A Highly Scalable Graph Clustering Library based on Parallel Union-Find

Karthik Senthil

Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

12 April 2018

16th Annual Workshop on Charm++ and its Applications 2018
Problem Statement

Graph clustering or connectivity is the process of detecting connected components in a given graph.

- **Connected component**: Maximal-size subgraph where a path exists between every pair of vertices in the subgraph.

![Connected components in a graph](image)

**Figure 1**: Connected components in a graph

Two schools of algorithms:

- Graph traversal algorithm
- Union-Find based algorithm
1. Related Work
2. Parallel Union-Find in Charm++
3. Path Compression
4. Implementation
5. Performance Evaluation
6. What’s In Store
1 Related Work

2 Parallel Union-Find in Charm++

3 Path Compression

4 Implementation

5 Performance Evaluation

6 What’s In Store
Related Work

Connectivity in a graph is a well-studied problem


Our motivation: A scalable parallel implementation using union-find data structures in a distributed asynchronous environment
Outline

1. Related Work
2. Parallel Union-Find in Charm++
3. Path Compression
4. Implementation
5. Performance Evaluation
6. What’s In Store
Algorithm

- Given a graph $G = (V, E)$, with $n = |V|$ and $m = |E|$.
- An edge $e = (v_1, v_2)$ represents a union operation.

Our algorithm:

1. Message $v_1$ for the operation $find(v_1)$
2. $v_1$ messages parents till $boss_1 = find(v_1)$
3. $boss_1$ messages $v_2$ for operation $find(v_2)$ and carries info of $boss_1$
4. When $boss_2 = find(v_2)$, align parent pointers of bosses.

- Effectively we are constructing a forest of inverted trees; each tree is a unique connected component.
- Root of a tree (boss) = Representative of the component.
Figure 2: Asynchronous union-find algorithm
Solving Race Conditions

An example scenario

- Enforce a strict ordering in the union operation based on vertex ID
- Brings in an additional min-heap like property to the inverted trees
  - ID of a parent node is always lesser than IDs of its children
  - A possible cycle edge can be detected if a node with lower ID is asked to point to node with higher ID

Consider 3 PEs, one chare on each PE
union(1, 2) on chare 0
union(2, 3) on chare 1
union(3, 1) on chare 2
union_request(v_1, v_2) {
    if (v_1.ID > v_2.ID)
        union_request(v_2, v_1)
    else
        find_boss1(v_1, v_2)
}

Listing 1: union_request

find_boss1(v_1, v_2) {
    if (v_1.parent == -1)
        find_boss2(v_2, boss_1)
    else
        find_boss1(v_1.parent, v_2)
}

Listing 2: find_boss1

find_boss2(v_2, boss_1) {
    if (v_2.parent == -1) {
        if (boss_1.ID > v_2.ID)
            union_request(v_2, boss_1)
        else
            v_2.parent = boss_1
    }
    else
        find_boss2(v_2.parent, boss_1)
}

Listing 3: find_boss2
Outline

1. Related Work

2. Parallel Union-Find in Charm++

3. Path Compression

4. Implementation

5. Performance Evaluation

6. What’s In Store
Local Path Compression

- Make the local subtree constructed in every chare completely shallow i.e. rooted star

- During **Find**, if next parent on current path is on a different chare then sequentially update parent pointer for all nodes on path

- Increases amount of sequential work per chare but greatly boosts speed of future Find operations
Global Path Compression

- Pointer jumping operation to grandparent
- Short circuits paths that are spanning across multiple chores

- Increases communication due to more messages, but overhead is reduced by aggregating messages using TRAM
Outline

1 Related Work
2 Parallel Union-Find in Charm++
3 Path Compression
4 Implementation
5 Performance Evaluation
6 What’s In Store
Library Design

