# ECE 565: Performance Optimization & Parallelism
# Homework 5

**Prathikshaa Rangarajan (pr109)** **Rijish Ganguly (rg239)**

November 6, 2019

# 1 Sequential Code Organization

The main data structures used in the code are wrapped in a struct defined using `sim_data` that is declared in main on the heap. This pointer to the data on the heap is passed onto every subsequent function call and is not declared as a global variable.

```
struct simulation_struct
{
    int P; // num_threads
    int M; // rain steps
    int num_steps; // total simulation steps
    float A; // absorption
    int N; // landscape size
    int **landscape; // landscape array - input
    float **current_rain; // keep track of rain through simulation
    float **trickle; // keep track of trickle in each time-step
    float **rain_absorbed; // rain absorbed in each tile output
    const char *elevation_file; // name of input file
} typedef simulation;
```

Some general purpose functions used:

```
double calc_time(struct timespec start, struct timespec end);
// For timing data
void print_data(FILE* stream, int N, float **data_struct);
// prints any float 2D array of size N to the specified file
void usage(const char *prog_name);
// print usage message
size_t str_to_num(const char *str);
// error checking string->invert conversion wrapper around strtoul()
float str_to_float(const char *str);
// error checking string->float wrapper around strtof()
int get_nums(int size, const char *line, int *landscape_row);
// reads the numbers from the string line provided and writes them
```

High-level steps in main to accomplish the task:

```
read_landscape(sim_data); // reads the input file into landscape array
run_simulation(sim_data); // does the actual simulation
write_result(sim_data);   // writes out the result
```

The function `run_simulation()` in turn does these tasks on a high-level:

```
// Loop until rain stoppped and all water is absorbed - using all_absorbed() to check
// In the loop:
calculate_trickle(sim_data, rain_drop=1/0);
// writes to sim_data->trickle array, rain_drop is based on whether it is still raining
update_trickle(sim_data);
// updates sim_data->current_rain array based on sim_data->trickle array
```

The `calculate_trickle()` function uses a little too many local variables such as `north`, `south`, `east`, `west` and `north_trickle`...`east_trickle` which could've been saved in a less verbose 4 element array. But this was implemented for clarity and readability when implemented. This part of the code also has several conditional statements in order to determine where the water should trickle to. There may be potential case overbloat that could be analyzed and optimized further.

# 2 Parallelization Strategy

In order to identify the code that would best benefit from paralellism, we first profiled the code to identify pertinent functions. The expectation was that the `run_simulation()` would be the most time consuming task of the operation. The profiler matched that expectation while giving additional insight that the first iteration within the `run_simulation()`, i.e. the `calculate_trickle()` was in fact the most time consuming as seen in Fig.1.

```
core@vcm-11541:~/hw5/rainfall$ gprof rainfall_seq gmon.out |head -n 2
0
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 82.27     65.70    65.70     1041    63.11    63.11  calculate_trickle
 15.51     78.09    12.39                             run_simulation
  2.23     79.87     1.78                             write_result
  0.06     79.92     0.05                             read_landscape

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

core@vcm-11541:~/hw5/rainfall$
```

Figure 1: Profile of the sequential code after running it on the 4096 file

This narrowed our approach from parallelizing all of `run_simulation()` to just focusing on `calculate_trickle()`. Considering this information and with the objective of improving performance without drastically increasing the synchronization overhead, we made two pertinent decisions:

- The task distribution between threads would be the rows of the matrices. This is due to the observation that operating on the rows of a matrix would improve the data locality for the thread. Also, breaking up the labour based on rows would reduce the data contention between threads and help reduce synchronization and conflict.
- The number of threads actually used/created in the program will be limited to the number of rows in the matrices. For instance, the 4x4 file would not benefit from breaking up by more than 4 rows.

## 2.1 Other discarded ideas

As discussed, the first instinct to parallelize the code was to create threads in the entire `run_simulation()` function. In fact, this was even attempted, but discarded due to the high complexity in synchronization and maintaining correctness of code since not everything in the function is parallelizable. Some parts of it should in fact be executed by only a single/master thread. This strategy included using `pthread_barrier_wait` as additional synchronization in between `calculate_trickle()` and `update_trickle()` to maintain the determinism of the results.

