Work-Efficient Parallel Union Find

Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, Kun-Lung Wu
Presented By: Sanjana Mupparaju

Problem: Incremental Graph Connectivity (IGC)

- Maintain a data structure that can efficiently answer whether two vertices of a graph are connected
  - Must handle the insertion of edges
- Why?
  - Rise of large **linked** and **dynamic** graph data – social networks, recommendation graphs, communication networks (IoT devices)
    - These graphs are updated continuously as new connections (edges) arrive
    - Many applications need to query connectivity in real time
    - Sequential algorithms cannot keep up with this scale

Equivalence to Union-Find

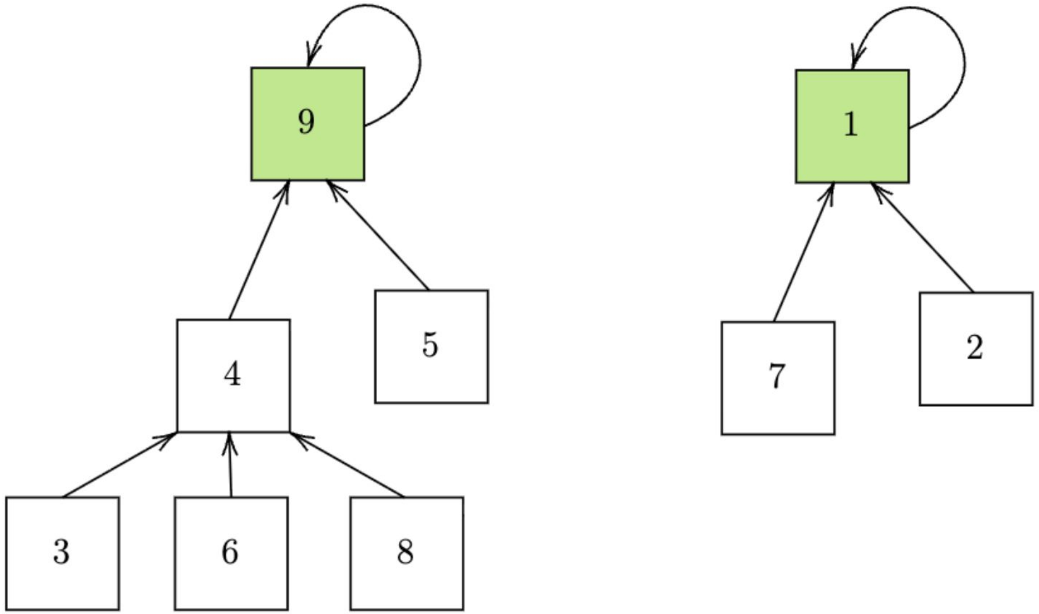
- **Union-Find (Disjoint Set Union)** is a data structure representing disjoint sets supports which supports two key operations.
  - **union(u, v)** - given elements u and v, combine the sets of u and v into one, and return a handle/pointer to the combined set.
  - **find(v) -** given element v, return a handle/pointer to the set containing v. If two items, u and v, are in the same set, then it is guaranteed that find(u) = find(v).
- Natural equivalent to IGC
  - Disjoint sets = connected components of input graph
  - If two vertices of graph are connected, then find(u) = find(v).
  - Adding edge (w, x) to graph is equivalent to union(w, x) operation.

Motivation to Parallelize

- In the applications we mentioned, this is usually a streaming problem.
  - Edges arrive continuously
  - Edges arrive in a large volume
- Scalable, parallel, dynamic algorithm are needed to properly utilize modern streaming systems which provide softwares for parallel processing.
  - IBM Streams
  - Spark Streaming

Review: Sequential Union Find Data Structure

- (Tarjan, 1975) - Optimal Sequential Data Structure
    - O(α(m, n)) **amortized** time per find(u)
    - α is the inverse Ackermann function, *n* is number of vertices in graph, and *m* is the number of operations
- Forest-Of-Trees Implementation with Optimizations
    - Union-By-Rank - parent of root of smaller ranked tree is set to root of larger ranked tree
    - Path Compression - find(u) updates pointer of every node it walks along to root node
- Parallelizing is difficult because of shared structure that is modified every operation.



