HOMEWORK 5

Zihao Wu, Kaiwen Wang
zw154, kw284

Introduction

The rainfall program is running a simulation on a given landscape, measuring number of rain drop absorption to each unit point of the landscape. There is a chance to parallelize the simulation process. We have develop skills in parallelizing program using programming model OpenMP. This time is we will develop the parallelism using another programming model, pthread provided by POSIX, which is more compatible across platforms and intuitively has higher performance than OpenMP.

Abstract

We designed two data structures, `Point` represents every piece of ground and stores the information of this piece, `Simulator` represents the whole process. We developed two versions of this program, sequential and paralle, to study the performance improvement after using `pthread` library to parallel the core code segment. In this project, we analyzed two different performances to get a clear understanding of `pthread` and parallel programming.

Implementation

Object-Oriented Programming will be used our design.

Serial Rainfall Implementation

Two main classes in our design: Point and Simulator where Point represnets each entry on our landscape matrix and simulator holds those configuration parameters and the Points container.

Point Class

```
class Point {
   private:
    int x;
    int y;
    int height;
    double absorbRate;
    double totalAbsorption;
    double remain;
    double trickleAmount;
    vector<pair<int, int>> neighbors;
  public:
    //construct and destruct
    Point(int x_,
          int y_,
          int height_,
          double absorbRate_,
          vector<pair<int, int>> neighbors_);
    //getters
    vector<pair<int, int>>& getNeighobors();
    double getTrickleAmount();
    double getTotalAbsorption();
    double getRemain();
    //Processing point during simulation
    void receive();
    void absorb();
    void calculate();
    void update(double trickleAmount);
    bool isNotFinished() { return remain; }

    //used to test content in the Point object
    friend ostream& operator<<(ostream& stream, const Point& data);
}
```

- Constructor, getters, << operator overloading
- Utility function during simulation

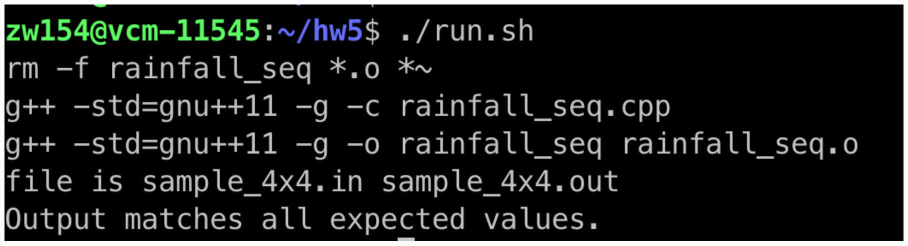
Simulator Class

```
class Simulator {
   public:
    //construct and destruct
    Simulator(int N_, int timeSteps_, double absorbRate_, int**
elevationMatrix);
    ~Simulator();
    // start the simulation and print result
    void run();
    void printResult();

   private:
    Point** points;
    int N;
    int timeSteps;
    double absorbRate;
    int totalTime;
    double elapsed_ns;

    //return vectors of coordinates indicating the set of lowest neighbors
    vector<pair<int, int>> findLowestNeighbors(int row, int col, int** matrix);
    void processDuringRain();
    void processAfterRain();
  // used to test container initialization
    void printPoints1();
    void printPoints2();
    void printNeighbors();

};
```

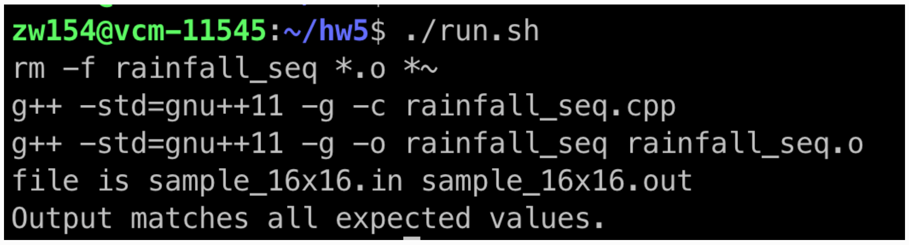
- Constructor, destructor (since the type of the container holds a pointer)
- run(), printResult() for simulation
- Some private methods are sub-process for the run() or implemented for debugging.

