1 Describe your sequential algorithm and data structures for the rainfall simulation.

The overall process of my sequential code and parallelism code follows strictly with the requirements in the pdf:

1.1 Requirement

- *Traverse over all landscape points*
  - *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)*
- *Make a second traversal over all landscape points*
  - *3b) For each point, use the calculated number of raindrops that will trickle to the lowest neighbor(s) to update the number of raindrops at each lowest neighbor, if applicable*

1.2 Data Structures

1. Two dimensions int array *topo*, where topo[i][j] is the elevation at row i col j.
2. Two dimensions float array, representing remaining water at row i col j.
3. Two dimensions float array absorbed, representing the total amount of water absorbed so far. It is also one of the final output.
4. Two dimensions float array trickled. Representing the total amount of water that will trickled into the current location in one step. It will be added to the capacity array.
5. A class *Neighbors* representing the trickle direction of each location.

```
class Neighbors {
 public:
  int n;
  int* neighbor_xs;
  int* neighbor_ys;
  Neighbors() : n(0), neighbor_xs(nullptr), neighbor_ys(
     nullptr) {}
  Neighbors(int num) : n(num) {
    int* neighbor_xs = new int[n];
    int* neighbor_ys = new int[n];
  }
  ~Neighbors() {
    delete[] neighbor_xs;
    delete[] neighbor_ys;
  }
};
```

where n is the total number of neighbors. Int array neighbor_xs is the row index of its neighbors. Int array neighbor_ys is the col index of its neighbors.

Since the trickle direction is only related to the elevations. It is static through the whole program. Hence we can calculate the neighbors information for each location before we start simulation. We do not have to calculate every step during simulation.

1.3 Algorithm

For details of the sequential code, each simulation step contains 3 part:

1. Receive raindrops. If there will be raindrop, it will traverse the capacity array and add 1.0 to every element in it.
2. Absorb raindrops. If there are water in the location, we will absorb water by subtracting the water from capacity array and add same amount of water to absorbed array. Also during traversal, the check if there is no water after absorbing in a location. If all locations has no water, it will return a boolen variable *all_absorbed* indicating there are no water. If so, the simluation will be terminated.
3. Trickle. This function will first subtracting trickled water from capacity array. Then add the trickled water to its corresponding element in *trickled*. Then it will traverse the capacity array, adding trickled[i][j] to capacity[i][j].

2 Parallel Code

2.1 Profile

Before I parallel our code, I profile the code. The result is as followed:

```
    Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 85.47     35.36    35.36      678    52.15    52.15  trickle(float
    **, Neighbors**, int)
 14.19     41.23     5.87                             main
  0.22     41.32     0.09        1    90.00   120.00  find_neighbors(
     Neighbors**, int, int**)
  0.07     41.35     0.03  6792510     0.00     0.00  void std::
     vector<int, std::allocator<int> >::_M_realloc_insert<int const
     &>(__gnu_cxx::__normal_iterator<int*, std::vector<int, std::
     allocator<int> > >, int const&)
  0.05     41.37     0.02                             _init
  0.00     41.37     0.00        5     0.00     0.00  frame_dummy
  0.00     41.37     0.00        3     0.00     0.00  int __gnu_cxx::
     __stoa<long, int, char, int>(long (*)(char const*, char**, int),
      char const*, char const*, unsigned long*, int)
```

It seems like the compiler has inline absorb and rain_drop function into main function. The majority amount of time is used in trickle function.

