# 618 Final Project Final Report

## Abstract

We implement a parallel map solver for the four-color problem using openMP and perform a detailed analysis of performance characteristics. We measure time and memory costs on different region maps and compare the performance between different scheduling algorithms.

**Source Codes**: https://github.com/Elena-Qiu/15618-Final-Project

## Background

### *Problem Description*

The probelm we are targeting is called four-color problem. It derives from four-color theorem, which states that any map is colorable with 4 different colors, such that after coloring, any two adjacent countries have different colors. The whole algorithm is divided into two phases: convert map to graph and color graph.

### Phase 1: Convert Map to Graph

Given a map image consisting of pixels, we need to first convert it to a graph consisting of nodes and edges. A Node represents a country in the map, and an edge connecting two nodes represents the fact that two countries corresponding to those two nodes are adjacent in the map.

While it's natural to represent a map image as a 2d array, we use a 1-dimensional vector to simplify the implementation (after all a 2d array is essentially rolled out as a 1d array in underlying memory structure. Nevertheless, it's still totally okay to keep viewing the map as a 2d array in our abstract conceptual diagram, and in later discussion we'll refer to this array as a "2d array" instead of a "1d vector".) This vector consists of integers with length equal to width `w` times height `h`, where each integer is either -2 (representing border) or -1 (representing country).

```
int w;
int h;
vector<int> map(w * h);
```

#### Phase 1.1 Find Nodes

To find nodes, we use the Depth First Traverse (DFS) to identify countries in the map, each corresponding to a node respectively. Note that we also collect marginal points of each node in this stage, but those marginal points will not be used till the find-edges step. The country is defined as a set of connected white pixels that are bound by the black pixels (borders). The marginal points are those whose neighbors contain black pixels, namely the

outer most boundary of the country. The following two figures shows an example map, with borders colored red, a country colored gray on left figure, and marginal points colored green on right figure.

For the sequential algorithm, we simply traverse all of the pixels in the map array and if we meet a white pixel, we call the `fillArea` helper function to find current country and assign its pixels with a new `nodeId` in the map using DFS. The pesuedo code is as follows.

```
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++;
```

### Phase 1.2 Find Edges

To find edges, we need to identify node pairs whose marginal pixels are nearby enough. We do that by traversing marginal points of each node, and detect if there are pixels from other nodes nearby (i.e. within a certain distance `edge_distance`), if so, we consider these two nodes as connected, i.e. there is an edge between them. To prevent duplicate edges, we only save the edge when `i < j`. The pesuedo code is as follows.
We also take into account a case where different countries only share a single common point in their borders (like a pie chart). In this case, countries should not be considered adjacent unless they really share an edge (namely share more than a single point). Thus we only treat nodes as connected if the number of "close-pixel-pairs" between them reach a certain threshold `edge_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})

```

## Phase 2: Color Graph

To color the graph using less than four colors and make sure that connected nodes have different colors, we use the brute-force backtracking algorithm. The initial `colors` of the nodes are undefined (set to `-1`). Then we implement a recursive function that for each node `n`, we traverse all of the possible colors that are different from its neighbor nodes. The recursive function returns when a solution is found or it reaches time out. The pesuedo code is as follows.

```
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
```

Based on the four color theorem, there should always be a solution for any map so actually it's impossible to return `FAILURE`.

# *Parallelism Analysis*

## Phase 1.1 Convert Map to Graph - Find Nodes

To analyze the bottleneck in this phase, we run the testcases as shown in the table below. Comparing testcase A and B, we find that its time cost is highly correlated with the number of pixels in the map. However, comparing testcase B and C, we get that its runtime is not sensitive to the number of nodes in the map.

| 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 |

Therefore, we derive that the main reason of its time cost is that it needs to traverse all the pixels in the map to identify countries. In this case, it can greatly benefit from data-parallelism. Instead of letting one thread be responsible for all the pixels, we can divide the workload into different portions of pixels and assign them to different threads for parallel execution.

However, due to the arbitrariness of node shapes, it's very challenging to maintain consistency of different portions when they are processed independently, for example, when we are segmenting the map, it is highly likely that one country may be partitioned into different segments. Then we need to figure out how to identify and connect those segments back to ensure the correctness. Moreover, there are dependencies in the memory as well, such as how to assign the global node ID and how to correctly maintain the marginal points of each country. We will discuss our approach in details in next section. Considering these issues, we expect there will be synchronization overhead for parallelizing this step, so the speedup will not be linear.

We also take into consideration the possibility of false sharing between different threads, since it's likely that a cache line spans across two segments that correspond to two threads. Thus we also experimented with a version that is resilient to false-sharing by converting the 2d array into a 4d array such that elements in each segment is continuous in its memory layout. This approach, however, turns out to be not very helpful, and we'll discuss about it in "Discussion on Different Parallelization Configurations" section.

## Phase 1.2 Convert Map to Graph - Find Edges

To analyze the bottleneck in this phase, we run the same testcases as in Phase 1.1. The results are shown in the table below. Comparing testcase A and B, we find that its time cost is highly correlated with the number of pixels in the map. Comparing testcase B and C, we get that its runtime is also highly correlated to the number of nodes. The results are expected because if the number of nodes is large, we need to find edges for more nodes, and if the map is large, we need to traverse more marginal points of each node.

| 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 |

This phase is highly parallelizable because the its workload are quite independent. Based on the algorithm, we traverse the marginal points of each node to find its connected nodes, which does not have any dependencies in between. Therefore, we can parallel over all nodes, namely each thread is responsible for finding the edges of each node. We expect linear speedup for this step because the workload is highly separable and there is little synchronization overhead.

### Phase 2 Color Graph

By running a bunch of different testcases, we find this phase is only time-consuming when the graph of the map is quite complex (e.g. the graph is densely connected, thus posing strict constraints on the coloring attempts. In this case, the backtracking algorithm needs to traverse almost all of the search space ($O(4^n)$) before it finds a solution, i.e. it cannot early prune). This kind of testcase is hard to find in real life so we write a testcase generator by ourselves to find such testcases.

For such testcases, there is a lot of space for speedup using parallelism. Instead of letting one thread traversing all the possible color assignments sequentially in the search space, we can let several threads be responsible for them respectively and searching them in parallel. In this way, if one thread find a possible solution, it can notify the other threads to stop searching. In this way, we can realize early pruning, which saves great amount of work in searching incorrect color assignments.

There are two difficult parts in parallelizing this phase. Firstly, since we are using a recursive function, which cannot direcly use a `parallel for`, then we need to figure out how to paralleize in this scenario. Secondly, we need to figure out a mechanism to let the first thread who finds a solution notify the other threads to exit their normal execution flow.

## Approach

We implement our algorithm in *C++*. For parallelizing, we use the *openMP* framework and we are targeting at CPU machines with multiple cores like *GHC*.

To make our demo more interactive, we write a web application which visualizes the coloring results of the map and also allows users to draw their own maps and solve the coloring for them. It consists of

- a frontend written for user interaction and map rendering, written in *JavaScript* with library *P5.js*, and
- a backend server for data communication between frontend and parallel map solver, written in *C*.

### Phase 1.1 Convert Map to Graph - Find Nodes

As shown in the following figure, we use 4 steps to find all nodes from a map image in parallel.

**Step 1: (Sequential)** We take inputs of the map image (either from file or from web frontend) and divide it into smaller segments. In order to prevent from false sharing between threads, we structure the pixel array in a way such that pixels in each segment is continuous in memory layout. We give some comparisons and discussions between this version against a naive linear vector in section Discussion on Different Parallelization Configurations.