## 2.2 Implementation

As described in Sec.1, `run_simulation()` calls the functions `calculate_trickle()` and `update_trickle()`. Since the identified function for parallelization opportunity was `calculate_trickle()`, we wrote a wrapper function around it that would create the requested number of threads and break down the labour.
Additional structs include:

```
struct calc_trickle_args_t {
    simulation *sim_data;
    int *thread_id;
    int rain_drop;
}typedef calc_trickle_args;
```

This is the structure used to pass in arguments to the `void *thread_calc_trickle(void *arguments)` function used in `pthread_create()`. This function is a thin wrapper around the actual `calculate_trickle()` updated in the parallel version to take some additional arguments including a bounds array that contains the lower and upper row bounds as `bounds[0]` for min row index and `bounds[1]` for the upper row index.

Special functions used in the parallel version of the code:

```
void read_landscape(simulation *sim_data);
// same as sequential version

------------ START OF PARALLELSIM ------------
int parallel_calculate_trickle(simulation *sim_data, int rain_drop);
// function called from run_simulation() in place of original calculate trickle
// It also generates a pool of threads on the thread_calc_trickle() below and destroys them
    after

void *thread_calc_trickle(void *arguments);
// Thin wrapper on the modified calculate_trickle() to process arguments and pass them to
    the function

int calculate_trickle(int * bounds, simulation *sim_data, int rain_drop);
// Modified version of seq function of same name
// Takes the bounds as additional information to determine the lower and upper bounds of the
     rows the function operates on
// 1) Add new rain, if raining
// 2) Absorb rain
// 3) Calculate trickle to neighbours
// The above steps are done by each thread for a given set of rows on the landscape,
    current_rain, rain_absorbed
// trickle is updated for neighbours so needs to be protected using synchronization

void get_bounds(simulation *sim_data, int thread_id, int *bounds);
// get the lower and upper bounds of the rows thread can operate on based on thread_id
------------- END OF PARALELLISM -------------
void run_simulation(simulation * sim_data);
// The function that calls parallel_calculate_trickle()

void update_trickle(simulation *sim_data);
// update trickle in current_rain using trickle array - same as seq
int all_absorbed(simulation *sim_data);
// check if simulation should stop - same as seq
void write_result(simulation *sim_data);
// write out result - same as seq
```

### 2.3 Synchronization Technique

In order to manage the contention for data between threads at row boundaries, row locks were implemented when updating the `sim_data->trickle` data structure (since the trickle flows to neighbours and they must be updated). Whenever a thread updates a value either near the bounds of the rows it owns, it acquires a lock to perform the update.

This was implemented using `pthread_mutex_t` locks.

## 3 Testing

Included in the code structure are scripts that can be used for testing. They make use of the provided `check.py` to generate the the output files and then test them against the validation output files.

One may find the `hw5/run_all.sh` and `hw5/pt_run_all.sh` useful to test the serial and parallel versions of the code respectively. The number of threads for the parallel version however needs to be manually modified in `hw5/rainfall/pt_{1..7}.sh`.

All scripts must be run in the directory where they live for correct behaviour.

## 4 Performance Testing and Analysis

| Index | Size | Sequential | 1 thread | 2 threads | 4 threads | 8 threads |
|---|---|---|---|---|---|---|
| 1 | 4X4 | 0.000082 | 0.014413 | 0.018509 | 0.018514 | 0.022063 |
| 2 | 16X16 | 0.000234 | 0.013726 | 0.021629 | 0.023102 | 0.070517 |
| 3 | 32X32 | 0.002994 | 0.02424 | 0.033389 | 0.03024 | 0.093669 |
| 4 | 128X128 | 0.08921 | 0.339045 | 0.289759 | 0.241842 | 0.55841 |
| 5 | 512X512 | 0.296602 | 0.543698 | 0.419436 | 0.177353 | 0.19037 |
| 6 | 2048X2048 | 13.569805 | 19.04348 | 13.647546 | 9.473639 | 7.400026 |
| 7 | 4096X4096 | 84.793159 | 108.956767 | 71.634238 | 51.397029 | 42.453876 |

