Work-efficient parallel union-find

Natcha Simsiri[1] | Kanat Tangwongsan[2] | Srikanta Tirthapura[3] | Kun-Lung Wu[4]

[1]College of Information and Computer Sciences, University of Massachusetts-Amherst, Amherst, MA, USA
[2]Computer Science Program, Mahidol University International College, Nakhon Pathom, Thailand
[3]Department of Electrical and Computer Engineering, Iowa State University, Ames, IA, USA
[4]IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

6.506 Paper Presentation

Thomas Bergamaschi

Motivation and Results

The Incremental Graph Connectivity Problem is Considered:

Given a graph G in which one receives edge updates dynamically, how can one answer connectivity queries between pairs of vertices?

- This can be solved using the Union-Find data structure sequentially easily, but what about in parallel?
- The main contribution to this paper is a parallel data structure for Union-Find, which guarantees work efficiency (inverse Ackermann and polylog depth)
- They also implement this algorithm and show how the performance scales practically with number of cores and graph properties.

Outline

1. 6.046/6.1220 Recap: Union-Find data structure
2. Preliminaries and Notation for Parallel Edge Streams
3. Prior Work
4. Simple Parallel Data Structure – Without Path Compression
5. How to include Path Compression?
6. Implementation and Results
7. Conclusion

6.046 Recap: Union Find Data Structure

Recall the Union-Find Data Structure:

- Want to maintain a collection of disjoint sets and support two operations:

  union(u, v) and *find(v)*

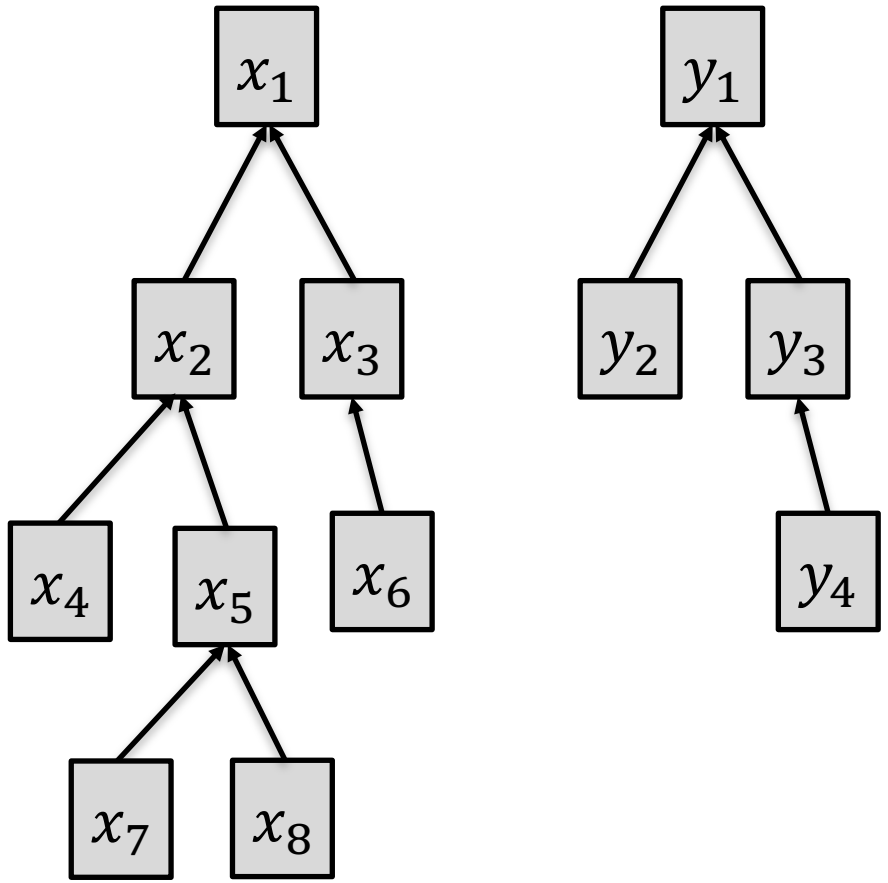
  union(u, v): combine the sets containing *u* and *v*
  find(v): return a representative of the set containing *v*

- Idea: use a forest of trees representation with parent pointers
- The representative of a set is the root node