```
int w;
int h;
int segment_size;
vector<vector<int>> map(
    w * h / segment_size,
    vector<int>(segment_size)
);
```

**Step 2: (Parallel)** we spawn a openMP thread group and let each thread pick up a segment of the map image and do a local depth-first-search to identify all "partial nodes". Each "partial node" may be a complete node, or just a segment of an actual node in the global scope. Each "partial node" is also assigned with a local node ID that is unique in its own segment, but may duplicate with other segments.

We also find marginal points for each "partial node" in this step. These marginal points will be used for find-edges step.

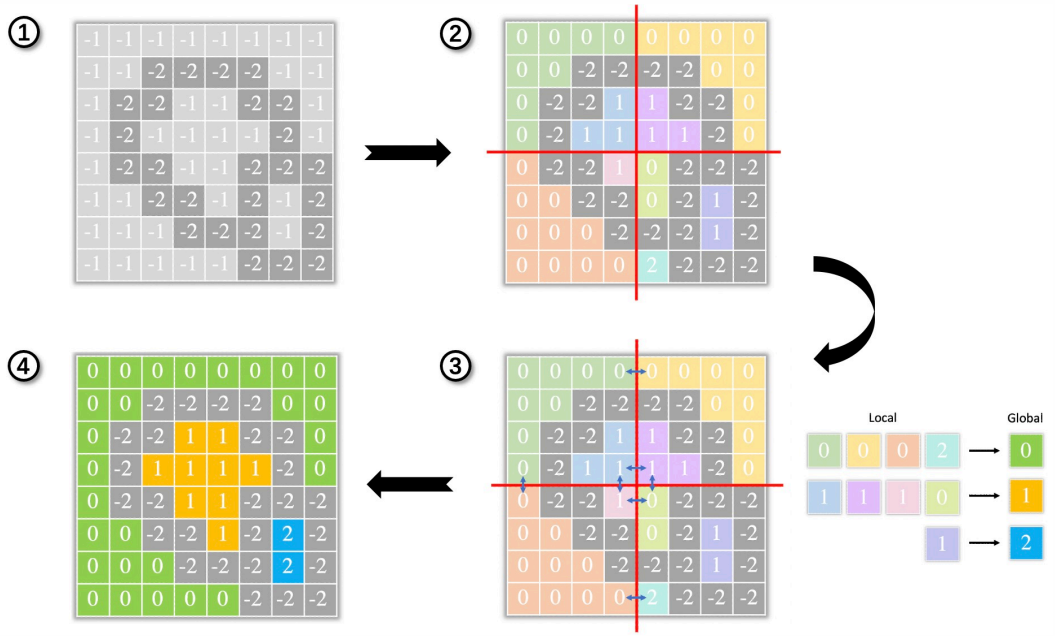
**Step 3.1: (Parallel)** each thread then again picks up a segment of the map, and it walks along the boundary of the segment and examine pixels on both sides of the segment boundary pair by pair to see whether the pixel from the neighboring segment actually belongs to the same node as the pixel in its own segment. We use "conflict pair" to name a pair of pixels from different segments that belong to the same node in global view. Once the thread detects a "conflict pair", it records that pair in a list. With those "conflict pairs", we could reconstruct nodes that are "cut apart" when dividing map into segments.

*Note:* in this step we also identify marginal points on boundaries. These marginal points cannot be detected in step 2 because their neighboring boundry pixel belong to other segments, so they can only be supplemented in this step.

**Step 3.2: (Sequential)** The master thread collects all local node IDs and "conflict pairs" and decides a global node ID for each node. In specific, the master thread uses a UnionFind data structure to compress conflict pairs into a mapping from local node ids to global node ids.

**Step 4.1: (Parallel)** Each thread, once more, picks up a segment of the map, and updates its local node id to the global node id following the mapping given by step 3.

**Step 4.2: (Sequential)** The master thread merges all the local marginal points into the global marginal points vector following the mapping given by step 3.




## Phase 1.2 Convert Map to Graph - Find Edges

Thanks to the lack of dependencies between tasks, in this approach we do not need to meticulously device the algorithm to ensure consistency between threads. Thus adding an `omp parallel for` for node traversal is sufficient to parallelize this part of the program, as shown in the pseudocode below. In this way, we are mapping the workload of finding edges for each node to each thread. Since `edges` are shared between threads, we need to add `omp critical` when we are pushing a new edge into it to ensure the atomicity. Since the size of each country varies a lot from each other, the number of marginal points for each node is quite different from each other. Therefore, the workload of finding edges for each node has great imbalance, thus **dynamic** scheduling policy is the most suitable option here.