- Library designed using bound-array concept
- Connected components detection
  - **Phase 1**: Build the forest of inverted trees using our asynchronous union-find algorithm
  - **Phase 2**: Identify the bosses of each component and label all vertices in that component
  - **Phase 3**: Prune out insignificant components
- Used TRAM to aggregate all messages in Phase 1 and Phase 2
- Tested and verified with protein structures from RCSB PDB
- Large scale testing with synthetic and real-world graphs
Phase 3 - Discussion

- Perform a global reduction to gather membership statistics for each component from all the chares
- Initially implemented using a custom reducer with each chare contributing an `std::map`
- Reduced final map is broadcast and rebuilt on each PE (using a group)

Figure 3: Overheads in map-based reducers for Phase 3
**Phase 1**: Build the forest of inverted trees using our asynchronous union-find algorithm

**Phase 2**:
(a) Parallel prefix scan to get total boss count and relabel all bosses with sequential identifiers
(b) Identify the bosses of each component and label all vertices in that component

**Phase 3**: Prune out insignificant components
- Use fixed size array based reduction for the counts
- Arrays can be sparse, but this implementation is very scalable and has minimal overhead
Outline

1. Related Work
2. Parallel Union-Find in Charm++
3. Path Compression
4. Implementation
5. Performance Evaluation
6. What’s In Store
Experiments

Experiments performed:

1. Phase runtime evaluation
   - Mesh configurations: $1024^2$ (1M), $2048^2$ (4M), $4096^2$ (16M), $8192^2$ (64M)
   - Probabilities: 2D40, 2D60, 2D80
   - Problem size per chare fixed at: 128x128 mesh piece

2. Strong scaling performance
   - Mesh configuration: $8192^2$ (64M), $16384^2$ (256M), 2D60
   - Number of cores: 64, 256, 1024, 4096

3. Real world graphs
   - com-Orkut: 3M vertices, 117M edges
   - com-Amazon: 330K vertices, 925K edges

All experiments were performed on the Blue Waters (Cray XE) supercomputer maintained by NCSA.
Figure 4: Mesh size 1024x1024 on 64 cores
Figure 5: Mesh size 8192x8192 on 4096 cores
Results - Strong Scaling

Figure 6: Strong scaling runs

Mesh 8192x8192

Mesh 16384x16384
### Comparison

<table>
<thead>
<tr>
<th>Mesh Size</th>
<th>Last Workshop</th>
<th>Current Workshop</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4096^2$</td>
<td>113.730437 s</td>
<td>0.815045 s</td>
</tr>
<tr>
<td>$8192^2$</td>
<td>195.767054 s</td>
<td>1.749127 s</td>
</tr>
<tr>
<td>$16384^2$</td>
<td>NA</td>
<td>9.178887 s</td>
</tr>
</tbody>
</table>

**Table 1**: Improvements in performance

Kudos to path compression optimizations and TRAM!
Results - Real World Graphs

Figure 7: Experiments with real world graphs

**com-Orkut**

**com-Amazon**

---

Karthik Senthil (PPL)  
Charm++ Workshop 2018  
12 April 2018  
19 / 22
Current Issues

Figure 8: Bottleneck will be observed at boss$_1$ when edges ($v_1$, $v_3$) and ($v_0$, $v_2$) are processed during later stages of Phase 1

- Potential bottlenecks at the root of the biggest inverted tree for dense graphs with very few number of components
- Cases where component roots are unevenly distributed among the chares leading to load imbalance in Phase 2
Outline

1. Related Work
2. Parallel Union-Find in Charm++
3. Path Compression
4. Implementation
5. Performance Evaluation
6. What’s In Store
Future Work

On the to-do list:

- Optimizing Phase 1 to remove bottleneck and improve weak scalability
- Performance evaluation using large ChaNGa datasets
- Implement a Python interface for library using Charmpy

Code and examples on Gerrit: users/karthik/unionFind

Acknowledgements: This material is based in part upon work supported by the NSF, SI2-SSI: Collaborative Research: ParaTreet: Parallel Software for Spatial Trees in Simulation and Analysis (NSF #1550554).
Thank You