Work-Efficient Parallel Union-Find

By Simsiri, Tangwongsan, Tirthapura, Wu

Reviewed by Kasra Mazaheri

The Problem

- **Incremental Graph Connectivity (IGC) or the Union-Find Problem:** A problem where the goal is to determine if two vertices in a graph are connected, with the graph dynamically growing as edges are added.

- **Challenge:** Traditional sequential solutions don't scale for large, rapidly-changing graphs typical in many real-world applications, necessitating a parallel solution.

- **Real-World Applications:** Parallel Union-Find algorithms are crucial in social network analysis for identifying communities, in network connectivity for understanding the robustness of the internet infrastructure, and in clustering algorithms for data analysis across diverse datasets.

Related Work and State of the World

- **Sequential Union-Find:** The sequential case has been thoroughly been studied, with Tarjan's algorithm [5] having been shown optimal (O(α(m,n)) per find).
  - These algorithms cannot exploit parallelism however.

- **Memory-Constrained Union-Find:** Demetrescu et al. present a multipass memory-constrained union-find.

- **Fully Dynamic Parallel IGC:** McColletal [11] solve the fully dynamic IGC, but with no theoretical bounds.

- **Distributed-Memory Parallel Union-Find:** Manne and Patwary [12]

- **Other Batched Parallel Union-Find:** Shiloach and Vishkin [17]

Definitions and Preliminaries

- **The Minibatch Streaming Model:** Input is streamed in mini batches of varying size, with each minibatch only containing union operations or finds. Mini batches are received sequentially, but each minibatch can itself be solved in parallel.

- **CRCW PEM Model:** Threads are assumed to have the **concurrent-read concurrent-write** model of shared memory. The algorithm can work in EREW with an extra logarithmic factor, because of parallel integer sort.

- **n:** The total number of vertices.
- **m:** The total number of edges.
- **q:** The total number of queries.
- **b:** The number of queries within a minibatch.

Contributions

1. **Simple Parallel Algorithm**
    a. Bulk-Union
    b. Bulk-Same-Set
2. **Work-Efficient Parallel Algorithm**
3. **Implementation and Evaluation**

The Simple Parallel Union-Find

The basic idea is **Union-Find with size comparison.**

1. **Bulk-Same-Set:** In parallel answer all queries by comparing the result of finds.
   - The parallel depth: O(log n)
   - The total work: O(q log n)

2. **Bulk-Union:** A bit more complicated.

Bulk-Union:

1. Relabel queries to component **supernodes**, i.e., the identifier.
2. Remove self-loops with supernodes.
3. Add the new edges between supernodes to create some connected components (CC).
4. For each CC, unite the supernodes using **Parallel-Join**.

Algorithm 3: `Parallel-Join`(U, C)

Input: U: the union-find structure, C: a seq. of tree roots
Output: The root of the tree after all of C are connected

1: **if** $|C| == 1$ **then**
2: | **return** $C[1]$
3: **else**
4: | $\ell \leftarrow \lfloor |C|/2 \rfloor$
5: | $u \leftarrow$ `Parallel-Join`$(U, C[1, 2, \ldots, \ell])$ **in parallel with**
| $v \leftarrow$ `Parallel-Join`$(U, C[\ell + 1, \ell + 2, \ldots, |C|])$
6: | **return** U.`union`(u, v)

Bulk-Union: Work and Depth Analysis

1. **Relabeling:** O(b log n) work and O(log n) depth.
2. **Filtering self-loops:** O(b) work and O(log b) depth.
3. **Create the supernode connected components (CC):**
   O(b) work and O(log max(b,n)) depth using Gazit's algorithm [16].
4. **Parallel-Join each component:** O(b) work and O(log n) depth.

Algorithm 3: `Parallel-Join`(U, C)

Input: U: the union-find structure, C: a seq. of tree roots
Output: The root of the tree after all of C are connected

```
1: if |C| == 1 then
2:     return C[1]
3: else
4:     ℓ ← ⌊|C|/2⌋
5:     u ← Parallel-Join(U, C[1, 2, ..., ℓ]) in parallel with
         v ← Parallel-Join(U, C[ℓ + 1, ℓ + 2, ..., |C|])
6:     return U.union(u, v)
```

The Simple Parallel Union-Find

The basic idea is **Union-Find with size comparison.**

1. **Bulk-Same-Set:** In parallel answer all queries by comparing the result of finds.
   - **The parallel depth:** O(log n)
   - **The total work:** O(q log n)

2. **Bulk-Union:** A bit more complicated.
   - **The parallel depth:** O(log max(n, b))
   - **The parallel work:** O(b log n)