```
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

For the recursive function, we cannot directly use `omp parallel for`. After reading documentation, we find that`omp task` is most suitable for paralleizing recursive functions. We also find that we can use `omp cancel` and `omp cancellation point` to notify later threads that a solution is found and ask them to exit their normal execution flow. As shown in the pseudocode below, the parallelizing algorithm is realized as follow:

1. When we first call the resursive function, we create a taskgroup.

2. During recursion, when we find a new possible color assignment, we create a new task for checking it and add it to the taskgroup.
3. When a thread finds the solution, it calls `omp cancel taskgroup` to cancel the whole taskgroup so that other threads can be notified to exit.
4. At the beginning of each task, we add `omp cancellation point taskgroup` so that the thread running it will check whether the taskgroup has been canceled or not. If so, it will stop creating new tasks and exit early from the normal execution flow.

There are two dependency issues when parallelizing this step and they are resolved as follows:

1. For the sequential algorithm, we use a global vector `colors` to track the color assignment. However, it is not feasible in the parallelized version as different threads may try to modify the `colors` vector at the same time. If we make this section critical, it will cause a lot of synchronization overhead. To resolve this, for each task, we copy a new vector `privateColors` from `curColors` and set it as private for this task. Here `curColors` is a newly added argument to the recursive function and is passed as the color assignment from last recursion. When a solution is found, we copy the values of `curColors` to the global `colors` to save the solution.
2. For the sequential algorithm, when we traverse the possible colors in each rescursive call, if we find the solution, we will return `SUCCESS` directly. However, it is not allowed in parallelizing. To resolve this, in each recursive call, we create a variable called `rst` and set its initial value as FAILURE. During traverse, if we find a solution, we will set `rst` to SUCCESS and call `omp cancel taskgroup` to stop creating new tasks (i.e. break from the for loop). Finally, we return `rst` in the end. Since `rst` should be a shared variable for all threads, we need to add `omp critical` when we are modifying it.

```
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
```

```
29
30              privateColors[n] = c
31              if colorGraph(n + 1, privateColors) == SUCCESS:
32
33                  // atomic operation for accessing shared variable
34                  #pragma omp critical
35                  {
36                      rst = SUCCESS;
37                  }
38
39                  // first thread to find the solution, cancel the taskgroup
40                  #pragma omp cancel taskgroup
41
42              privateColors[n] = -1
43          }
44      return rst
```

# Results

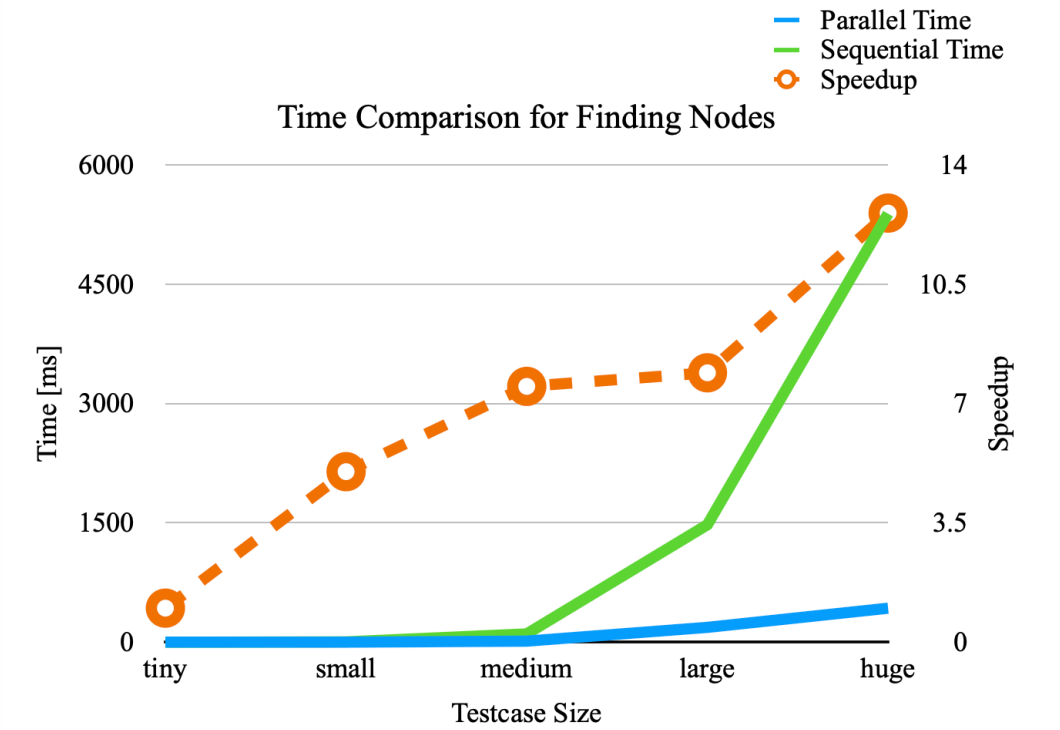
## *Testcases*

In all experiments regarding map-to-graph conversion, we use the following 7 different testcases to test our parallelization performance. They are generated by P5.js codes in our frontend web application using random shapes.

