(2+|E|/|V|)}|V|)$ [38] and it performs the best in practice as experimentally evaluated in [37]. To handle larger graphs, Union-Find has also been adapted for several computing environments such as parallel [29], [33]–[35], external-memory [36], and distributed-memory [29]–[32] systems.

### B. Distributed Union-Find Algorithms

Over the years, several Union-Find algorithms running on distributed memory systems have been proposed to efficiently find connected components in large graphs. The first practical distributed Union-Find algorithm is known to be *Union-Find Merging (UFM)* proposed in [29]. UFM first makes each processor build a forest array from edges residing in the processor. Then, it merges the forest arrays two by two over $\log \rho$ steps to resolve the global connectivity, where $\rho$ is the number of processors. However, UFM does not scale well because it does

not fully exploit all processors and uses a lot of memory; the number of active processors is halved every step, so only two processors are used in the last step, and a forest array occupies $O(|V|)$ memory space for each processor. *Distributed Union-Find (DUF)* [30] addresses the issues by partitioning the graph. DUF partitions the vertex set into $\rho$ blocks of size $O(|V|/\rho)$. Each processor builds a forest array from the subgraph induced by a block and updates it repeatedly until convergence; in each step, each processor updates the parent pointer of each vertex to point to the grandparent. If the parent belongs to a different processor, network communication occurs. *Distributed Rem (D-Rem)* [31] speeds it up with a different update strategy; it updates the parent pointer of each vertex to point directly to its root. Each update occurs network communication multiple times if the path to the root spans several processors. D-Rem reduces the number of network communication occurrences by using Rem's zigzag union operation that terminates the search early if possible. However, DUF and D-Rem suffer from load balancing problems since the parent pointers get concentrated on very few vertices as the algorithms proceed. *Asynchronous and Load Balanced Union-Find (ALBUF)* [32] alleviates the load balancing problems by redistributing vertices of the initial graph so that processors have similar numbers of vertices, but does not completely resolve the problems. Still, the parent pointers get concentrated on very few vertices during the computation, resulting in up to 4.8 times the difference in running time between processors (see Fig. 5 in [32]). Our experimental result in Fig. 5b shows that the initial rebalancing alone does not address the load balancing problems perfectly. We emphasize that, in this paper, we dramatically reduce the running time by fundamentally resolving the load balancing problem that DUF, D-Rem, and ALBUF suffers from (see Sections IV-C and V-B).

### C. Other Approaches

Besides Union-Find, several works propose other approaches to finding connected components in large graphs. We categorize the approaches into graph traversal, label propagation & shortcutting, and matrix-vector multiplication.

- **Graph Traversal**: Several studies adapt breadth-first search (BFS) or depth-first search (DFS) to parallel [9], [10], external [10], distributed-memory (including Pregel) [11], [13], [14], and MapReduce [12] systems. Graph traversal methods are optimal in time ($O(|V|+|E|)$) in sequential systems but not that efficient in parallel and distributed systems. They duplicate the entire graph for each processor or raise massive I/Os due to the nature of graph traversal requiring a large amount of random access.
- **Label Propagation & Shortcutting**: Label propagation propagates vertex labels through edges iteratively. Shortcutting reduces the number of iterations by sending a label to two-hop neighbors at once. Since label propagation at each vertex can work independently, label propagation algorithms in parallel and distributed systems [15]–[26] perform better than graph traversal algorithms. Still, to access all neighbors for each vertex every iteration, label propagation requires storing the entire graph into memory or raising massive I/Os.
- **Matrix-Vector Multiplication**: Label propagation can be represented by iterative multiplication between an adjacency matrix and a label vector. LACC [40] and FastSV [27], [41] use a distributed linear-algebra tool to perform label propagation. Pegasus [28] implements a generalized matrix-vector multiplication on MapReduce and uses it to find connected components. On parallel and distributed systems, matrix-vector multiplication has essentially the same problems as label propagation.

Due to the load balancing problems of existing distributed Union-Find algorithms, as described in Section II-B, various approaches introduced above for finding connected components in large graphs have been mainly studied. Note that we efficiently resolve the load balancing problem by keeping each edge inside a processor as much as possible. Consequently, our method outperforms other algorithms in different approaches and environments (see Sections V-E and V-F).

TABLE II: Table of symbols

| Symbol | Definition |
|---|---|
| $G=(V,E)$ | Undirected graph with vertex set $V$ and edge set $E$ |
| $u,v,w$ | Vertices |
| $(u,v)$ | Edge between $u$ and $v$ such that $u>v$ |
| $p(u)$ | Parent pointer of $u$ |
| $p(u,G')$ | Parent pointer of $u$ in a graph $G'$ |
| $\Lambda(u,G)$ | Connected component containing vertex $u$ in $G$ |
| $m(S)$ | Most preceding vertex in vertex set $S$ |
| $\xi$ | Random hash function: $V \to \{0,\cdots,\rho-1\}$ |
| $\xi(u)$ | Processor to which vertex $u$ belongs |
| $[S]_i$ | Vertex set S in processor $i$: $\{v \in S \mid \xi(v)=i\}$ |
| $\rho$ | Number of processors |
| $C$ | Number of connected components |

## III. Problem Definition

In this section, we define the problem of finding connected components. The symbols used in this paper are listed in Table II. Let $G=(V,E)$ be an undirected graph where $V$ and $E$ are the sets of vertices and edges, respectively. We assume that any two vertices are comparable and are not identical; $u<v$ means vertex $u$ precedes vertex $v$. An edge connecting two vertices $u$ and $v$ is represented as $(u,v)$ if $u>v$, and as $(v,u)$ if $u<v$. In other words, an edge $(u,v)$ is an ordered pair such that $u>v$; we call $u$ the source and $v$ the sink. We say vertices $u$ and $v$ are connected if a path exists between $u$ and $v$ in $G$. A connected component of a graph is a maximal subset of vertices where every vertex is connected. For example, the input graph in Fig. 2 has two connected components of 12 and 6 vertices. Each vertex belongs to exactly one connected component. We denote the connected component containing vertex $u$ by $\Lambda(u,G)$. Our method represents each connected component as a tree where parents precede their children. We denote by $p(u)$ the parent pointer of $u$ in a tree structure; i.e., an edge $(u,v)$ in a tree means $p(u)=v$. Additionally, we denote by $p(u,G')$ the parent pointer of $u$ in an arbitrary graph $G'$. We let the root vertex of a tree be the self-pointing vertex at the topmost; i.e.,

