HW5 Report

Yuanrui Zhang (yz545), Yuchuan Li (yl645)

1 Parallelization strategy

1.1 Sequential algorithm and data structure

Data structure

We created a class called *Landscape*, representing the whole area to address rainfall simulation. In this class, we have the following fields:

```
int dim;
bool is_dry;
double* raindrops;
double* absorbed_drops;
double* trickled_drops;
std::vector<std::pair<int, int> >* trickling_directions;
```

- *dim*: the dimension of the landscape
- *raindrops:* an array to indicate the number of raindrops at each point. It is to represent a 2d space in a way of 1d.
- *absorbed_drops:* an array to show how many raindrops have been absorbed at each point. It is to represent a 2d space in a way of 1d.
- *trickled_drops*: an array to store the number of drops that will be trickled to a certain point. It can be considered as an intermediate storage for step 3a.
- trickling _directions: as soon as a landscape is initialized, we can know which direction(s), from a certain point, can raindrops trickle to, based on the elevation. Therefore, we introduce this field as an array of vectors of coordinates. Or to put it more concrete, it is basically a 2d array, elements of which are the possible directions where raindrops can trickle.

Member functions

Several related member functions are listed.

```
void receive_rain_drop(int row, int col);
void absorb(int row, int col, double absorption_rate);
void absorb_pt(int row, int col, double absorption_rate);
void trickle_to(int row, int col);
void calculate_trickled_drops(int row, int col);
```

- *receive_rain_drop:* at a certain point with index *row* and *col*, receive 1 full raindrop.

- *absorb:* at a certain point with index *row* and *col*, absorb some amount of raindrops given *absorption_rate*.
- *absorb_pt:* absorb method for multi-threading.
- *trickle_to:* add the number of raindrops from *trickled_drops* to current landscape.

Algorithm

Based on subsections above, our core algorithm is:

```
while(step == 1 || !landscape.has_been_dry()) {
      landscape.reset_is_dry();
      // Within one time step
      // First iteration
      for(int i = 0; i < curr_dim; i++) {
          for(int j = 0; j < curr_dim; j++) {
              // Receive raindrops
              if(step <= raining_time) {
                  landscape.receive_rain_drop(i, j);
              }
              // Absorb
              landscape.absorb(i, j, absorption_rate);
              // Calculate trickled drops
              landscape.calculate_trickled_drops(i, j);
          }
      }
      // Second iteration
      for(int i = 0; i < curr_dim; i++) {
          for(int j = 0; j < curr_dim; j++) {
              landscape.trickle_to(i, j);
          }
      }
      ++step;
  }
```

Our implementation follows exactly the homework documentation.

- Procedure 1), 2) and 3a) are wrapped in the loop under “First iteration”; Procedure 3b) is marked with “Second iteration”.
- Variable “step” indicates time steps to simulate the rainfall till end.
- “has_been_dry” is an attribute belonging to landscape class. This attribute will only be updated when carrying out *absorb* method. It is simply a boolean flag and will be checked against each point after absorption.

1.2 Parallel strategies & Reasoning

We parallelized tasks as two functions, first_iter and second_iter. The first_iter is responsible for step 1, 2 and 3a in the description, with a range of rows. The second_iter is responsible for step 3b with a range of rows.

For step 3a, calculating the trickling part would cause race conditions, so we separate the whole landscape into two groups of sections of the landscape. All sections in one group would not access the same point at the same time, thus no race conditions. Then the strategy here would be calculating all the trickling water drops in the first group of sections concurrently, then doing some synchronization, then calculating the other group of sections concurrently. Basically we avoid race conditions through separating tasks and doing synchronization. This would cause some overhead but the tasks are splitted evenly and the synchronization is rather easy.

For step 3b, updating the landscape with trickling water drops is embarrassingly parallel. So the second_iter() function responsible for step 3b is called only once, splitting the whole landscape into one group of sections, calculating concurrently. Also, synchronization after this iteration is needed.

We chose this strategy because we avoid race conditions in a rather simple way and use a minimal number of synchronizations, while the tasks are splitted evenly to all threads. This should expect a good performance for a parallel version.

