A Scalable Parallel Union-Find Algorithm for Distributed Memory Computers

Fredrik Manne and Md. Mostofa Ali Patwary

University of Bergen, Norway

SIAM, CSC, October 29-31, 2009

Applications

- To maintain a number of non-overlapping sets consisting of elements from a finite universe.
- Applications
  - image decompositions.
  - computing connected components.
  - computing minimum spanning trees in graphs.
  - clustering.
  - sparse matrix computations.
- Often referred as the Union-Find algorithm.

Main operations

- Maintains a number of non-overlapping sets.
- Each set is represented by a rooted tree.
- The element in the root node is the representative of the set.
- Two main operations.
  - To which set does a given element x belong $\Rightarrow$ *find*(x).
  - Create a new set from the union of two existing sets containing x and y $\Rightarrow$ *union*(x, y).
- $p(x)$ denotes the parent of node x.

The sequential algorithm

- With these operations the connected components of a graph $G = (V, E)$ can be computed as follows.

The sequential algorithm. . .

```
Union-find Algorithm
{


}
```

The sequential algorithm...

```
Union-find Algorithm
{
  S = emptyset









}
```

The sequential algorithm. . .

```
Union-find Algorithm
{
  S = emptyset
  for (each vertex x of V)
    p(x) = x;

}
```

The sequential algorithm. . .

```
Union-find Algorithm
{
  S = emptyset
  for (each vertex x of V)
    p(x) = x;
  for (each edge (x, y) of E)
  {




  }
}
```

The sequential algorithm. . .

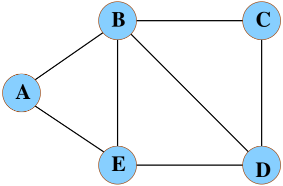
```
Union-find Algorithm
{
  S = emptyset
  for (each vertex x of V)
    p(x) = x;
  for (each edge (x, y) of E)
  {
      if(find(x) != find(y))


  }
}
```

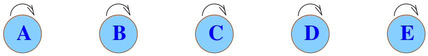
The sequential algorithm. . .

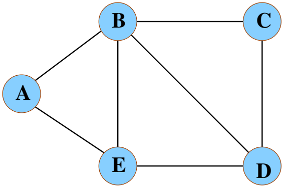
```
Union-find Algorithm
{
  S = emptyset
  for (each vertex x of V)
    p(x) = x;
  for (each edge (x, y) of E)
  {
      if(find(x) != find(y))
        union(x, y);
        S = S + {(x, y)};
  }
}
```

Example: Classical Union-Find algorithm

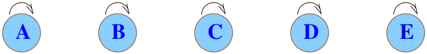


Example: Classical Union-Find algorithm

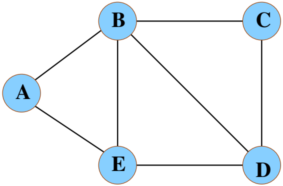




Example: Classical Union-Find algorithm

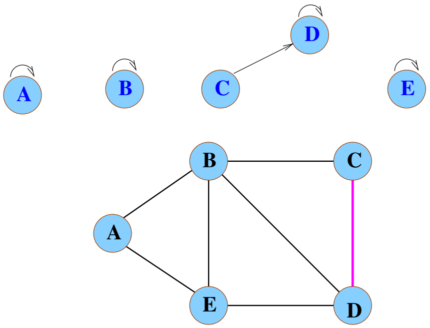


- (C, D)



S = { emptyset}

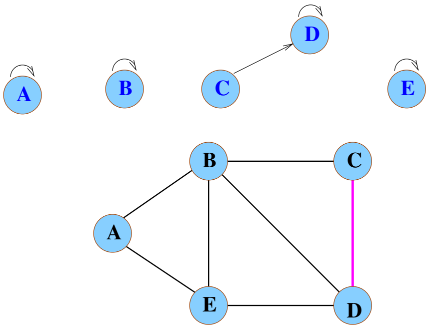
Example: Classical Union-Find algorithm



- (C, D)

S = {(C, D)}

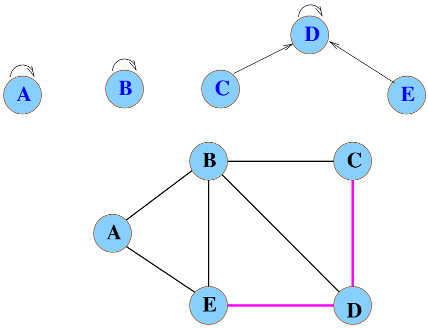
Example: Classical Union-Find algorithm



- (C, D)
- (D, E)

