UB SEAS course CSE 633: Parallel Computing

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}`

`union(1, 8) ⇒ {1, 2, 3, 4, 5, 6, 7, 8}`

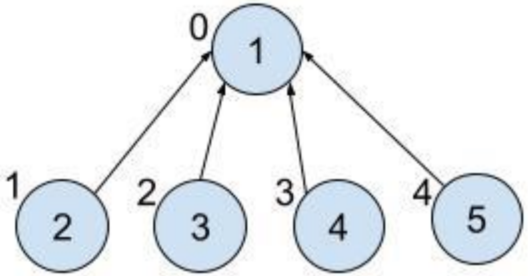
`find(1) ⇒ representative_of(`S_1`) ⇒ 1`

`find(7) ⇒ representative_of(`S_1`) ⇒ 6`

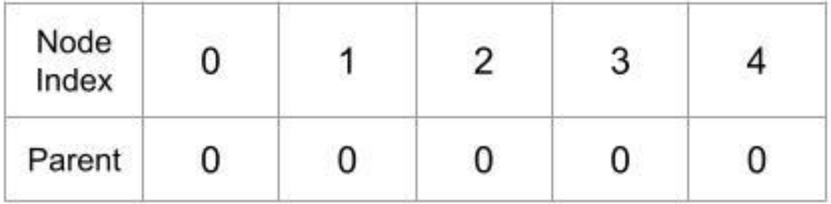
The Union-Find Forest (U)

- Union-Find usually uses the forest of directed trees data structure.It 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.
      root(T_i) = represenatative_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.



Node Index	0	1	2	3	4
Value	1	2	3	4	5



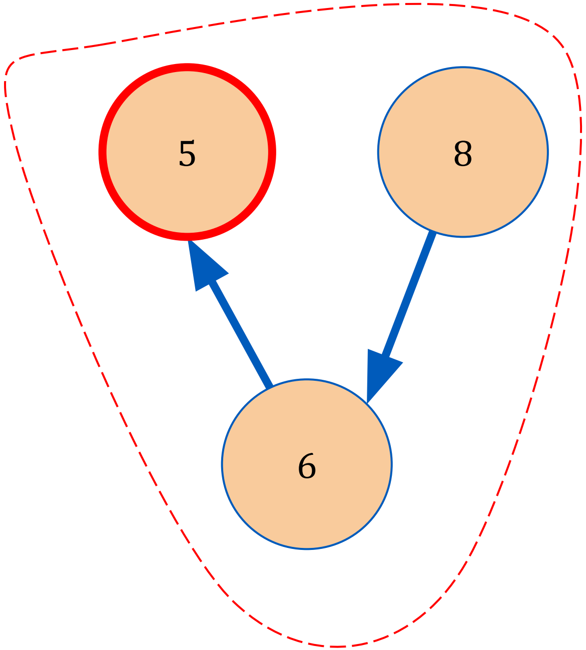
Node Index	0	1	2	3	4
Parent	0	0	0	0	0

Find Operation

FIND is the operation of getting the representative of a connected component.

location	5	6	8
root	5	5	6

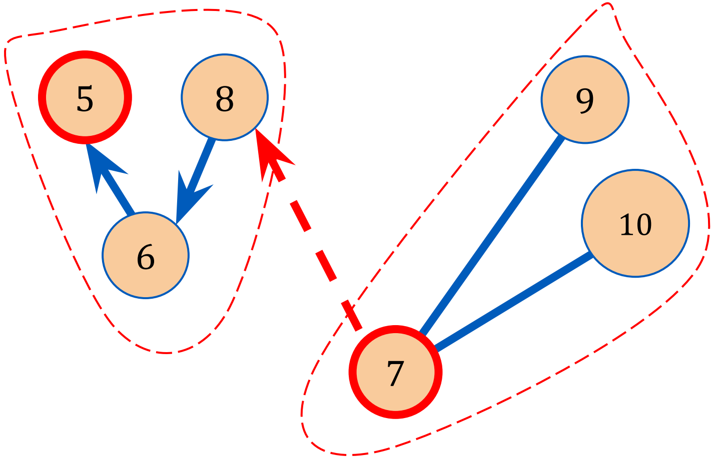
```
def find(roots: list[int], location_x: int) -> int:
    if roots[location_x] == None:
        return location_x
    else:
        root = location_x
        while root != roots[root]:
            root = roots[root]
        return root
```

