ECE 565 HW5

Yichuan Shi

ys205

Sequential Algorithm:

1. initialization
    a. initialize the Gird
    b. initialize the elevation
    c. initialize the trickle (pre-compute the trickle fraction for every Node in the Grid).

```
Grid<Node> grid(N + 2);
initializeGridElevation(grid, argv[4], N);
initializeGridTrickle(grid, N);
```

```
void initializeGridElevation(Grid<Node> &grid, const string& file_path, const int& N) {
  ifstream inputFIle(file_path);
  if (inputFIle.fail()) {
    cout << "fail doesn't exist" << endl;
    exit(EXIT_FAILURE);
  }
  string line;
  for (size_t i = 1; i < N + 1; i++) {
    getline(inputFIle, line);
    stringstream line_stream(line);
    for (size_t j = 1; j < N + 1; j++) {
      line_stream >> grid[i][j].elevation;
    }
  }
}
```

```cpp
void initializeGridTrickle(Grid<Node> &grid, const int& N) {
  for (int i = 1; i < N + 1; ++i) {
    for (int j = 1; j < N + 1; ++j) {
      initializeNodeTrickle(grid, i, j);
    }
  }
}
void initializeNodeTrickle(Grid<Node> &grid, const int& i, const int& j) {
  vector<int> neighbors_elevation(0);
  neighbors_elevation.push_back(grid[i - 1][j].elevation);
  neighbors_elevation.push_back(grid[i + 1][j].elevation);
  neighbors_elevation.push_back(grid[i][j - 1].elevation);
  neighbors_elevation.push_back(grid[i][j + 1].elevation);
  int lowest_elevation =
          *min_element(neighbors_elevation.begin(), neighbors_elevation.end());
  if (lowest_elevation >= grid[i][j].elevation) {
    grid[i][j].willTrickle = false;
    return;
  }
  grid[i][j].willTrickle = true;
  int num_of_lowest_elevation = 0;
  for (int neighbor_elevation : neighbors_elevation) {
    if (neighbor_elevation == lowest_elevation) {
      num_of_lowest_elevation++;
    }
  }
  grid[i][j].trickleNumber = num_of_lowest_elevation;
  if (lowest_elevation == grid[i - 1][j].elevation) {
    grid[i][j].topTrickle = true;
  }
  if (lowest_elevation == grid[i + 1][j].elevation) {
    grid[i][j].bottomTrickle = true;
  }
  if (lowest_elevation == grid[i][j - 1].elevation) {
    grid[i][j].leftTrickle = true;
  }
  if (lowest_elevation == grid[i][j + 1].elevation) {
    grid[i][j].rightTrickle = true;
  }
}
```

2. start the clock
3. simulation until all Nodes are considered as dry, loop variable is time steps, for every iteration:

```
int simulate(Grid<Node> &grid, const int &N, const int &M, const float &A) {
  int time_steps = 0;
  bool all_dry = false;
  while (!all_dry) {
    rainAndAbsorb(grid, time_steps, N, M, A);
    trickle(grid, N);
    time_steps++;
    all_dry = checkAllDry(grid, time_steps, N, A);
  }
  return time_steps;
}
```

a. if time steps lower than M, do a double for loop to simulate rain, every Node plus one on current rain amount field
b. do a double for loop to simulate absorb, if current rain amount on the Node larger than 0, calculate the absorb amount and calculate the amount that is gonna trickle this time.

```
void rainAndAbsorb(Grid<Node> &grid, const int &time_steps, const int &N, const int &M, const float &A) {
  for (size_t i = 1; i < N + 1; i++) {
    for (size_t j = 1; j < N + 1; j++) {
      if (time_steps < M) {
        grid[i][j].current++;
      }
      if (grid[i][j].current > 0) {
        float absorb_amount =
                (grid[i][j].current > A) ? A : grid[i][j].current;
        grid[i][j].current -= absorb_amount;
        grid[i][j].absorbed += absorb_amount;
      }
      if (grid[i][j].current > 0 && grid[i][j].willTrickle) {
        float total_amount_to_trickle =
                (grid[i][j].current >= 1) ? 1 : grid[i][j].current;
        grid[i][j].trickleAmount = total_amount_to_trickle;
        grid[i][j].current -= total_amount_to_trickle;
      } else {
        grid[i][j].trickleAmount = 0;
      }
    }
  }
}
```