S = {(C, D)}

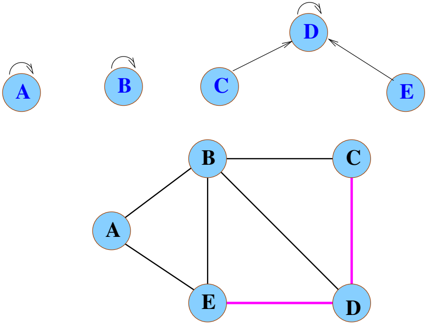
Example: Classical Union-Find algorithm



- (C, D)
- (D, E)

S = {(C, D), (D, E)}

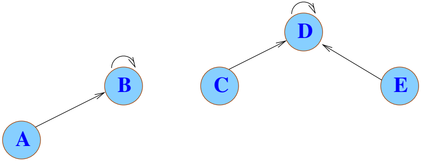
Example: Classical Union-Find algorithm



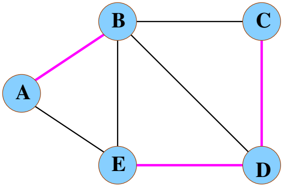
- (C, D)
- (D, E)
- (A, B)

S = {(C, D), (D, E)}

Example: Classical Union-Find algorithm

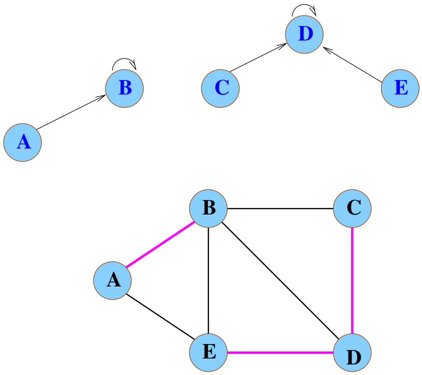


- (C, D)
- (D, E)
- (A, B)



S = {(C, D), (D, E), (A, B)}

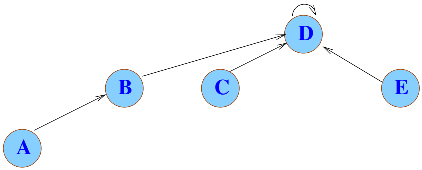
Example: Classical Union-Find algorithm

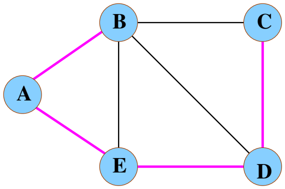


- (C, D)
- (D, E)
- (A, B)
- (A, E)

S = {(C, D), (D, E), (A, B)}

Example: Classical Union-Find algorithm

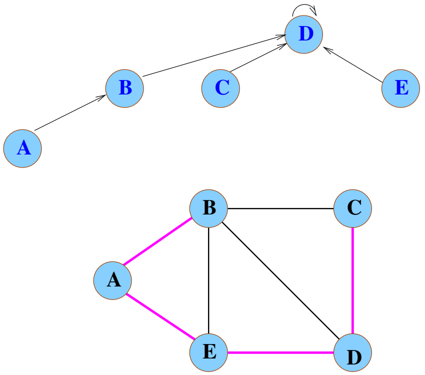




- (C, D)
- (D, E)
- (A, B)
- (A, E)

S = {(C, D), (D, E), (A, B), (A, E)}

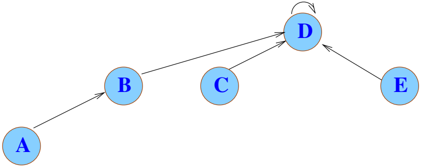
Example: Classical Union-Find algorithm

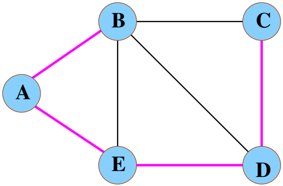


- (C, D)
- (D, E)
- (A, B)
- (A, E)
- (B, C)

S = {(C, D), (D, E), (A, B), (A, E)}

Example: Classical Union-Find algorithm

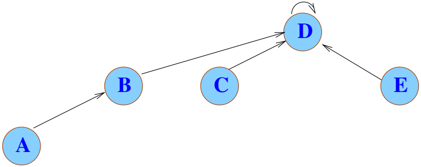


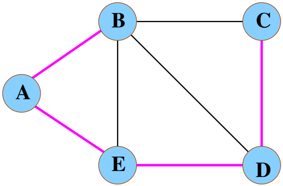


- (C, D)
- (D, E)
- (A, B)
- (A, E)
- (B, C)
- (B, D)
- (B, E)

S = {(C, D), (D, E), (A, B), (A, E)}

Example: Classical Union-Find algorithm





- (C, D)
- (D, E)
- (A, B)
- (A, E)
- (B, C)
- (B, D)
- (B, E)