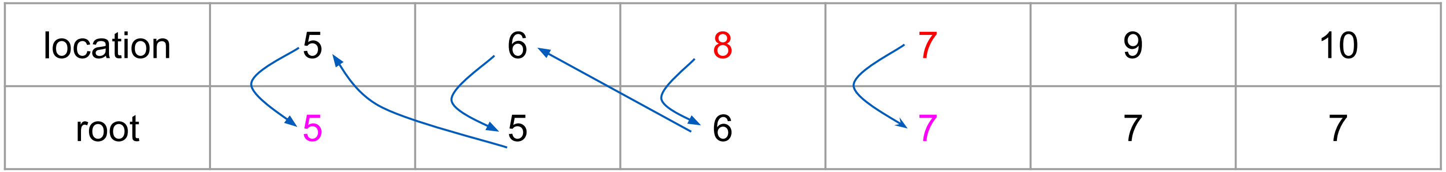


Union Operation

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)
```

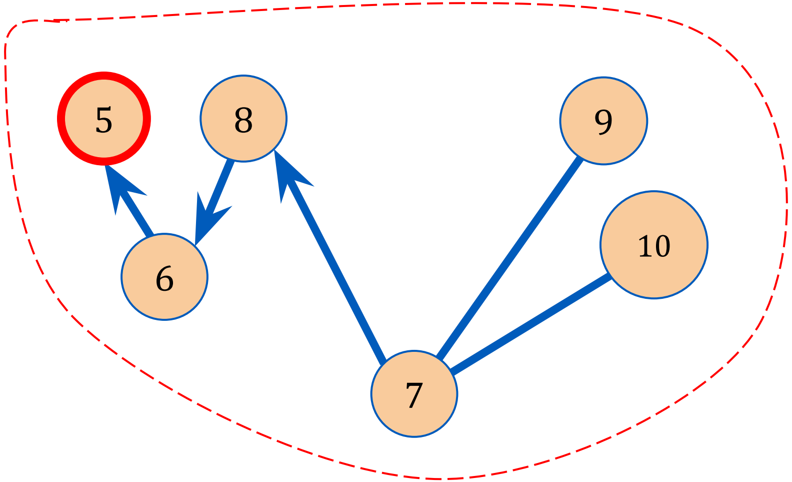




location	5	6	8	7	9	10
root	5	5	6	7	7	7

UNION Operation

