ECE 565 HW5

Sequential Part

The sequential algorithm starts with initializing elevation information. I used a 2-D matrix to read from input file, then store each cell's elevation. After that, I create a 2-D matrix, where each cell contains an array of coordinates pairs. Those coordinates pairs represent all neighbors of that cell where rain drop will trickle.

```
void initElevations(vector<vector<int>> & elevations, const char * filePath) {
    ifstream infile(filePath);
    string line;
    while(getline(infile, line)) {
        vector<int> vec;
        stringstream ss(line);
        string s = "";
        while(ss >> s) {
            vec.push_back(stoi(s));
        }
        elevations.push_back(vec);
    }
}

// each cell of lowests stores coordinates of neighbors that have same lowest
elevation
void initLowests(vector<vector<vector<pair<int, int>>>> & lowests,
vector<vector<int>> & elevations, const int N) {
    int dr[4] = {-1, 1, 0, 0};
    int dc[4] = {0, 0, -1, 1};
    for(int r = 0; r < N; ++r) {
        for(int c = 0; c < N; ++c) {
            vector<pair<int, pair<int, int>>> vec;
            for(int k = 0; k < 4; ++k) {
                int nextR = r + dr[k];
                int nextC = c + dc[k];
                if(nextR >= 0 && nextR < N && nextC >= 0 && nextC < N) {
                    vec.push_back(make_pair(elevations[nextR][nextC],
make_pair(nextR, nextC)));
                }
            }
            sort(vec.begin(), vec.end(), [](const pair<int, pair<int, int>> p1,
const pair<int, pair<int, int>> p2) -> bool {
                return p1.first < p2.first;
```

```
            });
            if(vec[0].first >= elevations[r][c]) {continue;}    // current cell's
elevation is the lowest among its neighbors
            int minVal = vec[0].first;
            for(size_t m = 0; m < vec.size(); ++m) {
                if(vec[m].first > minVal) {break;}
                lowests[r][c].push_back(make_pair(vec[m].second.first,
vec[m].second.second));
            }
        }
    }
}
```

After the initialization, I create three 2-D matrix to help simulation process. As below code snippet shows, the absorbed matrix records how much rain each cell has absorbed, which will be print at the end of program. States matrix represents how much rain drops each cell has at every time stamp. Trickled matrix records how much rain drops will be trickled into each cell at the end of every time stamp.

```
vector<vector<float>> absorbed(N, vector<float>(N, 0.0));
vector<vector<float>> states(N, vector<float>(N, 0.0));
vector<vector<float>> trickled(N, vector<float>(N, 0.0));
```

The actual simulation process are put inside an infinite while loop. At each time stamp, I will traverse through matrix to receive new rain drop, absorb certain amount of rain into ground and calculate the amount of rain that would be trickled to neighbor cells. After the trickled matrix is updated, I would traverse it to update each cell's state.

At the end of each iteration of the while loop, I will check if the cells of states matrix are all empty, if so, break out the loop, otherwise, keep looping.

```
while(true) {
        for(int i = 0; i < N; ++i) {
            for(int j = 0; j < N; ++j) {
                // Receive a new raindrop (if it is still raining) for each point
                if(timeStep <= timeOfRain) {
                    states[i][j] += 1.0;
                }
                // If there are raindrops on a point, absorb water into the point
                if(states[i][j] > 0.0) {
                    float absorbedVal = states[i][j] > absorbRate ? absorbRate :
states[i][j];
                    states[i][j] -= absorbedVal;
                    absorbed[i][j] += absorbedVal;
                }
```

```
                // Calculate the number of raindrops that will next trickle to
the lowest neighbor(s)
                if(lowests[i][j].size() > 0 && states[i][j] > 0.0) {
                    float trickledValTotal = states[i][j] >= 1.0 ? 1.0 :
states[i][j];
                    states[i][j] -= trickledValTotal;
                    float trickledVal = trickledValTotal / lowests[i][j].size();
                    for(size_t k = 0; k < lowests[i][j].size(); ++k) {
                        int neiR = lowests[i][j][k].first;
                        int neiC = lowests[i][j][k].second;
                        trickled[neiR][neiC] += trickledVal;
                    }
                }
            }
        }

        // update the number of raindrops at each lowest neighbor
        for(int i = 0; i < N; ++i) {
            for(int j = 0; j < N; ++j) {
               states[i][j] += trickled[i][j];
               trickled[i][j] = 0.0;
            }
        }

        // check if states of all points are zero, if so, break;
        if(isZero(states, N)) {
            break;
        }
        ++timeStep;
    }
```