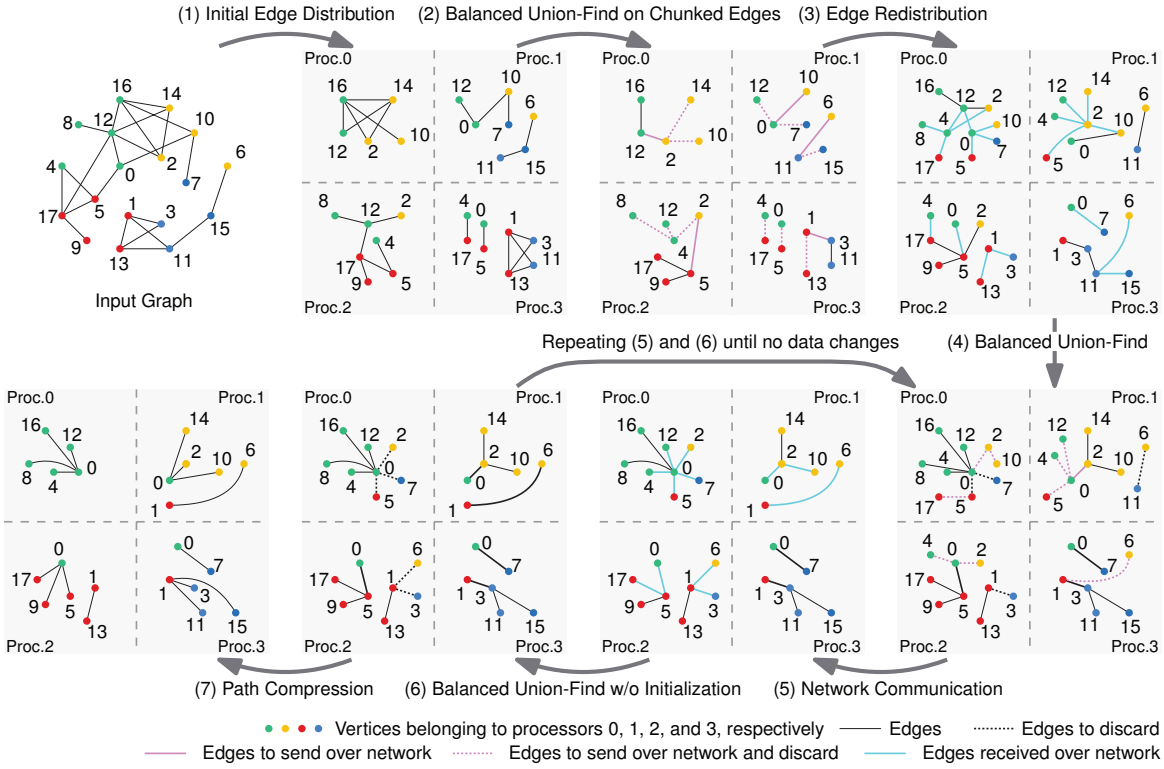



Fig. 2: An example of overall process of BTS.

$p(u) = u$. For a vertex set $S$, we denote the most preceding vertex in $S$ by $m(S)$.

The problem of this paper is to find all connected components in $G$, which is equivalent to transforming the graph like in Fig. 1a into a forest of one level trees, also known as star graphs, like in Fig. 1d. We formally define the problem as follows:

**Definition 1.** *The problem of finding all connected components in graph $G$ is to associate each vertex $u$ in $V$ to the most preceding vertex $m(\Lambda(u, G))$ in the connected component containing $u$.*

## IV. Proposed Method

We propose BTS, a novel distributed Union-Find algorithm that runs on distributed-memory systems. The main challenges to make BTS fast and scalable are as follows:

**(1) How can we balance the workload among processors?** Union-Find inherently concentrates pointers on a small number of vertices during the process, causing load balancing problems on distributed-memory systems as in the distributed Union-Find algorithms introduced in Section II-B.

**(2) How can we minimize network traffic?** How much network traffic an algorithm occurs determines its overall performance on distributed-memory systems. Meanwhile, updating parent pointers repeatedly across different processors, distributed Union-Find algorithms cause massive network traffic.

**(3) How can we bound the memory usage of processors?** In a distributed Union-Find algorithm, a processor requires memory space for vertices and edges that the processor deals with. It implies that a processor may have to occupy memory space for all the vertices and edges in the graph, such as when parent pointers are concentrated on the processor. To make an algorithm scalable, we need to bound the memory usage of processors.

We address the above challenges with our all-in-one solution, **edge rebalancing**, which balances the workload by keeping each edge inside a processor (see Section IV-C). It also reduces network traffic and memory usage as the number of edges spanning between processors decreases. We further optimize BTS by **excluding unchanged edges from network communication** to minimize network traffic and **discarding the parent pointers of outer vertices** to bound the memory usage (see Section IV-D).

### A. BTS: Overview

Fig. 2 shows the overall process of BTS. Given an input graph, vertices are colorized and assigned to one of the processors by a random hash function $\xi : V \to \{0, \cdots, \rho-1\}$. We denote by $\xi(u)$ the processor to which vertex $u$ belongs. In this example, vertices in the input graph colored in green, yellow, red, and blue belong to processors 0, 1, 2, and 3, respectively. BTS finds connected components by following steps: (1) BTS initially distributes the edges of the input graph evenly across the processors. (Section IV-B) (2) Each

**Algorithm 1:** BTS

**Input:** Undirected graph $G = (V, E)$ in external storage
**Output:** $\{(u, m(\Lambda(u, G)))|u \in V\}$

```
1 Initialization()                          ▷ Steps 1-3 (Algorithm 2)
2 BalancedUnionFind()                       ▷ Step 4 (Algorithm 3)
3 repeat
4 |   NetworkCommunication()                ▷ Step 5 (Algorithm 4)
5 |   BalancedUnionFind()                   ▷ Step 6 (Algorithm 3)
6 until no edge updates
7 PathCompression()                         ▷ Step 7 (Algorithm 5)
```

processor performs Balanced Union-Find (BUF) on the subgraph induced by the set of allocated edges. As a result, the subgraph turns into a forest of balanced trees with reduced edges and the same connectivity. We describe BUF in detail in Section IV-C. (3) Then, BTS redistributes edges over the network; each edge $(u, v)$ is sent to $\xi(u)$ and $\xi(v)$. (4) Each processor performs BUF again to build an initial forest and (5) sends every *updated* edge $(u, v)$ to $\xi(u)$ and $\xi(v)$ through the network. (6) Each processor receives some edges and conducts BUF on each received edge. BTS repeats (5) and (6) until no edge is updated. (7) BTS computes the final result by independently performing path compression for all vertices on each processor; no network communication occurs.

Algorithm 1 is the overview pseudocode of BTS. Given an undirected graph $G = (V, E)$, BTS progresses through a sequence of aforementioned steps to derive a set of pairs $\{(u, m(\Lambda(u, G)))|u \in V\}$ where $m(\Lambda(u, G))$ denotes the root vertex $u$ in the graph. The pseudocodes for each step are detailed in the following sections.

### B. Initial Edge Distribution & Redistribution

This section describes steps (1)-(3) in Fig. 2 in detail. Like the existing distributed Union-Find algorithms such as DUF, D-Rem, and ALBUF, BTS is a vertex partition-based algorithm. BTS partitions the vertex set into blocks, assigns each block to a processor, and has each processor repeatedly update the parent pointers of given vertices. Meanwhile, the input graph is initially stored as an edge list file in external storage such as Network Attached Storage (NAS) or Distributed File System (DFS). To quickly load data in parallel, BTS partitions the input edges into $\rho$ blocks evenly and has each processor load a block, where a block is a subset of edges existing consecutively in storage, and $\rho$ is the number of processors. For example, in step (1) of Fig. 2, 24 input edges are partitioned into $\rho = 4$ blocks as the input of processors, respectively. Step (2) transforms the subgraph induced by each block into a smaller subgraph with the same connectivity to reduce network communication; we explain this more in the next paragraph. Then, step (3) has each processor redistribute each edge $(u, v)$ to processors $\xi(u)$ and $\xi(v)$ to let them update their parent pointers with the edge. For example in Fig. 2, step (3) sends edge (4, 2) in processor 2 to $\xi(4) = 0$ and $\xi(2) = 1$.

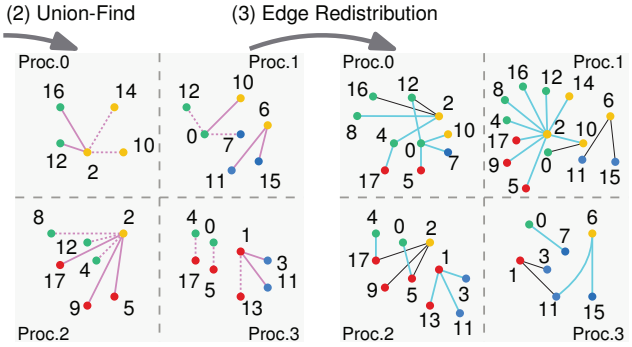



Fig. 3: An example of edge redistribution with plain Union-Find. Unlike when using BUF, edges are concentrated on vertex 2, causing a load balancing problem. Vertices and edges are color-coded in the same way as in Fig. 2.

