Scalable Asynchronous
Connected Components Detection Library

Senthil Kumar Karthik, Jaemin Choi, University of Illinois Urbana-Champaign

Overview

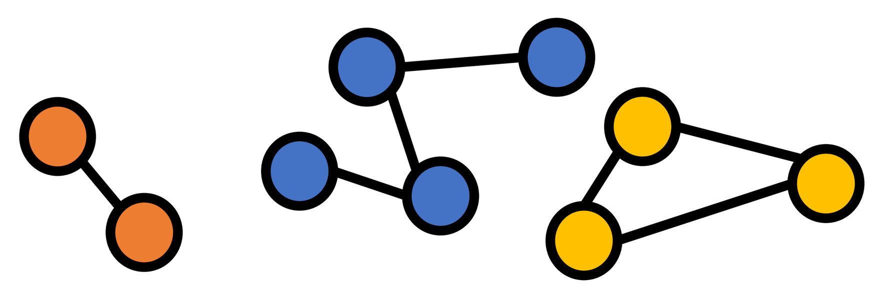
- Finding connected components: popular graph algorithm used in science and engineering
- A Union-Find based parallel library for distributed memory machines
- Scalable implementation using Charm++
- Performance evaluation on NCSA Blue Waters

Charm++

- Migratable object and task-based parallel programming model
- Adaptive runtime system
- Decompose problem domain into communicating objects (**chares**)
- Overdecomposition: many more objects than PEs (CPU cores)
- Asynchronous method invocation via messages

Background

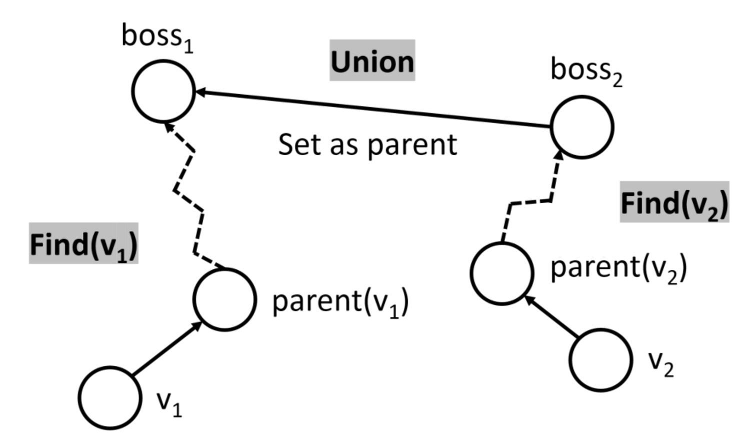
- **Connected component**: a subgraph where vertices are connected by paths, but are not connected to any other vertices outside the subgraph



- **Union-Find**
  - Operations performed on a disjoint-set data structure
  - Used to detect connected components
  - **Union(x,y)**: merge two sets where vertices x and y belong to each set
  - **Find(x)**: return the unique ID of the set containing x
  - If vertices of interest are in different sets (determined by **Find**) but the graph says otherwise, merge the sets (**Union**)

Algorithm

- Adapted version of **Shiloach-Vishkin** (SV) algorithm
  - Perform only tree-hooking step
  - Use asynchronous messaging on a distributed graph



- For each edge (v_1, v_2) in graph,
  1. Message v_1 to perform Find(v_1)
  2. Recursive parent messaging to reach $boss_1$
  3. $boss_1$ messages v_2 for Find(v_2)
  4. Recursive parent messaging to reach $boss_2$
  5. Set $boss_1$ as parent of $boss_2$

```
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)
}
```

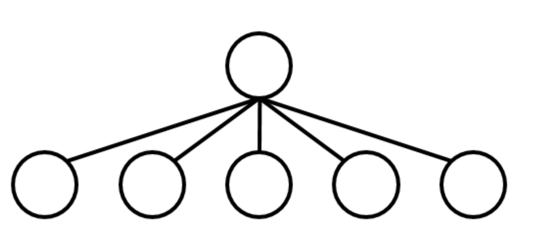
Listing 3: find_boss2

Implementation

- Library involves 3 phases for connected components detection
  - **Phase 1**: Build forest of inverted trees using asynchronous Union-Find
  - **Phase 2**: Label each vertex with ID of its boss
  - **Phase 3**: Prune out insignificant components
- Tested and verified with real-world graphs

Optimizations

- Motivation
  - Highly **communication-intensive**: lots of tiny messages (~1.5B messages for 16M vertices with 6M edges)
  - Deep trees causing slow Find operations
- **Locality-based tree climbing**
  - Sequentially traverse tree path for vertices in the same chare
  - Increases work per chare, but drastically reduces number of messages
  - 25x speedup in tree construction
- **Message aggregation**
  - Topology-aware routing and aggregation of network communication using TRAM library
- **Local path compression**
  - Make local tree in each chare completely shallow
  - Provides one-hop access to bosses



Probabilistic Mesh

- Random graph built on a lattice structure
- Edge between two lattice points (vertices) determined from a probability value using vertex coordinates
- Easy to scale graph size, verify results and catch race conditions