c. do a double for loop to simulate trickle, using the precomputed trickle fraction and trickle amount

```
void trickle(Grid<Node> &grid, const int &N) {
  for (size_t i = 1; i < N + 1; i++) {
    for (size_t j = 1; j < N + 1; j++) {
      if (grid[i][j].willTrickle) {
        float each_trickleAmount = (grid[i][j].trickleAmount / grid[i][j].trickleNumber);
        if (grid[i][j].topTrickle) {
          grid[i - 1][j].current += each_trickleAmount;
        }
        if (grid[i][j].bottomTrickle) {
          grid[i + 1][j].current += each_trickleAmount;
        }
        if (grid[i][j].leftTrickle) {
          grid[i][j - 1].current += each_trickleAmount;
        }
        if (grid[i][j].rightTrickle) {
          grid[i][j + 1].current += each_trickleAmount;
        }
      }
    }
  }
}
```

d. check if all the Nodes are considered dry, if any Node needs to trickle then considered not dry, while if every Node doesn't need to trickle we consider this the end of the simulation and can just assume all the current amount will be absorbed and calculate the final answer. In this way, we can quit early and avoid doing a lot of useless array loop checking to save time.

```
bool checkAllDry(Grid<Node> &grid, int &time_steps, const int &N, const float &A) {
  for (size_t i = 1; i < N + 1; i++) {
    for (size_t j = 1; j < N + 1; j++) {
      if (grid[i][j].trickleAmount != 0) {
        return false;
      }
    }
  }
  int max_time_to_end_absorb = 0;
  for (size_t i = 1; i < N + 1; i++) {
    for (size_t j = 1; j < N + 1; j++) {
      max_time_to_end_absorb =
              max(max_time_to_end_absorb, (int) (grid[i][j].current / A));
      grid[i][j].absorbed += grid[i][j].current;
    }
  }
  time_steps += max_time_to_end_absorb;
  return true;
}
```

4. end the clock

Data Structure:

For calculating the time I use std::chrono::high_resolution_clock as the timer and use std::chrono::nanoseconds as the precision unit.

Node.h:

```
#ifndef RAINFALL_NODE_H
#define RAINFALL_NODE_H

#include <climits>

class Node {
public:
    int elevation;
    float current{};
    float absorbed{};
    bool willTrickle{};
    float trickleAmount{};
    int trickleNumber{};
    bool topTrickle{};
    bool bottomTrickle{};
    bool leftTrickle{};
    bool rightTrickle{};

    Node() : elevation(INT_MAX) {}
};


#endif //RAINFALL_NODE_H
```

This header file defines every point in the 2-dimensional grid. Every Node has:
an integer elevation
a float number that represents current rain amount
and float that represents absorbed rain amount
a Boolean that represents if this Node is going to trickle to its neighbors
a float rain amount that is going to trickle to its neighbors
an integer number that represents how many neighbors are going to be trickled from this Node
four Boolean that represents if the Node is going to trickle to each of the four directions.
And the constructor explicitly initializes the elevation to INT_MAX for future use, other fields just use value initialization.

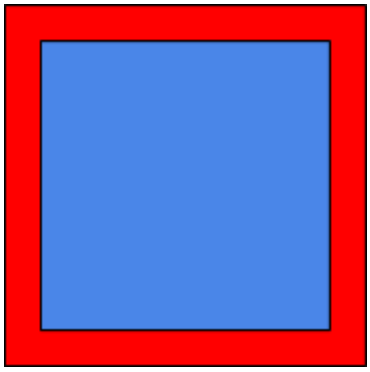
Grid.h:

```cpp
#ifndef RAINFALL_GRID_H
#define RAINFALL_GRID_H

#include <iostream>

template<typename T>
class Grid {
private:
    T *grid;
    int size;
public:
    explicit Grid(int n) {
        size = n;
        grid = new T[n * n];
    }

    ~Grid() {
        delete[] grid;
    }

    T *operator[](int x) {
        return (grid + x * size);
    }

    T *operator[](int x) const {
        return (grid + x * size);
    }
};

#endif //RAINFALL_GRID_H
```

This header file defines a templated array which contains a whole consecutive block of memory which equivalently represent a 2d array. I can simply use the regular [] operator to access the element in the 2d array. I believe this will be somewhat faster than the regular pointer to pointer 2d array approach because the memory is consecutive, so the cache performance could be better.