```
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)
```

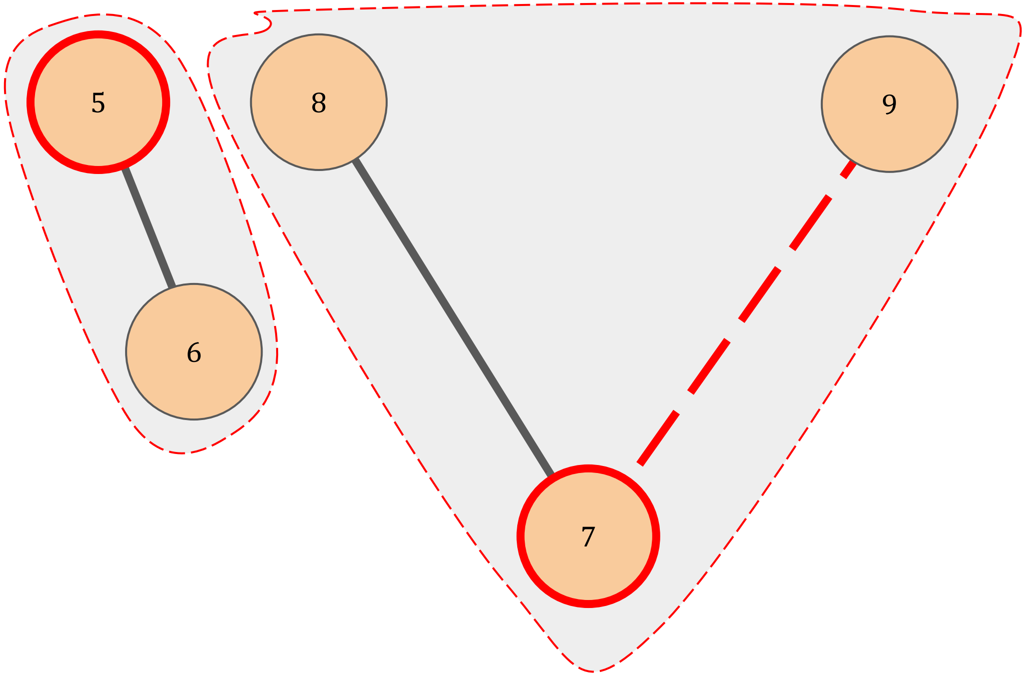


location	5	6	8	7	9	10
root	5	5	6	5	7	7

Creating Connected Component for a Graph

by iterating over edges

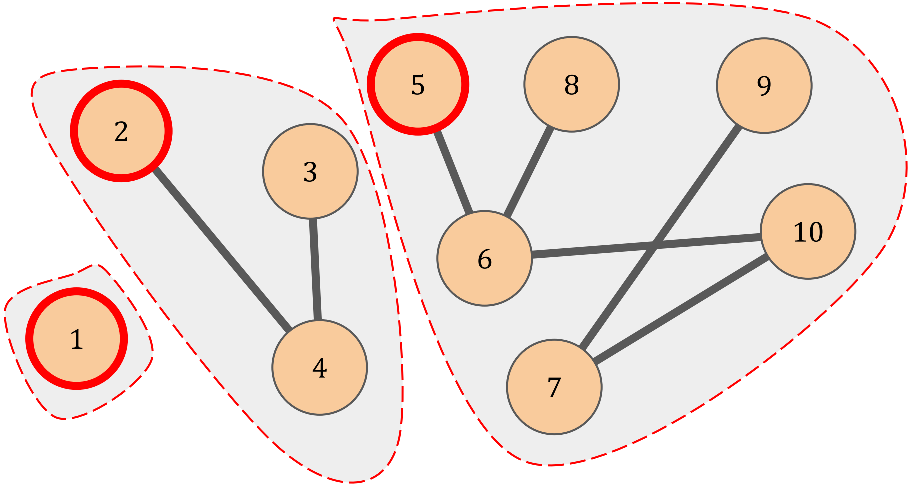
```
for edge in edges:
        if not(edge[0] in values):
            add_value(edge[0])
        if not(edge[1] in values):
            add_value(edge[1])
        union(edge[0], edge[1])
