ECE 565 Assignment 5 Writeup

Wenting Yang (wy77)

Junfeng Zhi (jz399)

Sequential Algorithm and data structures

For the sequential version, we use the following data structures:

1. `const vector<vector<int>> map` for storing the elevations in the input file;
2. `vector<pair<int, int>> lowestNeighbours` for the lowest neighbours of each point;
3. `vector<vector<float>> absorbedRainDrop` for the absorbed raindrops at each point;
4. `vector<vector<float>>& curRainDrops` for the current raindrops (on surface) at each point;
5. `vector<TrickleInfo> trickleDrops` for recording the trickle drops, where `TrickleInfo` is a struct including the trickle amount, the row and col of the destination point:

```
typedef struct TrickleInfo {
  float amount;
  int r;  // target row
  int c;  // target col

  TrickleInfo(float amt, int row, int col) {
    this->amount = amt;
    this->r = row;
    this->c = col;
  }
} TrickleInfo;
```

The algorithm is to first have a nested for loop to traverse each point. At each point, we will determine if the current timestep < the raining timestep, if so, we increment the raindrops of the current point:

```
for (int row = 0; row < args.dimension; row++) {
  for (int col = 0; col < args.dimension; col++) {
    // rain, add new drop to each point
    if (totalTimeStep <= args.timeStep) {
```

```
            curRainDrops[row][col]++;
          }
```

Then we determine if the current raindrop is more than 0. If so, we absorb by the specified absorption rate:

```
      if (curRainDrops[row][col] > 0) {
            float absRate = args.absorptionRate;
            if (curRainDrops[row][col] <= args.absorptionRate) {
              absRate = curRainDrops[row][col];
            }
            curRainDrops[row][col] -= absRate;
            absorbedRainDrop[row][col] += absRate;
          }
```

Then we decide trickle amount for this point:

```
      // decide trickle amount
          if (abs(curRainDrops[row][col]) < 1e-6) {
            continue;
          }
          trickleAmount =
              curRainDrops[row][col] > 1 ? 1.0 :
curRainDrops[row][col];
```

Then we calculate the lowest neighbours of the current point and store them in the `vector<pair<int, int>> lowestNeighbours` vector. If this vector is not empty, we deduct `trickleAmount` from `curRainDrops[row][col]` and push the corresponding `TrickleInfo` into the `trickleDrops` vector:

```
for (auto& point : lowestNeighbours) {
            int r = point.first;
            int c = point.second;
            trickleDrops.push_back(TrickleInfo(trickleAmount
/lowestNeighbours.size(), r, c));}
```

After the for loop is completed - each point has gone through the previous steps, we traverse the `trickleDrops` vector and add trickle raindrops to the corresponding points:

```
for (auto& it : trickleDrops) {
      curRainDrops[it.r][it.c] += it.amount;
    }
```

We end the simulation when `totalTimeStep > args.timeStep` and there are no raindrops in the `curRainDrops` vector.

Parallel strategies and reasons

We first profiled the codes using -DPROFILE. Here are the results:

```
---------------------------------------------------------------------------
Timer Results:                    #calls       avg(sec)     total(sec)
---------------------------------------------------------------------------
Read map                               1       0.020521     0.020521
Add rain drops                  57147392       0.000000     1.461347
Drops absorbed                  57147392       0.000000     1.557112
Record trickle direction        10419497       0.000000     0.569759
Reduce trickle drops            10419497       0.000000     0.265584
Add trickle drops                    218       0.000076     0.016556
isAllAbsorbed                        188       0.000005     0.000851
```

The results suggest that the “add rain drops” and “drops absorbed” processes are taking the largest portion of runtime. Also, the calculation of trickle drops takes a long time to execute. We should focus our parallelization effort on these three parts.

We parallelize the “add rain drops” and “absorb” parts by dividing all the points equally between all threads. Because there is no data dependency between the computation of different points, we don’t need any synchronization here.

For the lowest neighbours, we find it quite redundant to calculate the lowest neighbours over and over in each timestep. So instead we have a large vector of vector that is remembering the lowest neighbours for each point across timesteps. We do a one-time calculate before the simulation starts (before we start adding raindrops to each point). This calculation is also in parallel, but we have to lock when pushing back into the vector.

> `vector<vector<pair<int, int>>> lowestNeighbours` for the lowest neighbors of each point. Each point has an index = row * dimension + col, a vector of lowest neighbors for point i can be accessed by `lowestNeighbours[i]`, where each neighbor is a pair (row, column);

When calculating trickle raindrops, instead of pushing into a shared vector, each thread has its own vector. After calculating the trickle drops in parallel, we would traverse all the vectors produced by different threads in serial and add each trickle to . As the profile had shown, adding those trickles does not take a big portion of time. But calculation of the trickle drops consumes a lot of time. So we chose to speedup the calculation part.

Synchronization between threads

1. Barrier

   We use a barrier to wait for all the threads to complete. Because C++ 11 does not have a built-in barrier as pthreads, we implemented a barrier on our own.

