ECE 565 Assignment 5

repo: https://gitlab.oit.duke.edu/ece565/hw5

Qiheng Gao (qg45), Boyi Wang (bw224)

Project Overview

In this project, we design the program that carries out rainfall simulation in both sequential and parallelized version in modern C++.

Sequential Design Description

Data Structure

We designed a simulator class using Objective-Oriented Design mindset to implement the whole rainfall simulation process. Here is an overall view of our designed class:

```
class Simulator {
   protected:
    int rainSteps;  // rain drop time steps
    float absorbRate;  // amount of rain absorbed into ground/timestep
    vector<vector<int>> landscape;  // elevation of landscap
    // raindrop trickle directions on each coordinate
    vector<vector<vector<pair<int, int>>>> trickleDir;
    vector<vector<float>> absorbed; // absorbed raindrops at each point
    double totalTime; // simulation complete time
    int totalSteps; // simulation complete time steps
...
}
```

For each `Simulator` object, its `rainSteps`, `absorbRate`, `landscape`, and `trickleDir` fields will be set within its constructor according to the arguments we got from command line. During the simulation process, the `absorbed` 2D vector, `totalTime` and `totalSteps` fields will be updated by every iteration.

Sequential Algorithm

- Overall Logic in `main()`

  In the `main()`, we first parse all command line arguments and use them to create a `Simulator` object

  ```
  int main(int argc, char** argv) {
      ...
      unique_ptr<Simulator>
        simulator(new Simulator(rainSteps, absorbRate, N, file));
      simulator->simulate();
      simulator->printResult();
      ...
  }
  ```

Then, we call the member function `simulate()` to start the simulation process. Finally, we call the member function `printResult()` to print out results stored in the fields.

- Init Logic in Constructor

  When we construct `Simulator` object, its 2D vectors `landscape` field will be created using data read from input file.

```
void Simulator::initElevations(ifstream& elevationFile) {
    string line;
    int row = 0;
    int col = 0;
    int elevation;
    while (getline(elevationFile, line)) {
        stringstream ss(line);
        col = 0;
        while (ss >> elevation) {
            this->landscape[row][col] = elevation;
            ++col;
        }
        ++row;
    }
}
```

After we got each coordinate's elevation value in `landscape`, we can build another 2D vector field `trickleDir` to see how raindrop should be trickled on each points.

```
// init the lowest neighbor of each point
void Simulator::initTrickleDir() {
    // north/south/east/west
    vector<vector<int>> directions{{-1, 0}, {1, 0}, {0, 1}, {0, -1}};
    int N = this->landscape.size();
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            int lowest = INT_MAX;
            // find the lowerst neighbor value
            for (const auto& direction : directions) {
                int neighRow = i + direction[0], neighCol = j + direction[1];
                if (0 <= neighRow && neighRow < N && 0 <= neighCol && neighCol < N) {
                    lowest = min(lowest, this->landscape[neighRow][neighCol]);
                }
            }
            if (this->landscape[i][j] <= lowest) {
                continue;
            }
            for (const auto& direction : directions) {
                int neighRow = i + direction[0], neighCol = j + direction[1];
                if (0 <= neighRow && neighRow < N && 0 <= neighCol && neighCol < N) {
                    int neighVal = this->landscape[neighRow][neighCol];
                    if (neighVal == lowest) {
                        this->trickleDir[i][j].push_back({neighRow, neighCol});
                    }
                }
            }
```

```
        }
    }
}
```

- Simulation Logic in `Simulator::simulate()`

  Once the simulation start, we will first created two 2D vectors to record real-time datas for each step, where `N` is the size of landscape, and a boolean `isWet` to see whether the simulation should be finished.

