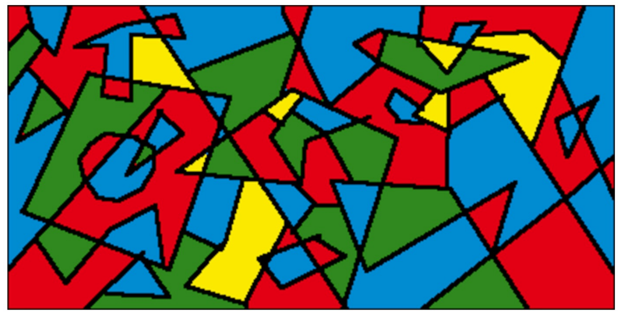
Parallel Four-Color Map-Solver

PLAY OUR DEMO !

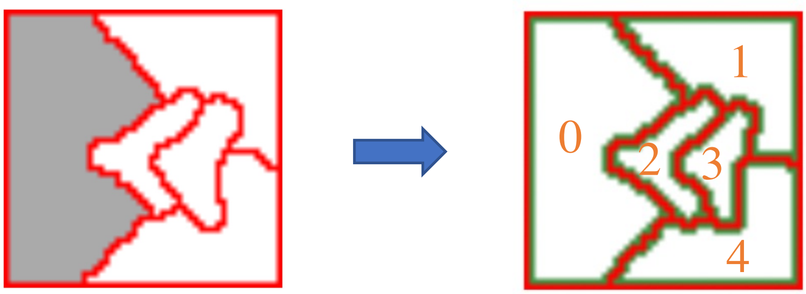
Background

Four Color Theorem: Any map is colorable with 4 different colors, such that after coloring, any two adjacent countries have different colors.

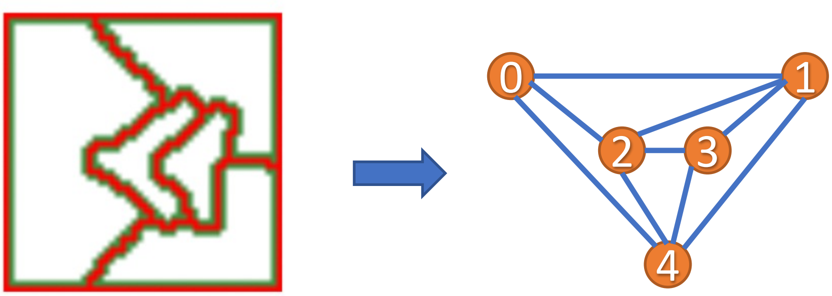


Phase 1 Convert Map to Graph

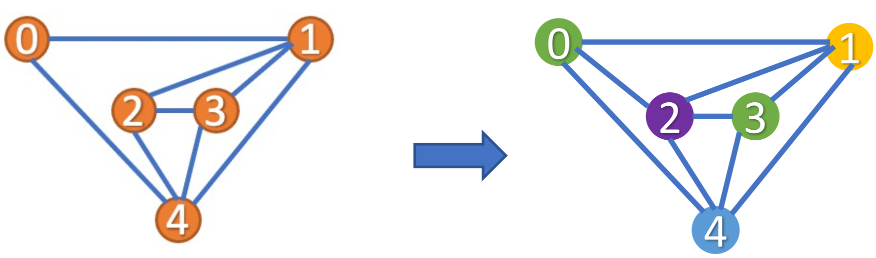
Phase 1.1 Find Nodes



Phase 1.2 Find Edges



Phase 2 Color Graph



Parallelism Analysis

Phase 1.1: Convert Map to Graph – Find Nodes

1. BFS over map to find countries
2. Keep track of marginal points for find edges

```
marginalPixels = [];
nodeId = 0;

func findNodes():
    for pixel in map:
        if pixel == -1:  // -1 represents node (i.e. country)
            curMarginalPixels = fillArea(pixel, nodeId); // BFS algorithm to find the area of current node and assign its pixels with nodeId
            marginalPixels.push(curMarginalPixels)
            nodeId++;
```

Bottleneck Analysis

Time cost correlates with map size, but not with the number of nodes

How to parallelize

Approach: Divide map into segments and each thread find nodes for one segment

Problem: 1. One country get segmented, how to merge back?