Test for correctness

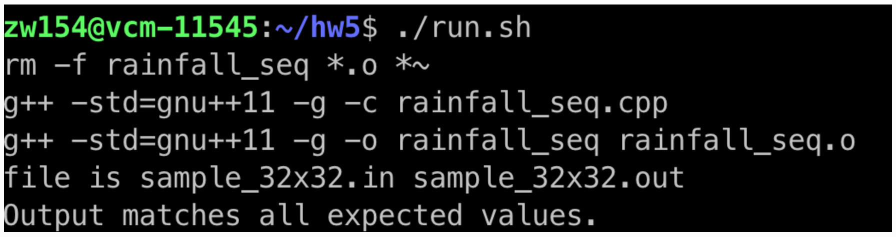
By running the `check.py <num of threads> <validation> <output_file>`



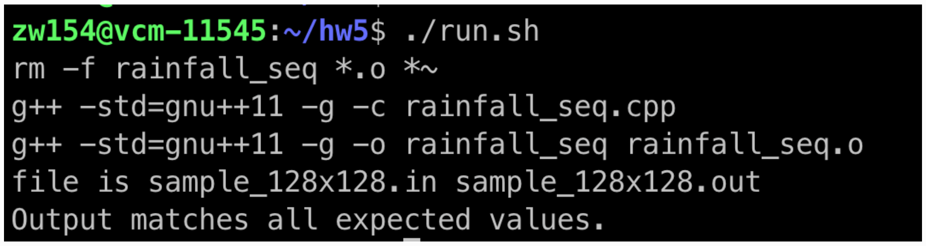
```
zw154@vcm-11545:~/hw5$ ./run.sh
rm -f rainfall_seq *.o *~
g++ -std=gnu++11 -g -c rainfall_seq.cpp
g++ -std=gnu++11 -g -o rainfall_seq rainfall_seq.o
file is sample_4x4.in sample_4x4.out
Output matches all expected values.
```



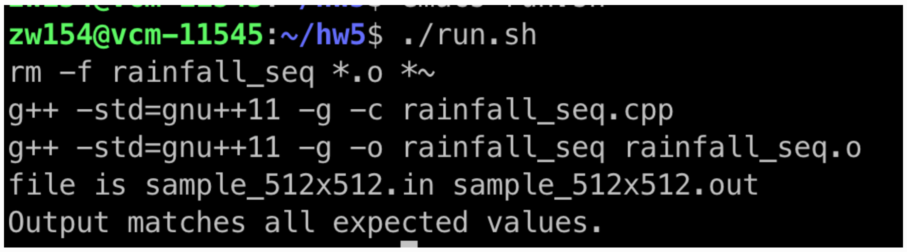
```
zw154@vcm-11545:~/hw5$ ./run.sh
rm -f rainfall_seq *.o *~
g++ -std=gnu++11 -g -c rainfall_seq.cpp
g++ -std=gnu++11 -g -o rainfall_seq rainfall_seq.o
file is sample_16x16.in sample_16x16.out
Output matches all expected values.
```



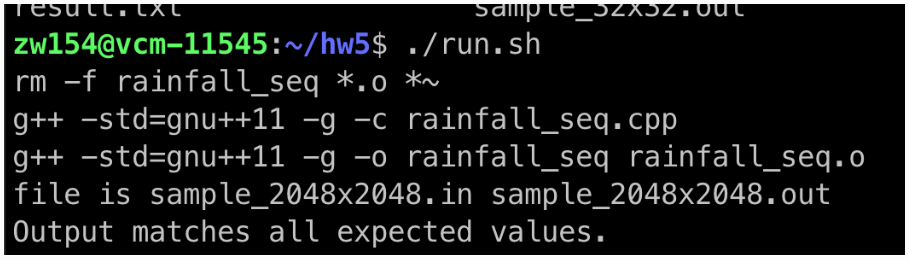
```
zw154@vcm-11545:~/hw5$ ./run.sh
rm -f rainfall_seq *.o *~
g++ -std=gnu++11 -g -c rainfall_seq.cpp
g++ -std=gnu++11 -g -o rainfall_seq rainfall_seq.o
file is sample_32x32.in sample_32x32.out
Output matches all expected values.
```



