Work-Efficient Parallel Union-Find

PRESENTED BY BRIAN WHEATMAN

What is Union-Find

Data structure with 2 operations

- union(u,v)
- find(v)

Used for Incremental graph connectivity on undirected graphs

Sequential approach

Maintain a union find forest with each tree representing a partition

Find(u) returns the root

Union(u,v) joins the roots of the two containing trees

Path Compression gives better performance

- Each time find is called connect that child to the root found

best approach has work $O((m+q)\alpha(m+q, n))$

Sequential Approach

```
function MakeSet(x)
  if x is not already present:
    add x to the disjoint-set tree
    x.parent := x
    x.rank   := 0
```

```
function Find(x)
  if x.parent != x
    x.parent := Find(x.parent)
  return x.parent
```

```
function Union(x, y)
  xRoot := Find(x)
  yRoot := Find(y)

  // x and y are already in the same set
  if xRoot == yRoot
      return

  // x and y are not in same set, so we merge them
  if xRoot.rank < yRoot.rank
    xRoot.parent := yRoot
  else if xRoot.rank > yRoot.rank
    yRoot.parent := xRoot
  else
    xRoot.parent := yRoot
    yRoot.rank   := yRoot.rank + 1
```

Contributions of this work

Simple practical algorithm

Provably work-efficient algorithm

Simple Bulk-Parallel Data Structure

O(n) memory

for b unions in a minibatch

- O(b log n) work
- O(log max(b,n)) depth

For q queries in a minibatch

- O(q log n) work
- O(log n) depth

Simple approach -- queries

Queries are read-only

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** $\langle a_1, a_2, \ldots, a_q \rangle$

Simple approach -- unions

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

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

▷ *Remove self-loops*

$A'' \leftarrow \langle (u, v) \; : \; (u, v) \in A'$ where $u \neq v \rangle$

$\mathcal{C} \leftarrow \mathrm{CC}(A'')$

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

  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

if $|C| == 1$ **then**

  return $C[1]$

else

  $\ell \leftarrow \lfloor |C|/2 \rfloor$

  $u \leftarrow$ Parallel-Join$(U, C[1, 2, \ldots, \ell])$ **in parallel with**

   $v \leftarrow$ Parallel-Join$(U, C[\ell + 1, \ell + 2, \ldots, |C|])$

  return $U.\mathrm{union}(u, v)$

Work-Efficient Parallel Algorithm

Path Compression

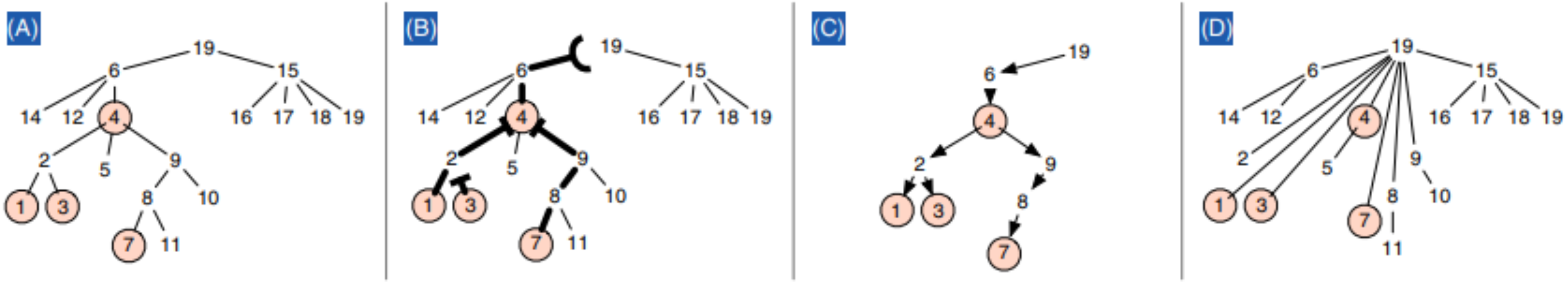

FIGURE 1 A, An example union-find tree with sample queries circled; B, bolded edges are paths, together with their stopping points that result from the traversal in Phase I; C, the traversal graph R_U recorded as a result of Phase I; and D, the union-find tree after Phase II, which updates all traversed nodes to point to their roots

Path Compression

Algorithm 4: $\texttt{Bulk-Find}(U, S)$—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*

$R_0 \leftarrow \langle (S[k], \textbf{null}) \;:\; k = 0, 1, 2, \ldots, |S| - 1 \rangle$

$F_0 \leftarrow \texttt{mkFrontier}(R_0, \emptyset)$, $roots \leftarrow \emptyset$, $visited \leftarrow \emptyset$, $i \leftarrow 0$

while $R_i \neq \emptyset$ **do**

- $visited \leftarrow visited \cup F_i$
- $R_{i+1} \leftarrow \langle (\texttt{parent}[v], v) \;:\; v \in F_i$ and $\texttt{parent}[v] \neq v \rangle$
- $roots \leftarrow roots \cup \{v \;:\; v \in F_i$ where $\texttt{parent}[v] = v\}$
- $F_{i+1} \leftarrow \texttt{mkFrontier}(R_{i+1}, visited)$, $i \leftarrow i + 1$

▷ *Set up response distribution*

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*

$D_0 \leftarrow \{(r, r) \;:\; r \in roots\}$, $i \leftarrow 0$

while $D_i \neq \emptyset$ **do**

- For each $(v, r) \in D_i$, in parallel, $\texttt{parent}[v] \leftarrow r$
- $D_{i+1} \leftarrow \bigcup_{(v,r) \in D_i} \{(u, r) \;:\; u \in RD.\texttt{allFrom}(v)$ 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.
- $i \leftarrow i + 1$

For $i = 0, 1, 2 \ldots, |S| - 1$, in parallel, make $res[i] \leftarrow \texttt{parent}[S[i]]$

return res

def $\texttt{mkFrontier}(R, visited)$:

// nodes to go to next

1: $req \leftarrow \langle v \;:\; (v, \_) \in R \wedge$ **not** $visited[v] \rangle$

2: **return** $\texttt{removeDup}(req)$

Implementation

Only implemented simple approach

- 2 optimizations
- Path compression after each round
- Faster connected components
  - Not as good theoretical guarantees

Results – overhead

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 – scalability

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

Future Work

Fully dynamic graphs

Smaller minibatchs