N+2 Grid Size:

One interesting thing in the code is that I am not using an N*N array but an (N+2)*(N+2) array, and when I do the array operation I will only do to the index 1-N which could be represented by this picture:
I don't actually care about the red area so this means I don't need to do boundary check in the code and that reduces a lot of if-else checks so my branch misses should be lesser.



threadArgs Struct:

```
struct threadArgs {
  int P;
  int M;
  float A;
  int N;
  Grid<Node> *grid;
  pthread_mutex_t *mutex_array;
  int threadId;
};
```

```
for (int i = 0; i < P; ++i) {
  thread_args = (struct threadArgs *)malloc(sizeof(struct threadArgs));
  thread_args->P = P;
  thread_args->M = M;
  thread_args->A = A;
  thread_args->N = N;
  thread_args->grid = &grid;
  thread_args->mutex_array = mutex_array;
  thread_args->threadId = i;
  pthread_create(threads + i, nullptr, worker, (void *)(thread_args));
}
```

I use this simple struct to pass in the argument every thread needs to work.

Parallel:

I choose to parallel the rain part, the absorb part and the trickle part. Because most of the simulation is array access using double for loop so I think I should parallel them.

Rain and Absorb Part:

For the rain and absorb part, there is no dependency on the array operations inside the loops so those loops could be perfectly parallelized. In order to provide better cache performance, I decided to parallel them by rows so each thread will evenly perform on different rows of the 2d array.

```
void rainAndAbsorb(Grid<Node> &grid, const int &time_steps, const int &N,
                   const int &M, const float &A, const int &workload,
                   const int &thread_id) {
  for (int i = thread_id * workload + 1; i < (thread_id + 1) * workload + 1;
       i++) {
    if (i >= N + 1) {
      return;
    }
    for (int j = 1; j < N + 1; j++) {
      if (time_steps < M) {
        grid[i][j].current++;
      }
      if (grid[i][j].current > 0) {
        float absorb_amount = (grid[i][j].current > A) ? A : grid[i][j].current;
        grid[i][j].current -= absorb_amount;
        grid[i][j].absorbed += absorb_amount;
      }
      if (grid[i][j].current > 0 && grid[i][j].willTrickle) {
        float total_amount_to_trickle =
            (grid[i][j].current >= 1) ? 1 : grid[i][j].current;
        grid[i][j].trickleAmount = total_amount_to_trickle;
        grid[i][j].current -= total_amount_to_trickle;
      } else {
        grid[i][j].trickleAmount = 0;
      }
    }
  }
}
```

For example, if there are 4 rows and 2 threads:
Thread1: row 1, row2
Thread2: row 3, row4
And I use the workload parameter with the thread id parameter to calculate the current thread's working rows.

In this way when each thread fetches full rows, it can fetch the whole cache line and pre-fetch could be used to speed up the cache performance.

If we perform the parallel by column the cache performance will be bad because when we entered the next row, every thread needs to fetch the whole new block of memory not connected to the previous block of memory which cause cache miss on every row switch.

Trickle Part:

For the trickle part, when we calculate the trickle amount for every Node we may potentially access the Node at its top, bottom, left and right so there is dependency inside the loop and if the parallel section collapse they may be updating the same Node so causing a race condition. So we need to provide synchronization here.