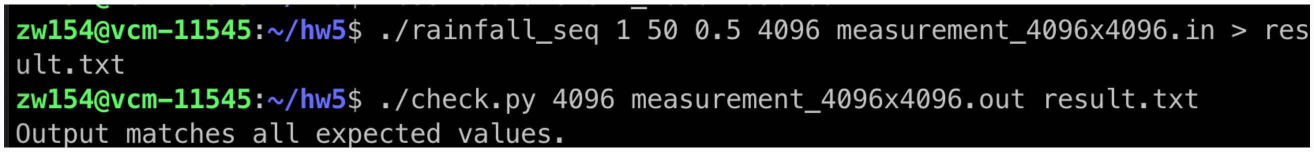
```
zw154@vcm-11545:~/hw5$ ./run.sh
rm -f rainfall_seq *.o *~
g++ -std=gnu++11 -g -c rainfall_seq.cpp
g++ -std=gnu++11 -g -o rainfall_seq rainfall_seq.o
file is sample_128x128.in sample_128x128.out
Output matches all expected values.
```



```
zw154@vcm-11545:~/hw5$ ./run.sh
rm -f rainfall_seq *.o *~
g++ -std=gnu++11 -g -c rainfall_seq.cpp
g++ -std=gnu++11 -g -o rainfall_seq rainfall_seq.o
file is sample_512x512.in sample_512x512.out
Output matches all expected values.
```



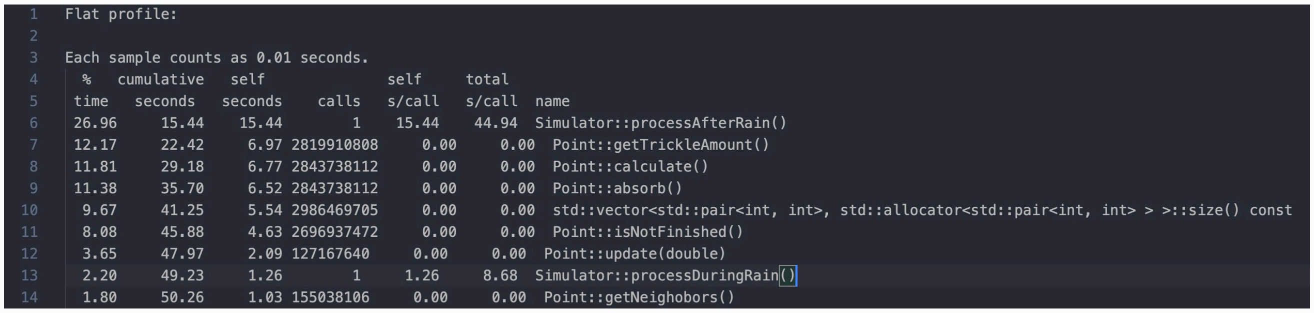
```
result.txt                    sample_32x32.out
zw154@vcm-11545:~/hw5$ ./run.sh
rm -f rainfall_seq *.o *~
g++ -std=gnu++11 -g -c rainfall_seq.cpp
g++ -std=gnu++11 -g -o rainfall_seq rainfall_seq.o
file is sample_2048x2048.in sample_2048x2048.out
Output matches all expected values.
```