union(u, v): climb tree for both *u* and *v* and set pointers ⇒ $O(h)$ time

find(v): climb tree for *v* and returns root ⇒ $O(h)$ time

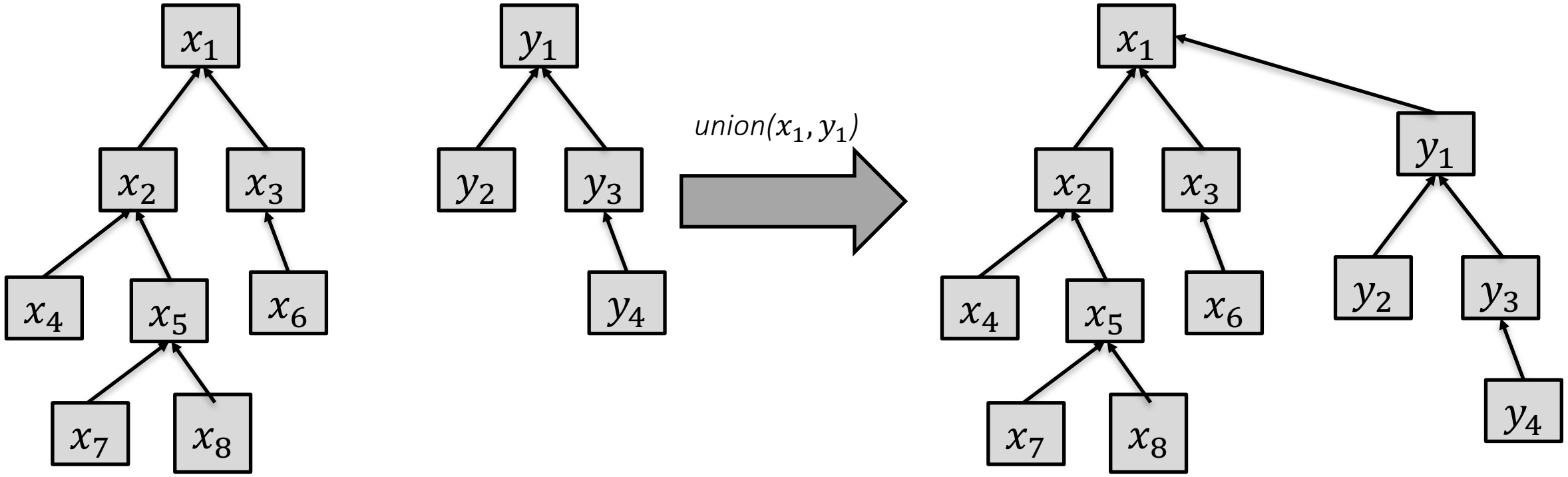
Worst case performance is $O(n)$
How to improve?



6.046 Recap: Union By Size

Union-Find Data Structure:

1. Union by size – Merge the smaller height tree onto the higher one:
   - Guarantees heights bounded by $O(\log n)$ ⇒ Amortized Work now $O(\log n)$

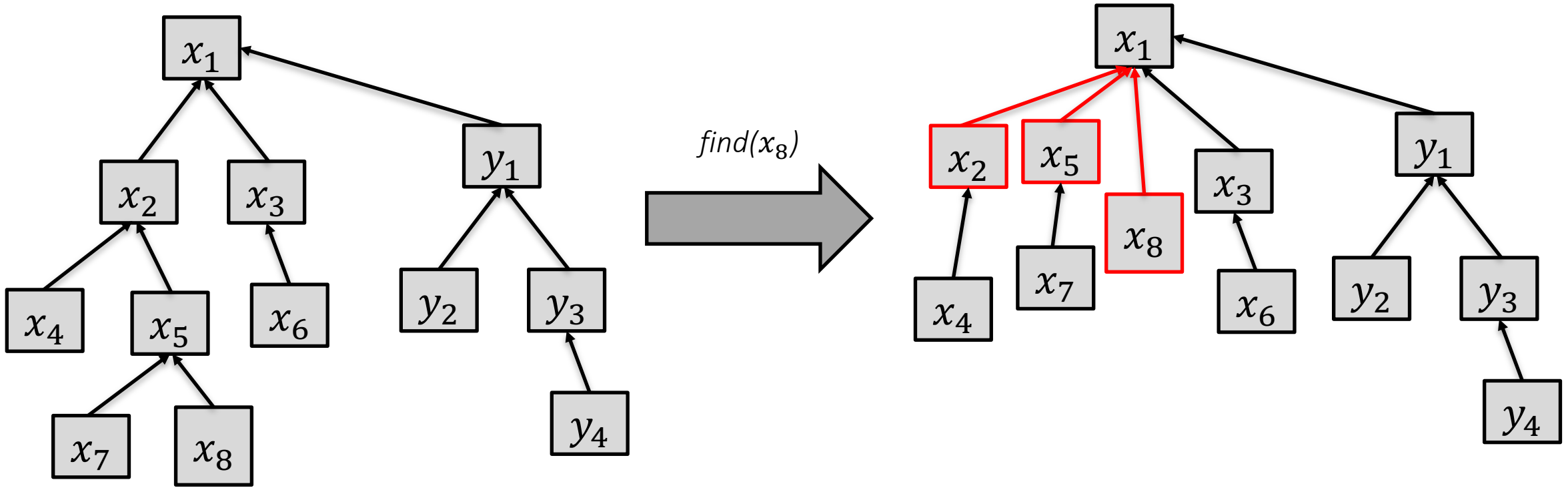


Imagine repeated queries involving nodes at the bottom of tree: can we avoid wasting work?

6.046 Recap: Path Compression

Union-Find Data Structure:

2. Path Compression – Redirect Parent Pointers to Avoid Repeated Tree climbing
   - Via potential functions can show amortized $O(\log n)$ cost



Path Compression can Shorten Long Paths which would previously involve wasting work.