| 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 |

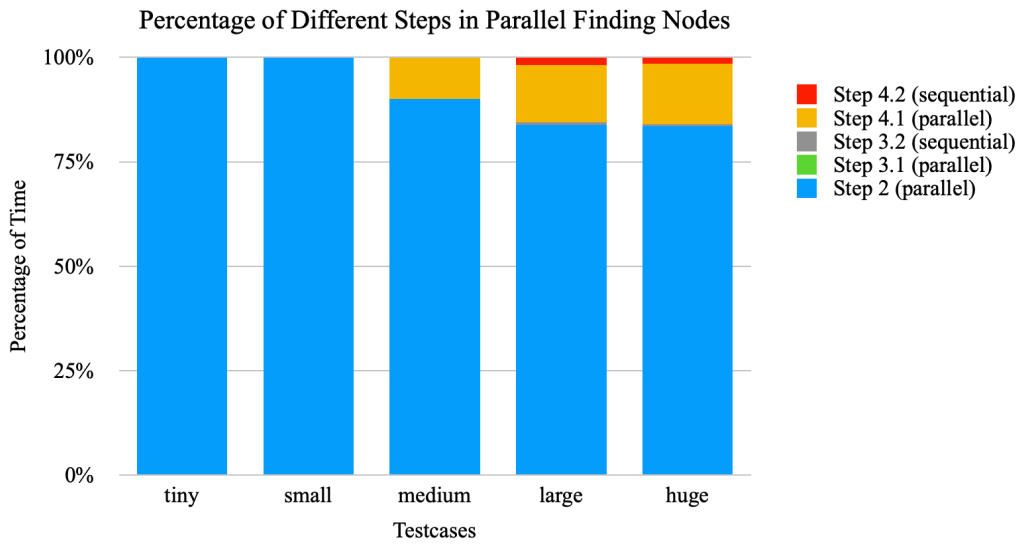
## *Speedup in Phase 1.1 Finding Nodes*

The time comparison between parallel and sequential implementation of find-nodes program is shown below. We use 8 threads, and divide the image into 16 segments for parallel processing.




As we can see, our parallel version of the program achieves super linear speedup! This actually exceeds our expectation, since we have 8 threads in total so we expect it to achieve at most 8 times speedup, but for the huge testcase, the speedup is more than 12 times! We can first conclude that the synchronization overhead is not large, e.g. only a nigligible portion of our program is intrisically sequential, so according to Amdahl's law, our program cannot achieve almost full speedup. Secondly, we can assume that the super-linear speedup is brought by enlarged total cache size from multiple processors, which considerably reduces cache misses.

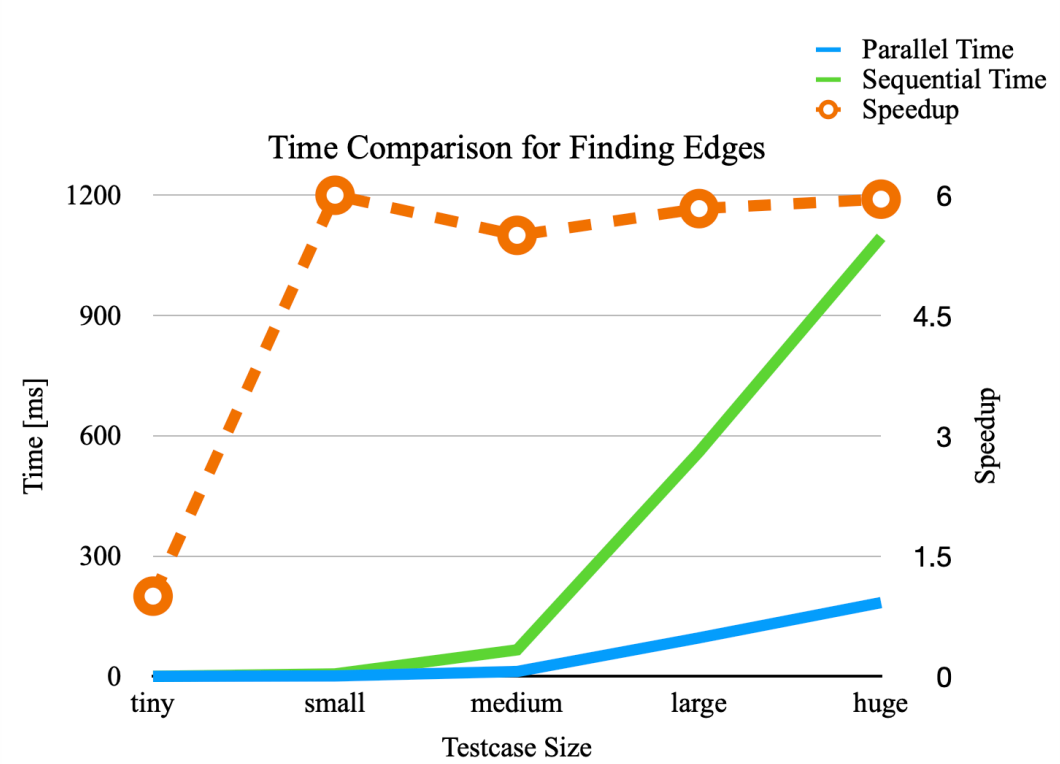
Indeed, after we measured the time breakdown of parallel and sequential phases in this step (as shown in the following figure), we realized that the overhead is actually negligible.




Note that for tiny and small testcases, the overhead still exists but is too small to be measured so on the plotting they are invisible.

## *Speedup in Phase 1.2 Finding Edges*

The time comparison between parallel and sequential implementation of find-edges program is shown below. We use 8 threads, and let each thread be responsible for finding edges of one node.




As we can see, our parallel version of the program almost achieves linear speedup (more than 6x speedup with 8 threads). This is expected since the task has no dependencies on each other so there is no synchronization overhead, and the number of tasks is greatly larger than the nubmer of threads so it can achieve good workload balance.

