ECE 590.02
Fall - 2016

Homework #5 - Pthreads Parallel Programming

Xiaoshuang Yin | Tananun Songdechakraiwut

*Contribution note: We work together through all parts

Rainfall Simulation

3. Describe your parallelization strategy:

- **Describe your sequential algorithm for the rainfall simulation, and how you chose to parallelize it:**

Precisely, our sequential algorithm steps are the following:

1) Parse user inputs including M, A, N, and elev_file.
2) Start initialize a matrix, which is essentially a 2D array of typedef struct point. Point structure is defined as following:

```
typedef struct {
 int elev;
 vector< pair<int, int> > lowest_neibrs;
 int num_lowest_neibrs;
 double drop_abs;
 double drop_remained;
 double drop_to_trickle;
}point;
```

3) Get line from the elev_file
4) Call find_neighbors, which finds the lowest neighbors for each point in matrix, and stores it into `vector< pair<int, int> > lowest_neibrs;` as well as initialize other fields including `int num_lowest_neibrs;`
5) Call rain_fall(...) function to execute the algorithm. It first initialize drop_sum to 0. This drop_sum will get updated as follows:

```
 if (M > 0){
    drop_sum += N * N;
  }
```

Then, it organizes exactly like mentioned in the HW5 instruction, consisting of two traversals. First traverses through all points and do 1) receive one rain drop if M > 0

2) absorbing water of A when double drop_remained; >= A else absorb what left.
Then we calculate water to trickle as follows:

```
if (matrix[i][j].drop_remained >= 1){
 matrix[i][j].drop_to_trickle = 1;
 matrix[i][j].drop_remained -= 1;
}
else{
 matrix[i][j].drop_to_trickle = matrix[i][j].drop_remained;
 [i][j].drop_remained  = 0;
}
```

6) Now we execute stop condition; algorithm stops when drop_sum <=0 and M <=0
7) Decrement M, then do the second traversal, which updates double drop_remained; of the lowest neighbor(s) - see HW5 instruction for details
8) We print output according to the format. That's it!

How you chose to parallelize it?

Our goal was to parallelize portions of the program that contain loops. As the first step towards the goal, we observed that there are the following portions matter to performance optimization:

1. loop inside find_neighbors()
2. two traversals inside rainfall() function

As you can see, these loops depend on N. When we traverse on really large matrix, total execution time will largely depend on these portions of the code. Thus, we decided to parallelize them using pthreads & mutex.

- **Discuss what parts of your simulation code you parallelized:**

As mentioned, we parallelize two portions including:

1. Loop inside find_neighbors()
2. Two traversals inside rainfall()

The way we parallelize them are as follows:

1. For loop inside find_neighbors(), it found that it exhibits DOALL parallelism. The reason is because vector< pair<int, int> > lowest_neibrs; and int num_lowest_neibrs; of each point in a matrix is independently updated by reading int elev; from its surrounding neighbors. Since int elev; doesn't get updated, it is safe to conclude that Loop inside find_neighbors() exhibits DOALL parallelism. What we did was to divide loop iterations equally across each pthreads when assigning task to each one.

2. To parallelize two traversals, we had to create a dedicated function for each traversal, since they both located inside the same rain_fall function. Thus, we created two functions: RecAbsTric() and updateTrickle(), respectively. Now that we have each function separately, we can assign threads to work on those functions and distribute loop iterations equally just like what we did to find_neighbors(). But since both don't exhibit DOALL parallelism, we had to use synchronization object to handle this. We used pthread_mutex, in this case.

   a) **RecAbsTric()**
   After analysis, we found that only drop_sum global variable needs to be protected inside RecAbsTric(). The reason is because each pthread only read its corresponding points in a matrix (so matrix is read-only). So we create a single mutex object to lock drop_sum, and call pthread_mutex_lock every time a thread wants to update drop_sum.

   b) **updateTrickle()**
   After analysis, we found that `double drop_remained;` makes a matrix a read-write conflicting variable, thus we had to look it with pthread_mutex_lock. Since each point need to get updated safely one at a time, we create a 1D array of locks corresponding to each point. The formula to calculate which lock to use for each point is as follow:
   $x\ coordinate \times N + y\ coordinate$
   So we call a corresponding lock each time we want to update `double drop_remained;` of each neighbors for a particular point.

- **Why did you choose those parts to parallelize in comparison to other parallelization strategies you considered?**

  Because those parts are most time consuming. From the flat profile obtained from running sequential rainfall program Gprof, the most time consuming function is **rainfall(),** which takes 99.96% of the execution time, the second most time consuming function is **find_neighbors()**, which takes 0.14% of the execution time. Thus, we chose to parallelize those two functions.