The Work-Efficient Parallel Union-Find

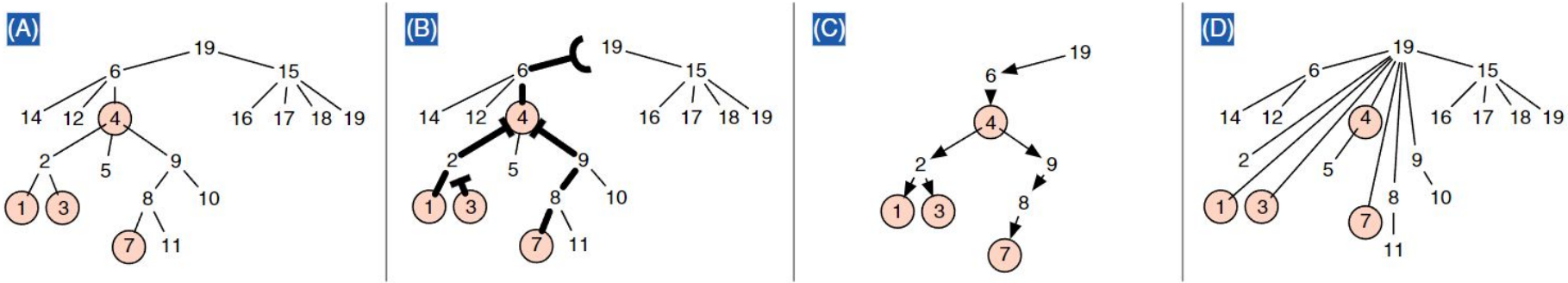
The basic idea is **Union-Find with size comparison, with path compression.**

1. **Bulk-Same-Set:** A bit complicated.
2. **Bulk-Union:** Same as the simple union-find algorithm.

Bulk-Find:

The general idea as an example.

A: An example union-find tree with sample queries circled.
B: bolded edges are paths, together with their stopping points that result from the traversal in Phase I.
C: the traversal graph R ∪ recorded as a result of Phase I.
D: the union-find tree after Phase II, which updates all traversed nodes to point to their roots



Bulk-Find:

Phase I:

- **Find the roots** of all queries, **coalescing *flows* as they meet**, through what's effectively a parallel BFS.
- **Record the traversed paths to distribute query responses** in Phase II.

Phase II:

- **Distribute responses through** a *backwards* BFS from all reached roots.
- **Compress the traversed paths** as the backwards BFS is run.

Bulk-Find:

The algorithm in greater detail, albeit unreadable:

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: $\quad visited \leftarrow visited \cup F_i$

5: $\quad R_{i+1} \leftarrow \langle (\texttt{parent}[v], v) \ : \ v \in F_i \text{ and } \texttt{parent}[v] \neq v \rangle$

6: $\quad roots \leftarrow roots \cup \{v \ : \ v \in F_i \text{ where } \texttt{parent}[v] = v\}$

7: $\quad F_{i+1} \leftarrow \texttt{mkFrontier}(R_{i+1}, visited)$, $i \leftarrow i + 1$

▷ *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: $\quad$ For each $(v, r) \in D_i$, in parallel, $\texttt{parent}[v] \leftarrow r$

12: $\quad D_{i+1} \leftarrow \bigcup_{(v,r) \in D_i} \{(u, r) \ : \ u \in RD.\texttt{allFrom}(v) \text{ and } u \neq \textbf{null}\}$. 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: $\quad i \leftarrow i + 1$

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)

Bulk-Find:

Phase I:

- **Find the roots** of all queries, **coalescing *flows* as they meet**, through what's effectively a parallel BFS.
- **Record the traversed paths to distribute query responses** in Phase II.

Phase II:

- **Distribute responses through** a *backwards* BFS from all reached roots.
- **Compress the traversed paths** as the backwards BFS is run.

Technical Note: Phase I **records** the traversed edges **as a list of edges**, while the backwards BFS in Phase II **requires adjacency lists** to perform optimally. The authors therefore design a data structure using hashing and Parallel Integer Sort to perform this conversion efficiently.

The Work-Efficient Parallel Union-Find

Work Analysis:

The authors use the following powerful lemma to prove the work-efficiency of the Parallel Union-Find algorithm:

Lemma (11): For a sequence of queries S with which Bulk-Find(U, S) is invoked, there is a sequence S' that is a permutation of S such that applying U.find to S' serially in that order yields the same union-find forest as Bulk-Find's and incurs the same traversal cost of $O(|R_U|)$, where R_U is as defined in the Bulk-Find algorithm.

A Practical Implementation

The authors used the **Simple Parallel Union-Find algorithm as the base algorithm**, with the following modifications:

1. **Path Compression:** Parallel-Find operations independently find the root of their trees, and then after finding the root, they make a second iteration on the path to **compress** it.