**Algorithm 2:** Initialization (Steps (1)-(3))

```
1  Function Initialization()
       ▷ Step (1). Initial Edge Distribution
2      Let E^C be the set of edge chunks, each of size O(M)
       ▷ Step (2). Balanced Union-Find on Chunked Edges
3      for each E_i^C in E^C do in parallel
4          Initialize G' = (V', E') as an empty graph
5          Union-Find(E_i^C, G')                ▷ In Algorithm 3
6          Rebalancing(G')                      ▷ In Algorithm 3
           ▷ Step (3). Edge Redistribution
7          for each vertex u ∈ V' do
8              v ← p(u, G')
9              Send (u, v) to processors ξ(u) and ξ(v)
10         Discard G' from the main memory
```

Before redistributing edges, BTS reduces the number of edges to send by transforming the input edges into other edges with the same connectivity as D-Rem [31] does; D-Rem has each processor perform Union-Find on the input edges, transforming the edges into a forest. In contrast, BTS uses Balanced Union-Find (BUF), described in Section IV-C, instead of Union-Find for the sake of load balancing. Fig. 3 shows the result of edge redistribution when plain Union-Find is used instead of BUF for step (2) in Fig. 2. Union-Find links many edges to vertex 2 in each processor, and accordingly, after edge redistribution, edges flock to processor 1 in charge of vertex 2, causing a load balancing problem. In contrast, BUF prevents edges from being concentrated at a single vertex (see steps (2) and (3) in Fig. 2). We give a detailed description of BUF in Section IV-C.

If the graph is enormous, so a vast number of edges are passed to a processor, an out-of-memory error can occur. To avoid out-of-memory errors, BTS has each processor divide the input edge block into chunks of size $O(M)$ and perform

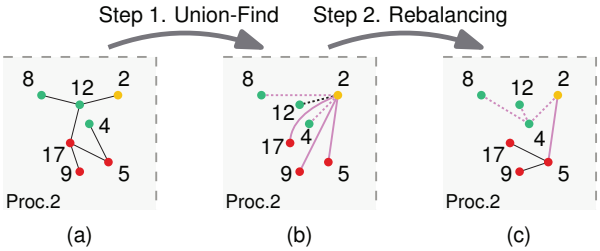



Fig. 4: An example of Balanced Union-Find. Vertices and edges are color-coded in the same way as in Fig. 2.

BUF on each chunk sequentially, where $M$ is the memory size of a processor. As soon as BUF computes a forest from each chunk, each processor immediately sends the edges to other processors through the network and then discards the edges to secure memory space.

Algorithm 2 is the pseudocode of Initialization, which is corresponding to steps (1) to (3). The function `Initialization` divides the input edge set $E$ into the chunk set $E^C$, where each chunk has $O(M)$ edges. Initializing $G' = (V', E')$ as an empty graph, the function performs Balanced Union-Find for each chunk $E_i^C$ in $E^C$. Balanced Union-Find consists of Union-Find and Rebalancing, detailed in Algorithm 3. As a result of the Balanced Union-Find, each vertex $u$ becomes connected to either the local root or the root vertex. Subsequently, the function sends the edge $(u, v)$ for each vertex $u$ and the updated parent pointer $v = (u, p(u, G'))$ to processors $\xi(u)$ and $\xi(v)$. Once the edge distribution is complete, $G'$ is discarded from the main memory to secure memory space.

### C. Balanced Union-Find

This section describes Balanced Union-Find (BUF) which is used in steps (2), (4), and (6) in Fig. 2. Existing vertex partitioning-based Union-Find algorithms suffer from load balancing problems due to the nature of Union-Find that gathers edges to a small number of vertices as it proceeds. BUF resolves the load balancing problems by conducting a rebalancing operation after Union-Find; the rebalancing operation modifies parent pointers not to focus on the same vertex. We describe the two steps that BUF does on each processor below.

*1) Initialization:* The input of a processor is edges transferred over the network from other processors. BUF initializes the parent pointer of each vertex in the subgraph induced by the input edges to point to the vertex itself. We use a hash table to store the parent pointers and assume that if a vertex is not in the hash table, the parent pointer of the vertex points to the vertex itself.

*2) Step 1: Union-Find on each processor:* For each input edge, BUF conducts the union operation of Union-Find, updating the parent pointers. After that, BUF conducts the find operation with path compression for each vertex in the hash table. As a result, each connected component of the subgraph induced by the input edges forms a star graph, where the root vertex is the only parent of the other vertices. Fig. 4b shows the star graph computed by Union-Find on the graph in Fig. 4a. Note that any sequential Union-Find algorithm can be used here, but we use Rem [39] because it is known to be the fastest so far.

*3) Step 2: Rebalancing parent pointers:* As a result of Step 1, all vertices in each star graph point to its root vertex. For example, all vertices in Fig. 4b point to vertex 2. BUF converts each star graph into a balanced tree for load balancing. Note that vertices are colorized by the hash function $\xi$. For each star graph, BUF modifies the parent pointer of each vertex to point to the local root, which is the most preceding vertex among vertices with the same color, and the parent pointers of local roots to point to the root. Fig. 4c shows the balanced tree computed from the star graph in Fig. 4b. Vertices 4 and 5 are local roots of green and red vertices, respectively. Accordingly, green vertices point to vertex 4, and red vertices point to vertex 5. Vertices 4 and 5 point to vertex 2, the root.

The pseudocode of Balanced Union-Find is listed in Algorithm 3. The `BalancedUnionFind` function has each processor $i$ receive edges $E_i'$ through the network and conducts two functions `Union-Find` and `Rebalancing` sequentially. `Union-Find` is used to integrate the received edges $E_i'$ into the local graph $G_i$, and `Rebalancing` is a critical step that ensures the balanced distribution of vertices across processors. Given an arbitrary edge set $E^\star$ and a graph $G'$, `Union-Find` incorporates each edge $(u, v)$ one by one into the graph $G'$ by linking the root $m(\Lambda(u, G'))$ of $u$ and the root $m(\Lambda(v, G'))$ of $v$. After that, a path compression is conducted locally so that each connected component forms a star graph. Then, `Rebalancing` re-updates the parent pointers of each vertex $u$ in $V'$ for load balancing. If $u$ is not the local root (i.e., $m([\Lambda(u, G')]_{\xi(u)})$), the parent pointer of $u$ is updated to $m([\Lambda(u, G')]_{\xi(u)})$. Otherwise, it is updated to $m(\Lambda(u, G'))$ representing the root vertex; this condition addresses the case where local roots are connected to the root vertex, ensuring a balanced tree structure. For the sake of intelligibility, the pseudocode of `Union-Find` is written similarly to the standard Union-Find algorithm, but it can be interchangeable with Rem [39] for better performance, as we mentioned.

### D. Network Communication

This section describes step (5) in Fig. 2 in detail. From the point of view of processor $i$, we call a vertex $u$ an outer vertex if $\xi(u) \neq i$, and an edge $(u, v)$ an outer edge if $\xi(u) \neq i$ or $\xi(v) \neq i$. Conversely, a vertex $u$ with $\xi(u) = i$ is called an inner vertex. After BUF, each processor sends outer edges to other processors over the network to inform that the edges might bridge different trees. In Fig. 4c, the pink edges are sent over the network because they are outer edges incident to non-red vertices, which do not belong to processor 2; edges $(5, 2)$ and $(4, 2)$ are sent to processor 1 (yellow), and edges $(8, 4)$, $(12, 4)$, and $(4, 2)$ are sent to processor 0 (green). Note that, without rebalancing, all the edges in Fig. 4b are sent to processor 1 to which vertex 2 belongs, causing a load

**Algorithm 3:** Balanced Union-Find

```
Function BalancedUnionFind()
    for i ← 0 to ρ − 1 do in parallel
        Let G_i = (V_i, E_i) be the graph subject to processor i
        Let E'_i be the edges received by processor i via network
        Union-Find(E'_i, G_i)        ▷ Integrating E'_i into G_i
        Rebalancing(G_i)             ▷ Rebalancing G_i
Function Union-Find(E*, G')
    for each edge (u, v) ∈ E* do
        V_i ← V' ∪ {u, v}
        r_u ← FindRoot(u, G')
        r_v ← FindRoot(v, G')
        if r_u < r_v then
            p(r_v, G') ← r_u
        else if r_v < r_u then
            p(r_u, G') ← r_v
    ▷ Path Compression
    for each vertex u ∈ V' do
        r ← FindRoot(u, G')
        a ← u
        while a ≠ r do
            t ← p(a, G')
            p(a, G') ← r
            a ← t
Function FindRoot(u, G')
    r ← u
    while r ≠ p(r, G') do
        r ← p(r, G')
    return r
Function Rebalancing(G')
    for each vertex u ∈ V' do
        if u ≠ m([Λ(u, G')]_{ξ(u)}) then
            p(u, G') ← m([Λ(u, G')]_{ξ(u)})
        else
            p(u, G') ← m(Λ(u, G'))
```

balancing problem. In this example, the number of edges sent to processor 1 reduces from 6 to 2 by rebalancing.