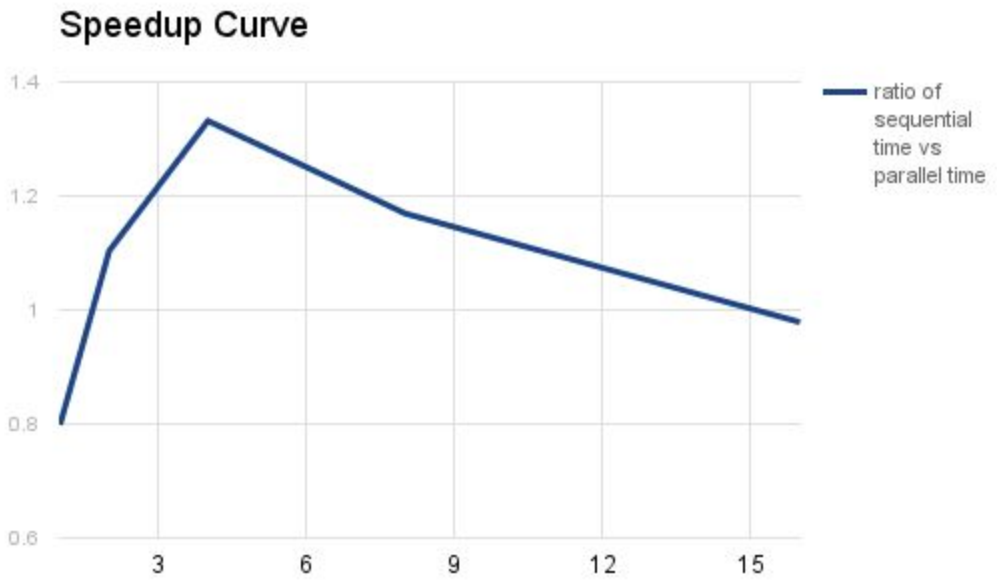
```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 99.96    889.23   889.23                             rain_fall(point**, double, int)
  0.14    890.45     1.22                             find_neighbors(point**)
  0.14    891.66     1.20 67092484    17.93    17.93  void std::vector<std::pair<int, st
d::pair<int, int> > const&>(std::pair<int, std::pair<int, int> > const&)
  0.01    891.79     0.13 17446551     7.47     7.47  void std::vector<std::pair<int, in
st&)
  0.00    891.79     0.00        1     0.00     0.00  _GLOBAL__sub_I_N
```

- **What types of synchronization across threads did you use?**

  As mentioned earlier, we use pthread_mutex_t and we initialize it with PTHREAD_MUTEX_INITIALIZER. We created a single mutex for drop_sum, which is used inside RecAbsTric(), and a 1D array of mutexes of size $N \times N$ to lock each point in a matrix.

4. Results

	Wall-clock execution time(ms)	Speedup Compare to sequential
Sequential	1049443.114544	1
Parallel with 1 thread	1314124.906288	0.7985870365
Parallel with 2 threads	950553.003064	1.1040342949433
Parallel with 4 threads	788052.254792	1.33169229345203
Parallel with 8 threads	897587.165584	1.16918239785793
Parallel with 16 threads	1072414.317632	0.97857991756514



- Describe your results. Were you able to obtain speedups?

  Yes. When the number of threads equals to 2,4,8 the program obtain speedup. When the number of threads equals to 4, the program obtain the maximum speedup which is 1.33. When the program has 16 threads, the program can not obtain speedup, the execution time is longer than the execution time of the sequential version of the program. When the number of thread is one, the program has gained speedup smaller than 1 and it's execution time is longest.

- Discuss your results, and how they matched or did not match what you expected.

  The result matches our expectation. Our expectation is :
  When there is only one thread, the execution time is longer than sequential code. Because of the overhead.

  When there are more than 1 threads, before certain point, the execution time is shorter than sequential code and the speedup will increase as the number of thread increases. Because the overhead is not too big and more threads to run portions of the program in parallel will make it run faster.

  When the number of threads reaches certain point, the speed up will begin to decrease. Because as the number of threads increases, the overhead increases: time spent on synchronization get increasingly larger.

  When the number of threads reaches that certain point, the execution time will be longer than sequential code.

- Describe possible reasons for why your results did or did not match expectations.

  The possible reasons are:

  1. When you add more threads, there is the operational cost such as acquiring and testing for locks for every synchronized method. This can impose a lot of overhead.
  2. Synchronization serializes execution of a set of statements so that only one thread executes at a time. Whenever several threads simultaneously try to execute the same synchronized block, those threads are essentially running together as one single

thread. This negates the purpose of having multiple threads in the first place.

Therefore, when we begin to run the program starting from 2 threads, there is less cost associated with synchronization as well as overhead cost, so the performance reasonably executes faster than the sequential. However, after a certain threshold, performance starts to degrade since costs of synchronization can be significant. Also, if there is only 1 thread, it makes sense that the program run slightly slower than the sequential since there is still an extra step of acquiring, testing for locks even only single thread exists.