Related Work

- McColl et al. (STINGER)
  - more general: maintain connected components in fully dynamic graphs
  - engineering focus: no theoretical guarantee
- Manne and Patwary
  - parallel union-find for distributed memory computers
- Patwary et al.
  - shared memory parallel algorithm for computing spanning forest using union find
  - no theoretical guarantees work efficiency
- Berry et al (X-stream)
  - methods for maintaining connected components in parallel graph stream model
  - respects sequential ordering of edges
  - ages out edges
  - no provable parallel complexity bounds
- Shun et. al + Hillel
  - algorithms for graph connectivity
  - edges know in advance
- Shiloach and Vishkin
  - not easy to perform parallel path compression

Parallel Setup

- V - fixed vertex set
- graph stream *A* - sequence of mini-batches A_1, A_2, A_3,..
  - each minibatch is a set of edges on V
    - edges can be viewed as a single union(u, v) operation
    - minibatch is a set of unions that are applied parallelly
  - minibatch can be of different sizes
- $G□ = (V, \cup_{i=1}^{t} A_i)$
  - graph at end of observing A_t
- Operations
  - Bulk-Union - takes minibatch as input, and adds edges to graph
    - one union operation/edges
  - Bulk-Same-Set - takes minibatch of vertex pairs, and for each pairs, returns whether the pair are connected in the graph
    - two find operations/vertex-pair
- Bulk-Union and Bulk-Same-Set must occur sequentially

Simple Parallel Union-Find Data Structure

- **Sequential Union Find Lemma:** Every union-find tree has height O(logn). O(logn) sequential time for union and find operations follow.
- Parallel data structure maintains instance of union-find data structure with union-by-size but *read-only finds* (no path-compression).
    - *Simple-Bulk-Set-Same:* O(logn) parallel depth, O(qlogn) parallel work where q is number of queries in mini-batch.

Simple Bulk-Union

Intuition: Safe to run multiple *unions* if they operate on different trees.

Step 1: Parallely, relabel edges (u, v) as (find(u), find(v)). Let these set of edges be called A’. Remove all edges where find(u) = find(v).

Step 2: Now, we are working with graph (H, A’) - where H is all the vertices appearing in the edges of A’. Compute connected components of H parallely, outputting C, the set of connected components of H.

Step 3: *Parallel-Join* each connected component of C

Simple Bulk-Union

Intuition: Safe to run multiple *unions* if they operate on different trees.

Step 1: Parallely, relabel edges (u, v) as (find(u), find(v)). Let these set of edges be called A'. Remove all edges where find(u) = find(v).

O(blogn + b) work + O(logn) depth

Step 2: Now, we are working with graph (H, A') - where H is all the vertices appearing in the edges of A'. Compute connected components of H parallely, outputting C, the set of connected components of H.

O(b) work and O(log max (b, n)) depth: Gazit's Algorithm

Step 3: *Parallel-Join* each connected component of C

O(b) work and O(logn) depth

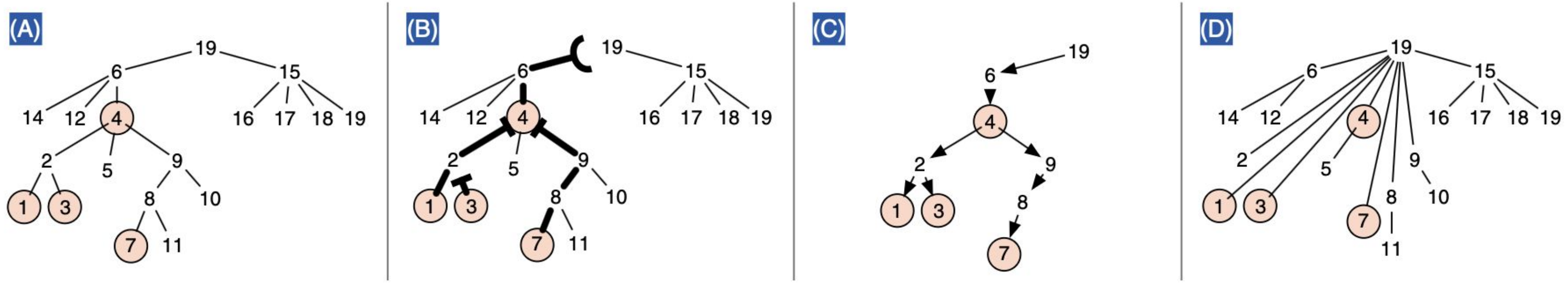
Bulk-Find

Phase 1: Parallely run BFS from all queried vertices.

- If multiple flows meet up, only one will move on
- If node was visited, flow will stop
- Maintain all edges traversed as parent child.

Phase 2:

- Reverse BFS from root, reassigning parent pointer.


