Mingxiang Dai (md382), Jiawei Sun (js896)

ECE 565 - Fall 2020 Assignment # 5

1: Sequential code

1. Core class

```
class Landscape{
private:
    int** elevations;
    double** water;
    double** change;
    double** absorbed;
    int N;
    int steps;
    double absorption_rate;
    void new_drop(int i, int j);
    void trickle(int i, int j);
    void absorb(int i, int j);
    bool check_dry();
public:
    int finished_steps;
    Landscape(int steps, double absorption_rate, int N, std::string path);
    void simulate();
    void printResult() const;
};
```

2. Fields of the class

- elevations: the elevation array from input file
- water: keep track of the change of water
- change: used as a temp array for updating water
- absorbed: sum up all water each point absorbed
- N: dimension
- steps: the number of steps during which one full drop of rain will fall to each point
- absorption rate: the absorption rate
- finished_steps: the final total steps when all points are dry

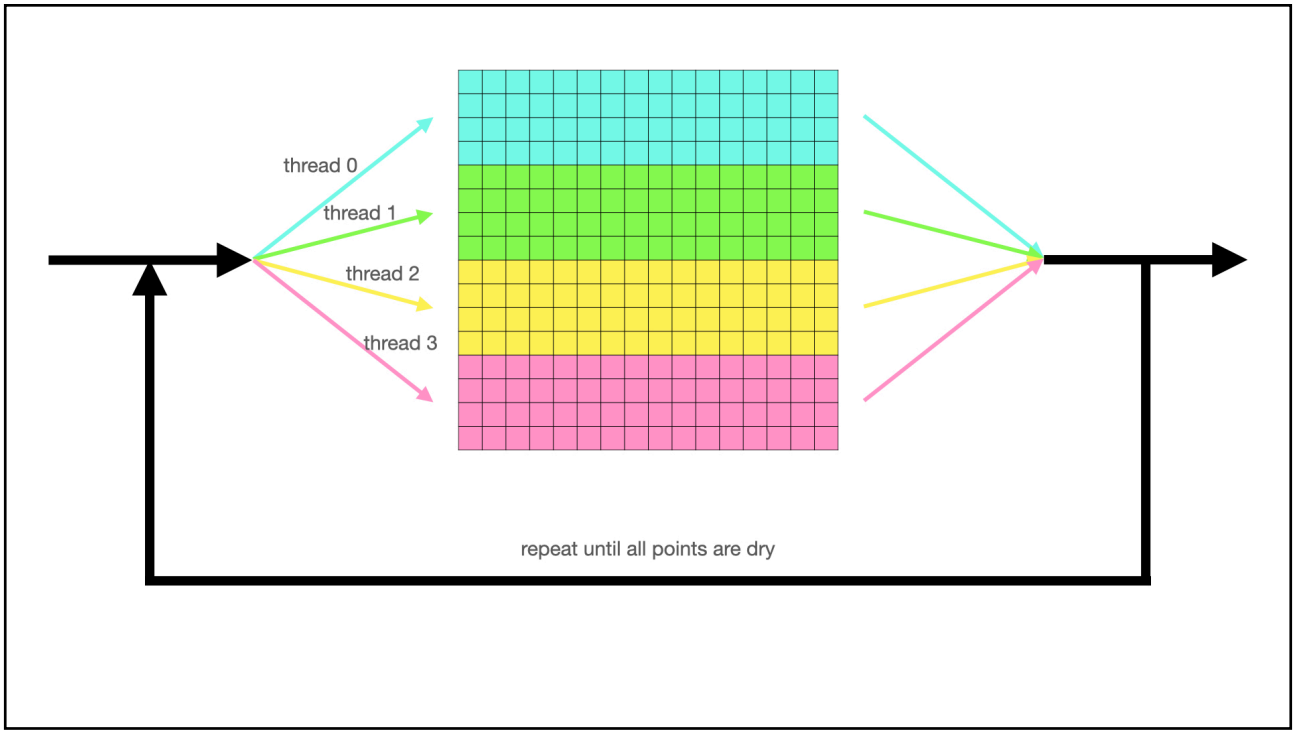
3. Key methods

- void new_drop(int i, int j): add 1 to water[i][j]
- void absord(int i, int j): calculate how much water each point should absorb
- void trickle(int i, int j): calculate how much water will trickle to its neighbors
- void simulate(): run the three methods above in order
  (1) Add one to each point
  (2) Calculate the absorption amount. Abstract water[i][j] by the corresponding amount and add absorbed[i][j] by corresponding amount.
  (3) Calculate the trickle amount. Get the change array of each step and update the water array.

2. Parallel Code

1. Strategy

We use multi-threads to calculate and update different parts (of lines) of the array (take a 4-thread program as an example):



The thread with number t_id should be in charge from line[t_id*(N num_threads)] to line[(t_id 1) (N this->num_threads)]-1.

2. Implemention

The high-level thought is the same as the sequential one. However, we must carefully make sure that there is no data race between different thread and the function the checks if all the points are dry should run when all threads are joined.

Here is the how we update the water array:

```
while (!landscape.check_dry()){
    for (int i = 0; i < num_threads; i++) {
    threads[i] = std::thread(&Landscape::reset_change, &landscape, i);
    }
    for (int i = 0; i < num_threads; i++) {
      threads[i].join();
    }
    for (int i = 0; i < num_threads; i++) {
    threads[i] = std::thread(&Landscape::drain, &landscape, i);
    }
    for (int i = 0; i < num_threads; i++) {
      threads[i].join();
    }
    for (int i = 0; i < num_threads; i++) {
    threads[i] = std::thread(&Landscape::water_change, &landscape, i);
    }
    for (int i = 0; i < num_threads; i++) {
      threads[i].join();
    }
    landscape.add_one_step();
}
}
```

Before each step, we make sure that the change array are reset to zero. Then the drain method should calculate the amount that the current step will update. Finally, the water_change method do the actual update.