2. How to map local ID to global ID?

3. Arbitrary node shape -> how to main consistency?

Testcase	Map Size	Node Num	Time Cost for FindNodes (ms)
A	200 x 200	369	2
B	1000 x 1000	382	24
C	1000 x 1000	877	28

Phase 1.2: Convert Map to Graph – Find Edges

1. Find “close-pixel-pairs” for each node
2. Consider as edge if reach threshold

```
edges = []

func findEdges():
    for i in range(nodeNum):
        cnt = {}
        curMarginalPixels = marginalPixels[i]
        nearbyPixels = findNearby(curMarginalPixels, edge_distance)
        for pixel in nearbyPixles:
            j = map[pixel]
            if j != i
                cnt[j]++
                if cnt[j] >= edge_threshold && i < j:
                    edges.push({i, j})

```

Parallelism Analysis

Bottleneck Analysis
Time cost correlates with map size and the number of nodes

How to parallelize
Approach: parallel over nodes
Workload is independent -> easy to parallelize

Testcase	Map Size	Node Num	Time Cost for FindEdges (ms)
A	200 x 200	369	1
B	1000 x 1000	382	9
C	1000 x 1000	877	13

Phase 2: Color Graph

1. Brute-force backtracking using recursive function
2. Return directly when found solution

```
colors = [-1 * nodeNum]

func colorGraph():
    colorGraphHelper(0);

func colorGraphHelper(n):
    if timeout:
        return TIMEOUT

    // find the solution
    if n == nodeNum:
        return SUCCESS

    // recursion
    for c in getPossibleColors(n):
        colors[n] = c
        if colorGraph(n + 1) == SUCCESS:
            return SUCCESS
        colors[n] = -1
    return FAILURE
```

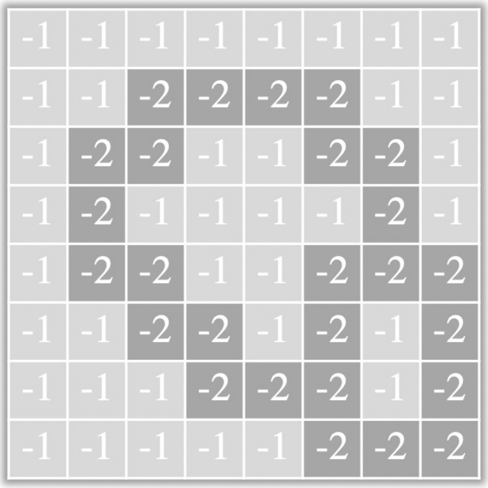
Bottleneck Analysis
Graph needs to be quite dense and complex so that cannot early prune

How to parallelize
Approach: parallel over color assignments
Problem: 1. How to parallelize with recursive function
2. How can the first thread who finds the solution notify others to exit early?