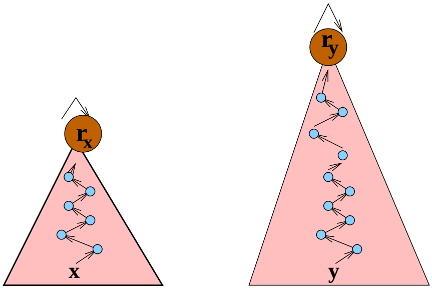
S = {(C, D), (D, E), (A, B), (A, E)}

Speeding up the Union-Find algorithm

- Union-by-rank
  - Rank is the upper bound on the height of the node in the tree. Lowest rank root set to point to the highest rank root.
- Path compression
  - Following any find operation, all traversed vertices are set to point to the root.
  - Compress the path and make a subsequent find operation using any of these vertices faster.
- Total running time of $O(m\alpha(m, n))$ to find the connected components.

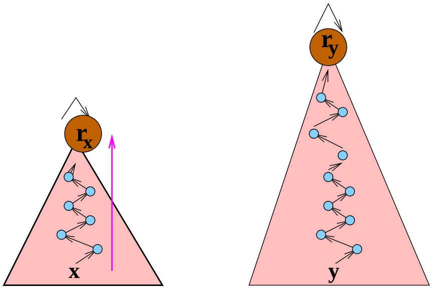
Further enhancements : Tarjan et al. [4]

- Terminate the *find*(y) operation early when x and y are in different sets.



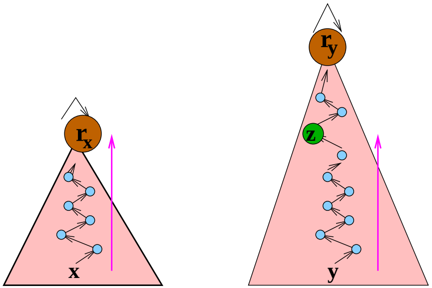
Further enhancements : Tarjan et al. [4]

- Terminate the *find*(y) operation early when x and y are in different sets.
- *find* the root of $x \Rightarrow r_x$



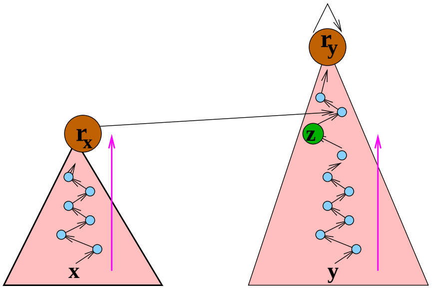
Further enhancements : Tarjan et al. [4]

- Terminate the *find*(y) operation early when x and y are in different sets.
- *find* the root of $x \Rightarrow r_x$
- *find* the root of $y \Rightarrow$ stop at z where $rank(z) = rank(r_x)$.



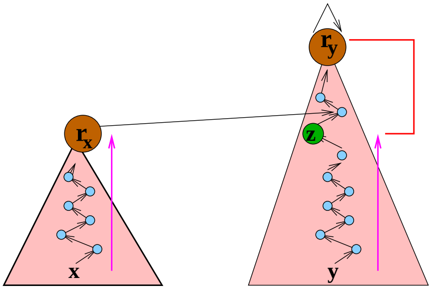
Further enhancements : Tarjan et al. [4]

- Terminate the *find*(y) operation early when x and y are in different sets.
- *find* the root of $x \Rightarrow r_x$
- *find* the root of $y \Rightarrow$ stop at z where $rank(z) = rank(r_x)$.
- Terminate by pointing $p(r_x) = p(z)$



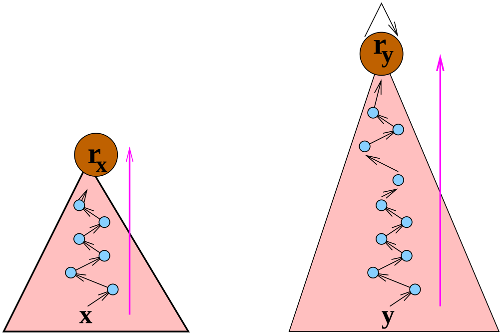
Further enhancements : Tarjan et al. [4]

- Terminate the *find*(y) operation early when x and y are in different sets.
- *find* the root of $x \Rightarrow r_x$
- *find* the root of $y \Rightarrow$ stop at z where $rank(z) = rank(r_x)$.
- Terminate by pointing $p(r_x) = p(z)$
- This will not violate the rank property



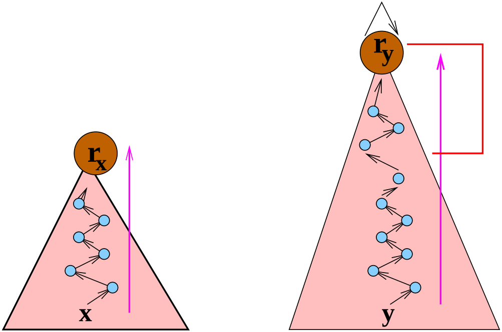
Further enhancements. . .