*1) Excluding unchanged edges from network communication:* BTS reduces network traffic by excluding edges not changed by BUF from network communication if the edge's source is not a local root. In Fig. 2, for example, outer edges $(5, 0)$ and $(7, 0)$ in processor 0 do not change in step (4); thus, they are not sent over the network in step (5). Excluding unchanged edges does not affect the final result. Assume processor $i$ receives an edge $(u, v)$ where $\xi(u) = i$ and $\xi(v) = j$ or vice versa. If the edge remains in processor $i$ as it is after BUF, we don't have to send the edge to processor $j$ because processor $j$ also has received the edge or already has it.

**Algorithm 4:** Network Communication

```
Function NetworkCommunication()
    for i ← 0 to ρ − 1 do in parallel
        Let G_i = (V_i, E_i) be the graph subject to processor i
        for each vertex u ∈ V_i do
            if p(u, G_i) is changed by BUF or
              u = m([Λ(u, G_i)]_{ξ(u)}) then
                v ← p(u, G_i)
                Send (u, v) to processors ξ(u) and ξ(v)
        ▷ Discarding spanning edges
        if ξ(u) ≠ i then
            V_i ← V_i \ {u}
            E_i ← E_i \ {(u, p(u, G'))}
```

*2) Discarding parent pointers of outer vertices:* As soon as the network communication ends, BTS discards all the parent pointers of outer vertices to secure the memory space. In other words, BTS discards every edge whose source is an outer vertex. In Fig. 2, the discarded edges are indicated by dotted lines. The absence of these edges does not change the connectivity of the graph because the edges are either sent to other processors or already exist on other processors. For example, in step (5) of Fig. 2, edge $(6, 1)$ discarded in processor 3 is sent to processors 1 and 2, and edge $(5, 0)$ discarded in processor 0 also exists in processor 2. After discarding those edges, each processor keeps only the parent pointers of the vertices belonging to the processor in memory, occupying $O(|V|/\rho)$ space in expectation.

The pseudocode for step (5), Network Communication, is presented in Algorithm 4. For each processor $i$ where $G_i = (V_i, E_i)$ represents the local graph associated with the processor, the algorithm checks each vertex $u$ in $V_i$ whether the parent pointer for $u$, denoted as $p(u, G_i)$, has been altered by a preceding BUF operation or if $u$ is $m([\Lambda(u, G_i)]_{\xi(u)})$. In either case, the algorithm send the edge $(u, v)$ to processors $\xi(u)$ and $\xi(v)$ where $v = p(u, G_i)$ through the network. Additionally, the algorithm removes the edge $(u, v)$ from $G_i$ to secure the memory space if $u$ does not belong to processor $i$, i.e., $\xi(u) \neq i$. This ensures that each processor maintains a consistent and up-to-date view of its local graph, facilitating efficient communication and synchronization across the processors.

### *E. Iteration Process & Finalization*

This section describes steps (6) and (7) in Fig. 2 in detail. When each processor receives edges, it uses the edges to update the parent pointers through steps 1 and 2 of BUF. If updated edges exist, they are sent again over the network. This process repeats until no more edges are updated. When the

**Algorithm 5:** Path Compression

```
1  Function PathCompression()
2      for i ← 0 to ρ − 1 do in parallel
3          Let G_i = (V_i, E_i) be the graph subject to
             processor i
4          for each vertex u ∈ V_i do
5              r ← FindRoot(u, G_i)        ▷ In Algorithm 3
               ▷ Linking vertices on the path to the root
                 to the root
6              a ← u
7              while a ≠ r do
8                  t ← p(a, G_i)
9                  p(a, G_i) ← r
10                 a ← t
```

iteration process ends, all vertices belonging to each processor form star graphs where the centers are local roots, and the star graphs include the root vertices (see the output of step (6) in Fig. 2). Finally, step (7) has each processor update the parent pointers to point to the root without network communication.

*1) Termination Detection:* How can we efficiently know if a change has occurred on each processor? A simple method is to back up all edges before updating them and compare them with updated ones. However, in this case, not only does it consume additional memory space, but it also incurs computational overhead because it requires checking that a change has occurred for every edge. To track changes efficiently, BUF marks 'updated' on each vertex if its parent pointer has changed in step 1 (Union-Find). After step 2 (Rebalancing) of BUF, each processor counts how many vertices marked with 'updated' are outer vertices or have their parent outer vertices. This is the number of edges changed in the processor. In Fig. 4, for example, all edges except for $(12, 2)$ in processor 2 are marked 'updated' in step 1; and after step 2, only $(8, 4)$, $(12, 4)$, $(4, 2)$, and $(5, 2)$ are counted as changed. Each processor broadcasts the number of changed edges to other processors. If the number of changed edges in all processors is 0, the iteration process ends.

Algorithm 5 outlines the pseudocode for step (7), Path Compression. Assuming $G_i = (V_i, E_i)$ represents the local graph associated with processor $i$, the `PathCompression` function finalizes the linkage of each vertex $u$ in $V_i$ to the root vertex. This linkage is established after $G_i$ forms a balanced tree structure in step (6) through the execution of Balanced Union-Find. Within each processor, the function utilizes the `FindRoot` function from Algorithm 3 to identify the root vertex of $u$. Subsequently, it updates the vertices along the path from $u$ to its root, effectively compressing the path. This compression is achieved by updating the parent pointer of each vertex to directly point to the root. Importantly, the `PathCompression` function operates independently within each processor, and as such, network communication is not required.

### F. Analysis

In this section, we present theoretical analyses of BTS. We first show that BTS guarantees load balancing by bounding the number of children a vertex can have.

**Theorem 1.** *The maximum number of children a vertex can have in the union of all balanced trees after BUF is $O(\rho^2 + |V|/\rho)$ in expectation where $\rho$ is the number of processors and $|V|$ is the number of vertices in the graph.*

*Proof.* After BUF, each processor has a forest of balanced trees. A balanced tree has three kinds of vertices: the root, local roots, and the other vertices. The children of a root are either local roots or vertices with the same color as the root. The children of a local root have the same color as the local root. A vertex that is neither a root nor a local root of a balanced tree has no child. The expected number of vertices with a specific color is $O(|V|/\rho)$ since the vertices are colorized by a random hash function. The maximum number of local roots in a balanced tree is $\rho - 1$. A vertex can be the root of $\rho$ different balanced trees across the processors. Accordingly, the maximum number of child vertices a vertex can have in the union of all balanced trees is $\rho(\rho - 1) + O(|V|/\rho) = O(\rho^2 + |V|/\rho)$. □

The rebalancing of BUF also reduces network traffic significantly.

