ECE 565 HW5 Report

Sequential:

Data structure:

In this problem we used mainly arrays to store different variables.

```
int timeSteps = stoi(argv[1]);
float absRate = stof(argv[2]);
int N = stoi(argv[3]);
string elevation_file = argv[4];
float runtime;
vector<vector<int>> elevation(N, vector<int>());
vector<vector<float>> absorb(N, vector<float>(N, 0));
struct timespec start_time, end_time;
int wholeSteps = 0; // Store the whole timesteps
```

```
int N = elevation.size();
vector<vector<float>> rain(
    N, vector<float>(N, 0)); // Store the current rain on the ground
vector<vector<float>> trickle(
    N, vector<float>(N, 0)); // Store the trickle of each step
vector<vector<float>> nextTrickle(
    N,
    vector<float>(N, 0)); // Store the trickle result to be added next round
vector<vector<float>> tempTrickle(
    N,
    vector<float>(N, 0)); // Store the trickle result to be added next round
vector<vector<float>> resetTrickle(
    N, vector<float>(N, 0)); // Used for resetting the tempTrickle
vector<vector<vector<vector<int>>>> neighborsToTrickle(
    N, vector<vector<vector<int>>>(N, vector<vector<int>>()));
```

Algorithm:

1. In main function:

First we initialize all the variables(elevation, absorb, wholeSteps...):

```
// Initialization of variables
int timeSteps = stoi(argv[1]);
float absRate = stof(argv[2]);
int N = stoi(argv[3]);
string elevation_file = argv[4];
float runtime;
vector<vector<int>> elevation(N, vector<int>());
vector<vector<float>> absorb(N, vector<float>(N, 0));
```

```
struct timespec start_time, end_time;
int wholeSteps = 0; // Store the whole timesteps

// open the file to read.
fstream in("./" + elevation_file);
string line;
int ele; // Store elevation for each input unit
if (in)  // file exists
{
  int row = 0;
  while (getline(in, line)) // line does not contain newline of each line
  {
    stringstream ss(line);
    while (ss >> ele) {
      elevation[row].push_back(ele);
    }
    ++row;
  }
} else // No such file
{
  cout << "No such elevation_file" << endl;
  return EXIT_FAILURE;
}
```

Then we start calculating time and the steps needed to fully absorb all the rain:

```
wholeSteps =
    calcRain(elevation, absorb, timeSteps, absRate, start_time, end_time);

float elapsed_ns = calc_time(start_time, end_time);
cout << "Rainfall simulation took " << wholeSteps
     << " time steps to complete." << endl;
cout << "Runtime = " << elapsed_ns / 1000000000 << " seconds" << endl;
cout << endl;
cout << "The following grid shows the number of raindrops absorbed at each "
        "point:"
     << endl;
```

2. calcRain function:

In each iteration, we call "**rainAbsorbTrickle**" to receive new raindrops, absorb and trickle the water until the ground is fully dry, then return the wholesteps.

```
float isDrain = 1;
```

```
clock_gettime(CLOCK_MONOTONIC, &start_time);

while (isDrain != 0) {
  isDrain = 0;

  // 1) Receive a new raindrop (if it is still raining) for each point.
  // 2) If there are raindrops on a point, absorb water into the point
  // 3a) Calculate the number of raindrops that will next trickle to the
  // lowest neighbor(s)
  rainAbsorbTrickle(rain, absorb, trickle, timeSteps, absRate, isDrain,
                    nextTrickle, elevation, neighborsToTrickle, tempTrickle);
  nextTrickle = tempTrickle;
  tempTrickle = resetTrickle;
  --timeSteps;
  ++wholeSteps;
}

clock_gettime(CLOCK_MONOTONIC, &end_time);

return wholeSteps;
```

Here some some variables we pass into “**rainAbsorbTrickle**”:

```
int N = elevation.size();
vector<vector<float>> rain(
    N, vector<float>(N, 0)); // Store the current rain on the ground
vector<vector<float>> trickle(
    N, vector<float>(N, 0)); // Store the trickle of each step
vector<vector<float>> nextTrickle(
    N,
    vector<float>(N, 0)); // Store the trickle result to be added next round
vector<vector<float>> tempTrickle(
    N,
    vector<float>(N, 0)); // Store the trickle result to be added next round
vector<vector<float>> resetTrickle(
    N, vector<float>(N, 0)); // Used for resetting the tempTrickle
vector<vector<vector<vector<int>>>> neighborsToTrickle(
    N, vector<vector<vector<int>>>(N, vector<vector<int>>()));
for (int i = 0; i < N; ++i) {
  for (int j = 0; j < N; ++j) {
    vector<vector<int>> neighToTrickle = countNeighbor(i, j, elevation);
    neighborsToTrickle[i][j] = neighToTrickle;
  }
```

```
}
```