After disabling optimization, the profile results are

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]     100.0    0.01     1.82                 main [1]
                 1.03     0.24     218/218         trickle(float**,
                    Neighbors**, int) [2]
                 0.43     0.00     218/218         absorb(float**, int,
                     float**, float) [3]
                 0.01     0.10       1/1           find_neighbors(
                    Neighbors**, int, int**) [5]
```

Since the rain drop step is very small compared to the total steps, the rain_drop function does not take too much time compared to absorb and trickle function.

2.2 Parallel Overview

I parallel each step by dividing the overall matrix into several parts by rows. Each thread will only simulate a certain rows, indicated by start_x and end_x in function simulate_step. Figure 1 shows an overview figure of our parallelization strategy.

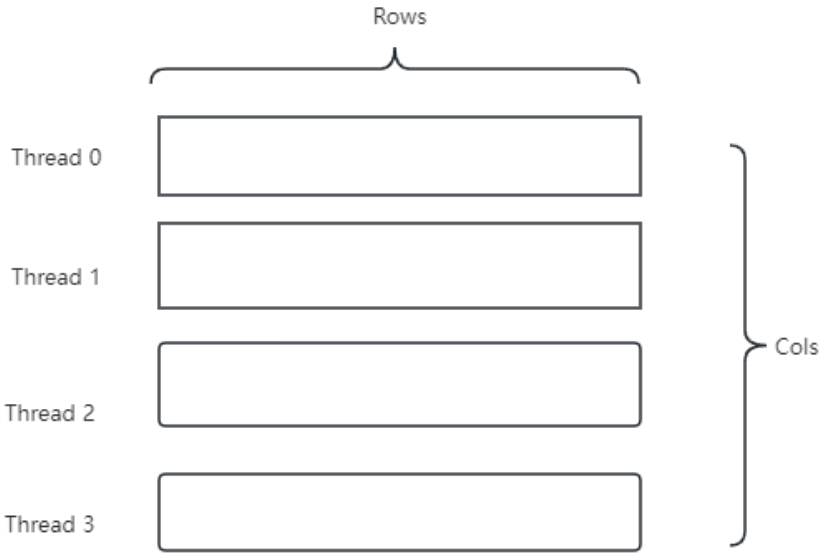

Figure 1: Parallel Overview

For rain drop and absorb, there are no race conditions between different threads. Hence I do not need to make any changes in these 2 functions, just add 2 parameters: start_x and end_x to tell the thread which part of matrix it need to simulate.

As for trickle function. There are race condition in the first traverse when we adding the trickled water to its corresponding element in *trickled*.

Figure 2 shows the race condition. When adding water trickled from x=start_x, it can add water to x=end_x -1 which is used by another thread and vice versa. To control this, I tried two solutions: one is add mutex every time we deal with the data on the "edge", the gray block in Figure 2. This will work but it will reduce our efficiency because the program will enter "edge" many many times. I then tried to use thread local storage. Every thread has a tls variable: vector <pair <pair <int, int >, float >>& edge_trickled. Given an example of its element: 31, 2, 0.25. Which means there are 0.25 water trickled into location 31,2. 31,2 must be a location on the edge (gray block). We add them to its capacity in the main thread. Hence we do not have to use mutex and do context switch.

Another race condition is when we judge if there are no water in every location. In sequential code, we traverse all the locations and check if its capacity is 0. In parallel code, we set the boolen variable *all_absorbed* as an atomic variable to make it work correctly.

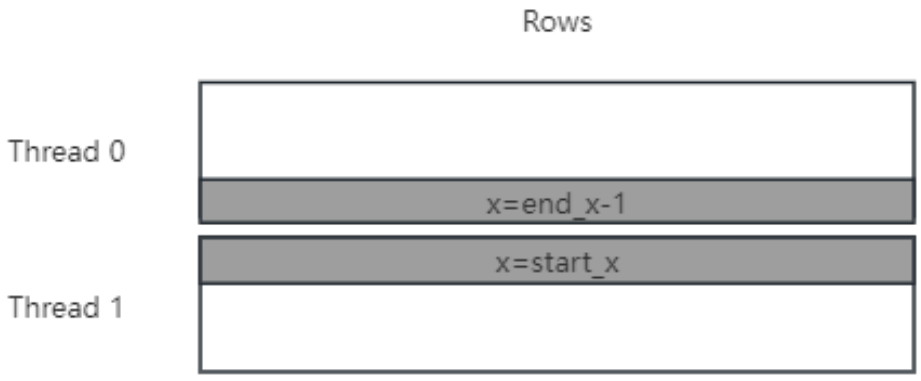

Figure 2: Race Condition

2.3 Barrier

In trickle function, we have