**Lemma 2.** *BTS without rebalancing requires $(|V| - C) \times \frac{\rho-1}{\rho}$ network communication in expectation upon convergence where $|V|$ and $|C|$ are the numbers of vertices and connected components, respectively, in the graph and $\rho$ is the number of processors.*

*Proof.* Upon convergence, every vertex $u$ resides in processor $\xi(u)$, and its parent pointer $p(u)$ points to the root $v$ after BUF without rebalancing. If $\xi(u) \neq \xi(v)$, processor $\xi(u)$ sends edge $(u, v)$ to processor $\xi(v)$. The probability that $\xi(u) \neq \xi(v)$ is $(\rho - 1)/\rho$ since the vertices are randomly colorized. The number of root vertices equals the number $C$ of connected components. Thus, the expected amount of network communication is the number $|V| - C$ of non-root vertices times the probability $\frac{\rho-1}{\rho}$ that two vertices have different colors. □

**Lemma 3.** *BTS with rebalancing requires at most $C \times (\rho - 1)$ network communication upon convergence where $C$ is the number of connected components in the graph and $\rho$ is the number of processors.*

*Proof.* Suppose that the graph has been converged by BUF with rebalancing. In contrast to BUF without rebalancing, only local roots point to its root, and the other vertices point to its local root. That is, inner vertices that are not local roots do not incur network communication. For each local root $u$ and its root $v$, processor $\xi(u)$ sends edge $(u, v)$ to processor $\xi(v)$. Each connected component has at most $\rho - 1$ local roots excluding the root. Thus, the maximum amount of network communication is $C \times (\rho - 1)$. □

TABLE III: The theoretical results

| | D-Rem [31], ALBUF [32] | BTS |
|---|---|---|
| Maximum number of children a vertex has | $O(\|V\|)$ | $O(\rho^2 + \|V\|/\rho)$ |
| Network communication per round | $O((\|V\| - C) \times \frac{\rho-1}{\rho})$ | $O(C \times (\rho - 1))$ |
| Memory space per processor | $O(\|V\| + \|E\|)$ | $O((\|V\| + \|E\|)/\rho)$ |

From the two lemmas above, we get how much rebalancing of BUF reduces network traffic.

**Theorem 4.** *Rebalancing reduces network traffic by a factor of $\frac{|V|-C}{C\rho}$ in expectation upon convergence where $|V|$ and $C$ are the numbers of vertices and connected components, respectively, in the graph and $\rho$ is the number of processors.*

*Proof.* By Lemmas 2 and 3, the rebalancing reduces network traffic by a factor of $\frac{(|V|-C)\times\frac{\rho-1}{\rho}}{C\times(\rho-1)} = \frac{|V|-C}{C\rho}$. □

Rebalancing edges and discarding parent pointers of outer vertices together bound memory usage of each processor to $O((|V| + |E|)/\rho)$.

**Theorem 5.** *BUF has each processor occupy $O((|V|+|E|)/\rho)$ memory space in expectation where $|V|$ and $|E|$ are the numbers of vertices and edges in the graph and $\rho$ is the number of processors.*

*Proof.* Each processor $i$ maintains the parent pointers of vertices belonging to it, and the expected number of such vertices is $|V|/\rho$ since vertices are assigned by a random hash function. Processor $i$ receives edges incident to vertices belonging to it. We let the edge set and its size be $E_i$ and $|E_i|$, respectively. Assuming that the edges are evenly distributed, the expected number of $|E_i|$ is $|E|(2\rho-1)/\rho^2$ since the probability that at least one vertex of an edge belongs to a specific processor is $1-(1-1/\rho)^2 = (2\rho-1)/\rho^2$. Then, processor $i$ has to maintain the parent pointers of outer vertices in the subgraph induced by $|E_i|$. The number of outer vertices in the induced subgraph is at most $|E_i|$ because each edge always has an inner vertex. Thus, processor $i$ additionally requires $|E_i| = |E|(2\rho-1)/\rho^2$ memory space. Totally, each processor $i$ occupies $|V|/\rho+|E|(2\rho-1)/\rho^2 = O((|V|+|E|)/\rho)$ memory space in expectation. □

In practice, each processor occupies $O(|V|/\rho)$ memory space as edge redistribution, described in Section IV-B, significantly reduces the number of edges. Accordingly, the number of outer edges is less than that of inner vertices in subsequent rounds. For example, in Fig. 7, the number of outer edges in round 1 is about 1 million, far less than 988.5 million, the number of vertices in the graph, SD.

Table III shows that BTS demonstrates superior theoretical results compared to D-Rem [31] and ALBUF [32]. Since neither D-Rem nor ALBUF addresses the load balancing problem, they have an identical theoretical outcome.

TABLE IV: The summary of datasets

| Datasets | $\|V\|$ | $\|E\|$ | Sources |
|---|---|---|---|
| LJ | 4.8M | 69M | SNAP* [44] |
| TW | 41.7M | 1.5B | Kwak et al.* [45] |
| FS | 65.6M | 1.8B | SNAP |
| SD | 89.2M | 2B | WebDataCommons* [46] |
| GSH | 988.5M | 33.9B | WebGraph* [47] |
| RMAT-21 | 1.1M | 31.5M | N/A (Synthetic graphs) |
| RMAT-23 | 4.1M | 125.8M | |
| RMAT-25 | 15.2M | 503.3M | |
| RMAT-27 | 56.1M | 2B | |
| RMAT-29 | 207M | 8B | |
| RMAT-31 | 762.8M | 32.2B | |

## V. Experiments

In this section, we experimentally evaluate the performance of BTS by answering the following questions:

**Q1** How well does BTS balance the workload? (Section V-B)

**Q2** How much network traffic does BTS reduce? (Section V-C)

**Q3** How much memory space does BTS require? (Section V-D)

**Q4** How does BTS scale up in terms of the input data size and the number of machines? (Section V-E)

**Q5** How well does BTS work on real-world graphs? (Section V-F)

### A. Setup

*1) Datasets:* Table IV shows both real-world and synthetic graphs we use in the experiments. LJ is a friendship network in LiveJournal. TW is a follower-following network in Twitter. FS is a friendship network in Friendster. SD is a domain-level hyperlink network. GSH is a page-level hyperlink network. RMAT-$k$ for $k \in \{21, 23, 25, 27, 29, 31\}$ graphs are realistic synthetic graphs that follow the Recursive Matrix model [42]; we generate RMAT-$k$ graphs using TegViz [43], an RMAT graph generating tool. We set RMAT parameters (a, b, c, d) = (0.57, 0.19, 0.19, 0.05).

*2) Algorithms:* We implement BTS using Message Passing Interface (MPI). We also test the algorithms written in bold in Table I; the other algorithms are excluded because at least one of the tested algorithms outperforms them. All distributed-memory algorithms considered in our experiments, such as D-Rem, D-Galois, and FastSV, are also based on MPI. We use the source codes written by the authors of the algorithms; all of them are available on the Web.

- BTS: the proposed distributed-memory Union-Find algorithm.
- ConnectIt [35]: the state-of-the-art multi-core algorithm.
- Mosaic [17]: the-state-of-the-art external-memory algorithm.
- D-Rem [31]: the representative distributed Union-Find algorithm.

*http://snap.stanford.edu/data/
*http://an.kaist.ac.kr/traces/WWW2010.html
*http://webdatacommons.org/hyperlinkgraph/
*https://law.di.unimi.it/datasets.php

- D-Galois [18]: the state-of-the-art distributed label propagation algorithm.
- FastSV [41]: the state-of-the-art distributed linear algebraic algorithm.
- PACC [21]: the state-of-the-art MapReduce algorithm.

*3) Machines:* We use a cluster consisting of a storage server and 10 computing machines; each is equipped with an Intel Xeon E3-1220 CPU (4-cores at 3.10GHz), 16GB RAM, 2 12TB HDDs. The machines are connected to a local network switch with 1Gbps ethernet speed. The storage server has 8 12TB HDD in RAID 6 and is connected to the same network switch with 5Gbps ethernet speed by Link Aggregation Control Protocol (LACP). MPICH v3.3 is installed on the cluster with all 10 machines and one of them acts as a master. We test single-machine algorithms on the master machine.