- What if we start with *find*(y) ??? ⇒



Further enhancements. . .

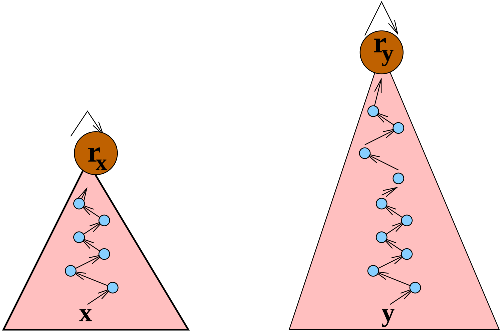
- What if we start with *find*(y) ??? ⇒
- We will not be able to terminate early.



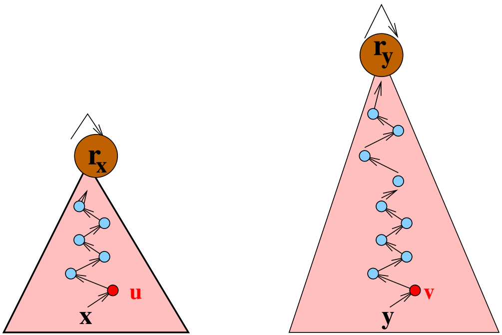
Further enhancements. . .

- Instead of doing two find operations separately, one can instead perform them in an interleaved fashion by always pursuing the node with the lowest rank.
- Hence the find operation terminates as soon as one reaches the root with the smallest rank.
- We call this the zigzag find operation.

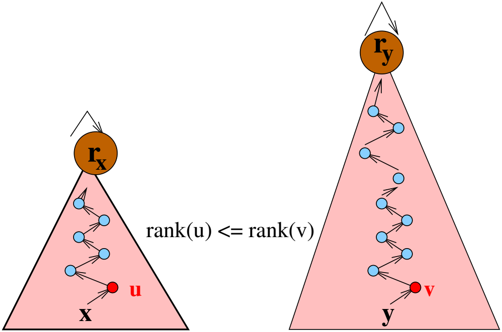
The Zigzag find operation : EXAMPLE



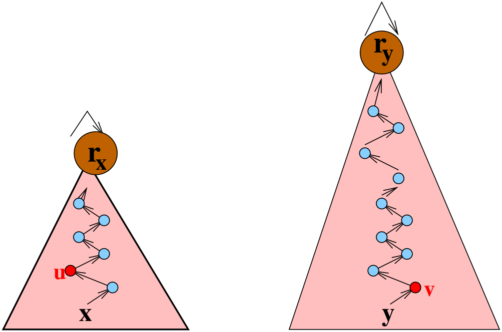
The Zigzag find operation : Different sets



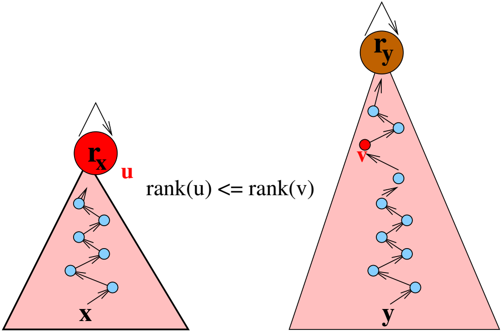
The Zigzag find operation . . .



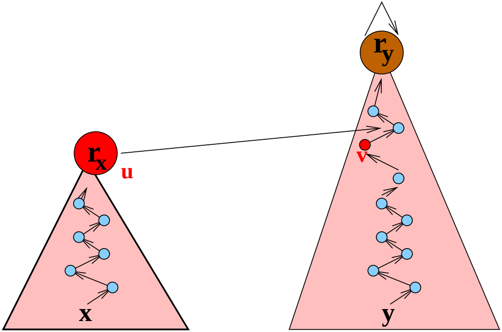
The Zigzag find operation . . .



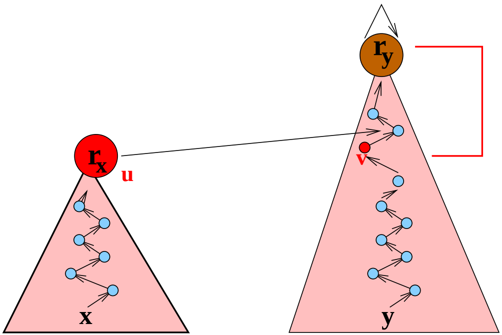
The Zigzag find operation . . .



The Zigzag find operation . . .



