ECE 565 - Fall 2020

Assignment #5 Parallel Programming with Pthreads

Team members: Yi Mi (**ym154**) Zewen Peng (**zp38**)

1. Sequential version

For algorithm, we followed the execution flow required in the assignment, which is:

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.

For data structures, we defined a class called ***Landscape***, which have four main private class members:

1. **Datamap** rainmap (stores current amount of raindrops on each point in the landscape)

2. **Datamap** nextTrickleMap (stores for each land point the numbers of raindrops calculated in step 3a)

3. **Datamap** absorbedMap (stores the total absorbed amount of raindrops for each point)

4. **Vectormap** directions (a data structure to store the information about trickle directions (details below))

As is shown above, these class members are of two custom class type **Datamap** and **Vectormap**.

Datamap:

A 2D array of doubles to store a single value for each point on the landscape, implemented with std::vector. The meanings of doubles differ for different members of class ***Landscape***, which were discussed above.

Vectormap:

When handling trickle steps, we considered creating a new class 'VectorMap' to hold the directions each point is going to trickle to while initializing our Landscape.

There are several different fields used to hold information:

1. X is a 3D vector of int used to hold horizontal trickling coordinates for each point in the land.

2. Y is a 3D vector of int used to hold vertical trickling coordinates for each point in the land.

3. trickleMatrix is a matrix of bool to hold if this point is going to trickle or not.

X and Y should be correspondents. Each time we initialize a new Landscape, we pass in the elevation matrix and calculate the lowest elevation among four directions and then create an instance of VectorMap. Therefore, in the following traversals, we can call the class methods to directly know if the point trickles or not and get the trickle directions, which will save time efficiently.

2. Parallelized version

We divided the matrix of landscape by rows, which means each thread is responsible for certain amounts of rows of the landscape. To calculate the rows for each thread, we simply divide the number of rows by the number of threads. Each thread goes through the same workflow on its respectable rows as the serial version. Because there is no evident difference of work across rows, the work is balanced among threads.

As for synchronization, there is no data race across rows for the first traversal, so there is no explicit synchronization for it. For the second traversal, however, there will be data race on the boarder of regions handled by different threads. When trickling raindrops, the update of rain amount on these boarder elements must be atomic. Thus, we assigned for each matrix element a mutex, the threads must acquire the corresponding mutex to update the value for an element. In other words, we maintained a matrix of mutexes the same dimensions as the landscape.

The other major synchronization we implemented is the barrier between the first traversal and the second one. This synchronization is needed because we must finish calculating all the amount of rain drops to trickle before we really start to trickle them, which is a true dependency. We implemented this synchronization by joining all the threads created for the first traversal.

We have thought of an alternative way of parallelizing the second traversal using Red-Black SOR (Successive Over Relaxation), because the trickle operation is similar to a 4-point stencil. The advantage of this parallel strategy is that we will not need to use mutex at all, since the interleaving operations on the black cells and red cells have no dependency on each other. The disadvantage is that we will at most achieve a 2X speedup on the second traversal. As for the mutex strategy, although there might be some contentions, it will not hurt performance a lot since these contentions are not that frequent, thus we can surely achieve more than 2X speedup when applying the mutex strategy. After the comparison made above, we finally chose the mutex strategy for parallelizing the second traversal.

3. Measurement

execution time (: seconds) among different threads number and speedup for 4096x4096

	Sequential	1 thread	2 threads	4 threads	8 threads
4x4	0.00007	0.01640	0.02186	0.03041	0.00007
16x16	0.00142	0.02449	0.03359	0.04331	0.10964
128x128	0.55729	1.18287	0.87891	0.55509	1.04483
512x512	1.52002	2.46142	1.58038	0.81609	0.54662
2048x2048	67.02913	82.07940	46.18433	26.66883	16.22793
4096x4096	407.58067	467.90200	245.44267	133.38067	73.44890
speedup	1.00000	0.87108127	1.66059419	3.0557702	5.54917319

4. Discussion

We were able to obtain decent number of speedups among different thread number, although it is not entirely proportional to the thread number. This exactly match what we expected.

For the speedup part, this whole workflow of rain simulation is almost closed to embarrassingly parallel, there is not much data race or contention. Thus, we have expected decent speedup and it really did.

For the not proportional part, the barrier between the two traversals is the main reason for the slowdown. Because the main thread must wait all the threads finish before starting on the second traversal for the correction of simulation results. The second reason is mutex contentions, although it is not frequent, there might be times when two threads wants to update the same elements on the matrix, which can cause slowdown. The minor reason might be the overhead of maintaining and accessing mutexes, the overhead of creating threads, which could also contribute to a little amount of slowdown.