### B. Load Balancing

BTS resolves the load balancing problems by rebalancing edges as described in Section IV-C. To show the effect of rebalancing edges, we implement three versions of BTS: (a) BTS without rebalancing edges (*BTS-nobal*), (b) BTS which rebalances edges only at the initial step (*BTS-init*), and (c) the original BTS which rebalances edges every round. BTS-nobal can be regarded as a variation of D-Rem [31] to which the network communication optimization technique of BTS is applied. BTS-init improves BTS-nobal with a load balancing strategy of ALBUF [32]. Since the source code of ALBUF is not publicly available, we consider BTS-init as a variation of ALBUF. Fig. 5 is a gantt chart showing the running time of 10 processors in the three versions of BTS. The steps listed in Fig. 2 are color-coded. SD is used. In each step, BTS synchronizes the processors; processors that have completed their work wait for the other processors to finish. Empty parts of the running time bars in Fig. 5 mean that the processors are idle or just receiving data from others. More empty parts of the running time bars suggest more serious load balancing problems. Fig. 5a shows that *BTS-nobal* suffers from load balancing problems; processor 0 is idle for 68.6% of its total running time. Since most of the network communication is concentrated on processor 0, the running time of the other processors for network communication is considerably long (see orange bars). This argument is supported by the fact that processor 0 has the longest running time for updating parent pointers (see dark blue bars). Fig. 5b shows that rebalancing edges at the initial step (*BTS-init*) alleviates a load balancing problem at that step. However, due to repeated updates of parent pointers, edges are concentrated on a small number of vertices again, and the load balancing problems reappear, indicating ALBUF [32] does not resolve the load balancing problems perfectly as the round proceeds. The original BTS eliminates the load balancing problems by rebalancing edges every round (see Fig. 5c). Compared to *BTS-nobal* and *BTS-init*, the original BTS dramatically reduces the running time for network communication by 522 and 553 times, respectively; accordingly, the total running time decreases by 2.7 and 2.5 times, respectively.

### C. Network Traffic

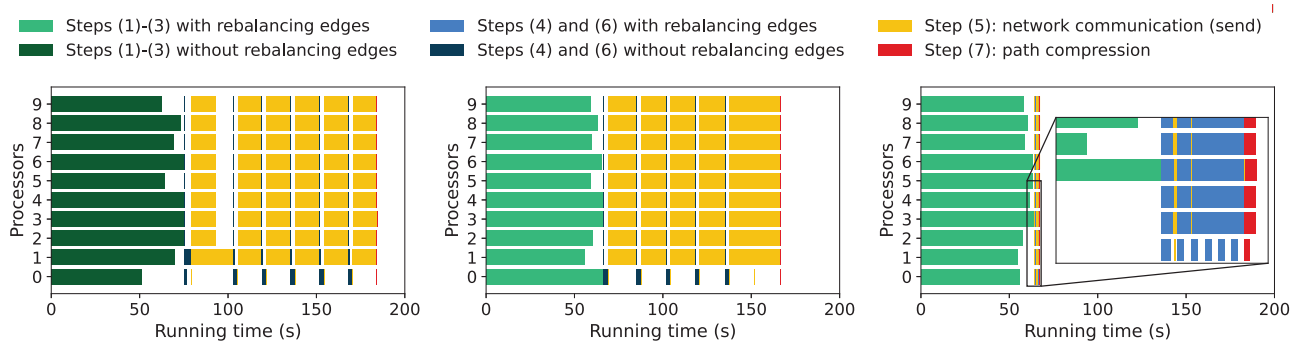
BTS reduces network traffic in two ways: rebalancing edges and excluding unchanged edges from network traffic. To show how much network traffic the methods reduce, we compare three versions of BTS: (a) BTS that neither rebalances edges nor excludes unchanged edges, (b) BTS that rebalances edges but does not exclude unchanged edges, and (c) the original BTS. It is worth noting that (a) and (b) can be regarded as implementations of D-Rem [31] and an improved version of ALBUF [32] within our framework, respectively; while ALBUF rebalances edges only at the initial step, (b) rebalances edges in every round, further reducing data traffic. Fig. 6 shows how many edges 10 processors send and receive for each round in the three versions of BTS. SD is used. Without rebalancing edges, processors differ by up to a factor of 14.4 and 29.6 in the numbers of edges sent and received, respectively, implying a load balancing problem (see red and blue lines). Edge rebalancing resolves the load balancing problem and significantly reduces the numbers of sent and received edges by a factor of 121.2 in round 1 and up to 514.3 in the subsequent rounds (see pink and sky blue lines). Excluding unchanged edges from network communication, the original BTS further decreases the numbers of sent and received edges by a factor of 3.3 (see orange and green lines).

### D. Memory Usage

BTS reduces memory consumption in two ways: rebalancing edges and discarding outer edges. To show how BTS is well-balanced by discarding outer edges so that reduce a large amount of memory, we compare three versions of BTS: (a) BTS that neither rebalances edges nor discards outer edges (red lines), (b) BTS that rebalances edges but does not discard outer edges (blue lines), and (c) the original BTS (black lines). Each processor stores the parent pointers of all vertices in memory, and outer edges take up memory additionally. So, Fig. 7 shows the number of outer edges in each processor immediately after network communication of each round to measure memory overhead. The initialization round is omitted from this figure because it works on chunked edges. The result shows that rebalancing edges dramatically reduces the number of outer edges by up to a factor of 71.2, and discarding outer edges further reduces the number of outer edges by up to a factor of 21.8.

### E. Scalability

We compare BTS to the other algorithms written in bold in Table I in terms of data and machine scalability in Fig. 8. Fig. 8a shows each algorithm's running time on various sized RMAT graphs listed in Table IV. Both axes are in a log scale. BTS scales up almost linearly with a slope of 0.94 and shows the fastest performance on graphs of any size; BTS is 3.1 to 163.3 times faster than other algorithms. Due to out-of-memory errors, the algorithms except BTS and PACC fail to process graphs larger than certain sizes. BTS and PACC handle 16 to 1024 times larger graphs than other algorithms. Meanwhile, BTS is 8 to 55.4 times faster than




(a) BTS-nobal (b) BTS-init (c) BTS

Fig. 5: Gantt chart showing the running time of 10 processors in three variations of BTS: (a) BTS without rebalancing edges, (b) BTS that rebalances edges only at the initial step, and (c) the original BTS, which rebalances edges every round. The steps listed in Fig. 2 are color-coded. Empty parts in bars indicate that the processors are idle or just receiving data from others. The original BTS resolves load balancing problems of BTS-nobal and BTS-init, reducing the running time for network communication by 522× and 553×, respectively.

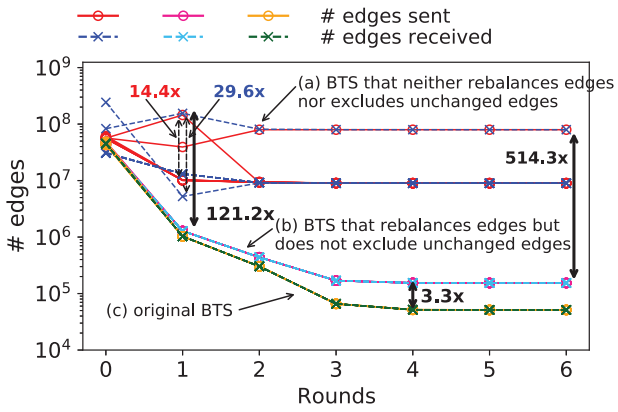



Fig. 6: The number of edges 10 processors send and receive for three methods; (a) BTS that neither rebalances edges nor excludes unchanged edges, (b) BTS that rebalances edges but does not exclude unchanged edges, and (c) the original BTS. While (a) suffers from a load balancing problem, (b) greatly reduces the network traffic by up to 514.3× and (c) further decreases the network traffic by up to 3.3×.