2. **Parallel Connected Components:** The Bulk-Union operation uses a parallel CC algorithm that while work-efficient, assumes random-access to a node's neighbors which in practice can be **costly**. The authors therefore use an algorithm by Blelloch et al. [12] with worse theoretical bounds that **works directly with a list of edges**.

Experimental Setup

The authors used the **Simple Parallel Union-Find algorithm as the base algorithm**, with the following modifications:

1. **Environment:** An Amazon EC2 instance with 20 cores and 2-way hyperthreading.

2. **Parallel Scheduling:** Intel Cilk's work-stealing scheduler built in Clang 3.4.

3. **Datasets:** A collection of synthetic graphs, including power-law-type graphs and more "regular" ones. These are similar to the graphs used by McColl et al. They also use graphs with varying rates of growing connectivity.

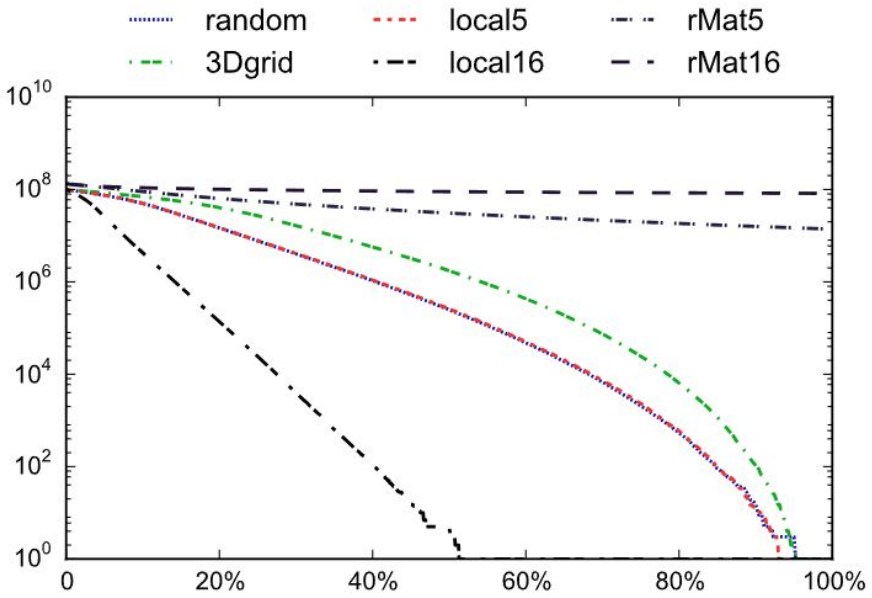
4. **Baseline:** The Sequential Union-Find algorithm, since most other work do not solve the same problem, notably with McColl et al.'s work solving the fully dynamic IGC problem.

Experimental Setup

On varying rates of growing connectivity:

TABLE 1 Characteristics of the graph streams used in our experiments, showing for every dataset, the total number of nodes (n), the total number of edges (m), and a brief description

Graph	#Vertices	#Edges	Notes
3Dgrid	99.9M	300M	3-d mesh
random	100M	500M	5 randomly-chosen neighbors per node
local5	100M	500M	small separators, avg. degree 5
local16	100M	1.6B	small separators, avg. degree 16
rMat5	134M	500M	power-law graph using rMat[23]
rMat16	134M	1.6B	a denser rMat graph



Results

Sequential Benchmarking:

When run sequentially, the Practical Parallel Union-Find is within **2.2-4.3x** of the optimal work-efficient algorithm's performance, exhibiting a **reasonably good** performance.

TABLE 2 Running times (in seconds) on 1 thread of the baseline union-find implementation (UF) with and without path compression and the bulk-parallel version as the batch size is varied

Graph	UF (no p.c.)	UF (p.c.)	Bulk-Parallel using batch size			
			500K	1M	5M	10M
random	44.63	18.42	65.43	66.57	75.20	77.89
3Dgrid	30.26	14.37	61.10	62.00	71.74	75.07
local5	44.94	18.51	65.84	66.77	75.33	78.23
local16	154.40	46.12	114.34	108.92	114.80	117.55
rMat5	33.39	18.47	66.98	68.48	74.97	78.69
rMat16	81.74	35.29	83.27	76.64	76.03	77.62

Results

Parallel Benchmarking:

When run parallely, the Practical Parallel Union-Find exhibits **8-11x speedups** for a batch size of **10M**.