I use omp_get_wtime() to calculate time consumed in simulation.

```
const double beginTime = omp_get_wtime();
// ... while loop
const double endTime = omp_get_wtime();
```

Parallel Part

Basically there are four parts in each time stamp. I parallelize the process of receiving rain drops, absorption, trickled amount calculation and updating states from trickled matrix. Each threads will be responsible for **N / ThreadNum** rows. The reason why I chose to parallelize this part of code is that except for trickled amount calculation, all other three parts are independent and no synchronization needed, because each thread is in charge of different group of cells.

There are two three types fo synchronization I used on parallel codes, lock, barrier and atomic operation. Below code snippet shows the usage of lock when calculating trickled matrix values. In order to get good performance, I created a matrix of mutex, so the lock will only be applied to each cell. If two threads are accessing different cells, they will not be blocked.

```
for(int i = id * threadN; i < (id+1) * threadN; ++i) {
  for(int j = 0; j < N; ++j) {
    // ... receive new rain drops
    // ... absorb rain drops

    if(lowests[i][j].size() > 0 && states[i][j] > 0.0) {
      float trickledValTotal = states[i][j] >= 1.0 ? 1.0 : states[i][j];
      states[i][j] -= trickledValTotal;
      float trickledVal = trickledValTotal / lowests[i][j].size();

      for(size_t k = 0; k < lowests[i][j].size(); ++k) {
        int neiR = lowests[i][j][k].first;
        int neiC = lowests[i][j][k].second;
        // lock the cell I used
        pthread_mutex_lock(&mtxes[neiR][neiC]);
        trickled[neiR][neiC] += trickledVal;
        pthread_mutex_unlock(&mtxes[neiR][neiC]);
      }
    }
  }
}
```

After the above process, I used barrier to synchronize each thread's progress. Updates to the state of cell should only be made after the calculation of trickled amount is finished. I also made a small modification here, compared to sequential version. I removed the explicit check of whether each cell has zero rain drop. Those checks are made after update to each cell's state, inside the same loop. So there is no time wasted on another for loop.

```
pthread_barrier_wait(&barrier1);
// update the number of raindrops at each lowest neighbor
for(int i = id * threadN; i < (id+1) * threadN; ++i) {
  for(int j = 0; j < N; ++j) {
    states[i][j] += trickled[i][j];
    trickled[i][j] = 0.0;
    if(abs(states[i][j]) > FLT_EPSILON) {
      isFinished = false;
    }
  }
}
```

If any cell's value is not zero, then the flag **isFinished** inside each thread will be marked false. Then each thread will update the value of global flag **globalFinished** by AND operation. If the **globalFinished** is true at the end of time stamp, then threads quits, otherwise, keep looping.

The atomic operation is used on global variable **roundNum**, which recored final result of how many time stamps are used. When the **globalFinished** flag is true, each thread will store their time stamp value to **roundNum**, since those values are same, the final result is correct.

```
pthread_mutex_lock(&globalFinishedMtx);
globalFinished = globalFinished && isFinished;
pthread_mutex_unlock(&globalFinishedMtx);
pthread_barrier_wait(&barrier2);
if(globalFinished) {
  roundNum.store(timeStep, memory_order_relaxed);
  break;
}
```

Test & Measurement

	Sequential	Thread 1	Thread 2	Thread 4	Thread 8
Runtime (second)	651.42	786.43	396.56	202.56	105.49

The speedup matches expectation. When executing parallel version with only one thread, the overhead induced by thread creation and destruction makes the result slower than sequential version. However, when we increased the number of threads to be 2, 4 and 8, runtime of parallel version codes are faster than sequential version.

Also, according to the table, runtime of parallel version is almost linear regrading to thread number, which means there are not much synchronization overhead for parallel execution.