The choice I use is an array of mutex, specifically, I provided a mutex for each row. And since I am dividing the parallel area by rows, the only possible collapse area for 2 threads is the upper thread's last row and the lower thread's top row.

```
181 void trickle(Grid<Node> &grid, pthread_mutex_t *&mutex_array, const int &N,
182                const int &workload, const int &thread_id) {
183   for (int i = thread_id * workload + 1; i < (thread_id + 1) * workload + 1; i++) {
184     if (i >= N + 1) {
185       return;
186     }
187     for (int j = 1; j < N + 1; j++) {
188       if (grid[i][j].willTrickle) {
189         float each_trickleAmount =
190             (grid[i][j].trickleAmount / grid[i][j].trickleNumber);
191         if (grid[i][j].topTrickle) {
192           if (i == thread_id * workload + 1) {
193             pthread_mutex_lock(&mutex_array[i - 1]);
194           }
195           grid[i - 1][j].current += each_trickleAmount;
196           if (i == thread_id * workload + 1) {
197             pthread_mutex_unlock(&mutex_array[i - 1]);
198           }
199         }
200         if (grid[i][j].bottomTrickle) {
201           if (i == (thread_id + 1) * workload) {
202             pthread_mutex_lock(&mutex_array[i + 1]);
203           }
204           grid[i + 1][j].current += each_trickleAmount;
205           if (i == (thread_id + 1) * workload) {
206             pthread_mutex_unlock(&mutex_array[i + 1]);
207           }
208         }
209         if (grid[i][j].leftTrickle) {
210           grid[i][j - 1].current += each_trickleAmount;
211         }
212         if (grid[i][j].rightTrickle) {
213           grid[i][j + 1].current += each_trickleAmount;
214         }
215       }
216     }
217   }
218 }
```

So I use the workload parameter with the thread id parameter to calculate the current thread's working rows. And then check if this row is the first row or the last row for this current thread, if so, we lock the corresponding mutex in the mutex array.
Other strategies could be using a mutex for every Grid Node but that cause too much memory, if there are 4096*4096 element then I need 4096*4096 mutexes which are insanely large and can cause bad performance in memory.
Also, you can simply lock the top, the bottom or the current row when you are accessing them but that would include a lot of useless locks because in theory they can't be accessed at the same time except for the top and the last row for 2 consecutive threads.

Time Step Part:

```
124  void simulate(Grid<Node> &grid, pthread_mutex_t *mutex_array, const int &N,
125                const int &M, const float &A, const int &P,
126                const int &thread_id) {
127    int workload = calculateWorkload(N, P);
128    while (!all_dry) {
129      rainAndAbsorb(grid, time_steps, N, M, A, workload, thread_id);
130      pthread_barrier_wait(&barrier);
131      trickle(grid, mutex_array, N, workload, thread_id);
132      if (thread_id == 0) {
133        time_steps++;
134        all_dry = checkAllDry(grid, N, A);
135      }
136      pthread_barrier_wait(&barrier);
137    }
138  }
```

For every thread to work at the same time step, I use a barrier to control there working progress. Explicitly when every thread finishes the rain and absorbs simulation, they synchronize at the barrier and then start to calculate the trickle. This avoids the situation that some thread starts to update the value after trickle but another thread is working on the last time steps' absorb which will cause a race condition.

Also to avoid using more locks and race condition, I only let thread 0 (master thread) to update the time step and calculate if we want to end the simulation. Here instead I could let every thread calculate and then update a global bool to check, but that needs additional synchronization and increased the development difficulty. I did test the result using that version and it is not much better than this version so I decided to stick to this simple and stable implementation.

After the master thread updated the loop condition, I use another barrier to synchronize all the threads so they can start to simulate the next time step at the same time.

Profiling:

First I used a check script to do a pressure test and see if any race condition happen.

```
#!/bin/bash
for i in {1..10000}; do
    ./rainfall 8 50 0.5 4096 measurement_4096x4096.in > result.out
    ./check.py 4096 measurement_4096x4096.out result.out
done
```

After it runs sometimes (more than 100 times) I found there is no evidence showing for race condition. So my parallel program is correct.