6.046 Recap: Combining Both Heuristics

Union-Find Data Structure:

Why not combine Union by Size and Path Compression?

- In 1973, amortized cost of $O(\log^* n)$ per operation.
- In 1975, Tarjan showed an amortized cost of $O\big(\alpha(m,n)\big)$ per operation - $\alpha(m,n)$ is the inverse Ackermann function and grows *incredibly* slowly.
- In 1989, an amortized $\Omega(\alpha(m,n))$ amortized cost lower bound was shown.
- Has applications in Kruskal's algorithm for Minimum Spanning Trees, Incremental Graph Connectivity Problem, Cilk's *Nondeterminator*, and many more

What about a parallel version of this problem?

Parallel Edge Streams and Notation

We consider the ***Incremental Connectivity Problem***:

Given a fixed set of vertices V, we will receive a *stream* of graph edges, and the queries are connectivity queries:

- We receive a graph stream $\mathcal{A}$, composed of sequential minibatches $A_1, A_2, \ldots, A_t$, where each A_i is a set of edges on V
- The graph after A_i is received is $G_i = (V, \bigcup_{t=1}^{i} A_t)$, and each A_i can have different sizes.
- Each minibatch can be thought of a set of union operations which need to be executed in parallel.
- We want to support two operations:

Bulk-Union(A_i): add the edges in A_i to the graph in parallel

Bulk-Find($\{u_i, v_i\}_{i=1}^{k}$): return for each pair of vertices if they are connected at that point in the stream

With m unions, and q find operations, Union-Find can handle this in $O((m+q)\alpha(m+q, n))$ work

Algorithm Assumptions

Notation and Assumed Algorithms:

We assume two parallel algorithms throughout the paper:

Parallel Integer Sort:

- Given integers $a_1, a_2 \ldots, a_n$ where $a_i \in [0, O(1) \cdot n]$, we can produce a sorted sequence in $O(n)$ work and $O(\text{polylog}(n))$ depth

Parallel Connectivity:

- Given a graph G, we can compute a sequence of connected components in $O(V + E)$ work and $O(\text{polylog}(V))$ depth – recall the (β, d) decomposition algorithm we saw in Lecture 6

Prior Work

A lot of Previous work has tackled variants of this problem:

- Some work on minimizing the storage requirement in a “streaming model” by Feigenbaum et al.
- Assuming only $o(n)$ workspace can compute connected components in $\Omega\left(\frac{n}{s}\right)$ passes using only $O(s)$ memory.
- Algorithms for parallel connected components by Shun et al which are work efficient and polylog depth (Lecture 6).
- Not much work on parallel algorithms for incremental connectivity…

Simple Parallel Data Structure: Queries

How to Respond to Find Queries?

Bulk-Find($\{u_i, v_i\}_{i=1}^{q}$): return for each pair of vertices if they are connected at that point in the stream

Initially, ignore Path Compression:

- Without Path Compression, connectivity queries are **read-only**, so run find queries entirely in parallel

Algorithm 1: `Simple-Bulk-Same-Set`$(U, \langle (u_i, v_i) \rangle_{i=1}^{q})$.

Input: U is the union find structure, and (u_i, v_i) is a pair of vertices, for $i = 1, \ldots, q$.
Output: For each i, whether or not u_i and v_i are in the same set (i.e., connected in the graph).

1: **for** $i = 1, 2, \ldots, q$ **do** in parallel
2: $\quad a_i \leftarrow (U.\texttt{find}(u_i) == U.\texttt{find}(v_i))$
3: **return** $\langle a_1, a_2, \ldots, a_q \rangle$

- The parallel complexity is simply $O(\log n)$, and work is $O(q \log n)$ – just by inheriting the properties of Union-Find

How to handle updates in parallel?

Simple Parallel Data Structure: Updates

How to handle updates in parallel?

Bulk-Union(A_i): add the edges in A_i to the graph in parallel

Initially, ignore Path Compression:

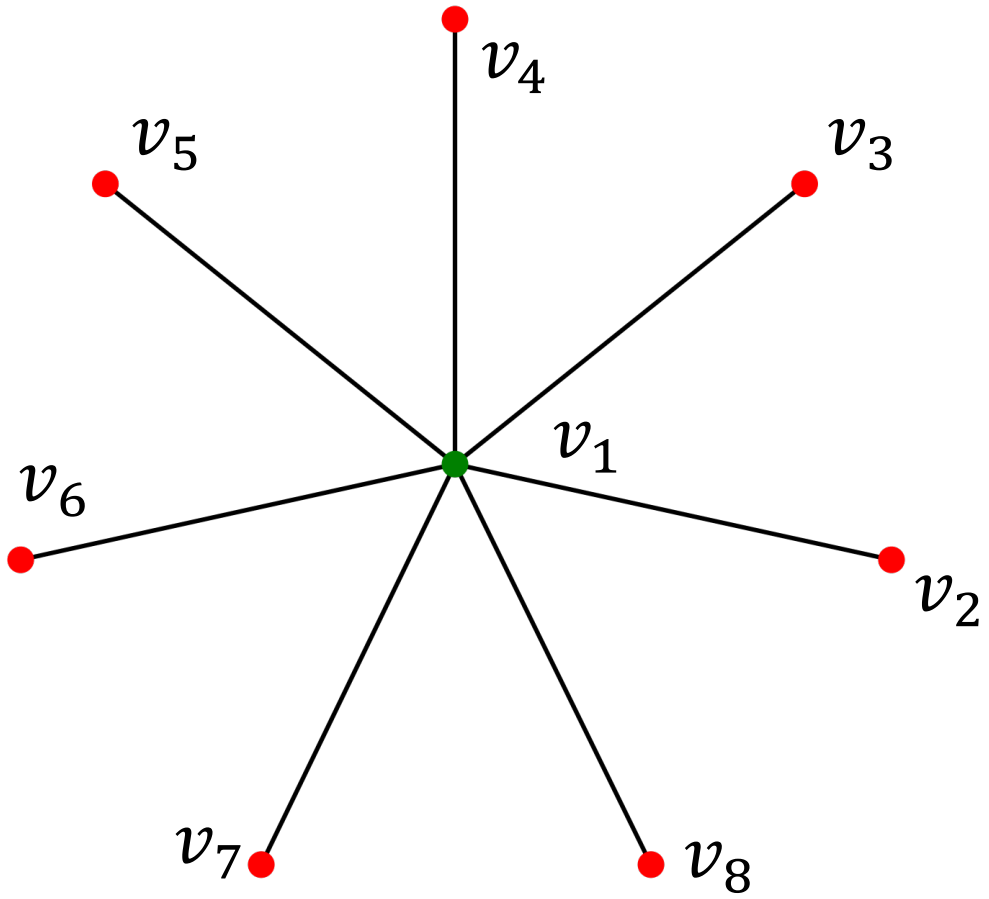
- At first sounds simple: just apply unions for each edge in A_i in parallel ⇒ bad because unions will update the forest, so unsafe to naively run
- However, if union operations are isolated to different trees they **can** run in parallel!
- In the worst case this can still lead to large parallel depth: just consider many operations on the same tree.

How can we create a schedule of unions representing the minibatch A_i but can be run in parallel?

Simple Parallel Data Structure: Scheduling Updates

How to schedule minibatch updates in parallel?

Consider a minibatch of 7 edges forming this star graph:



- Running *union*(v_1, v_2), *union*(v_1, v_3), … , *union*(v_1, v_8) is inherently sequential and will take 7 rounds of unions
- Can we run some unions in parallel? Consider running *union* on an equivalent set of vertices which can be ran in parallel:
  1. Run *union*(v_1, v_2), *union*(v_3, v_4), *union*(v_5, v_6), *union*(v_7, v_8)
  2. Run *union*(v_1, v_3), *union*(v_5, v_7)
  3. *Run union*(v_1, v_5)
- Each union involves different vertices and representatives so can be ran in parallel! How can we find such an equivalent set?

Simple Parallel Data Structure: Algorithm

How to handle updates in parallel?

Bulk-Union(A_i): add the edges in A_i to the graph in parallel

Algorithm 2: `Simple-Bulk-Union`(U, A)

Input: U: the union find structure, A: a set of edges to add to the graph.

▷ *Relabel each (u, v) with the roots of u and v*

1: $A' \leftarrow \langle (p_u, p_v) : (u, v) \in A \text{ where } p_u = U.\texttt{find}(u) \text{ and } p_v = U.\texttt{find}(v) \rangle$

▷ *Remove self-loops*

2: $A'' \leftarrow \langle (u, v) : (u, v) \in A' \text{ where } u \neq v \rangle$

3: $\mathcal{C} \leftarrow \mathsf{CC}(A'')$

4: **foreach** $C \in \mathcal{C}$ **do** in parallel

5: | `Parallel-Join`(U, C)

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: | $u \leftarrow \texttt{Parallel-Join}(U, C[1, 2, \ldots, \ell])$ **in parallel with**
| $v \leftarrow \texttt{Parallel-Join}(U, C[\ell + 1, \ell + 2, \ldots, |C|])$

6: | **return** $U.\texttt{union}(u, v)$

1. First create an equivalent minibatch of the new edges in terms of connected components – without self edges.

2. Over each set of representative nodes which will compose a connected component, join them in parallel

Work on a minibatch of b edges is $O(b \log n)$, and span is $O(\log \max(b, n))$.

Simple Parallel Data Structure: Overall Results

Combining Everything:

Bulk-Find($\{u_i, v_i\}_{i=1}^{k}$): return for each pair of vertices if they are connected at that point in the stream

Bulk-Union(A_i): add the edges in A_i to the graph in parallel

Initially, ignore Path Compression:

- Our simple parallel data structure achieves $O\big((m+q)\log n\big)$ work and polylog depth.
- We are manifestly not work efficient as we don't reach $O\big((m+q)\,\alpha(m+q,n)\big)$ work.