As a side note, we also ran a profiler to see which methods consume most time. Here shows the *gprof* result:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 37.09      0.22     0.22                             Landscape::absorb(int, int, double)
 33.64      0.41     0.20                             Landscape::calculate_trickled_drops(int, int)
 23.29      0.55     0.14                             Landscape::trickle_to(int, int)
  4.31      0.57     0.03                             Landscape::receive_rain_drop(int, int)
  1.72      0.58     0.01   262144    38.17    38.17  Landscape::calculate_trickling_directions(int, int, double*)
  0.00      0.58     0.00   122477     0.00     0.00  void std::vector<std::pair<int, int>, std::allocator<std::pair<int, int> >
  0.00      0.58     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z13validate_argci
  0.00      0.58     0.00        1     0.00     0.00  _GLOBAL__sub_I__ZN9LandscapeC2EiPKc
```

1.3 Type of Synchronization

So after each iteration, we need to synchronize all the tasks/threads we assigned. We use the thread pool's waitAll() function to ensure that all tasks are done before executing the next line of code in the main thread. Specifically, the thread pool library uses condition variables to achieve synchronization among threads.

2 Performance analysis & Debugging

In our very first version, we used the thread group class in the Boost library to build up a thread pool and observed that it actually slowed down when using multithreads. Since the version we installed is outdated, we thought this is caused by the library, not by our code.

Then we switched to using a thread pool implemented in ECE 751 with the same strategy. But we still could not get the expected speed-up in the parallel version.

To be more specific, we re-ran our program with the provided timer library from HW4. Simply speaking, what we did was wrapping each barrier with a timer, like

```
Timer_Start("Bar 1");
thread_pool.waitAll();
Timer_Stop("Bar 1");
```

Since we used three barriers, there would be “Bar 1”, “Bar 2” and “Bar 3“.

Here, we provide three screenshots of running with measurement_4096x4096.in against 1, 2, 4 threads respectively.

1 thread:

```
-------------------------------------------------------------------------
Timer Results:                    #calls       avg(sec)     total(sec)
-------------------------------------------------------------------------
Bar 1                               1040      0.100247   104.257309
Bar 2                               1040      0.099640   103.626027
Bar 3                               1039      0.039119    40.645094
```

2 threads:

```
-------------------------------------------------------------------------
Timer Results:                    #calls       avg(sec)     total(sec)
-------------------------------------------------------------------------
Bar 1                               1040      0.340701   354.328969
Bar 2                               1040      0.334972   348.370449
Bar 3                               1039      0.020153    20.938782
```

4 threads:

```
---------------------------------------------------------------------
Timer Results:                    #calls       avg(sec)     total(sec)
---------------------------------------------------------------------
Bar 1                               1040      0.478402   497.538226
Bar 2                               1040      0.468005   486.724958
Bar 3                               1039      0.013126    13.638232
```

From these, we observed that the first_iter (seen from Bar 1 and Bar 2) slowed down while the second_iter (seen from Bar 3) sped up if running with more threads, which was unintuitive and unexpected. Because those two iterations should be the same in the task point of view. They were both evenly spreaded tasks among threads.

Then, we asked Brian for help and he pointed out that there were two minor mistakes in our code:

The first one was about **false-sharing**, where we allocated a boolean array of the same size of num_threads, then assigned each boolean value to the corresponding thread. This approach will result in a slow-down because the size of the array is too small so it is stored in a single cache line that was shared among these threads. Plus, we were updating the value of each boolean variable all the time. So it would force the cache to update all the time. We fixed this with allocating a much larger boolean array with sparse values used to avoid false-sharing among threads. We got a speed-up after this improvement but it was still slower when using more threads.

The second was that we have a single field called is_dry in class landscape, which was the major cause to slow down our multithreading code. This value was also updated all the time by different threads. We identified this situation as a race condition but our solution was just ignore this value and use the boolean array aforementioned. This was a big mistake because although we got correct results, the updating of that single value among threads was a bad idea. All the threads running the first iteration would be updating the value so the cache consistency was too bad to speed up the program. We solved this problem by creating another member function in the class landscape to not use the is_dry variable to avoid the race condition and low performance of cache.

3 Results

After these two major improvements, we got the expected speed-up when running the multithreaded version of our program. Here are the results of running our improved code against *measuremen_4096x4096.in* with 1, 2, 4, 8 threads respectively after 5 tries.

Num_threads	1	2	4	8
Time (s)	236.889	121.057	61.2809	33.5769

Note that the results have been averaged.

Thus, our implementation is able to produce scalable speed-up with multiple threads.

4 Side notes

- Please note that our code only does basic input argument sanitization.
- Our multi-thread implementation only allows the size of input dimension is no smaller than 32.
- Our multi-thread code is not defensive enough, it can be promised to work well if the input dimension is equal to 2^n, where n >= 5 and n is an integer.