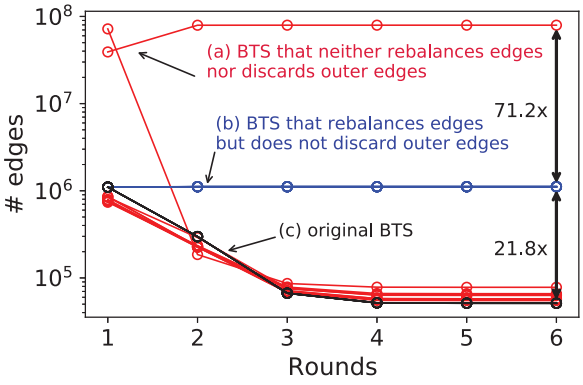


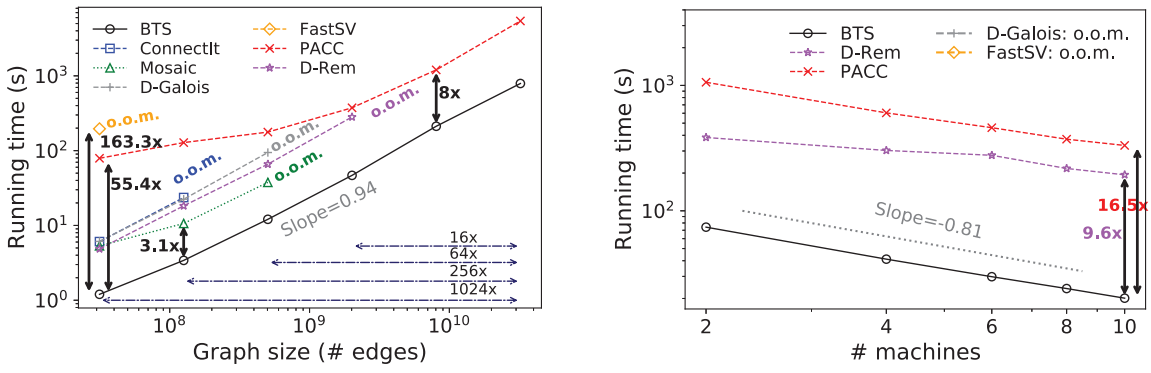

Fig. 7: The number of outer edges for (a) BTS that neither rebalances edges nor discards outer edges (red), (b) BTS that rebalances edges but does not discard outer edges (blue), and (c) the original BTS (black). The original BTS reduces the number of outer edges by a factor of 1552. Each line corresponds to a processor; each color has 10 lines, while blue and black lines perfectly overlap, implying that (b) and (c) are well balanced.

PACC. PACC is relatively slow due to a large amount of data shuffling, which causes massive disk and network I/Os. ConnectIt [35], the state-of-the-art algorithm that implements Union-Find in parallel, reportedly succeeds in handling 100 billion edges by exploiting an expensive server computer with 72 cores and 1TB memory. In our experimental environment where machines have moderate cores and memory, however, ConnectIt fails to handle graphs with more than 500 million edges, implying that ConnectIt requires a lot of memory.

Fig. 8b shows the machine scalability of all the distributed algorithms written in bold in Table I: BTS, D-Rem, D-Galois, FastSV, and PACC. TW is used. D-Galois and FastSV are omitted because they fail to process TW because of out-of-memory errors. We test on the numbers 2, 4, 6, 8, and 10 of machines. Both axes are in a log scale. BTS shows almost linear scalability; the slope is $-0.81$, meaning that doubling the number of machines decreases the running time by $2^{-0.81} = 1.75$ times. The slopes of D-Rem and PACC are -0.42 and -0.72, respectively. Regardless of the number of machines, BTS is always the best, showing up to 9.6 and 16.5 times faster performance than the competitive algorithms, D-Rem and PACC, respectively.

### *F. On Real-World Graphs*

We compare BTS to the algorithms written in bold in Table I on all real-world graphs in Table IV. Fig. 9 shows the running time of all algorithms in a log scale. Methods




(a) (b)

Fig. 8: (a) Data scalability of all algorithms on various sized RMAT graphs. BTS is 3.1 to 163.3 times faster and handles 16 to 1024 times larger graphs than other algorithms. (b) Machine scalability of all distributed algorithms. BTS is the fastest regardless of the number of machines, showing 9.6× and 16.5× faster speeds than D-Rem and PACC, respectively.

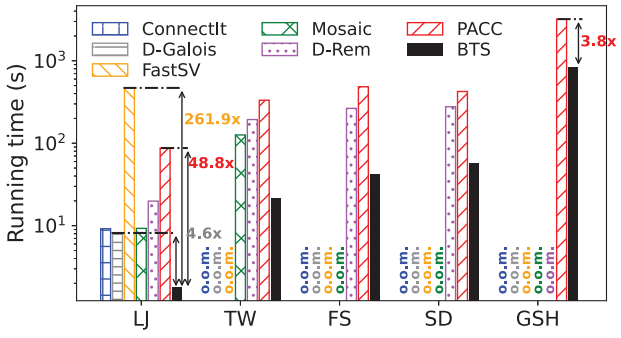



Fig. 9: The running time of all algorithms. BTS is the fastest showing up to 261.9 times faster speed.

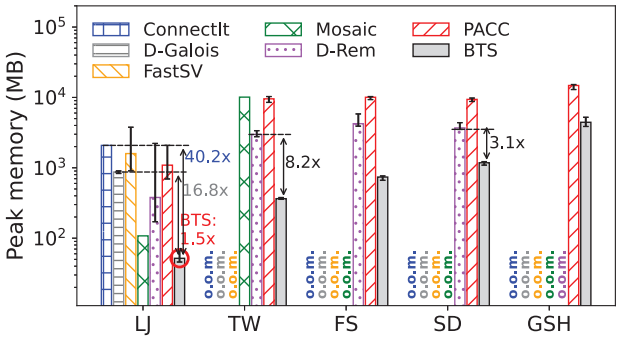



Fig. 10: Box: the average peak memory usage of 10 processors. Error bar: the maximum and minimum values. BTS consumes the least amount of memory on all graphs.

with o.o.m. for some datasets mean they fail to handle the datasets due to out-of-memory errors. Only BTS and PACC succeed in processing all the graphs, while BTS is 3.8 to 48.8 times faster than PACC. On LJ, which is the only graph that all algorithms succeed in handling, BTS outperforms all the algorithms reducing the running time by a factor of 4.6 (D-Galois) to 261.9 (FastSV).

Fig. 10 shows the memory usage of all algorithms in a log scale. Boxes indicate the average peak memory usage of 10 processors, and error bars do the maximum and minimum values. ConnectIt and Mosaic don't have error bars as they are shared-memory algorithms. On every graph, BTS requires the least amount of memory, and the difference in memory usage between processors is negligible, at most 1.5 times. Loading the input graph into memory, ConnectIt, D-Galois, and FastSV use a lot of memory by 16.8 to 40.2 times BTS's. Mosaic uses a lot of memory to manage states even though it is an external algorithm; we have tried various parameters according to the authors' guideline, but in the end, it fails to process FS, SD, and GSH. Loading only parent pointers and boundary edges in memory, D-Rem uses relatively little memory but still uses 3.1 to 8.2 times more memory than BTS; BTS reduces memory usage by rebalancing edges and discarding outer edges. PACC is measured to use a similar amount of memory on TW, FS, SD, and GSH, which appears to be due to the lazy garbage collection of Java.

## VI. Conclusion

In this paper, we propose BTS, a fast and scalable distributed Union-Find algorithm for finding connected components in large graphs. BTS introduces Balanced Union-Find, which rebalances edges to efficiently resolve load balancing problems, reducing the running time for network communication by 553 times and shrinking network traffic and memory usage simultaneously by up to 514.3 and 71.2 times, respectively. BTS further decreases network traffic by up to 3.3 times by excluding unchanged edges and memory usage by up to 21.8 times by discarding outer edges. As a result, BTS outperforms the state-of-the-art algorithms by processing 16-1024 times larger graphs with 3.1-261.9 times faster speeds.

## Acknowledgement

This work was funded by the Korea Meteorological Administration Research and Development Program "Developing Intelligent Assistant Technology and Its Application for Weather Forecasting Process" under Grant (KMA2021-00123). Ha-Myung Park is the corresponding author.

## REFERENCES


[1] Y. Lim, U. Kang, and C. Faloutsos, "Slashburn: Graph compression and mining beyond caveman communities," *TKDE*, vol. 26, no. 12, pp. 3077–3089, 2014.

[2] U. Kang and C. Faloutsos, "Beyond 'caveman communities': Hubs and spokes for graph compression and mining," in *ICDM*. IEEE Computer Society, 2011, pp. 300–309.