“**countNeighbor**” is used to sort and determine which neighbor(s) to trickle:

```
vector<vector<int>> countNeighbor(int i, int j,
                                  const vector<vector<int>> &elevation) {
 int N = elevation.size();
 vector<vector<int>> neighToTrickle; // <elevation, number>, number: Left: 0,
                                     // up: 1,right: 2, down:3
 vector<vector<int>> allNeigh;       // All neighbors that have less elevation
 if (j - 1 >= 0 && elevation[i][j - 1] < elevation[i][j]) { // left
   allNeigh.push_back({elevation[i][j - 1], 0});
 }
 if (i - 1 >= 0 && elevation[i - 1][j] < elevation[i][j]) { // up
   allNeigh.push_back({elevation[i - 1][j], 1});
 }
 if (j + 1 < N && elevation[i][j + 1] < elevation[i][j]) { // right
   allNeigh.push_back({elevation[i][j + 1], 2});
 }
 if (i + 1 < N && elevation[i + 1][j] < elevation[i][j]) { // down
   allNeigh.push_back({elevation[i + 1][j], 3});
 }
```

3. rainAbsorbTrickle function:

- 3.1 first we add the trickle calculated from last iteration, then add the new raindrop to absorb:

```
    // Add trickle from the previous step
    rain[i][j] += nextTrickle[i][j];
    // Reset the nextTrickle array
    nextTrickle[i][j] = 0;
    if (timeSteps > 0) {
      // 1) Receive a new raindrop (if it is still raining) for each point.
      ++rain[i][j];
    }
    if (rain[i][j] == 0) {
      continue;
    }
    // 2) If there are raindrops on a point, absorb water into the point
    if (rain[i][j] >= absRate) {
      rain[i][j] -= absRate;
      absorb[i][j] += absRate;
    } else if (rain[i][j] > 0) {
      absorb[i][j] += rain[i][j];
      rain[i][j] = 0;
```

```
      continue;
    }
```

- 3.2 Then we calculate the amount of rain that will trickle to the near lowest neighbor(s).

```
    // 3a) Calculate the number of raindrops that will next trickle to the
    // lowest neighbor(s)
    trickle[i][j] = 0; // Reset the trickle array
    if (rain[i][j] >= 1) {
      trickle[i][j] = 1;
    } else if (rain[i][j] > 0) {
      trickle[i][j] = rain[i][j];
    }
    isDrain += trickle[i][j];
```

(In here, we use **isDrain** to help check if the all the ground is dry)

- 3.3 For each point, we use the calculated trickle amount to update the actual raindrop that will trickle to the neighbor(s), if applicable.

```
    calcTrickle(i, j, rain, elevation, trickle, neighborsToTrickle,
                tempTrickle);
```

4. calcTrickle function

In this function, we trickle the rain on each point based on the evaluation of their neighbors, and store the results in array **tempTrickle**, this will be used to update the **nextTrickle** array, which is used at the beginning of **rainAbsorbTrickle** to absorb the trickled water.