(A)
19
6
15
14
12
4
16
17
18
19
2
5
9
1
3
8
10
7
11
(B)
19
6
15
14
12
4
16
17
18
19
2
5
9
1
3
8
10
7
11
(C)
19
6
4
2
9
1
3
8
7
(D)
19
6
15
14
12
4
16
17
18
19
2
5
9
1
3
8
10
7
11

Response Distributor

- Have a sequence of ƛ edges, $(\text{from}_i, \text{to}_i)$.
- To efficiently do the BFS in phase2 , need a way to efficiently find allFrom(f), where $\text{from}_i = f$.
- h() - hash function from from_i's to p = 3ƛ.
- Constructing RD
    - Compute hash for each $(\text{from}_i, \text{to}_i)$ and sort into ordered array A.
    - Create array *o* of length *p + 1*, where o_i marks beginning of pairs whose hash value is *i.* If none has to *i*, then $o_i = o_{i+1}$.
- allfrom(f).
    - Calculate h(f)
    - Look in A between $o_{h(f)}$ *and* o_{hf+1}*.-1.*

Algorithm 4: `Bulk-Find`(U, S)—find the root in U for each $s \in S$ with path compression.

```
Input: $U$ is the union find structure. For $i = 1, \ldots, |S|$, $S[i]$ is a vertex in the graph
Output: A response array $res$ of length $|S|$ where $res[i]$ is the root of the tree of the vertex
        $S[i]$ in the input.
▷ Phase I: Find the roots for all queries
1:  $R_0 \leftarrow \langle (S[k], \textbf{null}) : k = 0, 1, 2, \ldots, |S| - 1 \rangle$
2:  $F_0 \leftarrow \texttt{mkFrontier}(R_0, \varnothing)$, $roots \leftarrow \varnothing$, $visited \leftarrow \varnothing$, $i \leftarrow 0$
3:  while $R_i \neq \varnothing$ do
4:      $visited \leftarrow visited \cup F_i$
5:      $R_{i+1} \leftarrow \langle (\texttt{parent}[v], v) : v \in F_i$ and $\texttt{parent}[v] \neq v \rangle$
6:      $roots \leftarrow roots \cup \{v : v \in F_i$ where $\texttt{parent}[v] = v\}$
7:      $F_{i+1} \leftarrow \texttt{mkFrontier}(R_{i+1}, visited)$, $i \leftarrow i + 1$
▷ Set up response distribution
8:  Create an instance of $RD$ with $R_\cup = R_0 \oplus R_1 \oplus \cdots \oplus R_i$
▷ Phase II: Distribute the answers and shorten the paths
9:  $D_0 \leftarrow \{(r, r) : r \in roots\}$, $i \leftarrow 0$
10: while $D_i \neq \varnothing$ do
11:     For each $(v, r) \in D_i$, in parallel, $\texttt{parent}[v] \leftarrow r$
12:     $D_{i+1} \leftarrow \bigcup_{(v,r) \in D_i} \{(u, r) : u \in RD.\texttt{allFrom}(v)$ and $u \neq \textbf{null}\}$. That is, create $D_{i+1}$ by
          expanding every $(v, r) \in D_i$ as the entries of $RD.\texttt{allFrom}(v)$ excluding null, each
          inheriting $r$.
13:     $i \leftarrow i + 1$
14: For $i = 0, 1, 2 \ldots, |S| - 1$, in parallel, make $res[i] \leftarrow \texttt{parent}[S[i]]$
15: return $res$
```

```
def mkFrontier($R$, $visited$):
  // nodes to go to next
1: $req \leftarrow \langle v : (v, \_) \in R \wedge$
     not $visited[v] \rangle$
2: return removeDup($req$)
```

Theoretical Guarantees

- response distributor can be constructed in O(λ)work and O(polylog(n))depth

- Bulk-Find(U,S)does O(|R$\cup$|)work and has O(polylog(n))depth.

Experimental Setup

- Implemented all algorithms in C++ using the Ligra+ parallel graph processing framework.
- Experiments run on a 72-core Intel Xeon Gold 6252 processor (144 hyper-threads) with 1 TB memory.
- Graphs stored in compressed sparse row (CSR) format for efficient parallel access.
- Compared **Bulk-Parallel Union-Find** against:
- Sequential Union-Find (Tarjan, 1975)
- Existing parallel algorithms (e.g., Shun et al., Hillel & Vishkin)
- Evaluated performance on both **static** and **incremental/dynamic** graph workloads.
- Measured **speedup**, **work efficiency**, and **parallel depth** across varying batch sizes.

Experimental Results

Datasets: Evaluated on large real-world graphs (Twitter, LiveJournal, Orkut, Friendster).

Performance: Achieved up to **40–60× speedup** over the sequential union–find baseline.

Work efficiency: Total work remains close to sequential; scales nearly linearly with core count.

Batch size impact: Larger minibatches yield better parallelism and smaller depth.

Further Work/Limitations

- Algorithm currently handles **incremental connectivity** — extending to **fully dynamic graphs** (with deletions) remains open.
- Performance depends on **batch size selection** — very small batches offer limited parallel speedup.
- **Memory overhead** of auxiliary data structures (e.g., Response Distributor) could be reduced.
- Adapting approach to **heterogeneous systems** (GPUs, distributed memory) could further improve scalability.