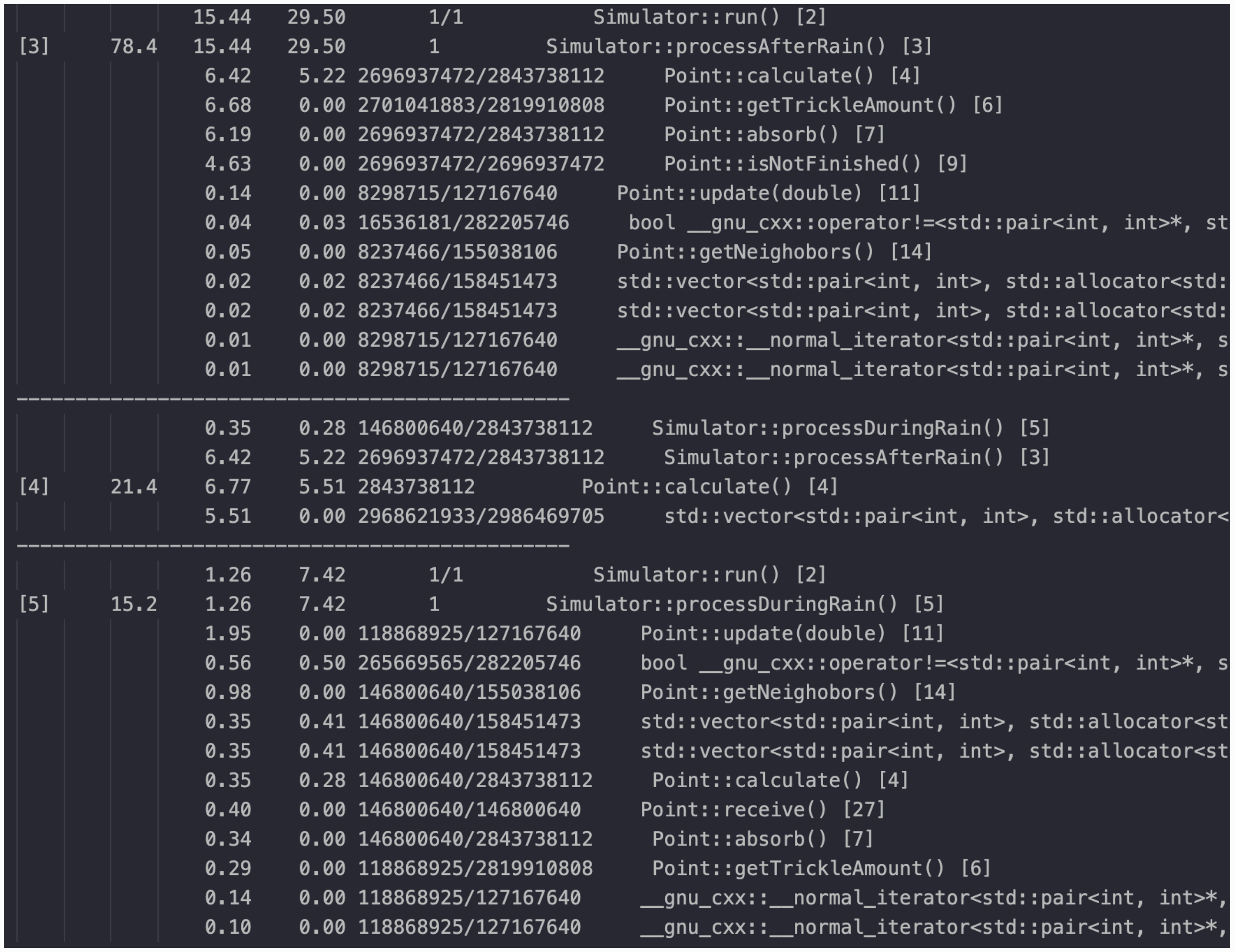
```
zw154@vcm-11545:~/hw5$ ./rainfall_seq 1 50 0.5 4096 measurement_4096x4096.in > res
ult.txt
zw154@vcm-11545:~/hw5$ ./check.py 4096 measurement_4096x4096.out result.txt
Output matches all expected values.
```

Profiling

```
$ gprof ./rainfall_seq gmon.out > analysis.txt
$ ./rainfall_seq 35 0.5 2048 sample_2048x2048.in
```



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 26.96     15.44    15.44        1    15.44    44.94  Simulator::processAfterRain()
 12.17     22.42     6.97 2819910808     0.00     0.00  Point::getTrickleAmount()
 11.81     29.18     6.77 2843738112     0.00     0.00  Point::calculate()
 11.38     35.70     6.52 2843738112     0.00     0.00  Point::absorb()
  9.67     41.25     5.54 2986469705     0.00     0.00  std::vector<std::pair<int, int>, std::allocator<std::pair<int, int> > >::size() const
  8.08     45.88     4.63 2696937472     0.00     0.00  Point::isNotFinished()
  3.65     47.97     2.09 127167640     0.00     0.00  Point::update(double)
  2.20     49.23     1.26        1     1.26     8.68  Simulator::processDuringRain()
  1.80     50.26     1.03 155038106     0.00     0.00  Point::getNeighobors()
```