How can Path Compression be implemented in parallel?

How to Include Path Compression?

How to run Path Compression in parallel:

Bulk-Find($\{u_i, v_i\}_{i=1}^{k}$): return for each pair of vertices if they are connected at that point in the stream

- Shortcutting paths cannot be naively implemented when *find* operations are ran in parallel.

- Key Idea is to split *find* into two phases: read only in Phase 1, write updates to tree safely in Phase 2

Phase 1:

1. For all queries run BFS up the tree in parallel, “merging” BFS flows whenever they meet. Keep a record of paths for path compression

Phase 2:

1. Distribute the recorded paths to the vertices for compression. Run BFS backwards from the roots reached in Phase 1.
2. We need to be able to efficiently find for every node the flows that arrived at this node

How to Include Path Compression?

How to run Path Compression in parallel:

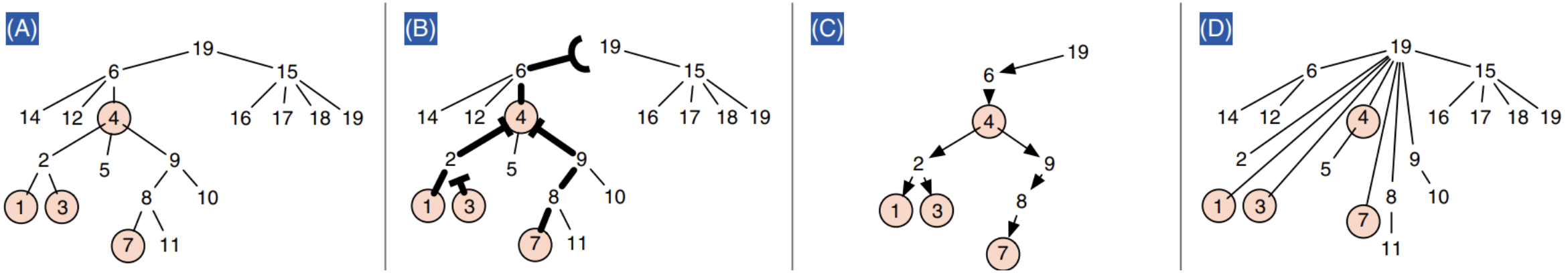
Bulk-Find($\{u_i, v_i\}_{i=1}^{k}$): return for each pair of vertices if they are connected at that point in the stream

Phase 1:

1. For all queries run BFS up the tree in parallel, “merging” BFS flows whenever they meet. Keep a record of paths for path compression

Phase 2:

1. Distribute the recorded paths to the vertices for compression. Run BFS backwards from the roots reached in Phase 1.
2. We need to be able to efficiently find for every node the flows that arrived at this node



Distributing Compressed Paths

We need to be able to efficiently find for every node the flows that arrived there:

- We need to maintain a Response Distributor $R = \langle \mathrm{from}_i, \mathrm{to}_i \rangle_{i=1}^{\lambda}$ that can be constructed in $O(\lambda)$ work and $O\big(\mathrm{polylog}(n)\big)$ depth
- It supports *allFrom*(f): returns a sequence of all to_i where $\mathrm{from}_i = f$.
- Each query must be answered in $O(\log \lambda)$ depth and all the work for requests throughout Phase 2 takes $O(\lambda)$ work
- Via hashing each $\langle \mathrm{from}_i, \mathrm{to}_i \rangle$ pair, and using parallel integer sorting on the hash values one can quickly construct a response distributor and respond to queries – more details in paper.
- Can show that we get all benefits from path compression in polylog depth for *Bulk-Find(*$\{u_i, v_i\}_{i=1}^{k}$*)* queries

Work Efficiency Proof

How to run Path Compression in parallel:

Bulk-Find($\{u_i, v_i\}_{i=1}^{k}$): return for each pair of vertices if they are connected at that point in the stream

Need to show that this algorithm is Work Efficient:

- Idea: what *Bulk-Find* does is equivalent to some sequential execution of find, so it does the same amount of work
- Paper argues that no matter the sequence of queries S, there is a permutation S' that gives the same Union-Find forest and incurs same work, and argue this bounds work at $O\big((m+q)\,\alpha(m+q,n)\big)$ work
- Combining previous results, they obtain a polylog(n) depth for *Bulk-Union* and *Bulk-Find* operations

Overall, they obtain a work efficient parallel algorithm for Incremental Graph Connectivity, and polylog depth for all batched updates and queries

Implementation: Graphs Considered

Implementation:

- A simpler path compression technique is considered:
  1. Run *find* operations independently in parallel.
  2. After root is found, go through the tree once more and update pointers along the path

- While this doesn't give all the benefits of Path Compression, as it still can waste work, it helps manage longest paths in the trees

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

Implementation: Graph Details

Implementation:

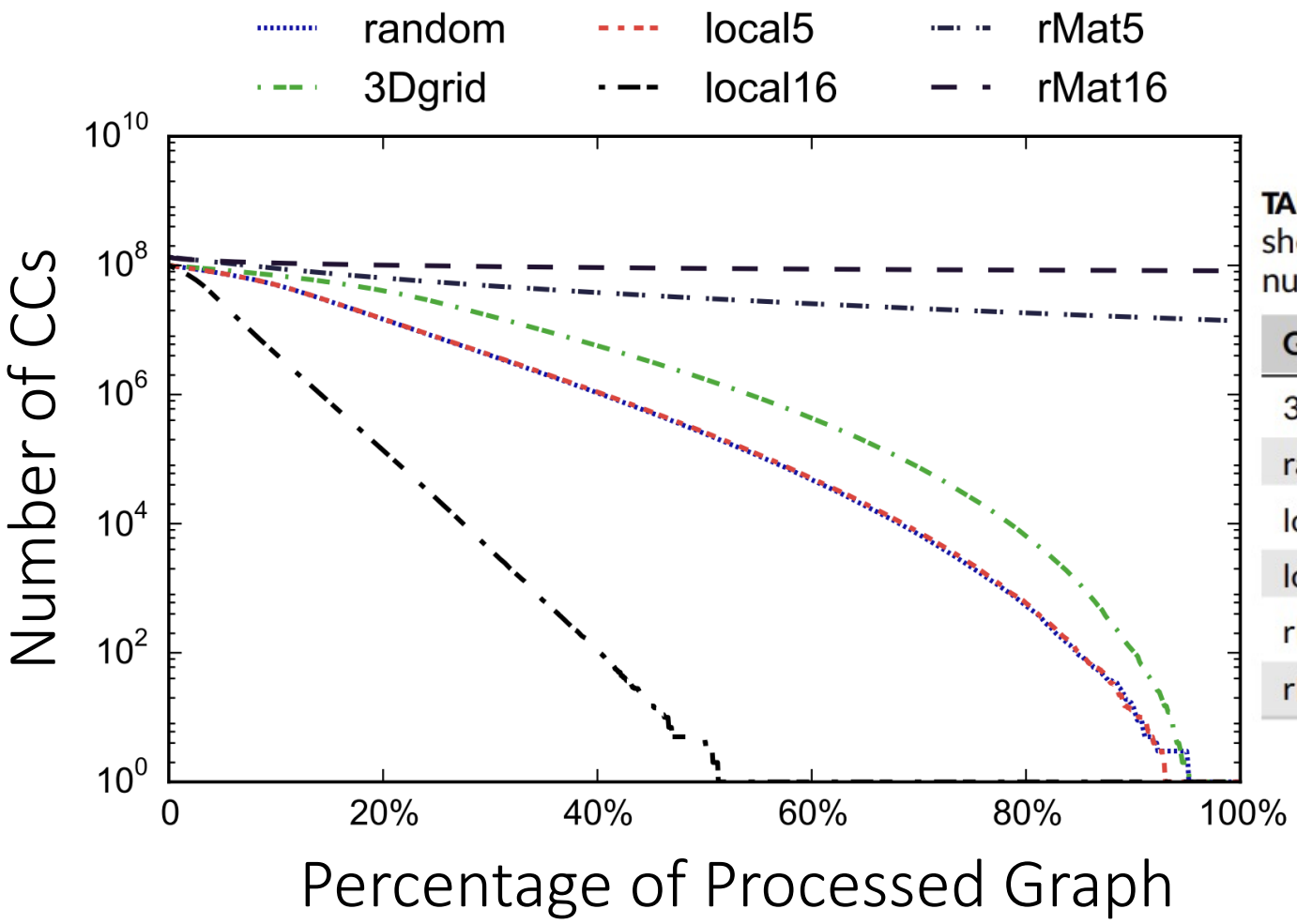


TABLE 1 Characteristics of the graph streams used in our experiments, showing for every dataset, the total number of nodes (n), the total number of edges (m), and a brief description

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: Work Comparison

Results:

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

In general, bulk-parallel implementation is slower serially

Results: Parallel Performance

Results

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

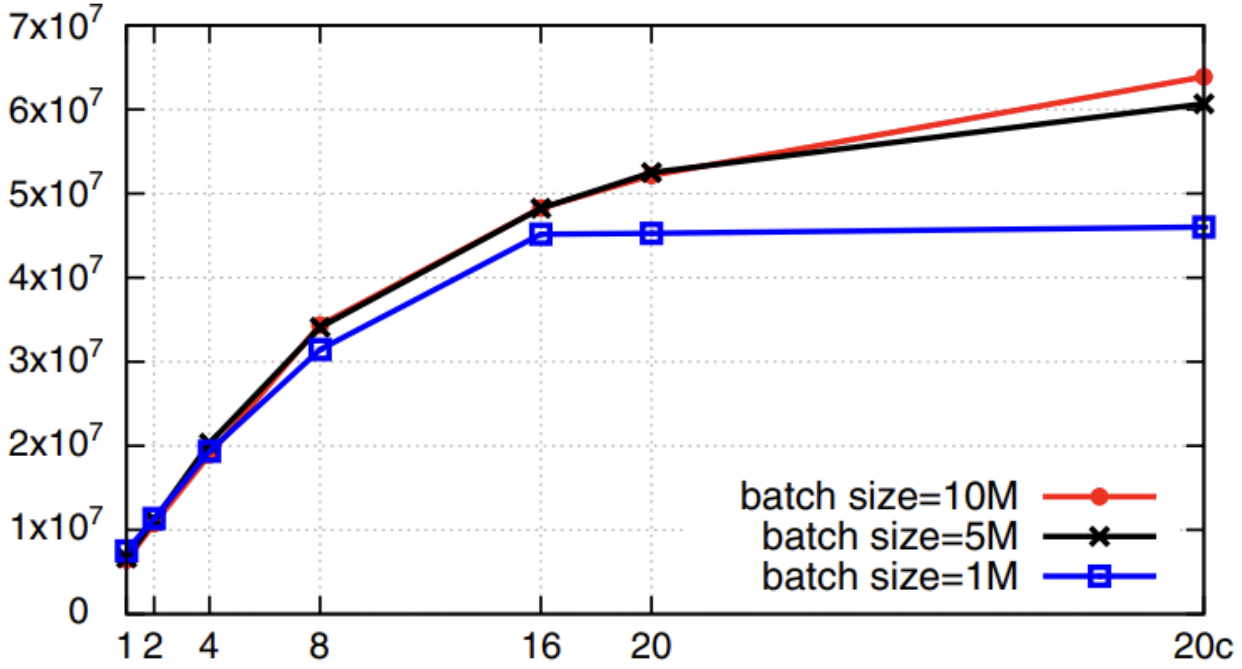
Graph	Using $b = 500K$			Using $b = 1M$			Using $b = 5M$			Using $b = 10M$		
	T_1	T_{20c}	T_{20c}/T_1	T_1	T_{20c}	T_{20c}/T_1	T_1	T_{20c}	T_{20c}/T_1	T_1	T_{20c}	T_{20c}/T_1
random	7.64	36.87	4.8x	7.51	46.02	6.1x	6.65	60.66	9.1x	6.42	63.90	10.0x
3Dgrid	4.91	27.97	5.7x	4.83	34.97	7.2x	4.18	44.27	10.6x	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