Run my simple test script test.sh

```
#!/bin/bash
./rainfall_seq 50 0.5 4096 measurement_4096x4096.in > result_seq.out
echo "sequential code:"
./check.py 4096 measurement_4096x4096.out result_seq.out
head -n3 result_seq.out
j=1
for i in {1..4}; do
    ./rainfall_pt $j 50 0.5 4096 measurement_4096x4096.in > result_pt_$j.out
    echo "using $j threads:"
    ./check.py 4096 measurement_4096x4096.out result_pt_$j.out
    head -n3 result_pt_$j.out
    let "j = j * 2"
done
```

The results are:
sequential code:
Output matches all expected values.
Rainfall simulation completed in 1040 time steps
Runtime = 53.7194 seconds

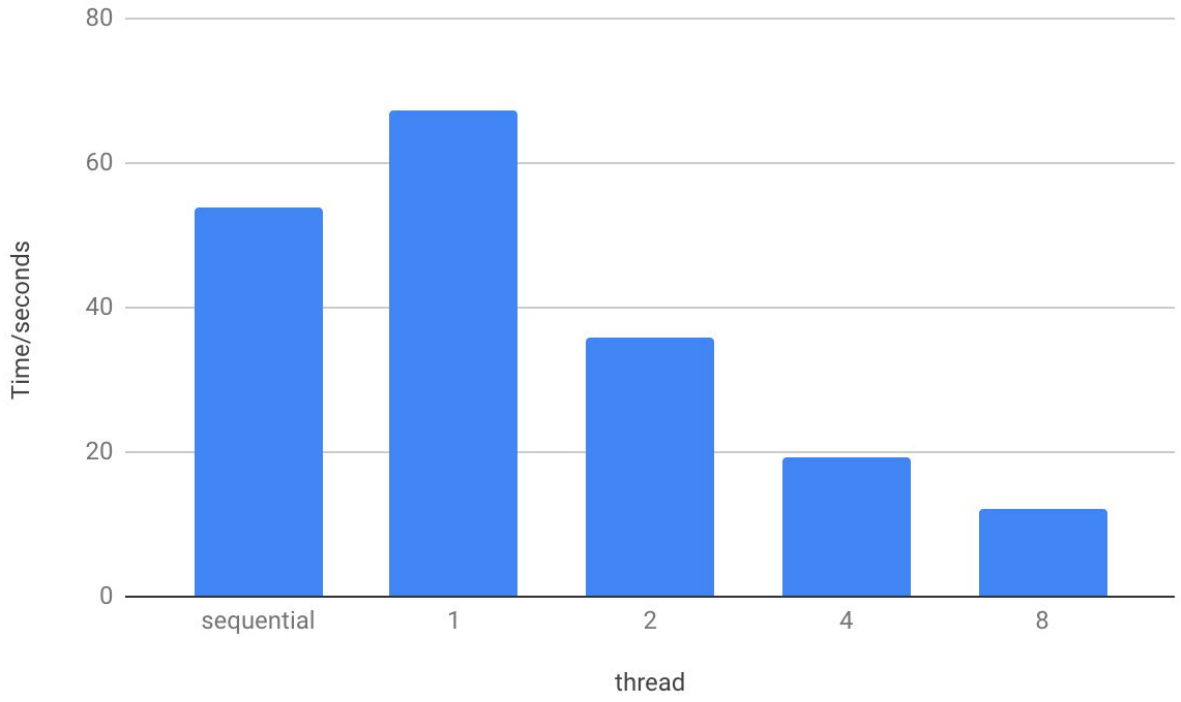
using 1 thread:
Output matches all expected values.
Rainfall simulation completed in 1040 time steps
Runtime = 67.4127 seconds

using 2 threads:
Output matches all expected values.
Rainfall simulation completed in 1040 time steps
Runtime = 35.7587 seconds

using 4 threads:
Output matches all expected values.
Rainfall simulation completed in 1040 time steps
Runtime = 19.3034 seconds

using 8 threads:
Output matches all expected values.
Rainfall simulation completed in 1040 time steps
Runtime = 12.1002 seconds

We can see the sequential code is 20% faster than the parallel version using only 1 thread because of the overhead for synchronization.
And between the parallel version using a different amount of threads, the improvement is very good.



It matches my expectation. As we can see the speed up between 2x thread change is nearly 2. The reason it can't achieve 2 is that:

1. I use a master thread to check the end condition for simulation
2. There are mutex and barrier for synchronization
3. Can't achieve perfect parallel because of other restriction like memory sharing and hardware sharing.