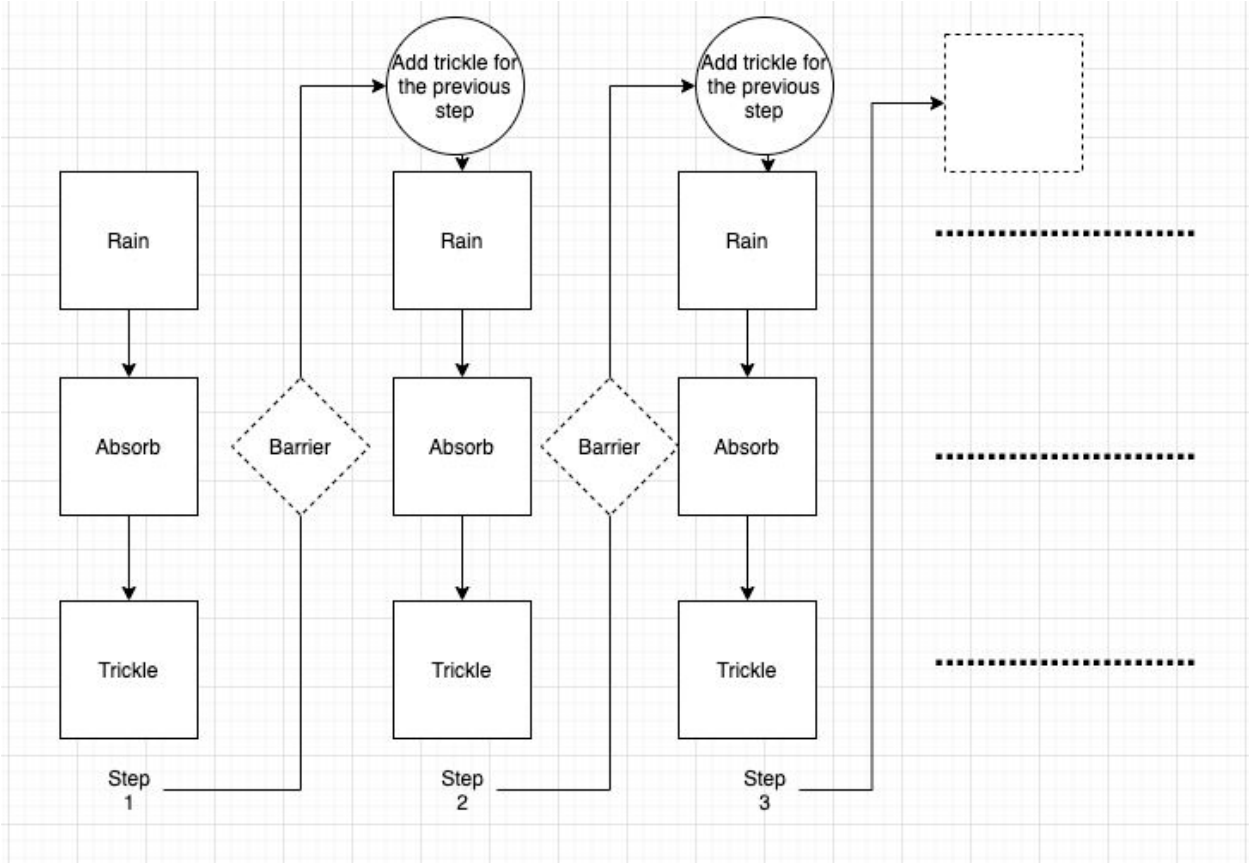
```
void calcTrickle(int i, int j, vector<vector<float>> &rain,
                 const vector<vector<int>> &elevation,
                 vector<vector<float>> &trickle,
                 vector<vector<vector<vector<int>>>> &neighborsToTrickle,
                 vector<vector<float>> &tempTrickle) {
 if (trickle[i][j] == 0 || neighborsToTrickle[i][j].size() == 0) {
   return;
 }
 vector<vector<int>> direct = {
     {0, -1}, {-1, 0}, {0, 1}, {1, 0}}; // Left, up, right, down

 float share = trickle[i][j] / neighborsToTrickle[i][j].size();
 rain[i][j] -= trickle[i][j];
 for (auto n : neighborsToTrickle[i][j]) {
   tempTrickle[i + direct[n[1]][0]][j + direct[n[1]][1]] += share;
```

Parallel:

In the parallel version, we used the pre-created threads from the thread pool. And used std::mutex to lock the variable.