```
1     void trickle(float **capacity, Neighbors **neighbors, int dim,
         vector<pair<pair<int, int>, float>>& edge_trickled,
2              vector<vector<float>> &trickled, int start_x, int end_x,
3              MyBarrier &barrier) {
4 //    cout<<start_x<<":"<<end_x<<endl;
5   for (int x = start_x; x < end_x; ++x) {
6     for (int y = 0; y < dim; y++) {
7 //subtracting trickled water from capacity array. Then add the
     trickled water to its corresponding element in \textit{trickled}.
8       }
9   }
10      barrier.Wait(); // wait until all threads finish calculating
           trickled
11      // second traversal
12      // traverse the capacity array, adding trickled[i][j] to
           capacity[i][j].
```

```
      for (int x = start_x; x < end_x; x++) {
        for (int y = 0; y < dim; y++) {
            capacity[x][y] += trickled[x][y];
        }
      }

}
```

If we do not use thread local storage. We have to add mutex and make a barrier to wait until all the threads finish calculating trickled[i][j]. This will require us to use barrier. Besides, when I use thread pool later on, it still require the main thread to know if the queue is empty. I use conditional variable to realize that, which is very similar to implementing the barrier. My barrier is implemented as follows:

```
class MyBarrier {
public:
  explicit MyBarrier(int num_threads)
      : count(num_threads), notified(false) {}

  void Wait() {
    std::unique_lock<std::mutex> lock(mutex);
    if (--count == 0) {
      // if it is the last thread, wake other threads up.
      notified = true;
      cv.notify_all();
    } else {
      // wait to be signaled.
      while (!notified) //to avoid spurious wakeup occurs
        cv.wait(lock);
    }
  }

private:
  int count;
  std::mutex mutex;
  bool notified;
  std::condition_variable cv;
};
#endif
```

Each Wait() function will subtract count by 1 and then the condition variable will wait until the count is 0. Then it will notify other threads which is waiting.

2.4 Thread Pool

If we do not use thread pool. It has to initialize threads every simulation step, as follows:

```
    vector<std::thread> threads;
    MyBarrier barrier(thread_n);
    for (int i = 0; i < thread_n; i++) {
```

```
        int end_x =  min(row_step+start_x, dim);

        threads.push_back(thread(std::bind(
          simulate_step, capacity, dim, absorbed, absorb_rate,
             neighbors,
          ref(trickled), start_x, end_x, ref(all_absorbed), rain, ref
             (barrier))));

        start_x = end_x;

    }
    for (thread &t : threads) {
      t.join();
    }
```

This will make extra cost for initializing these threads. Instead, we can initialize some threads at the beginning and each thread will loop, getting work from a queue, solving the problem and then wait for work again. I implemented a easy thread pool as follows:

```
class ThreadPool {
public:
    ThreadPool(int numThreads) : stop(false), completedTasks(0),
       totalTasks(numThreads){
        for (size_t i = 0; i < numThreads; ++i) {
            workers.emplace_back([this] {
                while (true) {
                    std::function<void()> task;
                    {                       std::unique_lock<std::mutex>
                        lock(queueMutex);
                        condition.wait(lock, [this] { return stop ||
                           !tasks.empty(); });

                        if (stop && tasks.empty()) {
                            return;
                        }
                        task = std::move(tasks.front());
                        tasks.pop();
                    }
                    task();
                    // to replace future
                    { std::unique_lock<std::mutex> lock(
                       completionMutex);
                        completedTasks++;
                    }
                    completionCondition.notify_one();
                }
            });
        }
```

```
    }

    void enqueue(std::function<void()> task) {
        {
            std::unique_lock<std::mutex> lock(queueMutex);
            // don't allow enqueueing after stopping the pool
            if (stop) {
                throw std::runtime_error("enqueue on stopped
                   ThreadPool");
            }
            tasks.emplace(std::move(task));
        }
        condition.notify_one();
    }
    ~ThreadPool() {
        {
            std::unique_lock<std::mutex> lock(queueMutex);
            stop = true;
        }
        condition.notify_all();
        for (std::thread &worker : workers) {
            worker.join();
        }
    }