```
                15.44   29.50       1/1           Simulator::run() [2]
[3]     78.4    15.44   29.50       1         Simulator::processAfterRain() [3]
                 6.42    5.22 2696937472/2843738112     Point::calculate() [4]
                 6.68    0.00 2701041883/2819910808     Point::getTrickleAmount() [6]
                 6.19    0.00 2696937472/2843738112     Point::absorb() [7]
                 4.63    0.00 2696937472/2696937472     Point::isNotFinished() [9]
                 0.14    0.00 8298715/127167640     Point::update(double) [11]
                 0.04    0.03 16536181/282205746     bool __gnu_cxx::operator!=<std::pair<int, int>*, st
                 0.05    0.00 8237466/155038106     Point::getNeighobors() [14]
                 0.02    0.02 8237466/158451473     std::vector<std::pair<int, int>, std::allocator<std:
                 0.02    0.02 8237466/158451473     std::vector<std::pair<int, int>, std::allocator<std:
                 0.01    0.00 8298715/127167640     __gnu_cxx::__normal_iterator<std::pair<int, int>*, s
                 0.01    0.00 8298715/127167640     __gnu_cxx::__normal_iterator<std::pair<int, int>*, s
-----------------------------------------------
                 0.35    0.28 146800640/2843738112     Simulator::processDuringRain() [5]
                 6.42    5.22 2696937472/2843738112     Simulator::processAfterRain() [3]
[4]     21.4     6.77    5.51 2843738112         Point::calculate() [4]
                 5.51    0.00 2968621933/2986469705     std::vector<std::pair<int, int>, std::allocator<
-----------------------------------------------
                 1.26    7.42       1/1           Simulator::run() [2]
[5]     15.2     1.26    7.42       1         Simulator::processDuringRain() [5]
                 1.95    0.00 118868925/127167640     Point::update(double) [11]
                 0.56    0.50 265669565/282205746     bool __gnu_cxx::operator!=<std::pair<int, int>*, s
                 0.98    0.00 146800640/155038106     Point::getNeighobors() [14]
                 0.35    0.41 146800640/158451473     std::vector<std::pair<int, int>, std::allocator<st
                 0.35    0.41 146800640/158451473     std::vector<std::pair<int, int>, std::allocator<st
                 0.35    0.28 146800640/2843738112     Point::calculate() [4]
                 0.40    0.00 146800640/146800640     Point::receive() [27]
                 0.34    0.00 146800640/2843738112     Point::absorb() [7]
                 0.29    0.00 118868925/2819910808     Point::getTrickleAmount() [6]
                 0.14    0.00 118868925/127167640     __gnu_cxx::__normal_iterator<std::pair<int, int>*,
                 0.10    0.00 118868925/127167640     __gnu_cxx::__normal_iterator<std::pair<int, int>*,
```

After profiling, we will draw a conclusion in focusing our effort on Simulator::processAfterRain(), and Simulator::processDuringRain() those takes top time consumption except main() and Simulator::run() which is the general processing method.

Parallel Rainfall Implementation

We partitioned the whole process by two parts: during rainfall and after rainfall. Each part has another two parts:

- During Rainfall:
  1. Each piece of ground receives a new rain drop, and absorbs a fixed amount of water based on given absorption rate. Then the particular piece will calculate and store the amount of rain it should pass to its lowest neighbor/neighbors.
  2. Each piece of ground will execute the operation of trickling the specific amount of rain to its lowest neighbor/neighbors.
- After Rainfall:
  1. Each piece of ground absorbs a fixed amount of rain, and calculates the amount of water it should trickle to its lowest neighbor/neighbors. Then check if this piece is finished in the whole process. If there's any piece of ground is not finished, the whole

process will continue.
2. Each piece of ground will execute the operation of trickling the specific amount of rain to its lowest neighbor/neighbors.

Parallel Parts

Since we have four small parts of code, and each part requires to traverse the whole matrix, then these traversals are the main areas that we focus on optimization.

Reason

We used profiler to find that these parts of code cost the most time of the whole program. As we can see, the `Simulator::processAfterRain()` and `Simulator::processDuringRain()` have only been called once for each function. However, these two functions cost 93.6% of all time.

Method

We used `pthread` library to initialize threads based on the number given by users.

We divided the matrix by row so that each thread would control the same number of matrix parts. We designed two structures to store arguments that we want to pass into each thread.