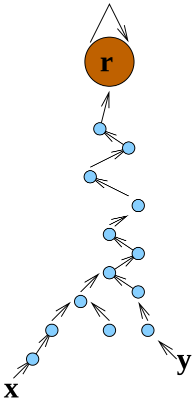
The Zigzag find operation . . .



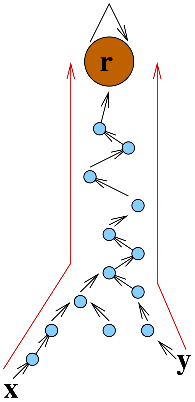
The Zigzag find operation . . .

- The Zigzag find operation can also be used to terminate the search early when the vertices x and y belong to the same set.
- Terminate at lowest common ancestor z.

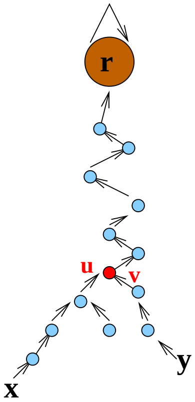
The Zigzag find operation : EXAMPLE



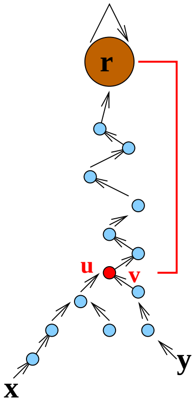
Classical find operation: Same set



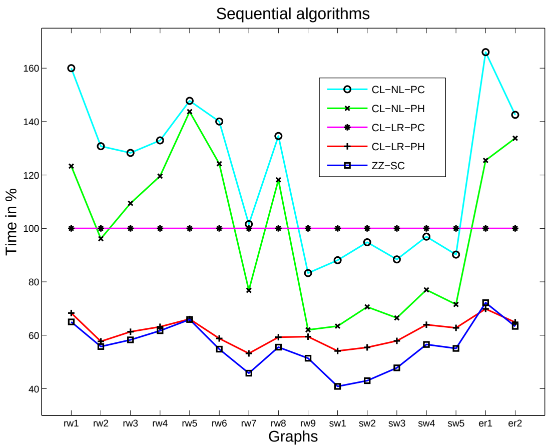
The Zigzag find operation : $u == v$



The Zigzag find operation



Sequential algorithms : real graphs and synthetic graphs



The parallel algorithm : Previous efforts

- Previous efforts to design parallel union-find algorithms has mainly focused on shared memory computers.
- The first effort was done by Cybenko et al. [3] ⇒ Experimental results were not promising and for a fixed sized problem, the running time increased with the number of processors used.
- Anderson and Woll [1] also presented an algorithm for shared memory computers ⇒ Violates the rank property and did not produce any experimental results.

The parallel algorithm : Previous efforts . . .

- Another shared memory implementation is presented by Bader and Cong [2]. This is the first code that gave speedup on arbitrary graphs ⇒ Implemented parallel spanning tree algorithm. But they did not parallelize the union-find algorithm.
- Our work is the first scalable parallel implementation of the union-find algorithm suitable for distributed memory computers.

Data distribution and notations . . .

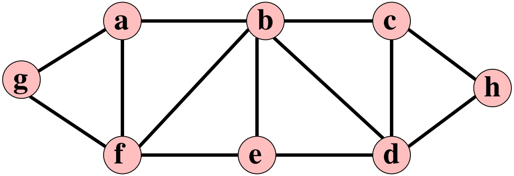


Figure: Graph $G = (V, E)$

Data distribution and notations ...

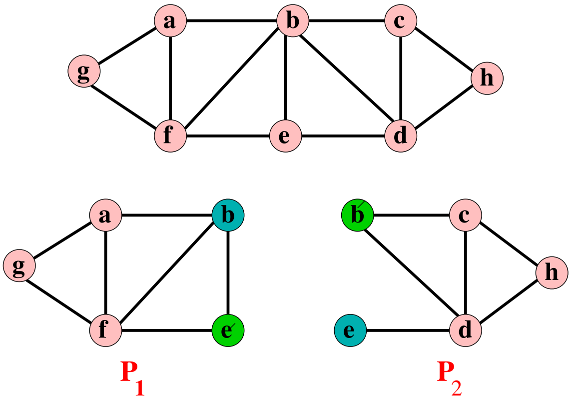


Figure: $G_i = (V_i, E_i)$: Original vertex and ghost vertex

Data distribution and notations . . .