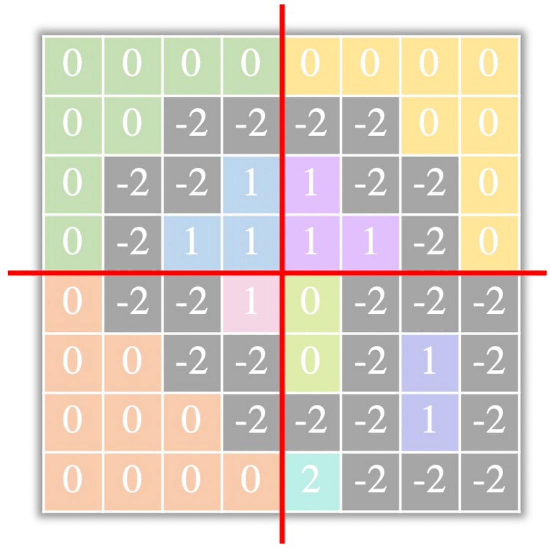
Approach

Phase 1.1: Convert Map to Graph – Find Nodes

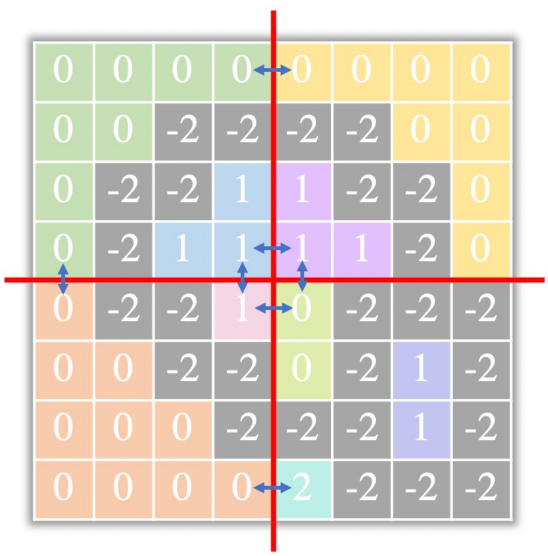
1 Read map as vector of pixels

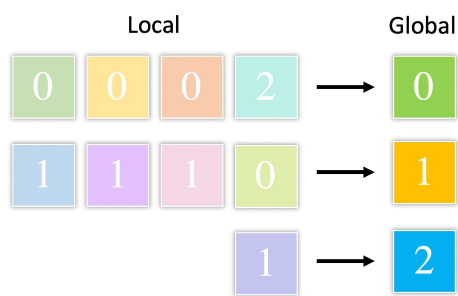


2 Grid map and each thread updates its local node ID (parallel)

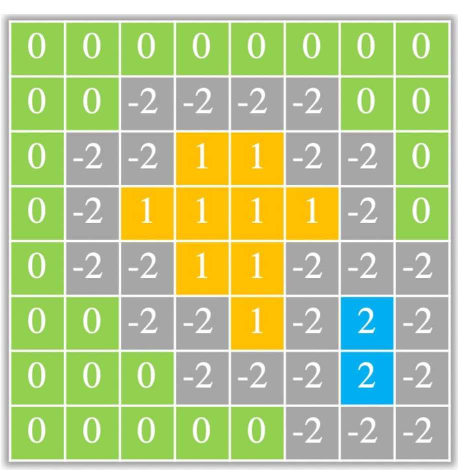


3 Find conflict pairs (parallel) and use UnionFind to map to global node ID





4 Update local node ID as global node ID (parallel)



Approach

Phase 1.2: Convert Map to Graph – Find Edges

Approach: Use **omp parallel for** to parallelize over nodes

```
edges = []

func findEdgesPar():

    // parallelizing over nodes
    #pragma omp parallel for shared(edges) schedule(dynamic)
    for i in range(nodeNum):
        ...

        // atmoic opertion
        #pragma omp critical {
            edges.push({i, j})
        }
```

Phase 2: Color Graph

Approach:

1. Use **omp task** to parallelize recursive function
2. Use **omp cancel** and **omp cancellation point** to notify other threads

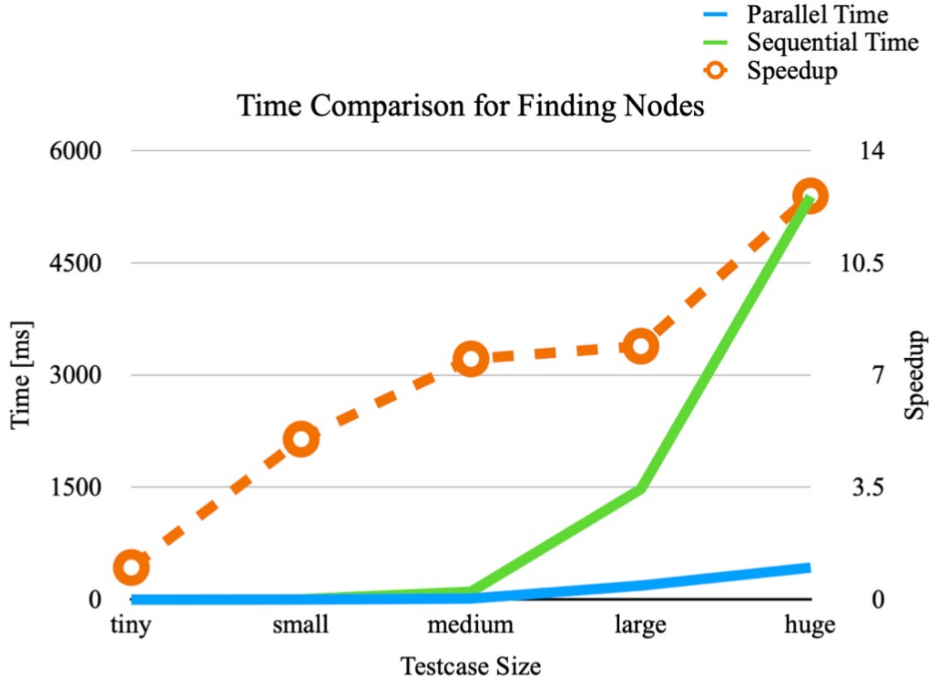
```
colors = [-1 * nodeNum]

func colorGraphPar():
    #pragma omp taskgroup
    {
        colorGraphHelperPar(0, colors);
    }

func colorGraphHelperPar(n, curColors):
    ...
    // the solution is found, save it in colors
    if n == nodeNum:
        colors = curColors
        return SUCCESS

    // recursion
    rst = FAILURE

    for c in getPossibleColors(n):

        // copy colors to each thread's own address space
        privateColors = curColors

        // create tasks for taskgroup, each task is checking a new color
assignment
        #pragma omp task firstprivate(privateColors) shared(rst) {

            // check whether any thread canceled the taskgroup, if so, exit early
            #pragma omp cancellation point taskgroup

            privateColors[n] = c
            if colorGraph(n + 1, privateColors) == SUCCESS:

                // atomic operation for accessing shared variable
                #pragma omp critical
                {
                    rst = SUCCESS;
                }

                // first thread to find the solution, cancel the taskgroup
                #pragma omp cancel taskgroup

            privateColors[n] = -1
        }
    return rst
```