[3] L. He, Y. Chao, K. Suzuki, and K. Wu, "Fast connected-component labeling," *Pattern Recognition*, vol. 42, no. 9, pp. 1977–1987, 2009.

[4] K. Wu, E. J. Otoo, and K. Suzuki, "Optimizing two-pass connected-component labeling algorithms," *Pattern Analysis and Applications*, vol. 12, no. 2, pp. 117–135, 2009.

[5] R. Jin, N. Ruan, Y. Xiang, and H. Wang, "Path-tree: An efficient reachability indexing scheme for large directed graphs," *TODS*, vol. 36, no. 1, pp. 7:1–7:44, 2011.

[6] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum, "FERRARI: flexible and efficient reachability range assignment for graph indexing," in *ICDE*. IEEE Computer Society, 2013, pp. 1009–1020.

[7] G. Even, J. Naor, S. Rao, and B. Schieber, "Fast approximate graph partitioning algorithms," *SICOMP*, vol. 28, no. 6, pp. 2187–2214, 1999.

[8] Y. Lim, W. Lee, H. Choi, and U. Kang, "MTP: discovering high quality partitions in real world graphs," *WWW*, vol. 20, no. 3, pp. 491–514, 2017.

[9] D. A. Bader and G. Cong, "A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps)," *JPDC*, vol. 65, no. 9, pp. 994–1006, 2005.

[10] R. A. Pearce, M. B. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *SC*. IEEE, 2010, pp. 1–11.

[11] C. Jain, P. Flick, T. Pan, O. Green, and S. Aluru, "An adaptive parallel algorithm for computing connected components," *TPDS*, vol. 28, no. 9, pp. 2428–2439, 2017.

[12] M. Asokan, "A robust, efficient, and balanced parallel algorithm for finding connected components," in *BigData*. IEEE, 2019, pp. 2113–2118.

[13] X. Feng, L. Chang, X. Lin, L. Qin, and W. Zhang, "Computing connected components with linear communication cost in pregel-like systems," in *ICDE*. IEEE Computer Society, 2016, pp. 85–96.

[14] X. Feng, L. Chang, X. Lin, L. Qin, W. Zhang, and L. Yuan, "Distributed computing connected components with linear communication cost," *Distributed Parallel Databases*, vol. 36, no. 3, pp. 555–592, 2018.

[15] Y. Shiloach and U. Vishkin, "An o(log n) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57–67, 1982.

[16] D. Zheng, D. Mhembere, R. C. Burns, J. T. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity ssds," in *FAST*. USENIX Association, 2015, pp. 45–58.

[17] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, and T. Kim, "Mosaic: Processing a trillion-edge graph on a single machine," in *EuroSys*. ACM, 2017, pp. 527–543.

[18] R. Dathathri, G. Gill, L. Hoang, H. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: a communication-optimizing substrate for distributed heterogeneous graph analytics," in *PLDI*. ACM, 2018, pp. 752–768.

[19] R. Kiveris, S. Lattanzi, V. S. Mirrokni, V. Rastogi, and S. Vassilvitskii, "Connected components in mapreduce and beyond," in *SoCC*. ACM, 2014, pp. 18:1–18:13.

[20] H. Park, N. Park, S. Myaeng, and U. Kang, "Partition aware connected component computation in distributed systems," in *ICDM*. IEEE Computer Society, 2016, pp. 420–429.

[21] H.-M. Park, N. Park, S.-H. Myaeng, and U. Kang, "Pacc: Large scale connected component computation on hadoop and spark," *PLOS ONE*, vol. 15, no. 3, pp. 1–25, 03 2020.

[22] V. Rastogi, A. Machanavajjhala, L. Chitnis, and A. D. Sarma, "Finding connected components in map-reduce in logarithmic rounds," in *ICDE*. IEEE Computer Society, 2013, pp. 50–61.

[23] S. Liu and R. E. Tarjan, "Simple concurrent labeling algorithms for connected components," in *SOSA*, ser. OASIcs, J. T. Fineman and M. Mitzenmacher, Eds., vol. 69. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 3:1–3:20.

[24] S. C. Liu and R. E. Tarjan, "Simple concurrent connected components algorithms," *ACM TOPC*, vol. 9, no. 2, pp. 9:1–9:26, 2022.

[25] G. Guo, H. Chen, D. Yan, J. Cheng, J. Y. Chen, and Z. Chong, "Scalable de novo genome assembly using a pregel-like graph-parallel system," *IEEE ACM TCBB*, vol. 18, no. 2, pp. 731–744, 2021.

[26] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, "Pregel algorithms for graph connectivity problems with performance guarantees," *VLDB*, vol. 7, no. 14, pp. 1821–1832, 2014.

[27] Y. Zhang, A. Azad, and A. Buluç, "Parallel algorithms for finding connected components using linear algebra," *JPDC*, vol. 144, pp. 14–27, 2020.

[28] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system," in *ICDM*. IEEE Computer Society, 2009, pp. 229–238.

[29] G. Cybenko, T. G. Allen, and J. E. Polito, "Practical parallel union-find algorithms for transitive closure and clustering," *International Journal of Parallel Programming*, vol. 17, no. 5, pp. 403–423, 1988.

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

[31] F. Manne and M. M. A. Patwary, "A scalable parallel union-find algorithm for distributed memory computers," in *PPAM*, R. Wyrzykowski, J. J. Dongarra, K. Karczewski, and J. Wasniewski, Eds., vol. 6067. Springer, 2009, pp. 186–195.

[32] J. Xu, H. Guo, H. Shen, M. Raj, X. Wang, X. Xu, Z. Wang, and T. Peterka, "Asynchronous and load-balanced union-find for distributed and parallel scientific data visualization and analysis," *TVCG*, vol. 27, no. 6, pp. 2808–2820, 2021.

[33] R. J. Anderson and H. Woll, "Wait-free parallel algorithms for the union-find problem," in *STOC*. ACM, 1991, pp. 370–380.

[34] N. Simsiri, K. Tangwongsan, S. Tirthapura, and K. Wu, "Work-efficient parallel union-find," *Concurrency Computation Practice and Experience*, vol. 30, no. 4, 2018.

[35] L. Dhulipala, C. Hong, and J. Shun, "Connectit: A framework for static and incremental parallel graph connectivity algorithms," *CoRR*, vol. abs/2008.03909, 2020.

[36] P. K. Agarwal, L. Arge, and K. Yi, "I/o-efficient batched union-find and its applications to terrain analysis," *TALG*, vol. 7, no. 1, pp. 11:1–11:21, 2010.

[37] M. M. A. Patwary, J. R. S. Blair, and F. Manne, "Experiments on union-find algorithms for the disjoint-set data structure," in *SEA*, ser. Lecture Notes in Computer Science, vol. 6049. Springer, 2010, pp. 411–423.

[38] R. E. Tarjan and J. van Leeuwen, "Worst-case analysis of set union algorithms," *JACM*, vol. 31, no. 2, pp. 245–281, 1984.

[39] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[40] A. Azad and A. Buluç, "LACC: A linear-algebraic algorithm for finding connected components in distributed memory," in *IPDPS*. IEEE, 2019, pp. 2–12.

[41] Y. Zhang, A. Azad, and Z. Hu, "Fastsv: A distributed-memory connected component algorithm with fast convergence," in *PPSC*. SIAM, 2020, pp. 46–57.

[42] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SIAM*. SIAM, 2004, pp. 442–446.

[43] B. Jeon, I. Jeon, and U. Kang, "Tegviz: Distributed tera-scale graph generation and visualization," in *ICDMW*. IEEE Computer Society, 2015, pp. 1620–1623.

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

[45] H. Kwak, C. Lee, H. Park, and S. B. Moon, "What is twitter, a social network or a news media?" in *WWW*, M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, Eds. ACM, 2010, pp. 591–600.

[46] (2012) Webscope. [Online]. Available: http://webdatacommons.org/hyperlinkgraph/

[47] (2015) Webgraph. [Online]. Available: http://law.di.unimi.it/datasets.php