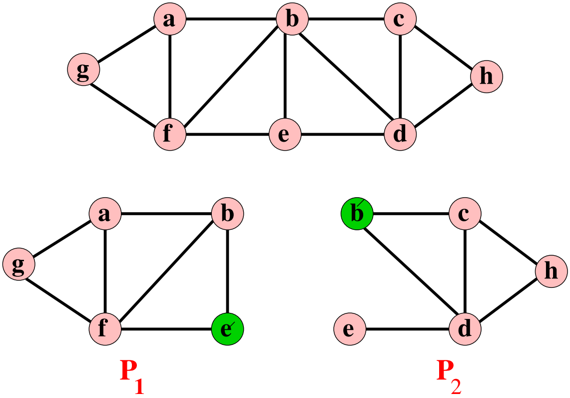


Figure: Original vertex and ghost vertex

Data distribution and notations . . .

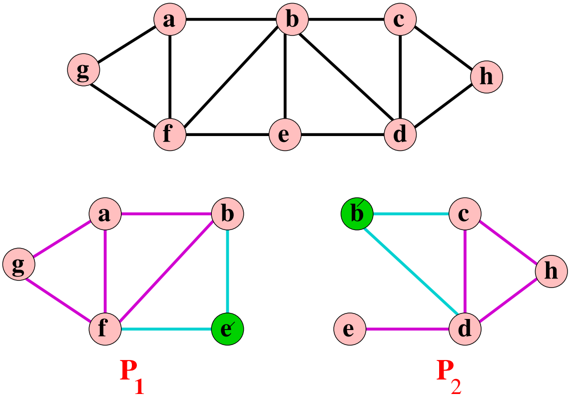


Figure: original edges ⇒ $E_i - E_i'$; crossing edges ⇒ E_i'

Data distribution and notations . . .

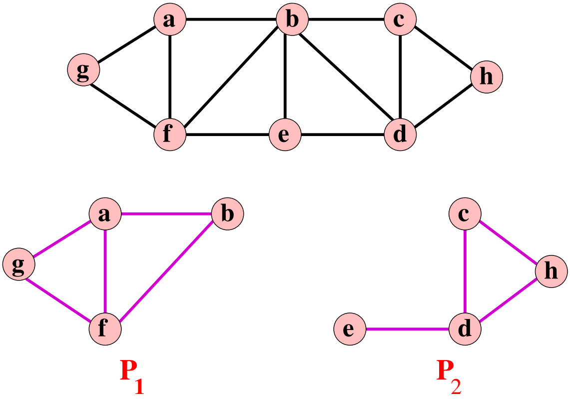


Figure: original edges ⇒ $E_i - E_i'$

Data distribution and notations . . .

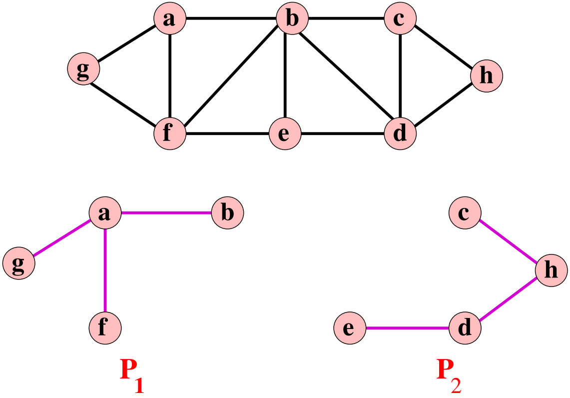


Figure: Spanning forest T_i of original edges $E_i - E_i'$

Data distribution and notations . . .

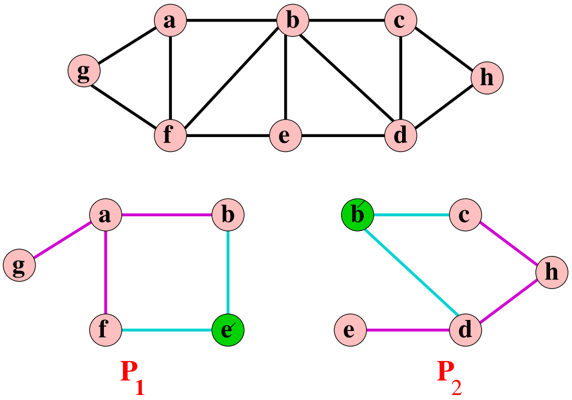


Figure: Spanning forest T_i and crossing edges $\Rightarrow E_i'$

Data distribution and notations . . .

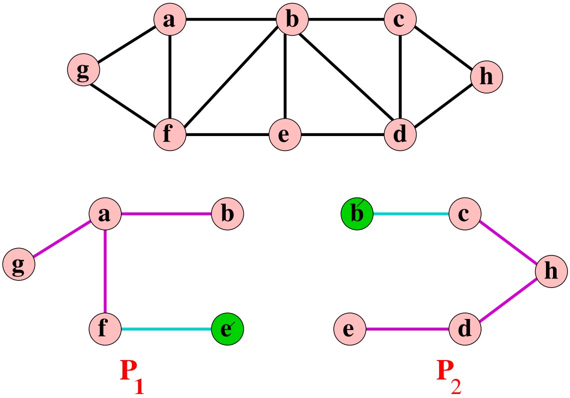


Figure: Spanning forest T_i and spanning forest T_i'

Data distribution and notations . . .