The implementation of the barrier is as follows: we have a global variable `done`, which is an atomic integer, when each thread is created in the for loop, we detach it, and call the barrier(), pass in the parameter `nThreads`, in the barrier, the main thread will check if done reaches the value `nThreads`, if not, it will wait on the condition variable.

```
void barrier(const int nThreads) {
  std::unique_lock<std::mutex> lk(mtx);  // mtx locks here
  while (done.load() != nThreads) {
    cv.wait(lk);
  }
  // unlock automatically when lk destructs.
}
```

Each thread will acquire the `mtx` lock before modifying the `done` variable, even if it is atomic, this is to prevent the case that `done` is modified between the condition check and `wait`.

```
mtx.lock();
  done++;
  mtx.unlock();
  cv.notify_all();
```

2. Vector of trickleDrops

   When calculating the trickle raindrops, each thread has its own vector to record the trickle drops, instead of pushing into a shared vector. This is to prevent resource contension, and also, std::vector is not thread safe, if we use a shared vector we would have to acquire locks, which reduces performance.

   In the main thread, we use an array of pointers to track the vector owned by each thread. Later on, the main thread would traverse all the vectors to add the trickledrops to each point.

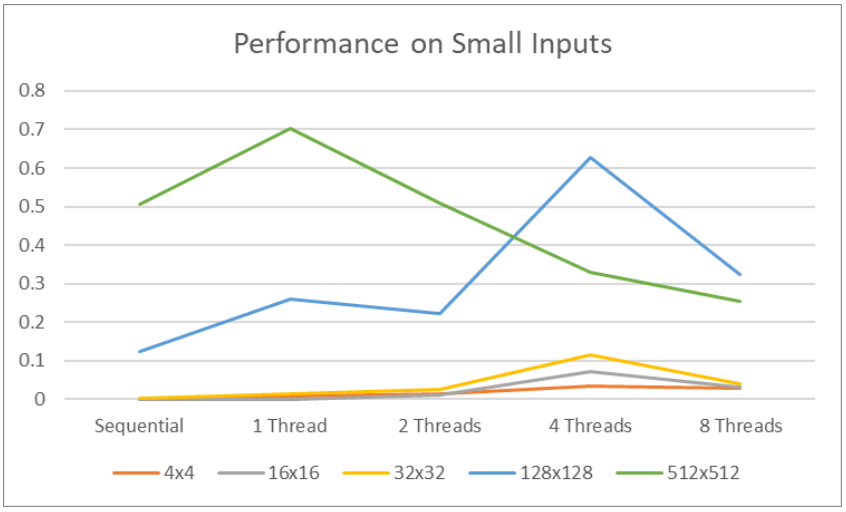
3. Lock

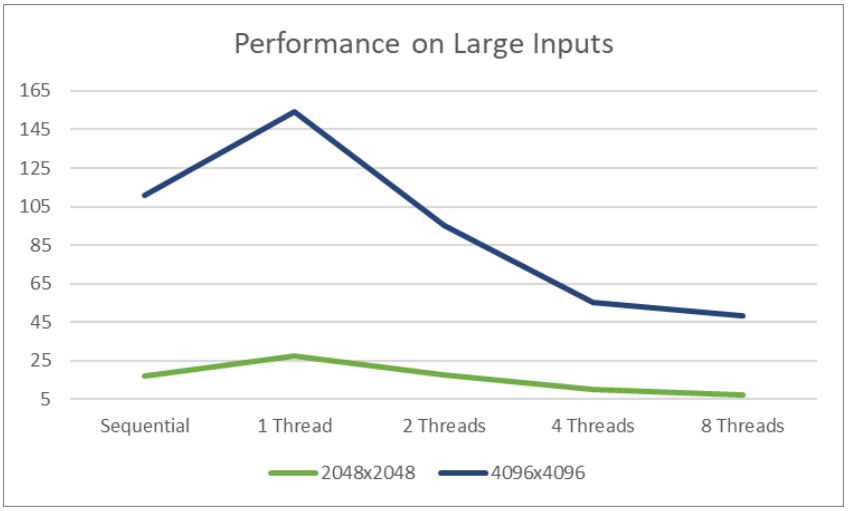
   When calculating the lowest neighbours, we use lock while pushing into a shared vector. Because this is a one-time calculation and the locked version already looks good enough, we did not further optimize it to use separate vectors per thread.

Performance Results

Input Size	Sequential	1 Thread	2 Threads	4 Threads	8 Threads
4x4	0.000072	0.0072229	0.0132821	0.0359077	0.02967

16x16	0.000555	0.007634	0.0127015	0.0727795	0.0322608
32x32	0.001704	0.0137531	0.025188	0.116533	0.0406636
128x128	0.122860	0.261052	0.223548	0.627155	0.324605
512x512	0.505005	0.704028	0.509856	0.329532	0.255222
2048x2048	17.210130	27.8223	17.4332	10.3855	7.5321
4096x4096	110.8664	154.058	95.2101	55.3846	48.3368





We managed to gain speedup when running with 2, 4 or 8 threads. The program is running more slowly given threadNum = 1, that's because there is overhead in creating and destructing the thread resources.

This matches our expectation because according to our profiling results, we have approximately paralleled 71% of the program, according to Amdahl's law, when running with 8 threads, the speedup should be: $\frac{1}{29\% + \frac{71\%}{8}} = 2.58$. In the experiment, we achieved 2.29× speedup, which is close to the limitation predicted by Amdahl's law. Given the overhead introduced by creating threads and resource contention, this speedup is pretty good.