  Then we will start the timer using `omp_get_wtime()` and enter into while loop. At each time step, we will use `simulate()` and `updateStatus` to traverse matrix respectively, and simulate the rainfall process. After simulation ends, stop the timer and save the execution time into `totalTime`

```
void Simulator::simulate() {
    int N = this->landscape.size();
    // in-time raindrop status for each point
    vector<vector<float>> status(N, vector<float>(N, 0.0f));
    // in-time will-be-trickled raindrops for each point
    vector<vector<float>> trickled(N, vector<float>(N, 0.0f));
    // check simulation stop condition
    bool isWet = true;
    const double begin = omp_get_wtime();
    while (isWet) {
        ++this->totalSteps;
        // 1st Traverse: rain, absorb, calculate trickled
        simulate(status, trickled);
        // 3b) 2nd Traverse: update # of raindrops at each lowest neighbor & check wet
        isWet = updateStatus(N, status, trickled);
    }
    const double end = omp_get_wtime();
    this->totalTime = end - begin;
}
```

  Within the override function `simulate(status, trickled)`, for each point, we will

  1) receive new raindrops (while it is still raining), update `status`

  2) absorb raindrop into ground, update `status` and `absorb`

  3a) calculate the number of rain that will be trickled on next step (saved in `trickled`)

```
void Simulator::simulate(vector<vector<float>>& status, vector<vector<float>>& trickled) {
  int N = this->landscape.size();
  for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
          // 1) Receive a new raindrop for each point while still raining
          if (this->totalSteps <= this->rainSteps) {
              status[i][j] += 1.0;
          }
          // 2) if points have raindrops, absorb
          if (status[i][j] > 0.0) {
              float toAbsorb = min(status[i][j], this->absorbRate);
              status[i][j] -= toAbsorb;
```

```
                this->absorbed[i][j] += toAbsorb;
            }
            // 3a) calculate # of raindrops will next trickle to
            // the lowest neighbors
            if (status[i][j] > 0.0 && this->trickleDir[i][j].size() > 0) {
                float dropToTrickle = min(status[i][j], 1.0f);
                status[i][j] -= dropToTrickle;
                float toNeigh = dropToTrickle / this->trickleDir[i][j].size();
                for (const auto& point : this->trickleDir[i][j]) {
                    trickled[point.first][point.second] += toNeigh;
                }
            }
        }
    }
}
```

Then using function `updateStatus(N, status, trickled)` to 3b) update trickled raindrops onto the `status` vector, and check if any rain still be held by each point to update `isWet`:

```
bool Simulator::updateStatus(int N, vector<vector<float>>& status, vector<vector<float>>& trickled) {
  bool isWet = false;
  for (int i = 0; i < N; ++i) {
      for (int j = 0; j < N; ++j) {
          status[i][j] += trickled[i][j];
          trickled[i][j] = 0.0;
          if (abs(status[i][j]) > FLT_EPSILON) {
              isWet = true;
          }
      }
  }
  return isWet;
}
```

Parallelization Strategy Description

As we noticed that the main workload of the simulation are centering around the 2-D matrixes, and different data elements can be processed in an independent fashion. Therefore, we decide to do the data parallelism to assign the `ThreadNum` threads to process the `N x N` matrixes, where each thread is responsible for `N / ThreadNum` rows, so that the entire matrixes updating task can be separated by multiple threads.

Reason

The reason why we chose these parts to parallelize is that most of the simulation logic is independent for each point, and can be processed in parallel without synchronization:

1) receive new raindrops

2) absorb rain into ground

3b) update trickled raindrops on `status` matrix

The only two parts we need to pay attention on (need synchronization) are:

3a) calculate the number of raindrops that will next trickle to the lowest neighbor(s)

4) check when simulation should be finished and update that flag

because different point may have raindrops trickled to same lowest neighbor(s). Besides, the simulation finish flag should depends on all threads: only when the all separated sections charged by each thread have no rain remained should the simulation stop.