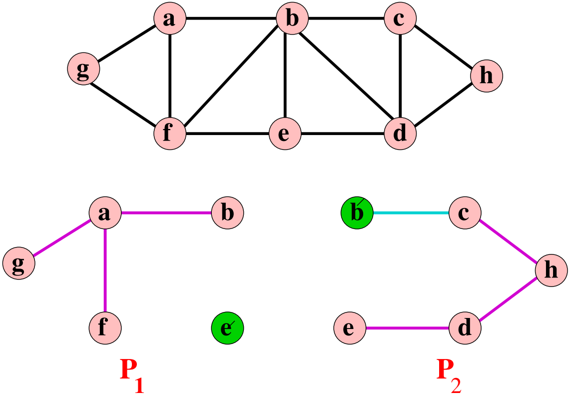


Figure: Final spanning forests : T_i and some edges from T_i'

Data distribution and notations . . .

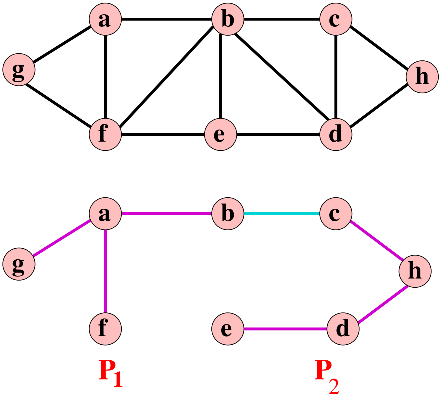


Figure: Final spanning forests

The Parallel algorithm: Each processor P_i

Par-Algorithm

- Stage 1:
- Stage 2:

The Parallel algorithm: Each processor P_i

Par-Algorithm

- Stage 1:
  - Compute spanning forest T_i from original edge set $E_i - E_i'$
- Stage 2:

The Parallel algorithm: Each processor P_i

Par-Algorithm

- Stage 1:
  - Compute spanning forest T_i from original edge set $E_i - E'_i$
  - Compute spanning forest T'_i from crossing edge set E'_i
- Stage 2:

The Parallel algorithm: Each processor P_i

Par-Algorithm

- Stage 1:
  - Compute spanning forest T_i from original edge set $E_i - E_i'$
  - Compute spanning forest T_i' from crossing edge set E_i'
- Stage 2:
  - Partition T_i' into subsets $T_{i,1}', T_{i,2}', \ldots, T_{i,\ell}'$, each of size s

The Parallel algorithm: Each processor P_i

Par-Algorithm

- Stage 1:
  - Compute spanning forest T_i from original edge set $E_i - E_i'$
  - Compute spanning forest T_i' from crossing edge set E_i'
- Stage 2:
  - Partition T_i' into subsets $T_{i,1}', T_{i,2}', \ldots, T_{i,\ell}'$, each of size s
  - For each subset $T_{i,j}'$

The Parallel algorithm: Each processor P_i

Par-Algorithm

- Stage 1:
  - Compute spanning forest T_i from original edge set $E_i - E'_i$
  - Compute spanning forest T'_i from crossing edge set E'_i
- Stage 2:
  - Partition T'_i into subsets $T'_{i,1}, T'_{i,2}, \ldots, T'_{i,\ell}$, each of size s
  - For each subset $T'_{i,j}$
    - For each edge $e = (u, v) \in T'_{i,j}$, execute parallel zigzag find-union and possibly add e to T_i.

The Parallel algorithm: Each processor P_i

Par-Algorithm

- Stage 1:
  - Compute spanning forest T_i from original edge set $E_i - E_i'$
  - Compute spanning forest T_i' from crossing edge set E_i'
- Stage 2:
  - Partition T_i' into subsets $T_{i,1}', T_{i,2}', \ldots, T_{i,\ell}'$, each of size s
  - For each subset $T_{i,j}'$
    - For each edge $e = (u, v) \in T_{i,j}'$, execute parallel zigzag find-union and possibly add e to T_i.
    - Send and Receive messages to other processors.
    - Process incoming messages.

