Work-efficient parallel union-find

Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, Kun-Lung Wu

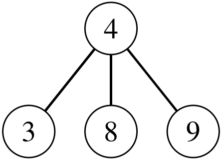
Presenter: Jessica Shi

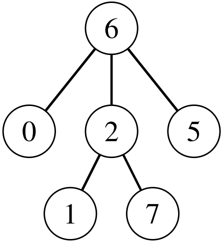
6.886 Algorithm Engineering
Spring 2019, MIT

Introduction

Union-find

- **Union-find**: Maintain a collection of disjoint sets supporting:
  - union(u, v): Combine sets containing u and v
  - find(v): Return set containing v
    - If u and v are in the same set, find(u) = find(v)

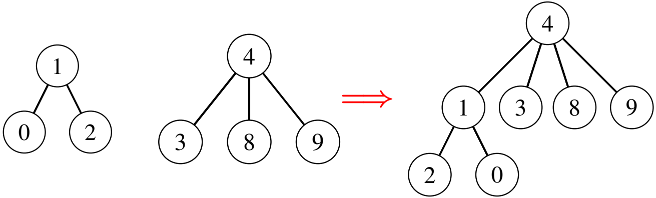




Goal: Incremental graph connectivity

- **Incremental graph connectivity**: Graph connectivity as edges are added over time

$$\text{find}(0) = 1 \implies \text{union}(0, 3) \implies \text{find}(0) = 4$$



Goal: Parallelization

- **Shared-memory parallelization**:
  - Communication overhead in distributed setting
  - Multicore machines can store large graphs
- **Work-efficiency**:
  - Guarantee worst-case performance

Previous work

- **McColl et al.** [1]: Parallel alg for fully dynamic connectivity
    - No theoretical bound
- **Manne and Patwary** [2]: Parallel union-find alg for distributed setting
- **Patwary et al.** [3]: Shared-memory parallel union-find alg
    - No theoretical bound
- **Shun et al.** [4] and **Gazit** [5]: Work-efficient parallel alg for connectivity
    - Only for static graphs

[1] McColl, Green, and Bader. 2013.
[2] Manne and Patwary. 2010.
[3] Patwary, Refsnes, and Manne. 2012.
[4] Shun, Dhulipala, and Blelloch. 2014.
[5] Gazit. 1991.

Main results: Union-find

- **Simple parallel algorithm**
  - *b* union/find: $O\big(b \log n\big)$ work, $O\big(\text{polylog}(n)\big)$ depth
  - $O\big(n\big)$ memory
- **Work-efficient parallel algorithm**
  - *m* union, *q* find: $O\big((m+q)\alpha(m+q,n)\big)$ total work, $O\big(\text{polylog}(m,n)\big)$ depth
  - $\alpha =$ inverse Ackermann's function
  - $O\big(n\big)$ memory
- **Implementation of simple parallel algorithm**

Preliminaries

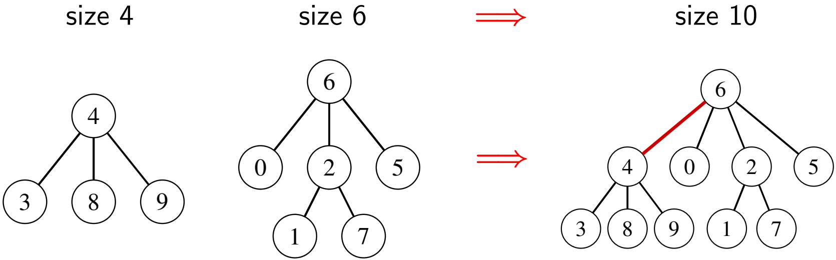
Preliminaries

- **Discretized stream input**: Sequence of minibatches
  - Each minibatch consists of either union queries or find queries
- **Parallel subroutines**:
  - Filter, prefix sum, map, pack: $O(n)$ work, $O(\log^2 n)$ depth
  - Duplicate removal: $O(n)$ work, $O(\log(2n))$ depth
  - Integer sort: $a_i \in [0, O(1) \cdot n]$: $O(n)$ work, $O(\text{polylog}(n))$ depth
  - Connectivity (static) [6]: $O(|V| + |E|)$ work, $O(\text{polylog}(|V|, |E|))$ depth

[6] Shun, Dhulipala, and Blelloch. 2014.

Union by size

- Always link tree with fewer vertices to tree with more vertices
  - Tree height $O(\log n)$
  - Each union/find $O(\log n)$



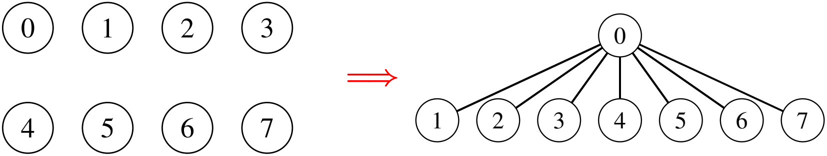
Simple parallel algorithm

Simple parallel algorithm: Find

- **Parallel find**: Perform finds in parallel
  - Read-only = no conflicts
- Work: $O\big(b \log n\big)$
- Depth: $O\big(\mathrm{polylog} n\big)$