Results

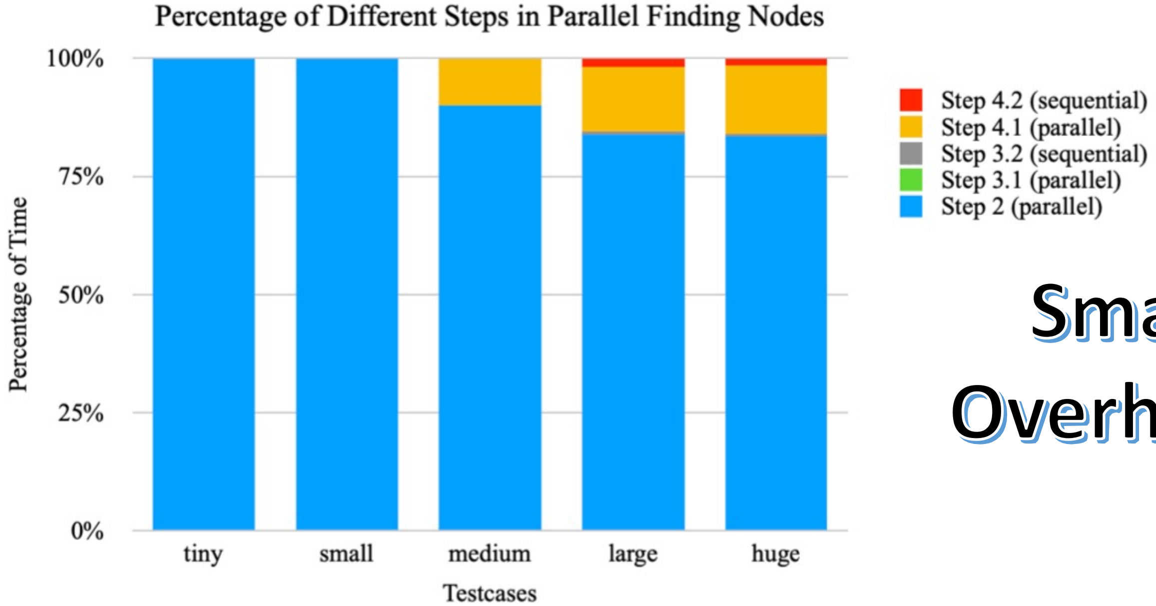
Testcases

Testcase	Map Size	Node Num	Edge Num
tiny	20 x 20	3	2
small	200 x 200	369	445
medium	1000 x 1000	877	1748
large	4000 x 4000	3109	7706
huge	6000 x 6000	6045	13926
medium-cornered	1000 x 1000	264	519
large-cornered	4000 x 4000	701	1672