TABLE 3 Average throughput (in million edges/second) and speedup of `Bulk-Union` for different batch sizes b, where T_1 is throughput on 1 thread and T_{20c} is the throughput on 20 cores

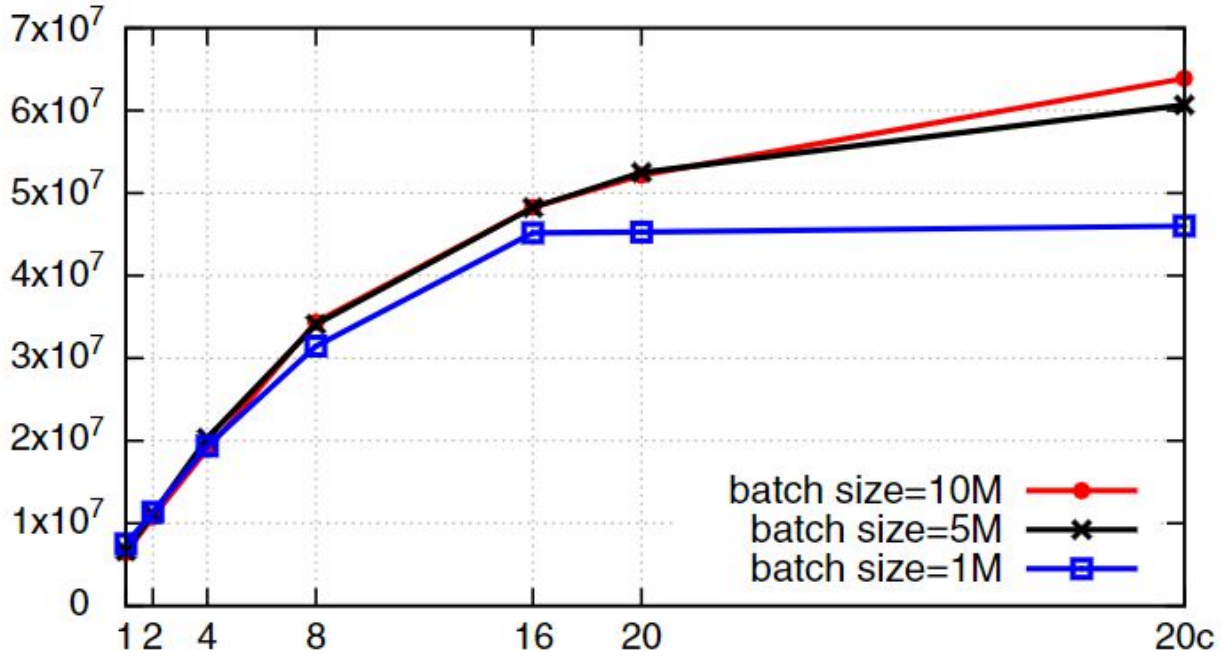
Graph	Using $b = 500K$			Using $b = 1M$			Using $b = 5M$			Using $b = 10M$		
	T_1	T_{20c}	T_{20c}/T_1	T_1	T_{20c}	T_{20c}/T_1	T_1	T_{20c}	T_{20c}/T_1	T_1	T_{20c}	T_{20c}/T_1
random	7.64	36.87	4.8x	7.51	46.02	6.1x	6.65	60.66	9.1x	6.42	63.90	10.0x
3Dgrid	4.91	27.97	5.7x	4.83	34.97	7.2x	4.18	44.27	10.6x	3.99	45.24	11.3x
local5	7.59	38.41	5.1x	7.49	48.32	6.5x	6.64	64.61	9.7x	6.39	64.09	10.0x
local16	13.99	78.83	5.6x	14.69	95.57	6.5x	13.94	122.69	8.8x	13.61	122.03	9.0x
rMat5	7.47	26.08	3.5x	7.30	34.19	4.7x	6.67	49.92	7.5x	6.35	50.37	7.9x
rMat16	19.21	54.94	2.9x	20.88	78.10	3.7x	21.05	143.63	6.8x	20.61	167.68	8.1x

Results

Parallel Benchmarking:

Average throughput (edges per seconds) for varying number of threads in random graphs are plotted on the right:

20c denotes 20 threads with **2-way hyperthreading**.



Future Work

- **A hybrid IGC solver** that can dynamically choose between a DFS and Union-Find based approach, to optimize performance based on batch size.
    - Not really practical since for all practical purposes, the inverse Ackerman's function is constant.
- **Extending the results** of work-efficient Union-Find **to fully dynamic IGC**, where edges deletions are allowed as well as edge additions.

Evaluation

- **Strengths**
  - Introduces a novel, work-efficient parallel algorithm for Union-Find with significant theoretical and practical contributions.
  - Demonstrates scalability and practicality through comprehensive benchmarks and experimental results.*
  - Opens new avenues for research in parallel algorithms for dynamic graph problems.
- **Weaknesses**
  - Lacks direct comparison with distributed systems, which could provide insights into its relative performance.
  - Does not fully address the challenges of graph dynamics that include edge deletions.