In general, larger batch size, better speedup factors

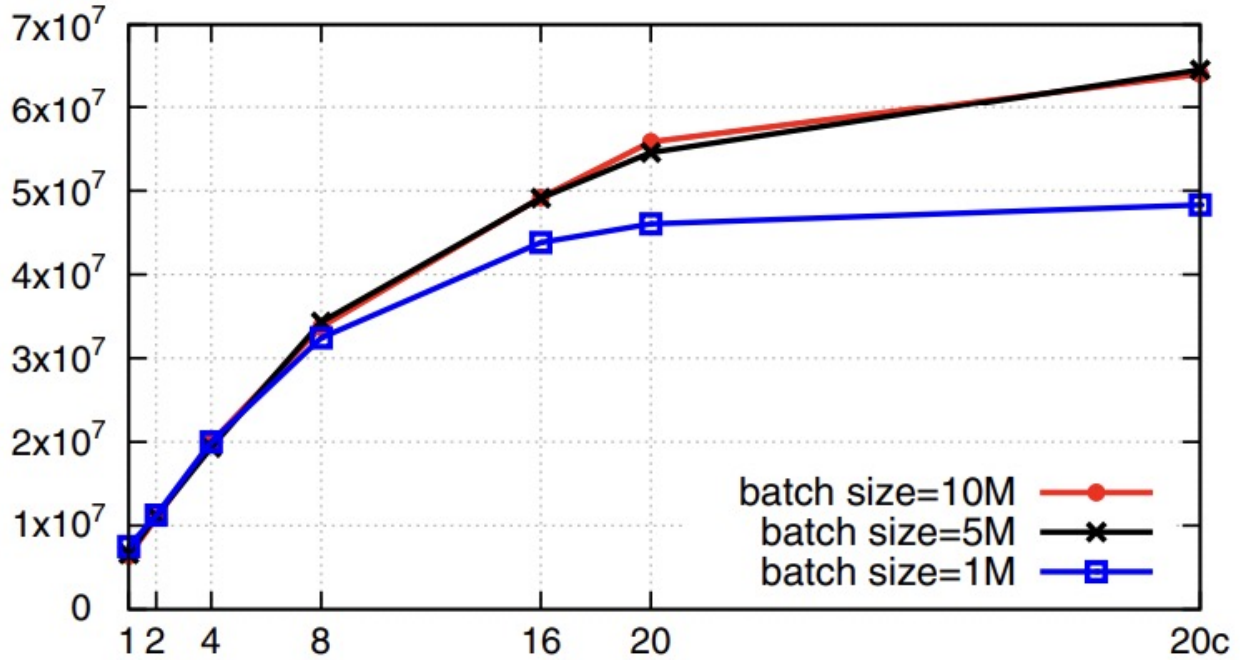
Results: Average Throughput

Results

Plot of Average Throughput(edges/s) as a function of the number of threads



Random Graph



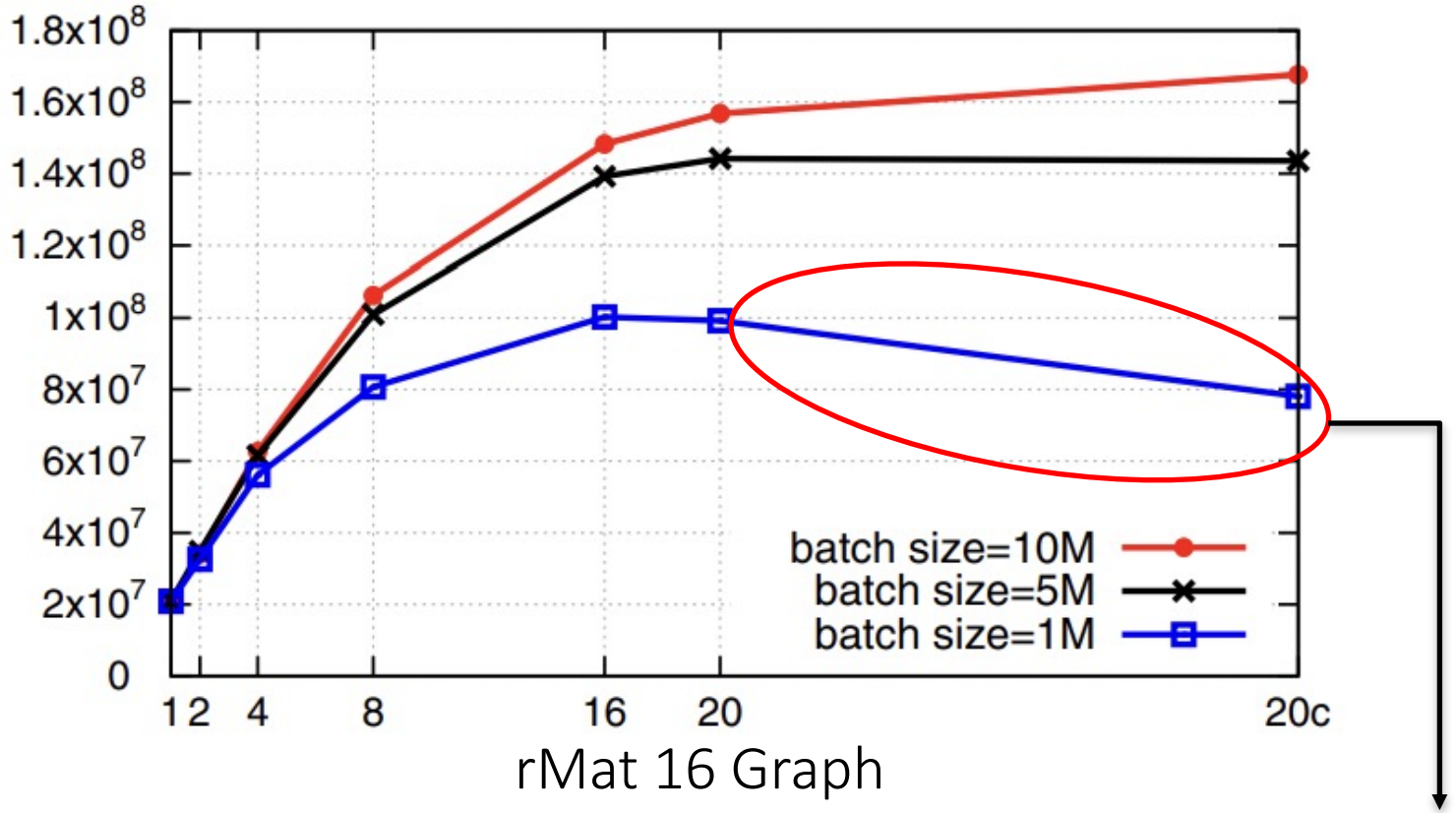
Local 16 Graph

In general, more cores, more average throughput

Results: Average Throughput

Results

Plot of Average Throughput(edges/s) as a function of the number of threads



rMat 16 Graph

An exception to this is rMat16 as it is sparsely connected, so not much work to be done per minibatch

Conclusion

Conclusion:

A parallel data structure for Union-Find was given and achieves:

- Work Efficiency: $O((m+q)\alpha(m+q,n))$ work
- Polylogarithmic Depth for batched operations

A simplified version of the algorithm was implemented – with reduced path compression:

- Performs between 1.01-2.50× slower than previous sequential algorithms on benchmark graphs
- Obtains between 8-10× parallel speedups on large batch sizes

Open Questions to be Analyzed:

- How can we support edge deletions? – There are other Union Find variants which support deletions, but they can't be done in parallel
- For very large batches our work is superlinear (albeit almost linear due to Ackermann), however, can be done in linear time. Can we have an algorithm which is linear work for large batches, and falls back to Union Find for small batches?