Phase 1.1: Convert Map to Graph – Find Nodes



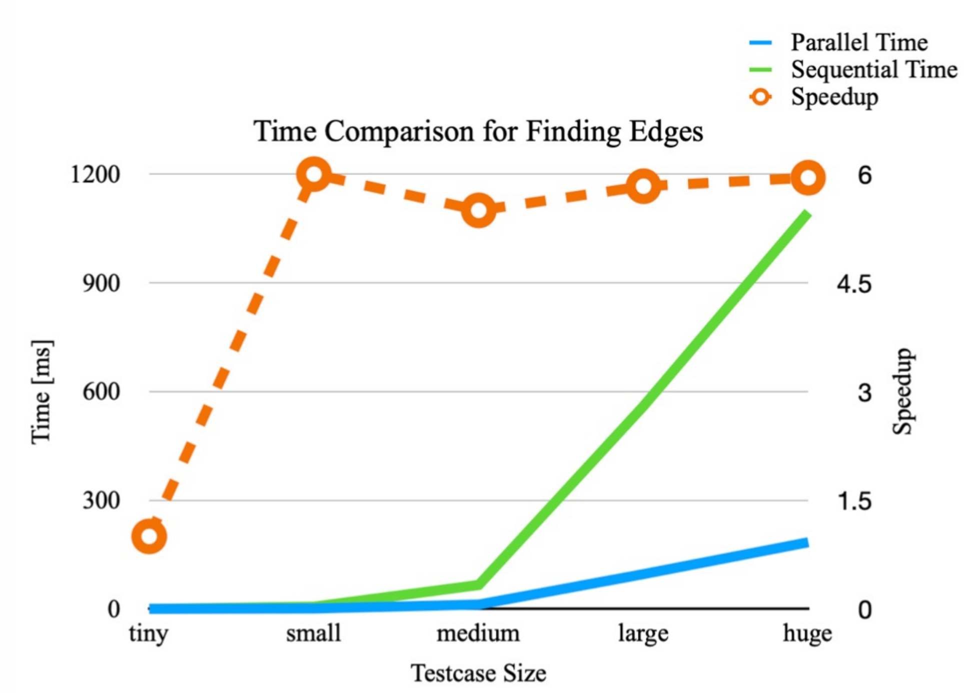
Super Linear Speedup !



Small Overhead!

Results

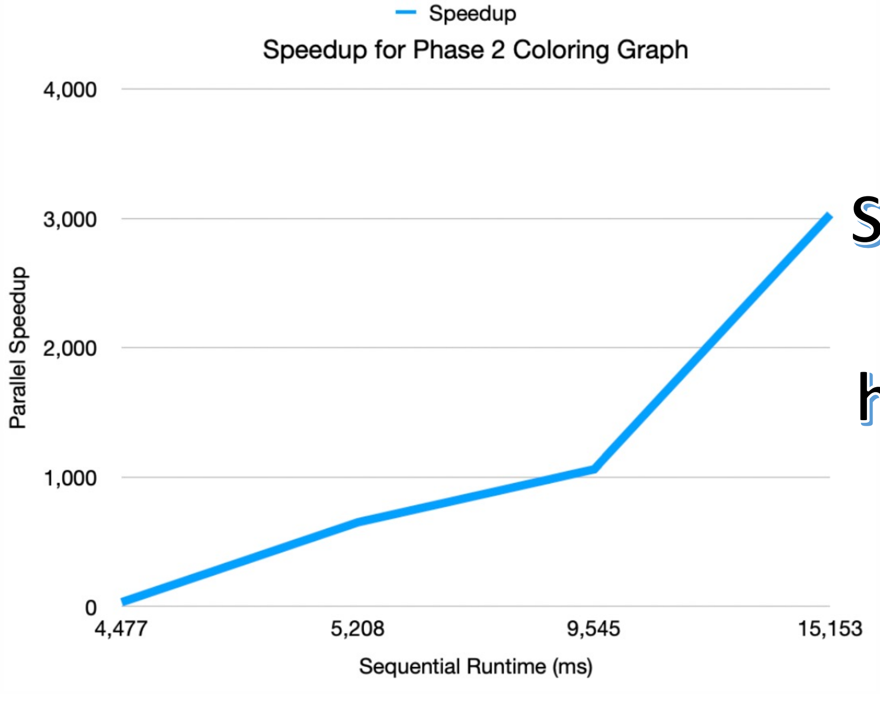
Phase 1.2: Convert Map to Graph – Find Edges



Linear Speedup !