The Parallel algorithm: Each processor P_i

Par-Algorithm

- Stage 1:
  - Compute spanning forest T_i from original edge set $E_i - E'_i$
  - Compute spanning forest T'_i from crossing edge set E'_i
- Stage 2:
  - Partition T'_i into subsets $T'_{i,1}, T'_{i,2}, \ldots, T'_{i,\ell}$, each of size s
  - For each subset $T'_{i,j}$
    - For each edge $e = (u, v) \in T'_{i,j}$, execute parallel zigzag find-union and possibly add e to T_i.
    - Send and Receive messages to other processors.
    - Process incoming messages.
- Global spanning forest is $T_1 \cup T_2 \cup \ldots \cup T_p$

Parallel zigzag find-union - Different sets

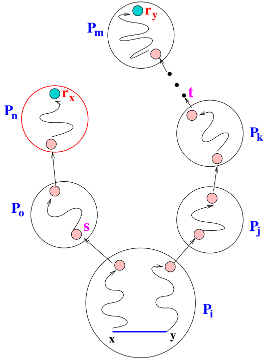


Figure: $rank(r_x) \leq rank(t)$; $p(r_x) \rightarrow p(t)$

Parallel zigzag find-union - Different sets . . .

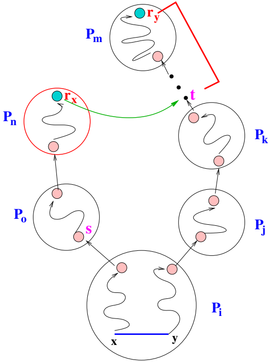


Figure: Do not need to traverse $(P_t \ldots P_m)$; add (x, y) to T_i

Parallel zigzag find-union - Same set

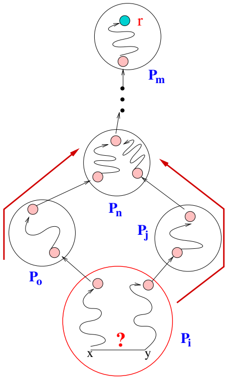


Figure: Traverse $(P_i, P_j, P_n, P_o, P_n)$; P_n informs P_i to discard edge (x, y)

Parallel zigzag find-union - Same set . . .

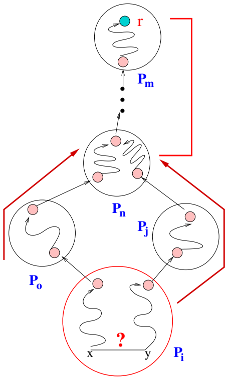
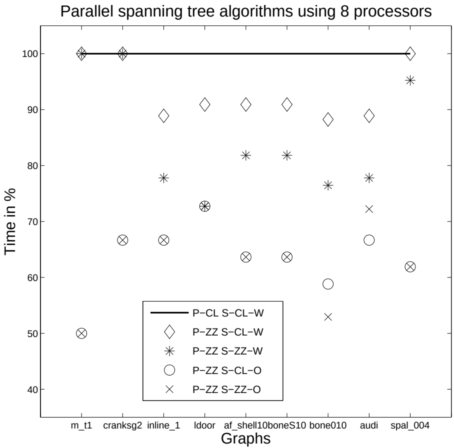
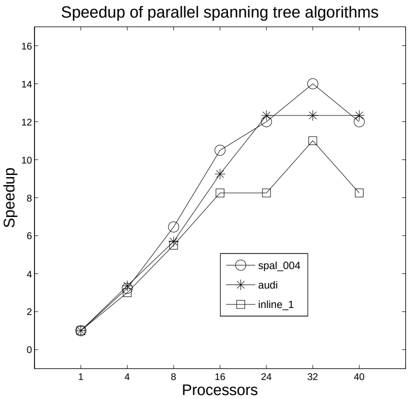


Figure: Do not need to traverse $(P_n \rightarrow \ldots \rightarrow P_m)$

Parallel algorithms using 8 processors : real graphs



Speedup of parallel algorithm : real graphs



Conclusion

- Developed enhanced faster sequential Union-Find algorithm.
- Looking for some analysis how our enhanced sequential Union-Find algorithm performs for different graph classes.
- Developed first parallel Union-Find algorithm for distributed memory computers.
- Investigating more applications where we can apply the algorithms.

Thank you.

Bibliography I

- R. J. Anderson and H. Woll, *Wait-free parallel algorithms for the union-find problem*, in Proceedings 23rd ACM Symposium on Theory of Computing, 1991, pp. 370–380.
- D. Bader and G. Cong, *A fast, parallel spanning tree algorithm for symmetric multiprocessors*, Journal of Parallel and Distributed Processing, 65 (2005), pp. 994–1006.
- A. T. G. Cybenko, G. and J. E. Polito, *Practical parallel union-find algorithms for transitive closure and clustering*, International Journal of Parallel Programming, 17 (1988), pp. 403–423.

Bibliography II

R. E. Tarjan and J. v. Leeuwan, *Worst-case analysis of set union algorithms*, in Journal of the ACM, 1984, pp. 245–281.