Besides the execution sequence, we need to add some locks to some critical data, for example, the change array:

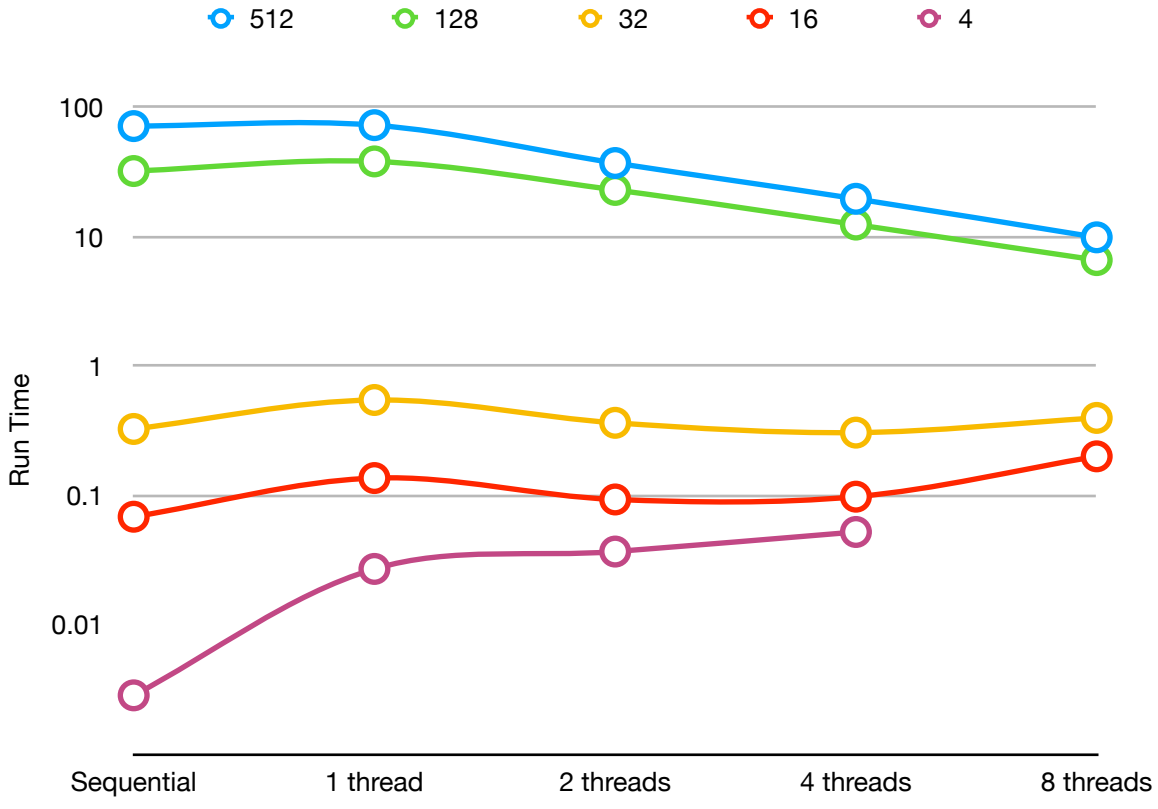
```
//north
            case 0:
                pthread_mutex_lock(&this->locks[i - 1]);
                change[i - 1][j] += amt/cnt;
                pthread_mutex_unlock(&this->locks[i - 1]);
                break;
            //south
            case 1:
                pthread_mutex_lock(&this->locks[i + 1]);
                change[i + 1][j] += amt/cnt;
                pthread_mutex_unlock(&this->locks[i + 1]);
                break;
            //west
            case 2:
                pthread_mutex_lock(&this->locks[i]);
                change[i][j-1] += amt/cnt;
                pthread_mutex_unlock(&this->locks[i]);
                break;
            //east
            default:
                pthread_mutex_lock(&this->locks[i]);
                change[i][j+1] += amt/cnt;
                pthread_mutex_unlock(&this->locks[i]);
                break;
            }
```

3. Results and Analysis

1. Results

	Sequential	1 thread	2 threads	4 threads	8 threads
4	0.00283017	0.0268984	0.0366473	0.0519431	
16	0.068156	0.135718	0.0926832	0.0974058	0.199826
32	0.325408	0.545038	0.362957	0.30489	0.396136
128	32.1756	38.1849	22.9808	12.3986	6.58752
512	71.6516	72.8262	37.109	19.6193	9.87822
2048*	?	?	?	?	?
4096*	?	?	?	?	?

*We keep getting a “segment fault (core dump)” error when we try to run our algorithm on bigger input (the 2048 one and the 4096 one). At first, we thought this is maybe because of the data structure we use, which is any array allocate in heap. Then we tried to change our store structure to vector in stack and vector in heap, but none of them worked.



2. Analysis

Note that the chart above is shown is log scale.

As it is shown in the results, all the program with 1 thread takes more time than the sequential one. The parallel version does worse on all small inputs. But there is about a 5~8x speedup on large inputs such as 512x512 and 128x128. This is somewhat expected due to the overhead introduced by the parallelism in the code in terms of initial thread, lock initialization, synchronization, etc.

When this input size is small or when the number of thread is not big enough, the benefit provided by the multithread is overwhelmed by the overhead. When the input size are increasing, overhead remains about the same level, but the benefit becomes more and more clear.

Overall, the results are not very surprising, but we do have some concrete details on what data size benefit the most from parallelism.