```



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

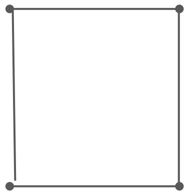
1. Two processors exchange the vertex values. ***P_i gets V_j and P_j gets V_i***
2. Both of them, check for vertex overlaps. ***P_i and P_j calculates $V_i \cap V_j$***
3. Both of them, generate edges of **(value, root[value]) for vertices in $V_i \cap V_j$**
4. Both of them, **exchange these new edges 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).

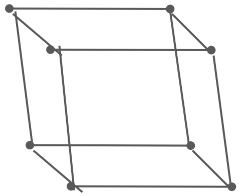
The Choice of Network - Hypercube

Some hypercubes with their dimensions:

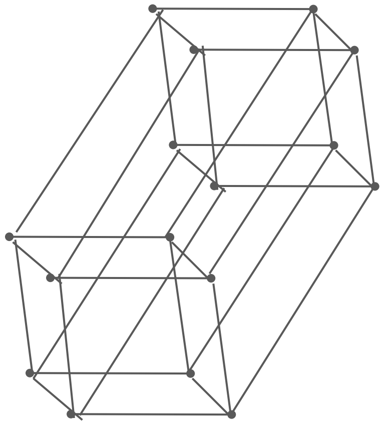
n=2



n=4

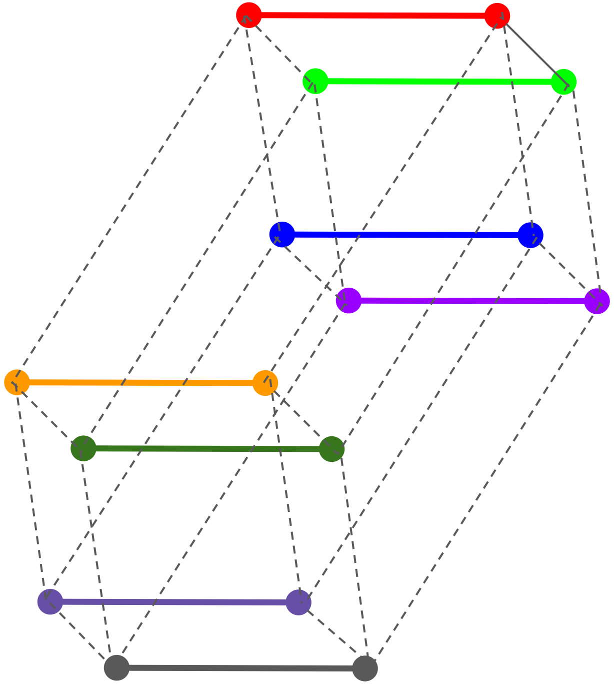


n=8



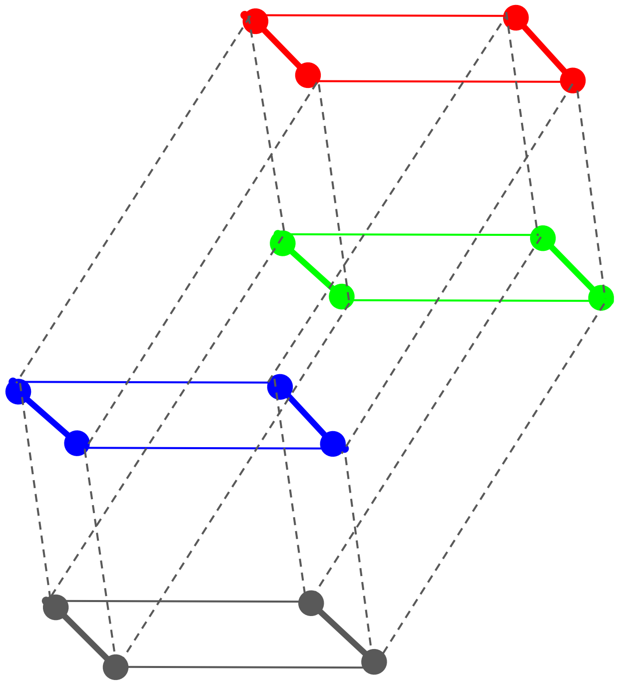
n=16

Iterations for n=16 hypercube



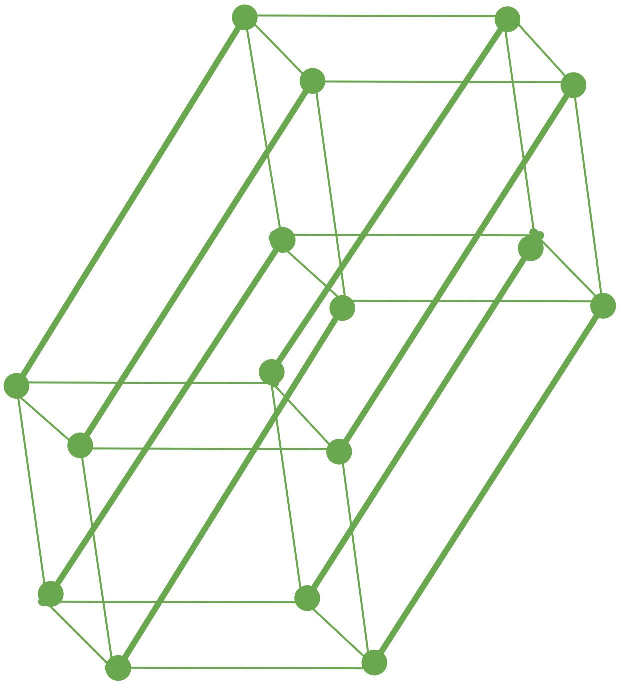
Iteration 1

Iterations for n=16 hypercube



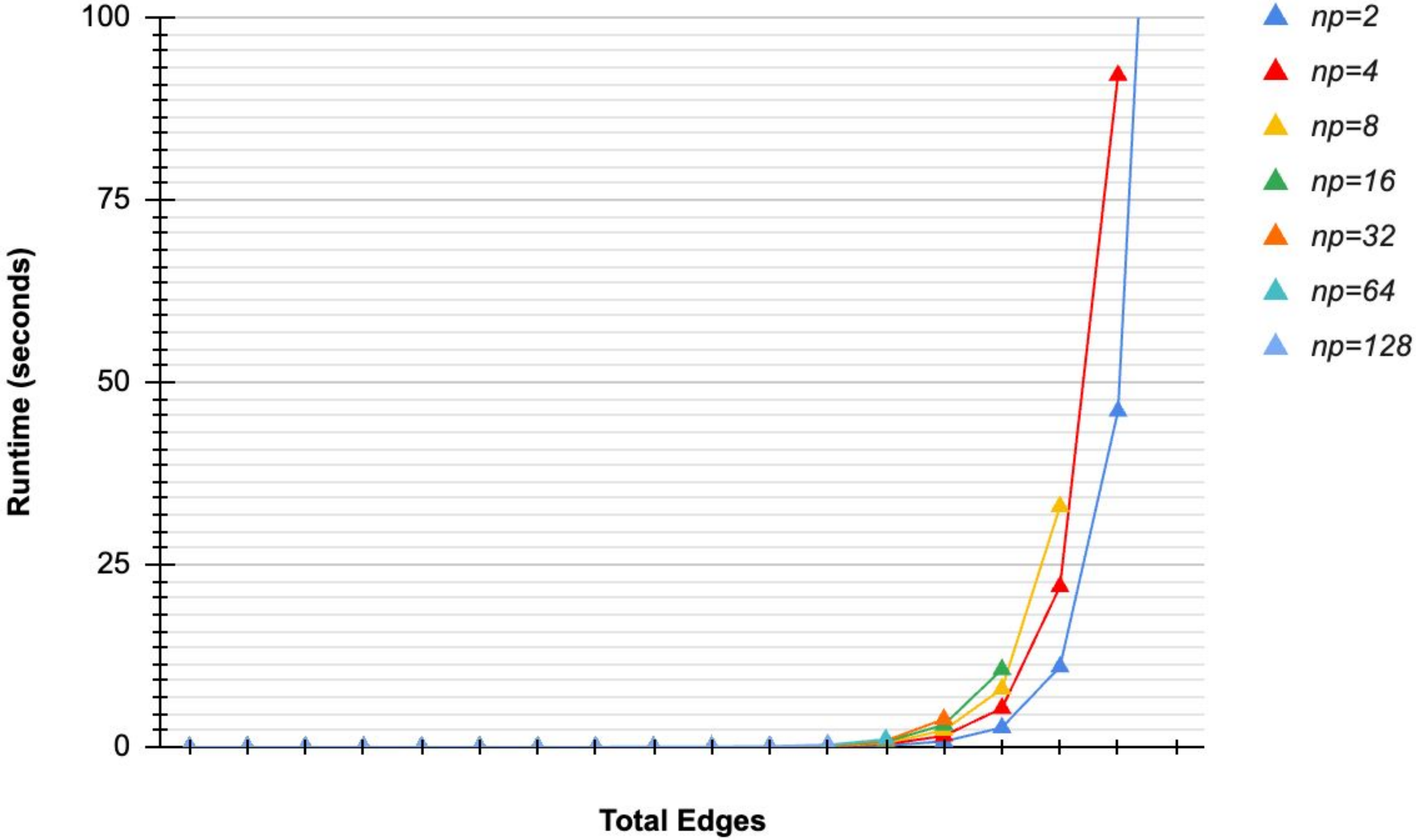