We choose to parallel the whole process of rain, absorb and trickle part in each step because that's where most of the calculation and loop happens.



future<void> results is used to join all the threads(works like a barrier)

```
clock_gettime(CLOCK_MONOTONIC, &start_time);

while (notDrain) {
  notDrain = false;
  future<void> results[numThreads]; // To join all the threads

  // 1) Receive a new raindrop (if it is still raining) for each point.
  // 2) If there are raindrops on a point, absorb water into the point
  // 3a) Calculate the number of raindrops that will next trickle to the
  // lowest neighbor(s)
  // cout << "---------------------------------" << wholeSteps << endl;
  for (int i = 0; i < numThreads; i++) {
    results[i] = p.push(rainAbsorbTrickle, ID++);
```

```
  }
  for (int i = 0; i < numThreads; i++) {
    results[i].wait(); // synchronize all threads
  }
  // cout << "---------------------------------" << endl;
  ID = 0;
  nextTrickle = tempTrickle;
  tempTrickle = resetTrickle;
  --timeSteps;
  ++wholeSteps;
}

clock_gettime(CLOCK_MONOTONIC, &end_time);
```

To prevent each thread from operating on the same point when raining and absorbing, we assign each thread with one unique block of the array based on their thread_id:

```
void rainAbsorbTrickle(int id, int threadId) {
 for (int i = threadId * N / numThreads;
      i < threadId * N / numThreads + int(ceil(float(N) / float(numThreads)));
      ++i) {
   for (int j = (threadId * N * N / numThreads) % N;
        j < N - ((threadId * N * N / numThreads) % N + N * N / numThreads) % N;
        ++j) {
...
```

Since different threads operate on different blocks while raining and absorbing, there is no race condition.

The only part we need to take care of is the trickle part, because even though threads are operating on different blocks, race conditions can still happen if they write to the same neighbors when calculating the trickle, so we need to lock the **tempTrickle** array, but locking the whole array will slow down the speed greatly. After some consideration, we realize since only when the threads operate on the edge of the block can they write to the same neighbor, we only need to lock the edge of the block.

For example, if there are total 16 * 16 points and there are 4 threads, we will assign 64 consecutive points for each thread, shown as the following

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
1	**Thread 1**															
2																
3	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock
4	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock
5	**Thread 2**															
6																
7	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock
8	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock
9																
10	**Thread 3**															
11	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock
12	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock	lock
13																
14	**Thread 4**															
15																

We only need to lock the edge part which is shown in the above picture with “lock” on the point

```
void calcTrickle(int i, int j, int threadId) {
 if (trickle[i][j] == 0 || neighborsToTrickle[i][j].size() == 0) {
   return;
 }
 vector<vector<int>> direct = {
     {0, -1}, {-1, 0}, {0, 1}, {1, 0}}; // Left, up, right, down

 float share = trickle[i][j] / neighborsToTrickle[i][j].size();
 rain[i][j] -= trickle[i][j];
 for (auto n : neighborsToTrickle[i][j]) {
   if (i == threadId * N / numThreads ||
       i == threadId * N / numThreads +
                int(ceil(float(N) / float(numThreads))) - 1) {
     mtx.lock(); // Only lock on edge
   }
   tempTrickle[i + direct[n[1]][0]][j + direct[n[1]][1]] += share;
```

```
   if (i == threadId * N / numThreads ||
       i == threadId * N / numThreads +
                int(ceil(float(N) / float(numThreads))) - 1) {
     mtx.unlock(); // Only lock on edge
   }
 }
}
```

Performance:

Sequential:

my_4096x4096.out

```
Rainfall simulation took 1040 time steps to complete.
Runtime = 259.656 seconds

The following grid shows the number of raindrops absorbed at each point:
      25      25      25    25.5      25      25    33.5     191      25     202      25    25.5    99.5
     100    25.5     224      50      25      25      25    25.5    25.5      64      25      25      25
      25      25      25      25    25.5   124.5      25      25      25    33.5      25    25.5      25
      25      50   199.5      25      25      25     100      25      50      25    25.5     199      25
      25    25.5      25      25      25     150      25    25.5      25      25      25      25   124.5
      25    33.5     216    25.5      50      25      25    99.5      25      42      25      25    25.5
      25      25      25      25      25      25  149.75      25      25   232.5      25    33.5      25
```

Parallel:

1 thread:

my_4096_1thread.out

```
Rainfall simulation took 1040 time steps to complete.
Runtime = 445.862 seconds

The following grid shows the number of raindrops absorbed at each point:
      25      25      25    25.5      25      25    33.5     191      25     202      25    25.5    99.5      25
     100    25.5     224      50      25      25      25    25.5    25.5      64      25      25      25     100
      25      25      25      25    25.5   124.5      25      25      25    33.5      25    25.5      25      25
      25      50   199.5      25      25      25     100      25      50      25    25.5     199      25      25
      25    25.5      25      25      25     150      25    25.5      25      25      25      25   124.5      25
      25    33.5     216    25.5      50      25      25    99.5      25      42      25      25    25.5      25
      25      25      25      25      25      25  149.75      25      25   232.5      25    33.5      25      25    3
```

