Parallel Union-Find using MPI

By Shubham Prasad Pednekar

Instructor: Dr. R. Miller
Problem Definition
The Union-Find Data Structure

Maintain a collection of sets supporting:

- **union(u, v)**
  
  Combine sets containing u and v

- **find(v)**
  
  Return set containing v usually indexed by a unique representative of the set.

Representative element is usually the smallest element of the set.

\[
S_1 := \{1, 2, 3, 4, 5\} \\
S_2 := \{6, 7, 8\} \\
\text{union}(1, 8) \Rightarrow \{1, 2, 3, 4, 5, 6, 7, 8\} \\
\text{find}(1) \Rightarrow \text{representative_of}(S_1) \Rightarrow 1 \\
\text{find}(7) \Rightarrow \text{representative_of}(S_1) \Rightarrow 6
\]
The Union-Find Forest (U)

- Union-Find usually uses the forest of directed trees data structure. It has the following properties:
  - Every tree $T_i$ in the forest represents the disjoint sets $S_i$ in $U$.
  - The root of every tree in $U$ is the representative of that group.
  - $\text{root}(T_i) = \text{representative_of}(S_i)$
  - All elements of set $S_i$ are the key values of the nodes of tree $T_i$

The forest of trees is represented as parent array and key array in memory.
FIND is the operation of getting the representative of a connected component.

<table>
<thead>
<tr>
<th>location</th>
<th>5</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

```python
def find(roots: list[int], location_x: int) -> int:
    if roots[location_x] == None:
        return location_x
    else:
        root = location_x
        while root != roots[root]:
            root = roots[root]
        return root
```
UNION is the operation of joining 2 components with an edge.

```python
def union(roots: list, road: tuple) -> None:
    location_a = road[0]
    location_b = road[1]
    root_a = find(roots, location_a)
    root_b = find(roots, location_b)
    roots[root_a] = min(root_a, root_b)
    roots[root_b] = min(root_a, root_b)
```

<table>
<thead>
<tr>
<th>location</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>7</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>
def union(roots: list, road: tuple) → None:
    location_a = road[0]
    location_b = road[1]
    root_a = find(roots, location_a)
    root_b = find(roots, location_b)
    roots[root_a] = min(root_a, root_b)
    roots[root_b] = min(root_a, root_b)
Creating Connected Component for a Graph

by iterating over edges

```python
for edge in edges:
    if not(edge[0] in values):
        add_value(edge[0])
    if not(edge[1] in values):
        add_value(edge[1])
    union(edge[0], edge[1])
```
Creating Connected Component for a Graph
by iterating over edges

def get_roots(N, M, roads, Q, queries) → list:
    roots = [(None) for x in range(N)]
    for road in roads:
        union(roots, road)
    return roots
Parallel Approach
Parallel Algorithm for Union-Find Generation

1. **Distribute edges equally** over the nodes of a network.

2. **Generate the partial forest** for each processor using its edges.

3. Synchronize the partial forests over connected nodes of the network using **Connect Subgroup Operations**.

4. **Iterate** equal to communication diameter of the network.
Connect Subgroup Operation

1. Two processors exchange the vertex values. \( P_i \text{ gets } V_j \text{ and } P_j \text{ gets } V_i \)
2. Both of them, check for vertex overlaps. \( P_i \text{ and } P_j \text{ calculates } V_i \cap V_j \)
3. Both of them, generate edges of \((\text{value}, \text{root[\text{value}]})\) for vertices in \(V_i \cap V_j\)
4. Both of them, **exchange these new edges and representatives of sets.**
5. Both of them, **add the new edges** to their own partial forest

At the end, both processors represent a single forest. (same root for same valued vertices in the partial forests).
The Choice of Network - Hypercube

Some hypercubes with their dimensions:
Iterations for n=16 hypercube

Iteration 1
Iterations for n=16 hypercube

Iteration 2
Iterations for n=16 hypercube

Iteration 3
Runtime VS Edge Count

- np=2
- np=4
- np=8
- np=16
- np=32
- np=64
- np=128

![Graph showing runtime vs edge count with different np values.](image-url)
262144 Total Edges = 262144

Runtime vs Processors
### Speed Up & Efficiency Measurements for Constant Input Size

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>Runtime</th>
<th>Input (Edge Count)</th>
<th>SPEEDUP (for 2X processor count)</th>
<th>EFFICIENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>199.9765223</td>
<td>262144</td>
<td>2*</td>
<td>1*</td>
</tr>
<tr>
<td>4</td>
<td>92.0984</td>
<td>262144</td>
<td>2.171335467</td>
<td>1.085667733</td>
</tr>
<tr>
<td>8</td>
<td>32.975811</td>
<td>262144</td>
<td>2.792907807</td>
<td>1.396453904</td>
</tr>
<tr>
<td>16</td>
<td>10.622648</td>
<td>262144</td>
<td>3.104292922</td>
<td>1.552146461</td>
</tr>
<tr>
<td>32</td>
<td>3.803315</td>
<td>262144</td>
<td>2.792997162</td>
<td>1.396498581</td>
</tr>
<tr>
<td>64</td>
<td>1.040988</td>
<td>262144</td>
<td>3.653562769</td>
<td>1.826781385</td>
</tr>
<tr>
<td>128</td>
<td>0.297374</td>
<td>262144</td>
<td>3.500601936</td>
<td>1.750300968</td>
</tr>
</tbody>
</table>

*Reference Measurement
Why is Efficiency >1 ?

Possible Reason:

- Underlying Serial Code not efficient.
  - It is evident by comparing different edge sizes for 2 processor setup, that the time roughly increases by a factor of 4 for 2X increase in input size.
  - The primary culprit seems to be the `for loops` used to perform set operations $[O(V^2) \text{ complexity}]$ instead of a `hashmap` implementation $[O(V) \text{ complexity}]$.
  - That made sending messages $[O(V) \text{ complexity}]$ much more efficient than performing computation on the same processor.

Future Scope

- Integrate a HashMap based set operation library to make set operations $O(V)$. 
Questions?
References

Work-efficient parallel union-find
Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, Kun-Lung Wu

Algorithms Sequential & Parallel: A Unified Approach 3rd Edition
by Russ Miller (Author), Laurence Boxer (Author)

MPI Tutorial
by Wes Kendall

SLURM reference guide
by UB CCR
THANK YOU