## *Speedup in Phase 2 Coloring Graph*

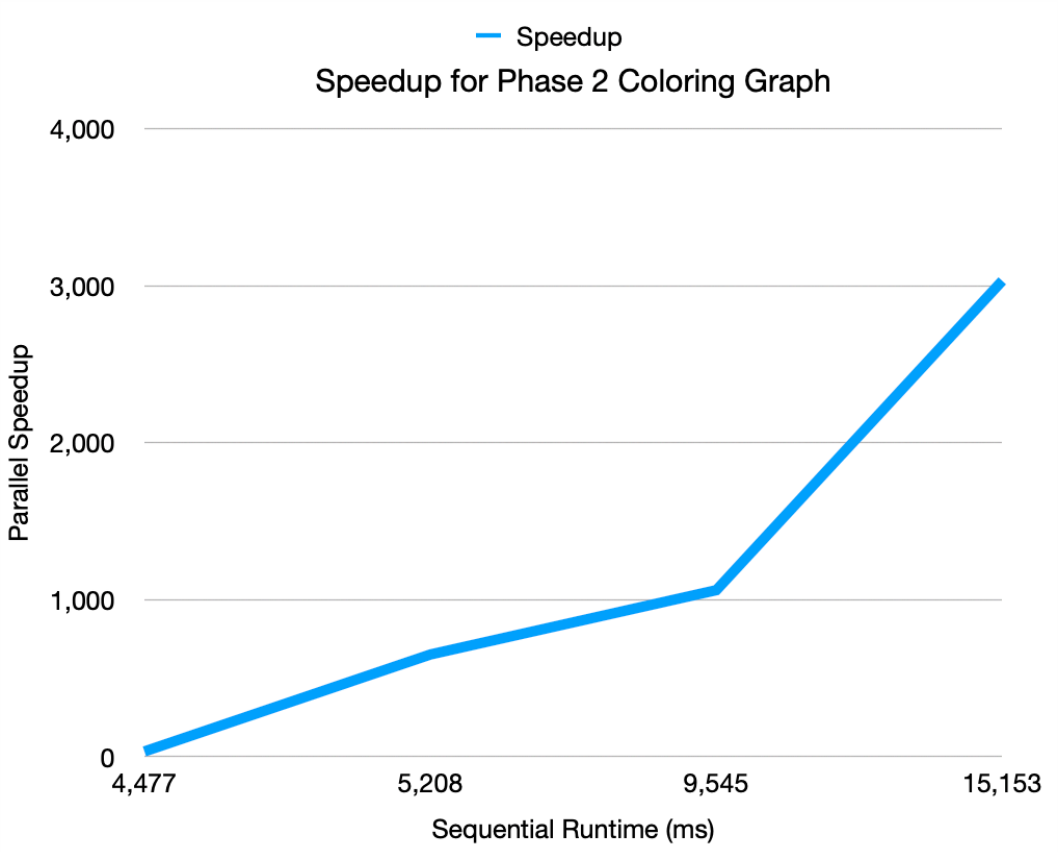
As discussed in the background session, based on our exploration, the testcases that are suitable for phase 2 coloring graph are quite different from those for phase 1. Specifically, the graphs that are time-consuming for coloring should be quite dense and complex. Since this kind of maps are rare to appear in real life, we implement a testcase generator by ourselves which randomly generates graph with nodes and edges that do not cross each other on a plane. Then we write a bash shell to continously generate testcases and try to color them. The shell only saves the testcases that costs the sequential algorithm 5 - 15 seconds to finish. During this process, we find that within reasonable runtime, 5 - 15 seconds, the ideal number of  nodes for such complex testcases is around 40 - 100.

After generating the testcases, we color them using parallel algorithm. However, we find that there exist very high randomness in the runtime. In some cases, parallel version can achieve over 1000 speedup but in other cases, it may be much slower than the sequential verison. Besides, even running the parallel version on the same testcase twice would produce quite different runtime results. The table below shows the runtime results on the four testcases we generate using both sequential and parallel version. The name of testcases are formatted as `nodeNum_edgeNum_sequentialRunTime` and the unit of all the runtime is `ms`. We run the parallel version three times and we can see that there exists high variance in its runtime for different trials. The reason for such

randomness is due to the random nature of task scheduling policy in openMP and huge variance between different search paths in backtracking.

| 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 |

To get more stable data, we run the parallelized version ten times and take its average runtime. We calculate the speedup it acheives compared to the sequential version and the draw the graph with Parallel Speedup vs Sequential Runtime as shown in the following graph. We can see the speedup increases as the sequential runtime increases. The reason may be that since we control the number of nodes as 40 and the number of edges as 100, then longer sequential runtimes implies the graph is more dense and complex. Therefore, there is more room for the parallel version to speedup as the early pruning it provides can help save more time of checking incorrect color assignment.




## Discussion on Different Parallelization Configurations

## *Should We Worry about False-Sharing?*

In step for converting a map image to a graph, we implemented two versions of data structure to store pixels, the first version is a linear vector:

```
int w;
int h;
vector<int> map(w * h);
```

and the second version is a vector of vector:

```
int w;
int h;
int segment_size;
vector<vector<int>> map(
    w * h / segment_size,
    vector<int>(segment_size)
);
```

According to our expectation, the second version should be able to prevent two threads working on neighboring segments from false-sharing the same cache line accross their boundaries. However, we did a bunch of experiments to test our hypothesis and the results (shown in the following table) suggests that the second version does not make improvements. We use 8 threads, and divide the image into 16 segments.