Iteration 2

Iterations for n=16 hypercube

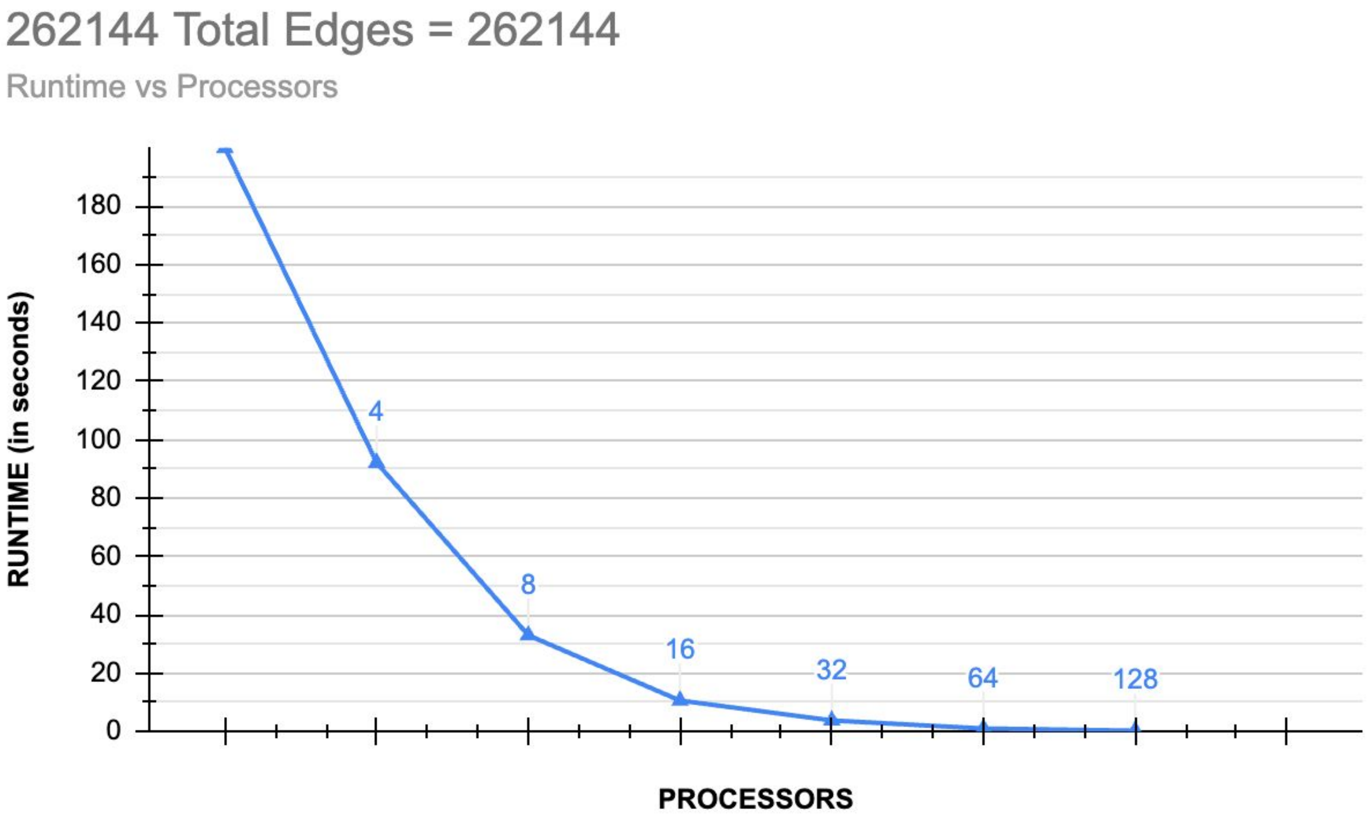


Iteration 3

Runtime VS Edge Count



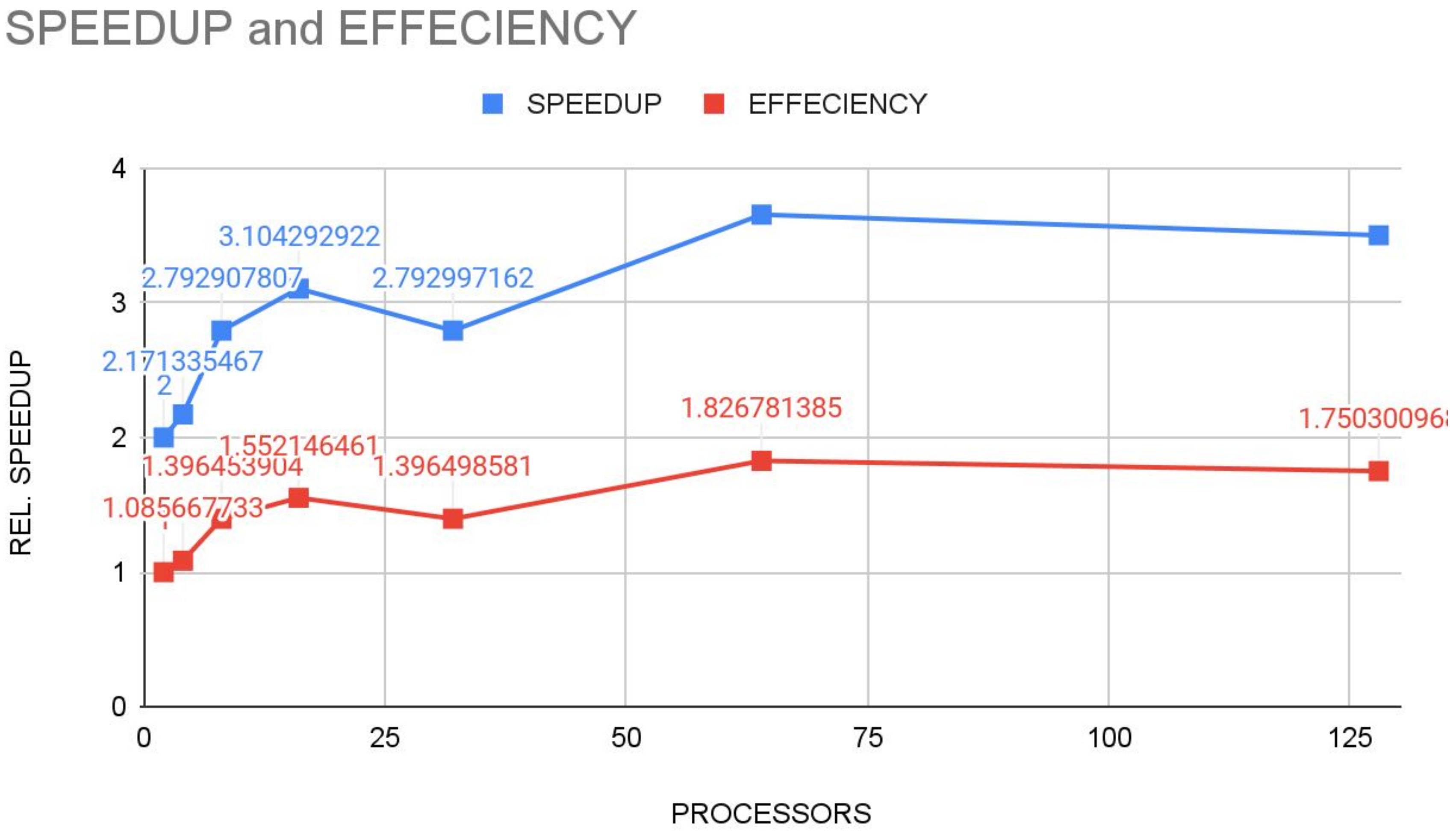
Runtime VS Processor Count



Speed Up & Efficiency Measurements for Constant Input Size

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

*Reference Measurement


SPEEDUP and EFFECIENCY
SPEEDUP
EFFECIENCY
REL. SPEEDUP
PROCESSORS
0
1
2
3
4
0
25
50
75
100
125
3.104292922
2.792907807
2.792997162
2.171335467
2
1.826781385
1.7503009
1.552146461
1.396453904
1.396498581
1.085667733

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) complexity] instead of a `hashmap` implementation [O(V) complexity].
  - That made sending messages [O(V) 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
(https://people.csail.mit.edu/jshun/6886-s19/lectures/lecture15-2.pdf)

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