Simple parallel algorithm: Union

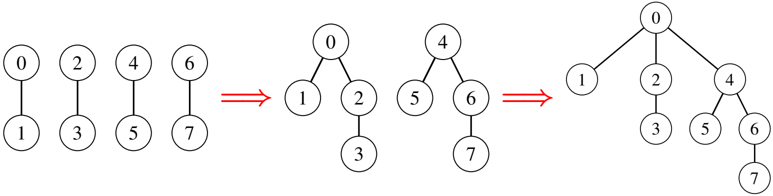
- Safe to run multiple unions in parallel if they belong to different trees
- **Worst case**: Star minibatch: $(0, 1), (0, 2), (0, 3), \ldots, (0, 7)$



Simple parallel algorithm: Union

- **Main idea**: Doesn't matter how we connect $\{0, \ldots, 7\}$
- 3 parallel rounds:

$$(0,1),(2,3),(4,5),(6,7) \implies (0,2),(4,6) \implies (0,4)$$



Simple parallel algorithm: Union

- **Parallel join**: Recursively join tree roots, so that they are all connected at the end
  - $u \leftarrow$ parallel join on first half of roots
  - $v \leftarrow$ parallel join on second half of roots
  - Return union(u, v)
- **Parallel union**:
  - Relabel each (u, v) with the roots of u and v
  - Remove self-loops
  - Compute the connected components among our edge pairs
  - For each connected component (in parallel):
    - Parallel join the roots

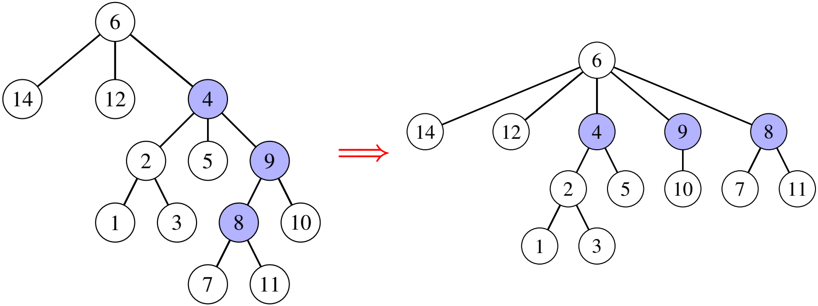
Simple parallel algorithm: Union

- **Parallel join**:
  - Work: $W(k) = 2W(k/2) + O(1) \Rightarrow O(k)$
  - Depth: $D(k) = D(k/2) + O(1) \Rightarrow D(k) = O(\log k)$
- **Parallel union**: b unions:
  - Work: $O(b \log n)$
  - Depth: $O(\log \max(b, n))$

Preliminaries 2.0

Path compression

- find(8)



- Path compression & union by size: Amortized $O(\alpha(n))$ union/find

Work-efficient algorithm

Work-efficient algorithm: Path compression

- **Parallel union**: Same as in the simple parallel algorithm
- **Parallel find**:
  - Find roots for all queries
    - BFS: When flows meet up, only one moves on (use remove duplicates)
  - Distribute roots back along BFS path for path compression
    - Response distributor

Response distributor

- Save all (from, to) pairs on BFS ($\mathbb{F} =$ set of all from values)
- Must construct function that finds all pairs from f
- **Response distributor**:
    - Hash all from values to range $[3 \cdot |\mathbb{F}|]$
    - Integer sort ordered pairs by hashed from value
    - Create array A of length $3 \cdot |\mathbb{F}| + 1$ s.t. i^{th} entry marks beginning of pairs where hashed from value is i
    - Work: $O(|\mathbb{F}|)$, Depth: $O(\text{polylog}(|\mathbb{F}|))$
- **Distributor function**:
    - Hash f and use A to find all pairs from f
    - Work: $O(|\mathbb{F}|)$, Depth: $O(\log |\mathbb{F}|)$

Work-efficient algorithm: Path compression

- **Parallel find**:
  - Work: Given by number of nodes encountered in BFS
  - Depth: $O(\text{polylog}(n))$
- Note: There exists an ordering of find queries s.t. serial find produces the same forest as parallel find, and traversal cost is equal
- $\therefore$ **work-efficient**!

Implementation

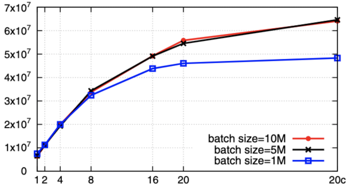
Implementation

- **Simple parallel algorithm**:
  - Simple path compression: After union minibatch, traverse tree one more time to distribute roots
  - Note: Does not give all benefits of path compression, esp within minibatch
  - Connected components: Use alg by Blelloch *et al.* [7] (worse theoretical bounds, good real-world perf)

[7] Blelloch et al. 2012.

Evaluation

- Amazon EC2 instance, 20 cores (40 hyperthreaded)
- Parallel overhead: 1.01x – 2.5x compared to seq w/o path compression
- Speedup: 4.6x with b =500K, 9.4x with b =20M



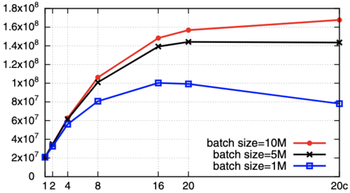


Figure: Average throughput (edges per second) of batch union over number of threads, of local16 (left) and rMat16 (right)

Conclusion

Conclusion

- Simple parallel algorithm
- Work-efficient parallel algorithm
- Implementation of simple parallel algorithm
- **Future work**:
  - Implementation of work-efficient parallel algorithm
  - Switch algorithms depending on batch size:
    - Linear work in # of edges given large union batch (e.g., DFS if all edges given in one batch – our alg is superlinear)
    - Fall back to union-find algorithm for smaller minibatch

Thank you!