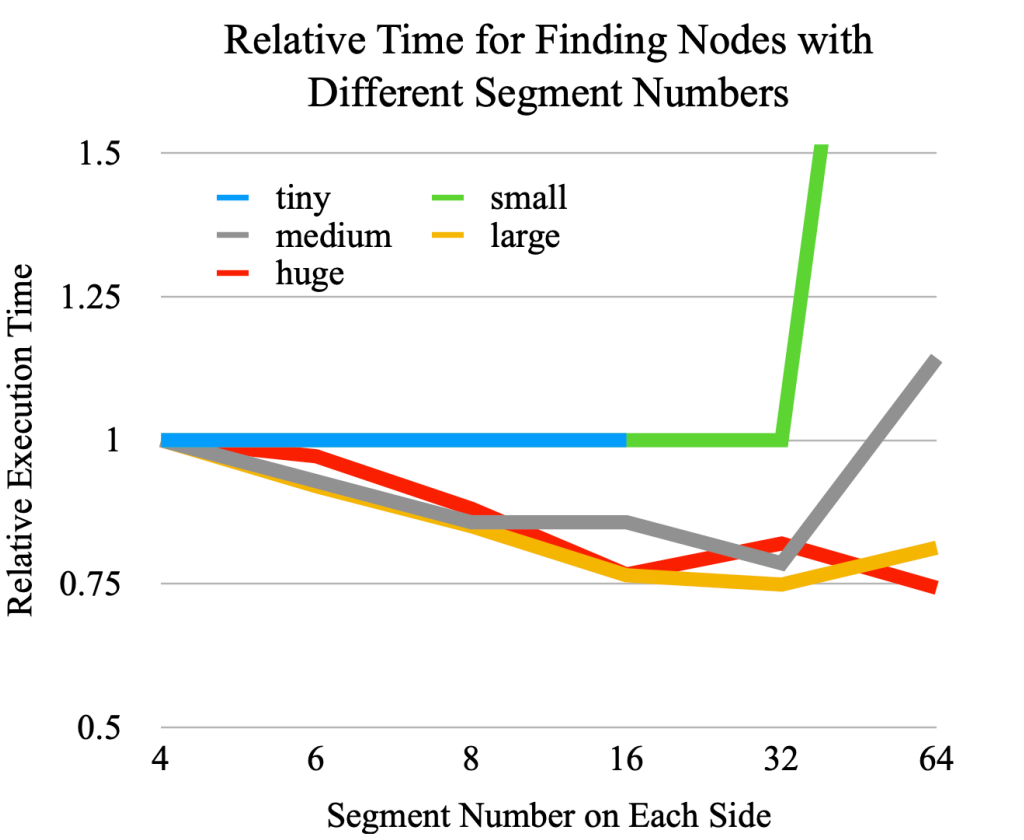
| **Size** | **Time for finding nodes (ms) (false sharing)** | **Time for finding nodes (ms) (without false sharing)** |
|---|---|---|
| tiny | 1 | 1 |
| small | 1 | 1 |
| medium | 14 | 14 |
| large | 185 | 187 |
| huge | 422 | 428 |

We suspect that it is because in the first version, although a cache line could span across two segments, it's actually quite seldom that two threads will be working on the same cache line at the same time. In another word, for most of the time, one thread accesses the cache line exclusively and make its updates while the other thread is working on other different cache lines. And at the time the other thread accesses the shared cache line, the first thread has finished all its updates. So the cache line is not frequently ping-panged between caches. So false-sharing is actually not a problem in our program even if we just use a linear vector to store the pixels.

## *Segment Number*

It matters how many segments we should divide the map image into. So we did a bunch of experiments on different segment numbers and the results are shown in the following plotting. We use 8 threads. Since different testcases have radically different absolute execution time, which are difficult to display on the same graph, so we show relative execution time (relative to seguemnt number equal to 4) in the plotting, namely:

$$\text{ralative\_time}(\text{segment\_number}) = \frac{\text{actual\_time}(\text{segment\_number})}{\text{actual\_time}(4)} \tag{1}$$