Data Structure & Synchronization Types

We used two types of synchronization across threads: **mutex** and **barrier.**

We created a new class `SimulatorPro` which inherits the former `Simulator` class:

```
class SimulatorPro : public Simulator {
   protected:
    int threadNum;  // # of threads
    vector<mutex> mutexes;  // each point has a mutex, N * N in total
    mutex globalStatusLock; //mutex for globalFinished update
    Barrier barrier;    //reusable barrier to synchronize all thrds
    bool globalFinished;    //simulation finish flag
...
}
```

- For this new child class, we add above new fields. As discussed above, we have a vector of mutex `mutexes` (size = N * N) so that each point in the matrix has a mutex to eliminate race condition for 3a).
- We also need another mutex `globalStatusLock` to update the simulation finish flag `globalFinished`.
- Beside, we implemented a reusable barrier class `Barrier` to achieve synchronization between each thread.

Parallelized Simulation Code

For 3a), having mutex for each point will give our `SimulatorPro` better performance because the mutex will be used only when different threads accessing the same point (we reduce the lock granularity). Otherwise, threads won't be blocked. After the first traverse, we then use barrier to synchronize all threads and make sure finish calculating the trickled amount for the whole matrix:

```
for (int i = id * rows; i < (id + 1) * rows; ++i) {
    for (int j = 0; j < N; ++j) {
        //... 1) Receive a new raindrop for each point while still raining
        //...2) if points have raindrops, absorb
        // 3a) calculate # of raindrops will next trickle to the lowest neighbors
        if (status[i][j] > 0.0 && this->trickleDir[i][j].size() > 0) {
            float dropToTrickle = min(status[i][j], 1.0f);
            status[i][j] -= dropToTrickle;
            float toNeigh = dropToTrickle / this->trickleDir[i][j].size();
            for (const auto& point : this->trickleDir[i][j]) {
                // lock the cell
                int lockID = point.first * N + point.second;
                unique_lock<mutex> lk(this->mutexes[lockID]);
```

```
                trickled[point.first][point.second] += toNeigh;
            }
        }
    }
}
// add barrier, wait untill all current absorb and trickle matric are processed
this->barrier.wait();
```

For the second traverse 3b), update `status` and `trickled` like what we did in sequential version. The difference is that each section has its own finish flag `localFinished` indicating whether rain remaining in that section. Then we need to update `globalFinished` based on each thread's `localFinished`, guarded by mutex `globalStatusLock`. Finally, use barrier to synchronize all threads.

```
bool localFinshed = true;
for (int i = id * rows; i < (id + 1) * rows; ++i) {
    for (int j = 0; j < N; ++j) {
        status[i][j] += trickled[i][j];
        trickled[i][j] = 0.0;
        if (abs(status[i][j]) > FLT_EPSILON) {
            localFinshed = false;
        }
    }
}
unique_lock<mutex> lk(this->globalStatusLock);
this->globalFinished = this->globalFinished && localFinshed;
lk.unlock();
// add barrier,
// wait untill global status are passed to && with [locaFinished] of all threads
this->barrier.wait();
```

When `globalFinished` updated by all threads, check whether simulation should be finished. If not, reset the flag and enter into next time step:

```
if (this->globalFinished) {
    this->totalSteps = steps;
    return;
}
// add barrier,
// wait untill all threads has passed the globalFinished determination,
// reset globalFinished
this->barrier.wait();
this->globalFinished = true;
```

Results Measurement & Discuss

Compile without any optimization and calculate the execution time starting from the simulation task.

Results

	Sequential	1 thread	2 threads	4 threads	8 threads

Runtime (second)	1020.46	1098.11	642.555	367.345	155.296

Discussion

Overall, the result **match** our expectation. We are able to obtain speedups: reduce almost 85% running time with 8 threads compared to sequential version, and the result has varying degrees of improvement when using 2 and 4 threads, thus we can conclude that the affect of synchronization overheads to performance is not much.

Moreover, it can be seen from the table that the parallelized version using 1 thread are slightly slower than the sequential version. We believe that this is acceptable; as compared to the entirely sequential processing, the parallelized version cannot circumvent overheads like lock acquire, lock release and barrier overheads.