    void waitAll() {
        std::unique_lock<std::mutex> lock(completionMutex);
        completionCondition.wait(lock, [this] { return completedTasks
            == totalTasks; });
        completedTasks = 0;
    }

private:
    std::vector<std::thread> workers;
    // the task queue
    std::queue<std::function<void()>> tasks;
    // synchronization
    std::mutex queueMutex;
    std::condition_variable condition;
    std::mutex completionMutex;
    std::condition_variable completionCondition;
    int totalTasks;
    int completedTasks;
    bool stop;
};
```

At the beginning, each worker will be assigned to one thread. This thread will keep getting task from

a queue. Since each thread has to wait until all other workers finish their task each simulation step. I use a strategy similar to Barrier: Each worker will increase completedTasks by one. ThreadPool has a waitAll() function. It will wait the condition variable. It will be signaled once the completedTasks equals to thread_numbers. It indicates the queue become empty and one simulation step is over. In this way, waitAll() acting as a barrier, waiting every thread to finish completing their tasks for one simulation step.

2.5 Thread Local Storage

Every thread has a local variable: vector <pair <pair <int, int >, float >>& edge_trickled. Given an example of its element: 31, 2, 0.25. Which means there are 0.25 water trickled into location 31,2. 31,2 must be a location on the edge (gray block). So it will not cause race condition because we do not add them to capacity, which will induce conflicts in gray area in Figure 2.

We add them to its capacity in the main thread as follows. Hence we do not have to use mutex and do context switch.

```
    for (int i = 0; i < thread_n; i++) {
        int end_x =  min(row_step+start_x, dim);
        pool.enqueue(std::bind(
          simulate_step, capacity, dim, absorbed, absorb_rate,
             neighbors, ref(edge_trickled[i]),
          ref(trickled), start_x, end_x, ref(all_absorbed), rain));
        start_x = end_x;
    }
    pool.waitAll();
    for(vector<pair<pair<int, int>, float>>& edge: edge_trickled){
      for(pair<pair<int, int>, float>& points: edge){
        capacity[points.first.first][points.first.second] += points.
           second;
      }
    }
```

2.6 atomic float

Another strategy to solve race condition is to use atomic float for trickled[x][y]. Thus adding to trickled[x][y] becomes an atomic operation. However, since there are no official implementation from STL in C++ 11. I did not use this approach. But I think this approach will be the most effective method.

3 Performance

3.1 Result

This result basically matches my expectation. For small problem size like 16, first let's focus on the compare between sequential and parallel code. Since we add extra cost for initializing threads, moving data, synchronizing result and this cost is the major factor for performance compared to the simulation task, we can see as we have more threads, the total time become larger and it is longer than sequential time. Let's then focus on compare between using thread pool and not. As we mentioned before, the major

Table 1: result

Size	Sequential	Parallel				Parallel with ThreadPool + Thread local Storage			
	/	1 thread	2 thread	4 thread	8 thread	1 thread	2 thread	4 thread	8 thread
16	0.0003s	0.014s	0.024s	0.038s	0.077s	0.006s	0.007s	0.009s	0.014s
512	0.92s	1.40s	1.01s	0.69s	0.51s	1.19s	0.74s	0.60s	0.53s
2048	50.98s	69.24s	44.22s	32.24s	25.48s	63.78s	43.79s	32.07s	23.31s
4096	355.50s	411.99s	254.61s	180.11s	140.70s	400.13s	250.85s	177.50s	140.92s

factor for performance is initializing threads, moving data, synchronizing result, if we use thread pool, we can save a lot of time for initializing threads every step, hence it will have shorter time compared to not using thread pool.

If the problem size grows larger, especially reaches 4096, the major factor impacting the performance become simulation. Hence when we have more threads, the time will shorten and it will achieve good speed ups when we have 8 threads. Since initializing threads only takes hundred ns, the speed up brought by thread pool is not very obvious. Besides, since there are only 8 threads at most, there are not too many conflict data. Hence using local storage does not bring too much speed up. So we can see the result of using thread pool and thread local storage and not using are very similar. The speed ups are not exactly proportional to more threads because we only speed up some part of function and we still need to wait, synchronize between different threads.