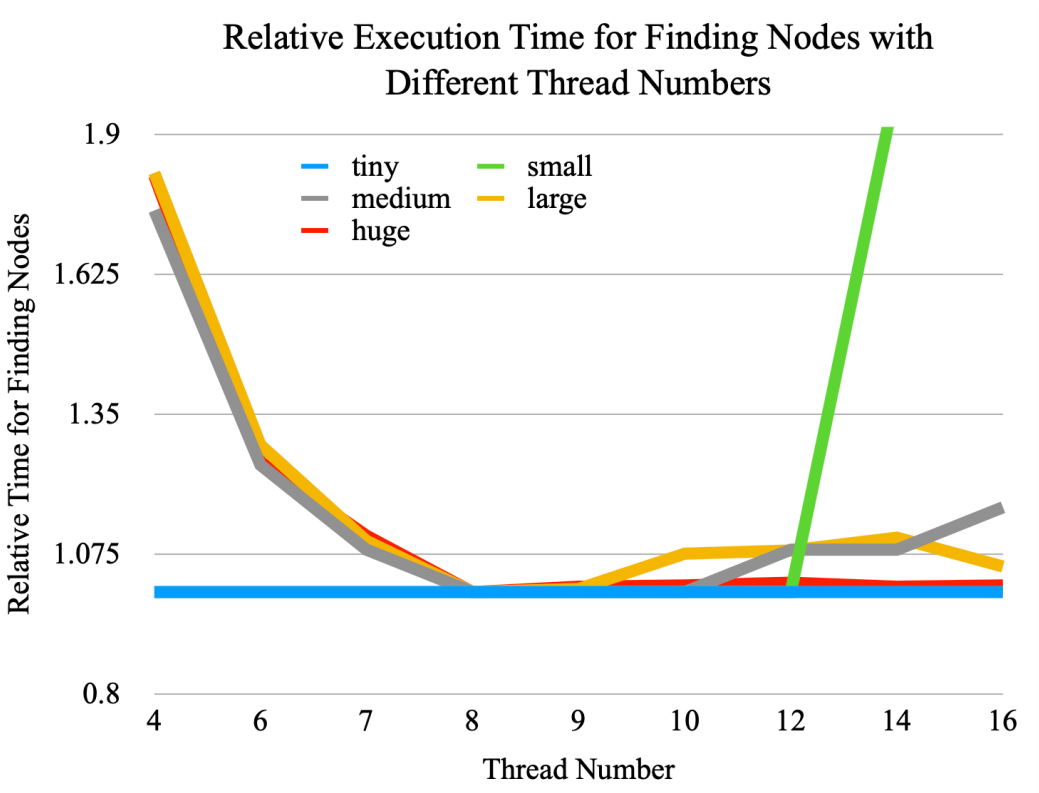


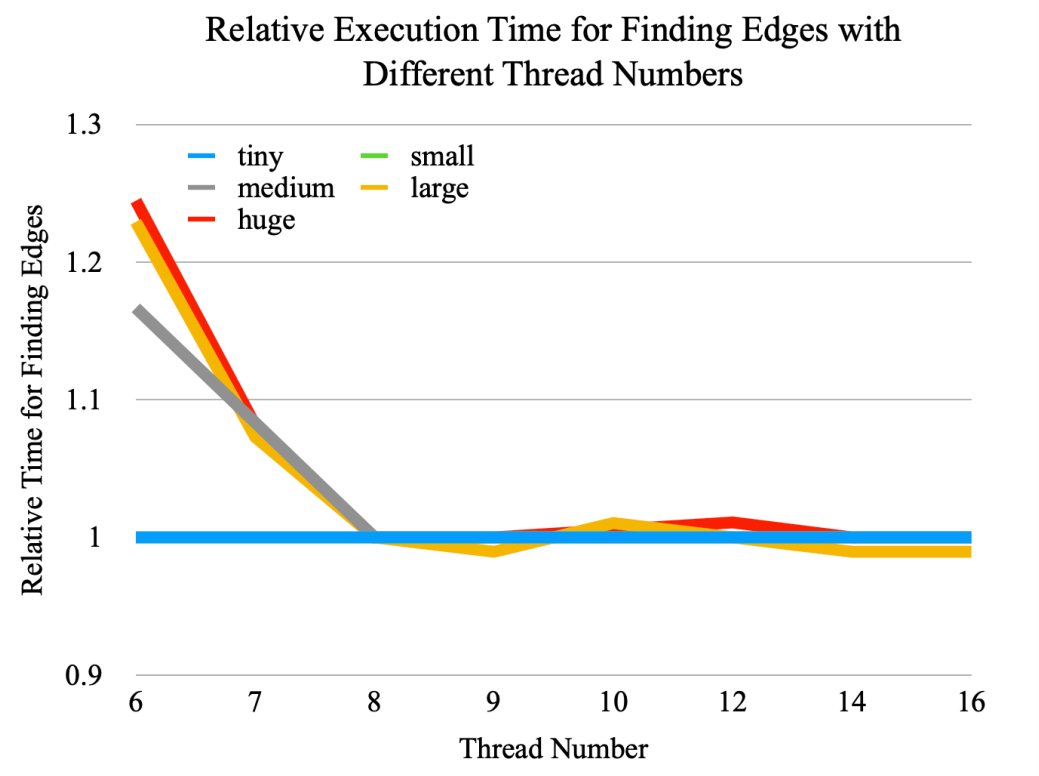

As we can see, for all testcases, setting segment number to 32 will give relatively optimal results for small, medium and large testcases. While huge testcase may need larger segment number to achieve optimal performance, 32 is also relatively good, so we'll set optimal segment number to 32.

## *Thread Number*

It matters how many openMP threads are spawned in the program. So we did a bunch of experiments on different thread numbers and the results for finding nodes and finding edges are shown in the following two plottings, relatively. We divide the image into 16 segments. Again, since different testcases have radically different absolute execution time, which are difficult to display on the same graph, so we show relative execution time (relative to thread number equal to 8) in the plotting, namely:

$$\text{ralative\_time}(\text{thread\_number}) = \frac{\text{actual\_time}(\text{thread\_number})}{\text{actual\_time}(8)} \tag{2}$$



