A Parallel Union-Find Library in Charm++

Karthik Senthil

Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

17 April 2017

15th Annual Workshop on Charm++ and its Applications 2017
Definition:
A union-find algorithm is an algorithm that performs two operations on a disjoint-set data structure

- **Find**: Determine which subset a particular element is in
- **Union**: Join two subsets into a single subset

![Figure 1: Connected Components in a graph](image)

Other applications: Kruskal’s minimum spanning tree algorithm
Outline

1. Related Work

2. A Charm++ Approach to Union-Find

3. Challenges

4. Optimizations

5. Current Status

6. What’s In Store
Outline

1. Related Work
2. A Charm++ Approach to Union-Find
3. Challenges
4. Optimizations
5. Current Status
6. What’s In Store
Related Work

Connectivity in a graph is a very well explored problem


Our motivation: A scalable union-find algorithm in a distributed asynchronous environment
Outline

1. Related Work
2. A Charm++ Approach to Union-Find
3. Challenges
4. Optimizations
5. Current Status
6. What’s In Store
Our 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 $\text{find}(v_1)$.
2. $v_1$ messages parents till $\text{boss}_1 = \text{find}(v_1)$.
3. $\text{boss}_1$ messages $v_2$ for operation $\text{find}(v_2)$ and carries info of $\text{boss}_1$.
4. When $\text{boss}_2 = \text{find}(v_2)$, 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 $=$ Representative of the component.
Figure 2: Asynchronous union-find algorithm
Challenges

RACE CONDITIONS

RACE CONDITIONS EVERYWHERE

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

Too much symmetry
Simplicity is the best way of dealing with complexity
Enforce a strict ordering in the union operation, say 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
  - We reprocess the union-request by flipping the order to comply with the ordering
Solution - 3 Functions

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
union_request(v₁, v₂) {
    if (v₁.ID > v₂.ID)
        union_request(v₂, v₁)
    else
        find_boss1(v₁, v₂)
}

Listing 1: union_request

find_boss1(v₁, v₂) {
    if (v₁.parent == -1)
        if (boss₁.ID > v₂.ID)
            union_request(v₂, boss₁)
        else
            find_boss1(v₁.parent, v₂)
    else
        find_boss2(v₂, boss₁)
}

Listing 2: find_boss1
union_request(v₁, v₂) {
    if (v₁.ID > v₂.ID)
        union_request(v₂, v₁)
    else
        find_boss1(v₁, v₂)
}

Listing 1: union_request

find_boss1(v₁, v₂) {
    if (v₁.parent == -1)
        find_boss2(v₂, boss₁)
    else
        find_boss1(v₁.parent, v₂)
}

Listing 2: find_boss1

find_boss2(v₂, boss₁) {
    if (v₂.parent == -1) {
        if (boss₁.ID > v₂.ID)
            union_request(v₂, boss₁)
        else
            v₂.parent = boss₁
    } else
        find_boss2(v₂.parent, boss₁)
}

Listing 3: find_boss2
Outline

1. Related Work
2. A Charm++ Approach to Union-Find
3. Challenges
4. Optimizations
5. Current Status
6. What’s In Store
Motivation to optimize:

- Tree construction is very communication-intensive
- Lots of tiny messages (≈1.5 billion messages for 16 million vertices, 6 million edges)
- We also found the trees to be very deep
  - Sequentially, path compression is used to get optimal performance
- Climbing long tree paths for each request slowed down tree construction
1. **Locality-based tree climbing**
   - Sequentially traverse the tree path until the next vertex lies on a different chare
   - This increases work per chare but drastically reduces number of messages
   - Observed 25x speedup in tree construction
Optimizations

1. **Locality-based tree climbing**
   - Sequentially traverse the tree path until the next vertex lies on a different chare.
   - This increases work per chare but drastically reduces the number of messages.
   - Observed 25x speedup in tree construction.

2. **Local path compression**
   - Make the local tree constructed in every chare completely shallow.
   - Provides a one-hop access to bosses.

More optimization if extended to PE-level or node-level.
Outline

1. Related Work
2. A Charm++ Approach to Union-Find
3. Challenges
4. Optimizations
5. Current Status
6. What’s In Store
Current Status

- 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
- Tested and verified with real-world graphs (protein structures from PDB)
- Large scale testing with probabilistic mesh concept
Probabilistic Mesh

- A class of graphs motivated by cluster dynamics in computational physics\(^1\) (2D Ising model)
- A random graph built on a lattice structure
- Edge between two lattice points (vertices) is determined by calculating a probability value using coordinate positions

Advantages:
- Easy to scale the size of graph
- Easy to verify results and catch race conditions
  - Fixed probability and lattice size produces same graph
  - Play with the number of chares and PEs

---

Experiments performed:

1. Phase runtime evaluation
   - Mesh configurations: $1024^2$ (1M), $2048^2$ (4M), $4096^2$ (16M), $8192^2$ (64M)
   - Probabilities: 2D00, 2D40, 2D60
   - Problem size per chare fixed at: 64x64 mesh piece

2. Scaling performance
   - Mesh configuration: $2048^2$, 2D40
   - Problem size per chare: 2x2 mesh piece
   - Number of physical nodes: 2, 4, 8, 16, 32, 64
Figure 4: Mesh size 1024x1024 on 2 nodes
Results - Phase runtime

Figure 5: Mesh size 2048x2048 on 2 nodes
Figure 6: Mesh size 4096x4096 on 16 nodes
Figure 7: Mesh size 8192x8192 on 32 nodes
Results - Scaling runs

Phase 1

Phase 2

**Figure 8**: Scaling runs on mesh size 2048x2048
Phase 3

Figure 9: Scaling runs on mesh size 2048x2048
Outline

1. Related Work
2. A Charm++ Approach to Union-Find
3. Challenges
4. Optimizations
5. Current Status
6. What’s In Store
Future Work

On the to-do list:

- Optimizing Phase 1 for very large graphs (planning on sub-phases)
- Priority for particular kinds of messages
- Global level path compression which is PE and node-aware
- Use TRAM library in Charm++
- Target ChaNGa for friends-of-friends based galaxy detection

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).