Phase 2: Color Graph

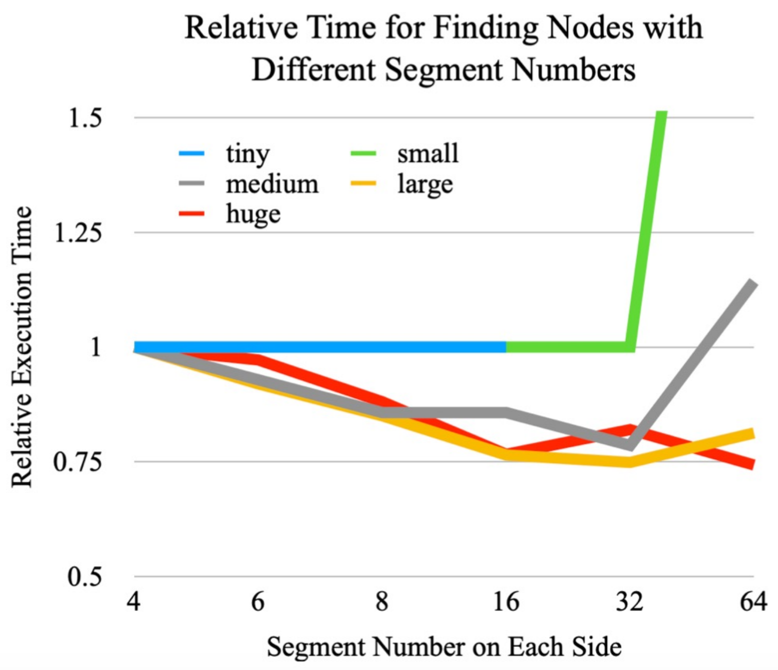
Testcase	Sequential	Parallel (Run 1)	Parallel (Run 2)	Parallel (Run 3)
40_100_4s	4,477	69	451	118
40_100_5s	5,208	7	4	12
40_100_9s	9,545	4	11	3
40_100_15s	15,153	7	9	12



Supersuperlinear Speedup but has Randomness

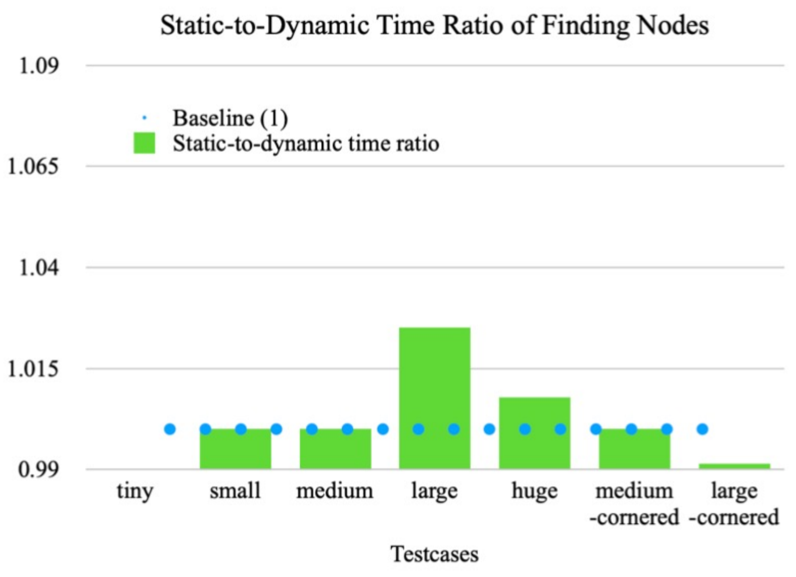
Deeper Analysis

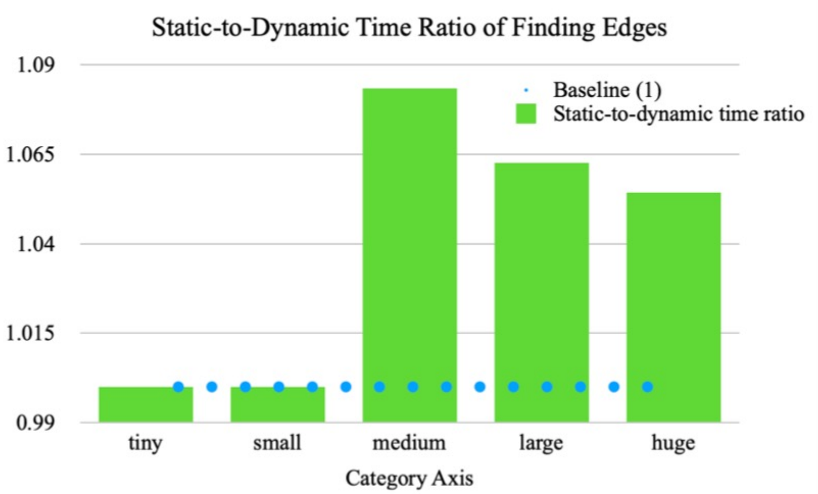
Effect of Segment Number



32 is optimal

Effect of scheduling policy





Dynamic is Better