2 threads:

my_4096_2threads.out

```
Rainfall simulation took 1040 time steps to complete.
Runtime = 276.395 seconds

The following grid shows the number of raindrops absorbed at each point:
      25      25      25    25.5      25      25    33.5     191      25     202      25    25.5    99.5      25     125      25      25      25     100
     100    25.5     224      50      25      25      25    25.5    25.5      64      25      25      25     100      25      25    33.5      25    33.5
      25      25      25      25    25.5   124.5      25      25      25    33.5      25    25.5      25      25      25      25   116.5      25     233
      25      50   199.5      25      25      25     100      25      50      25    25.5     199      25      25      42     208      25      25    33.5
      25    25.5      25      25      25     150      25    25.5      25      25      25      25   124.5      25      25      25      50      25      25
      25    33.5     216    25.5      50      25      25    99.5      25      42      25      25    25.5      25      25      25      25   191.5      25
      25      25      25      25      25      25  149.75      25      25   232.5      25    33.5      25      25    33.5      75      25    33.5      25
```

4 threads:

my_4096_4threads.out

```
Rainfall simulation took 1040 time steps to complete.
Runtime = 189.914 seconds

The following grid shows the number of raindrops absorbed at each point:
      25      25      25    25.5      25      25    33.5     191      25     202      25    25.5    99.5      25     125      25      25      25     100
     100    25.5     224      50      25      25      25    25.5    25.5      64      25      25      25     100      25      25    33.5      25    33.5
      25      25      25      25    25.5   124.5      25      25      25    33.5      25    25.5      25      25      25      25   116.5      25     233
      25      50   199.5      25      25      25     100      25      50      25    25.5     199      25      25      42     208      25      25    33.5
      25    25.5      25      25      25     150      25    25.5      25      25      25      25   124.5      25      25      25      50      25      25
      25    33.5     216    25.5      50      25      25    99.5      25      42      25      25    25.5      25      25      25      25   191.5      25
      25      25      25      25      25      25  149.75      25      25   232.5      25    33.5      25      25    33.5      75      25    33.5      25
     100      25      25     208      42      25    25.5      25      25    25.5      25     154      25      25   141.5      25      25      25      25
```

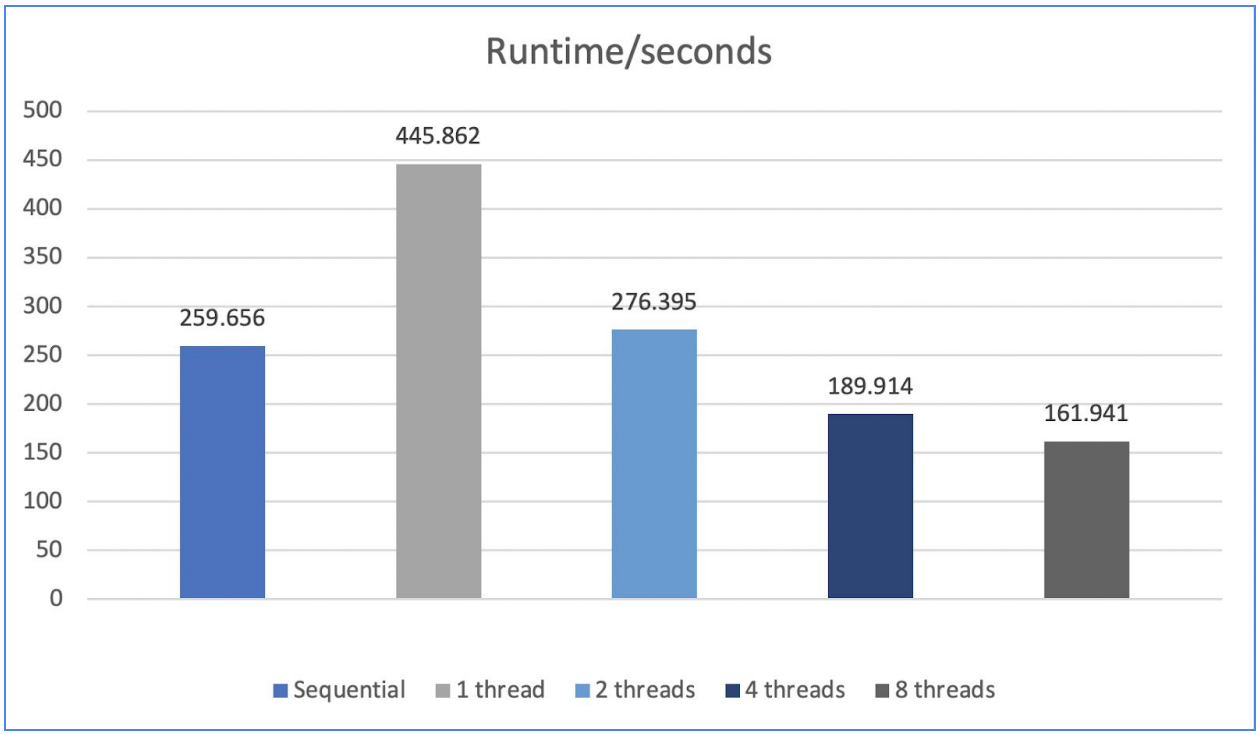
8 threads:

my_4096_8threads.out

```
Rainfall simulation took 1040 time steps to complete.
Runtime = 161.941 seconds

The following grid shows the number of raindrops absorbed at each point:
      25     25     25   25.5     25     25   33.5    191     25    202     25   25.5   99.5     25    125     25     25     25    100
     100   25.5    224     50     25     25     25   25.5   25.5     64     25     25     25    100     25     25   33.5     25   33.5
      25     25     25     25   25.5  124.5     25     25     25   33.5     25   25.5     25     25     25     25  116.5     25    233
      25     50  199.5     25     25     25    100     25     50     25   25.5    199     25     25     42    208     25     25   33.5
      25   25.5     25     25     25    150     25   25.5     25     25     25     25  124.5     25     25     25     50     25     25
      25   33.5    216   25.5     50     25     25   99.5     25     42     25     25   25.5     25     25     25     25  191.5     25
      25     25     25     25     25     25  149.75     25     25  232.5     25   33.5     25     25   33.5     75     25   33.5     25
     100     25     25    208     42     25   25.5     25     25   25.5     25    154     25     25  141.5     25     25     25     25
      25     25     50     25     25     25  149.25   25.5     25     25   29.5     25     25    150     25     25  224.5     50     25
     183     42     25     25   25.5  190.5     25     25     25     25  207.5   25.5     25     25   25.5     25     25   25.5     25
```

We can see that for 4 threads and 8 threads, the performance gets improved greatly. But for 2 threads and 1 thread, the performance actually gets worse.



At first, it doesn't match our expectation because performance of 1 thread is much worse and 2 threads performance is barely the same as sequential.

Then after some research, it starts to make sense to us. We think it's because of the overhead of threadpool. We can see that even when we only pre-create 1 thread in the thread pool, we still need to push the function using this single thread each step, make this single thread deal with the whole work, which will cause a lot of overhead and greatly slow down the speed. Mainly because the thread needs to reset its variables each step.

```
for (int i = 0; i < numThreads; i++) {
  results[i] = p.push(rainAbsorbTrickle, ID++);
}
```

```
  for (int i = 0; i < numThreads; i++) {
    results[i].wait(); // synchronize all threads
  }
```

We also used mutex in the array, the lock & unlock operation will have overhead as well. And other threads might need to wait for the current thread to finish the operation.

```
for (auto n : neighborsToTrickle[i][j]) {
  if (i == threadId * N / numThreads ||
      i == threadId * N / numThreads +
               int(ceil(float(N) / float(numThreads))) - 1) {
    mtx.lock(); // Only lock on edge
  }
  tempTrickle[i + direct[n[1]][0]][j + direct[n[1]][1]] += share;
  if (i == threadId * N / numThreads ||
      i == threadId * N / numThreads +
               int(ceil(float(N) / float(numThreads))) - 1) {
    mtx.unlock(); // Only lock on edge
  }
}
```

Only when we have more threads(4,8..), the parallelization is strong enough to overcome these overhead and show improvement in the performance.