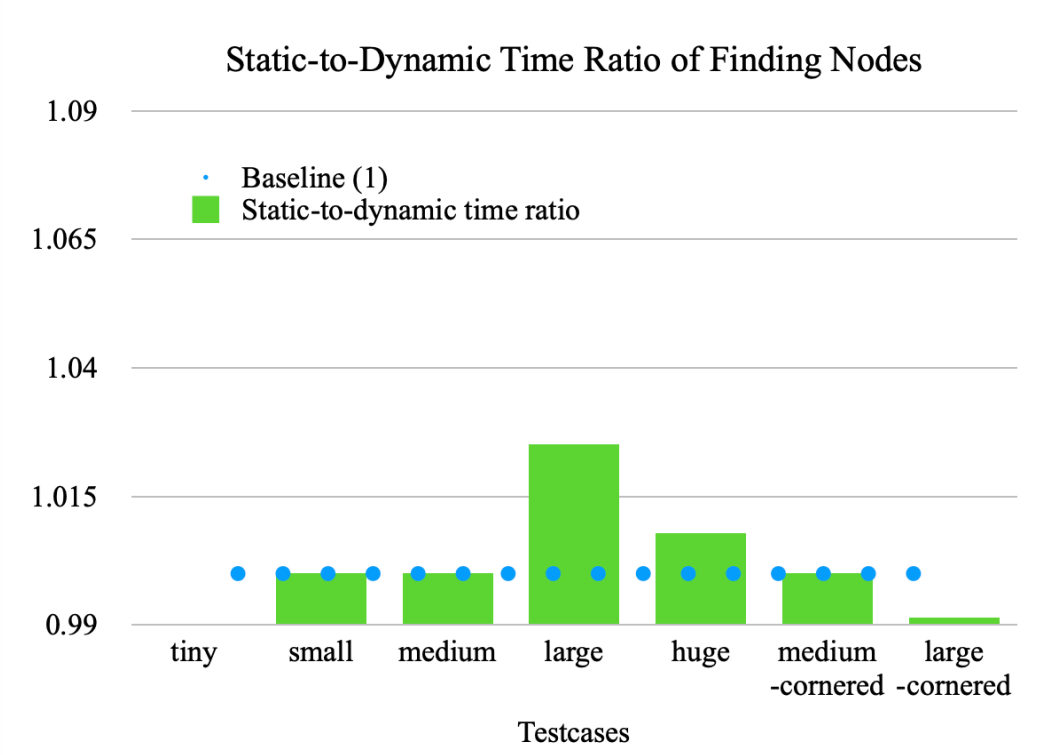



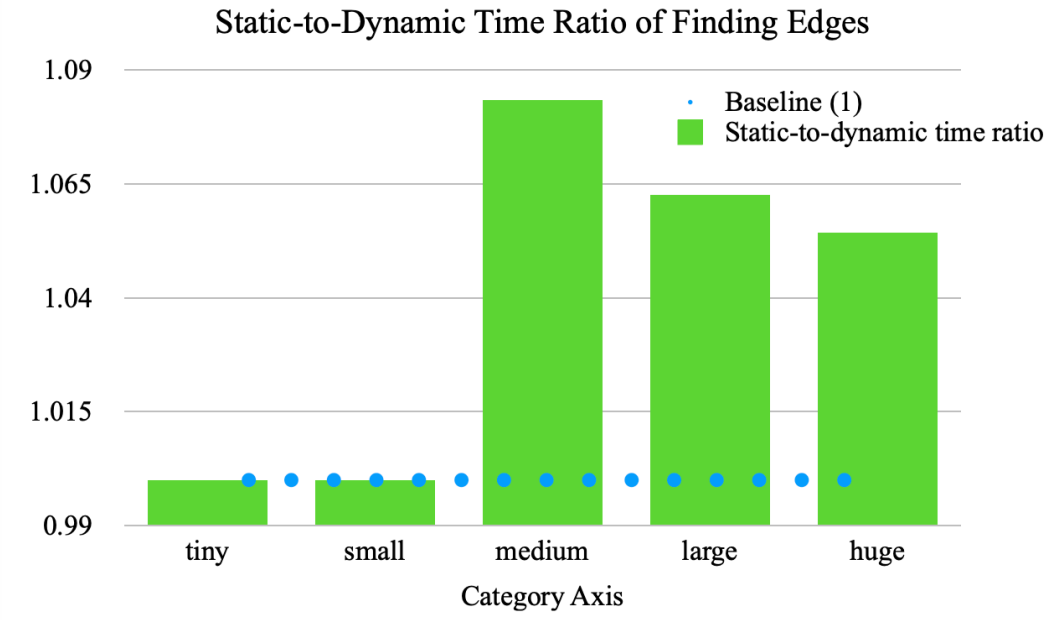

As we can observe, both finding nodes phase and finding edges phase are somewhat affected by thread numbers. Combining two sets of results together, we can see that when thread number is 8, it achieves optimal performance generally.

# *Scheduling Policy*

We tried two scheduling policies in this project -- dynamic vs static. Since dynamic scheduling is suitable for cases where task numbers are small and task workloads are imbalanced; on the contrast, static scheduling is more suitable for cases where task workloads are evenly distributed. Thus we expect that for find-nodes step, dynamic and static scheduling should not have a radical difference since the number of tasks is small (if segment number on each side is 4, then there are only 16 tasks) and the workload in each task is similar (because workloads for finding nodes is mainly determined by pixel number, instead of node number). For find-nodes phase, we included two testcases specially designed to be imbalanced in terms of node distributions.

We did some experiments to compare dynamic and static scheduling on find-nodes phase and find-edges phase, respectively. We use 8 threads and divide image to 16 segments, and we got the following results.



As we can see, for find-nodes phase, the time difference between static and dynamic scheduling is very insignificant.

For find-edges phase, the advantage of dynamic scheduling over staic scheduling is more significant, and this is because the workloads of different tasks is highly imbalanced (larger nodes needs longer time to find its edges). However, the time difference is still not significant, and this is due to the fact that the number of tasks is greatly larger than the number of threads (we have hundreds or thousands of nodes, but only 8 threads), therefore the threads are likely to get assigned a set of tasks with similar distribution of workloads.

### *Framework Choice*

In this application, we choose openMP over CUDA and MPI, because both processing image pixels and analyzing connectivity of graphs can benefit from shared-address communication pattern, so MPI is not efficient considering its heavy data transmission overhead and its limitations on explicit message communication. While CUDA is good at rendering pixels on some applications, the arbitrariness of map shapes makes it difficult to implement vectorized operations. Thus, openMP turns out to be the most suitable option.

## Reference

For the algorithm, we refer to the codebase here but we re-implement everything by ourseleves.

## Work Distribution

**Yuqing Qiu**:

- Sequential and parallel program of map solver;
- sequential program of map-to-graph converter;
- UnionFind data structure class and API;
- Overall pipeline between backend server and frontend web application;
- map solver testcase generation;

**Chenfei Lou**:

- TCP server for data transmission between frontend web application and map solver;
- Frontend web page and P5.js image renderer;
- parallel program for map-to-graph converter;
- convert graph to maps that could be displayed in web page;
- map-to-graph testcase generation;

**Distribution of Total Credits**: 50%-50%