```
// the lower and upper bound of matrix that the thread should handle
struct Arg {
    int start;
    int end;
};

// the argument pass into threads
struct BasicArg {
    void* obj;
    Arg arg;
};

// four parts we mentioned above
void processDuringRain1(Arg arg);
void processDuringRain2(Arg arg);
void processAfterRain1(Arg arg);
void processAfterRain2(Arg arg);

// functions that assigned to threads
// each function calls a part
void* worker1(void* basicArg);
void* worker2(void* basicArg);
void* worker3(void* basicArg);
void* worker4(void* basicArg);
```

Synchronization

Reason

Each piece of ground will trickle the remain water to its lowest neighbor/neighbors. And there's chance that some pieces may need to update its neighbor that belongs to other ground. That is when we need synchronization.

Method

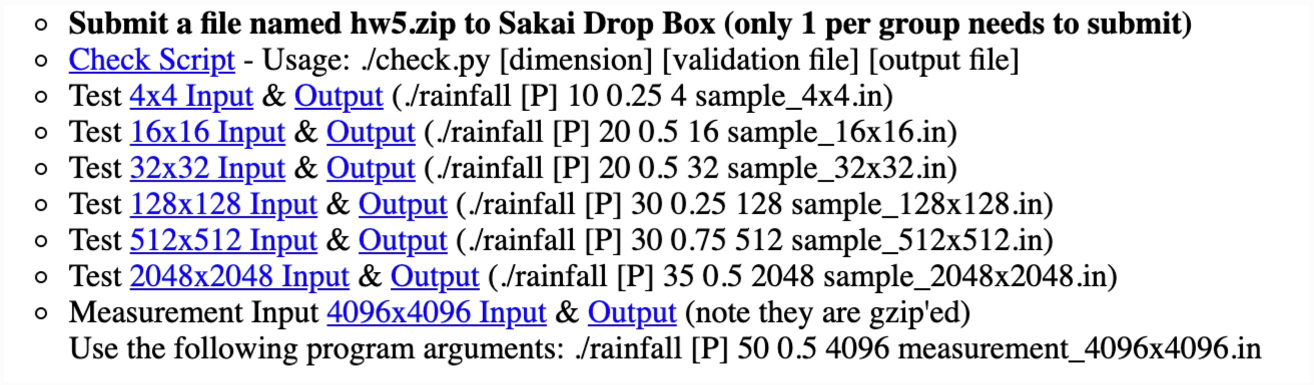
In the beginning of the program, we initialize a matrix of `pthread_mutex_t` for every piece of ground. And we only lock those pieces which are at the upper and lower boundary rows of each thread.