Performance Evaluation

- Test environment
  - NCSA Blue Waters

1. **Phase execution time** for different probabilities

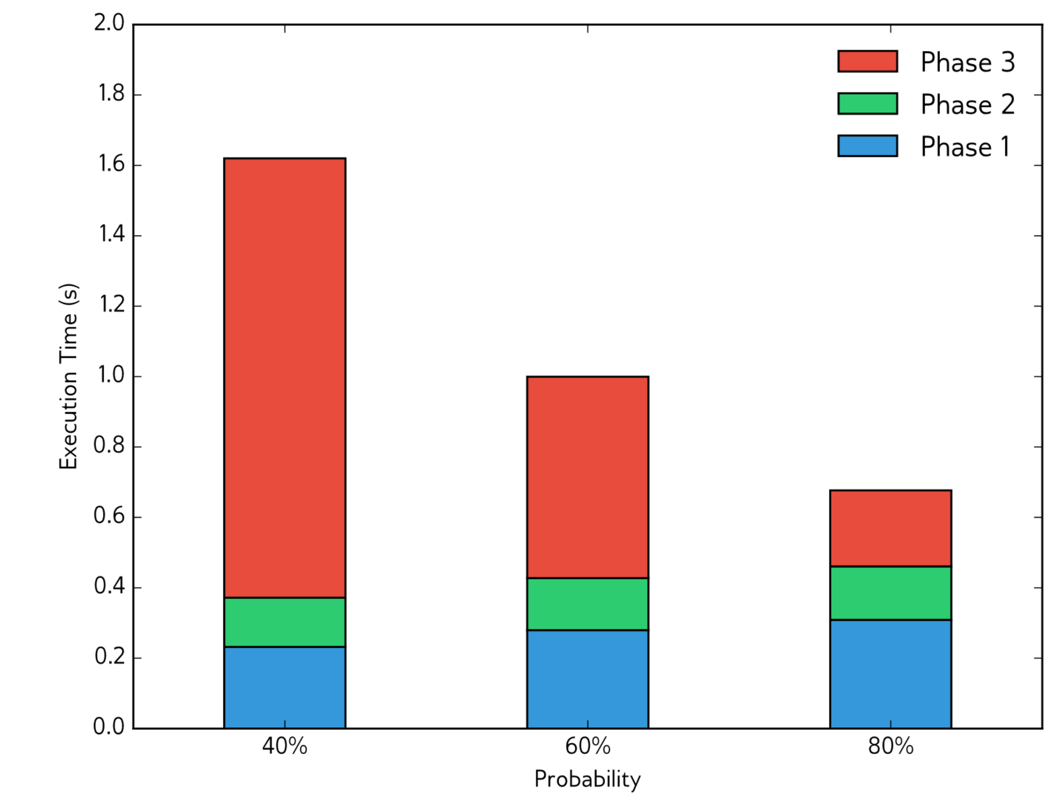


Figure 1: Mesh size 1024^2 on 96 cores

2. **Strong scaling**

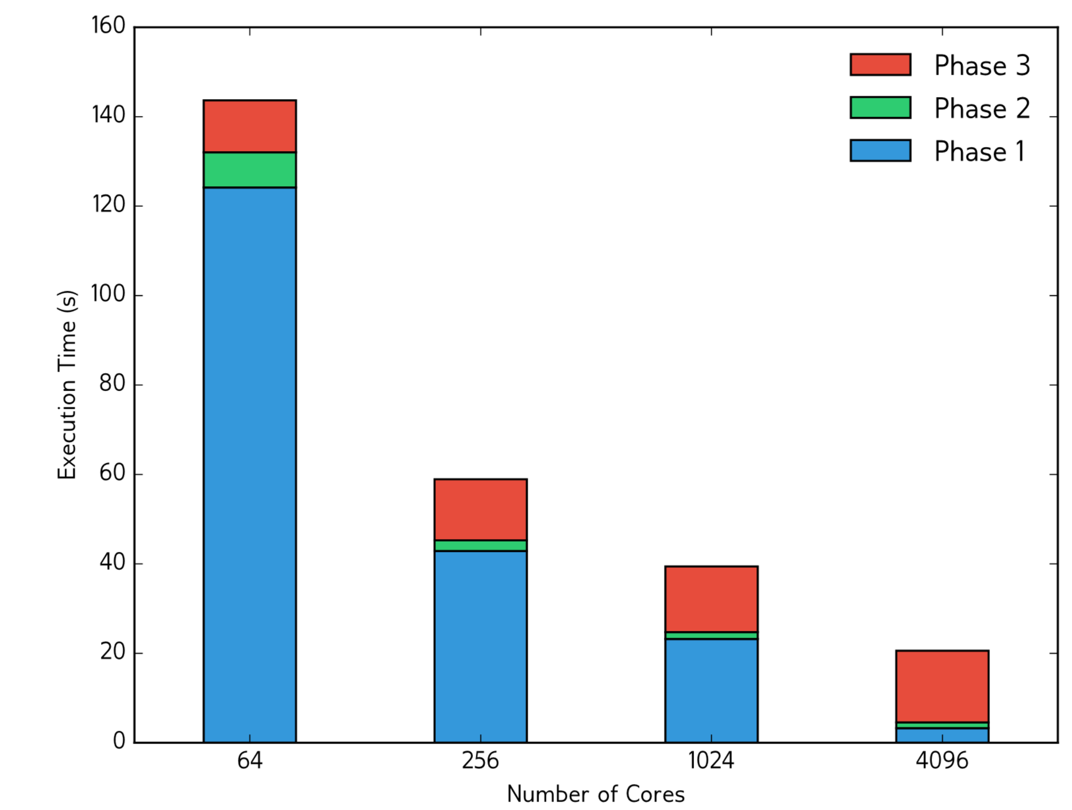
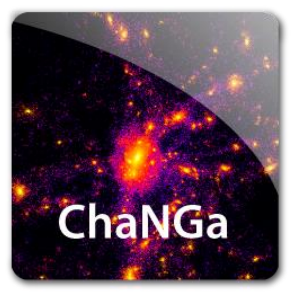


Figure 2: Mesh size 8192^2, 60% probability

Future Work



- Integrate with **ChaNGa**
  - Galaxy detection based on Friends-of-Friend algorithm
  - Detect clusters of stars and classify galaxies