```
if (coord.first == arg.start || coord.first == arg.end - 1
  || coord.first == arg.start - 1 || coord.first == arg.end) {
    pthread_mutex_lock(&locks[coord.first][coord.second]);
    points[coord.first][coord.second].update(curr->getTrickleAmount());
    pthread_mutex_unlock(&locks[coord.first][coord.second]);
}
```

In this way, we don't need to lock a whole row when updating critical points, so that other threads can still update irrelevant points on this row.

benchmark result

Testing the simulator on commands:

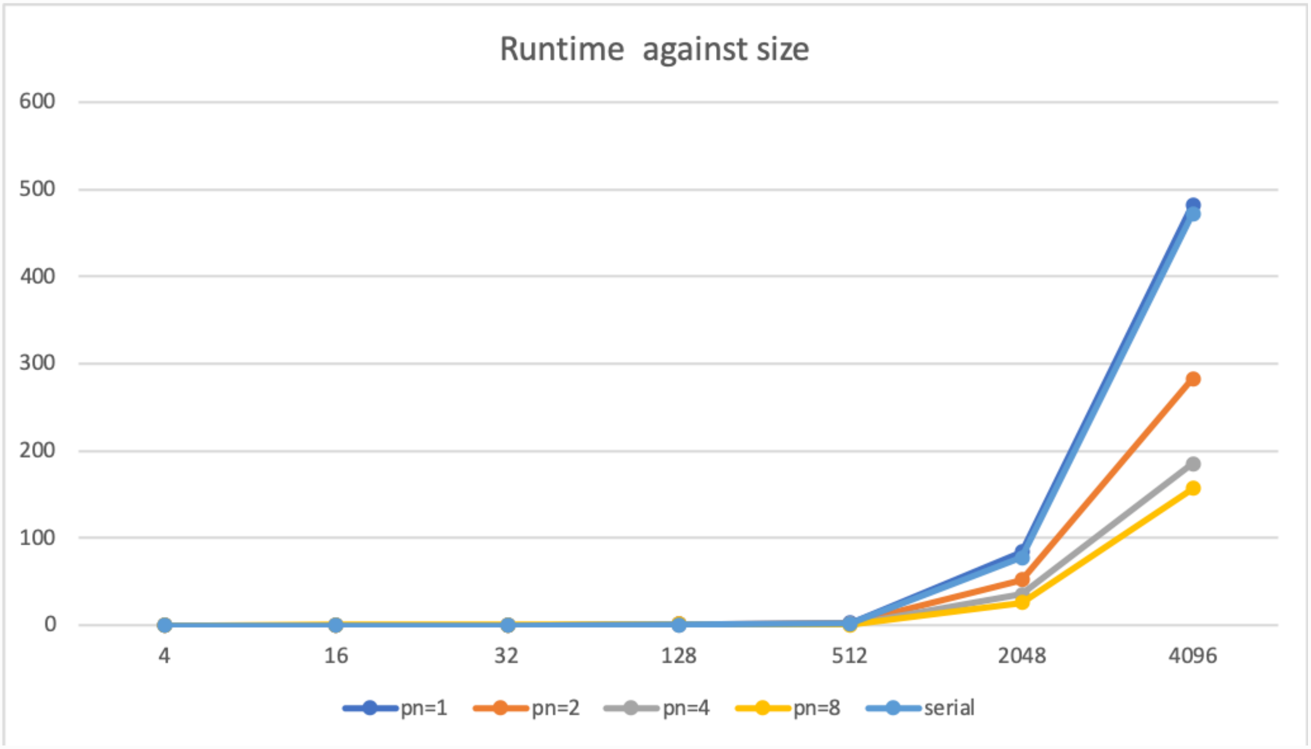


- **Submit a file named hw5.zip to Sakai Drop Box (only 1 per group needs to submit)**
- Check Script - Usage: ./check.py [dimension] [validation file] [output file]
- Test 4x4 Input & Output (./rainfall [P] 10 0.25 4 sample_4x4.in)
- Test 16x16 Input & Output (./rainfall [P] 20 0.5 16 sample_16x16.in)
- Test 32x32 Input & Output (./rainfall [P] 20 0.5 32 sample_32x32.in)
- Test 128x128 Input & Output (./rainfall [P] 30 0.25 128 sample_128x128.in)
- Test 512x512 Input & Output (./rainfall [P] 30 0.75 512 sample_512x512.in)
- Test 2048x2048 Input & Output (./rainfall [P] 35 0.5 2048 sample_2048x2048.in)
- Measurement Input 4096x4096 Input & Output (note they are gzip'ed)
  Use the following program arguments: ./rainfall [P] 50 0.5 4096 measurement_4096x4096.in

Measure runtime in seconds.

size	serial	# of threads=1	# of threads=2	# of threads=4	# of threads=8
4	0.000183055	0.0180515	0.0225215	0.0385853	0.0390366
16	0.00195519	0.0338951	0.0396985	0.0432577	0.187231

32	0.00617794	0.0518847	0.0478158	0.0613169	0.182695
128	0.537122	1.43511	1.06042	0.557307	1.16401
512	1.58023	2.70222	1.63165	1.05047	0.731575
2048	77.7396	84.5912	51.9928	35.5148	25.6884
4096	472.362	482.769	282.792	185.489	156.923



Analysis

Observation 1

From size = 4 to 32, runtime goes up with number of threads

At size = 128, the minimum runtime occurs at number of threads = 4.

From size = 512 to 4096, runtime goes down with number of threads

Analysis 1

For size = 4 to 32, the matrix is small enough to achieve a good performance without parallelism. The number of threads is close the size of matrix. By adding threads and locks, there comes more overhead time, delivering worse performance.

For size = 512 to 4096, the number of threads is far less than row count of matrix. Threading the matrix would have better performance as time overhead in thread creation and mutex behavior is relatively insignificant compared to traversal of large size matrix.

For size = 128, an equilibrium point denotes tradeoff between benefits from parallelism and overhead in threading and synchronization.

Observation 2

Serial only beats number of threads = 1 a little bit

Analysis 2

Both serial and number of threads = 1 do not take any benefit from parallelism. However, number of threads = 1 has time overhead in thread creation and mutex behavior.