Figure 2: The run-time for different inputs and varying number of Threads

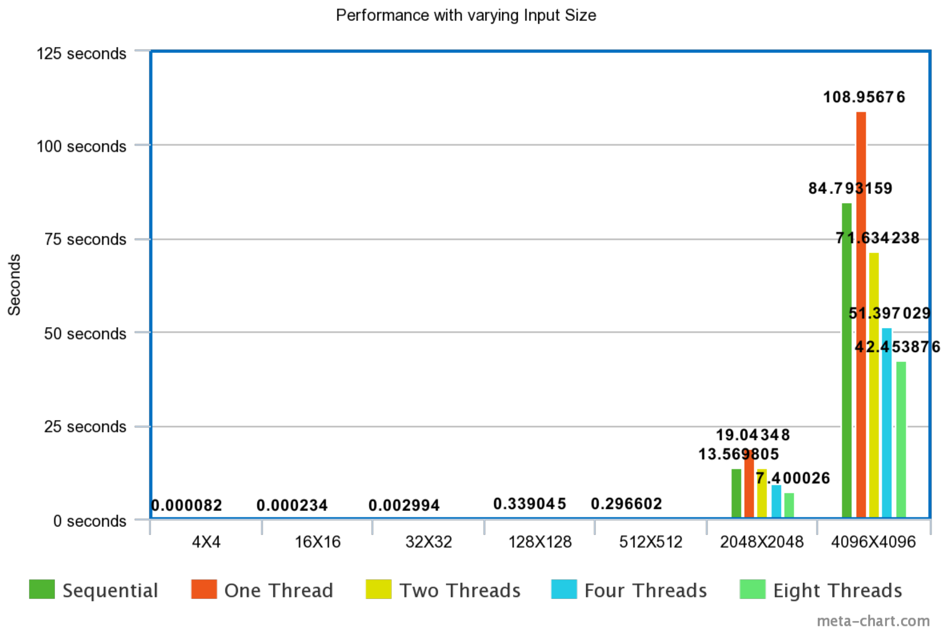


Figure 3: Performances with varying input sizes

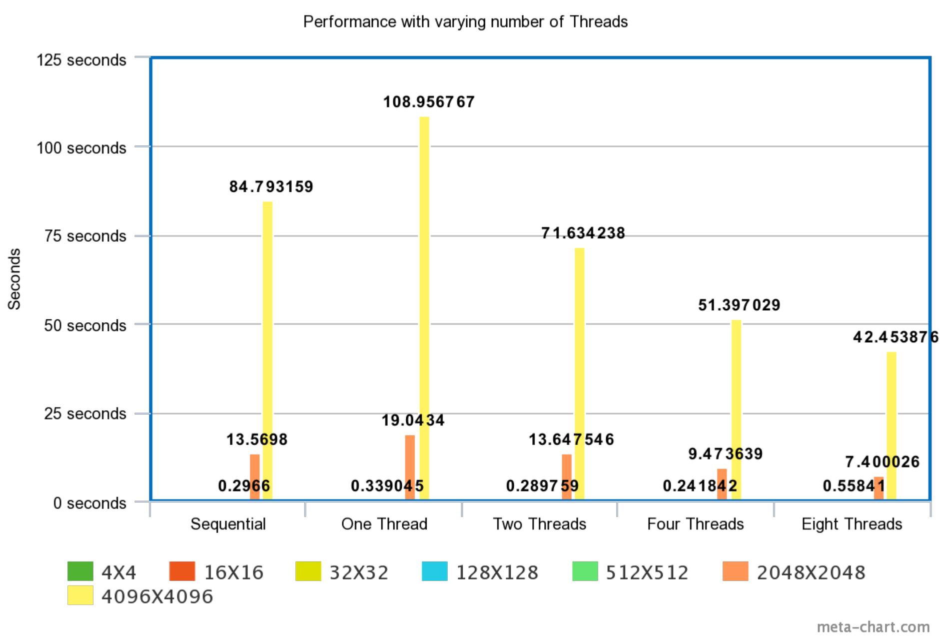


Figure 4: Performances with varying number of threads

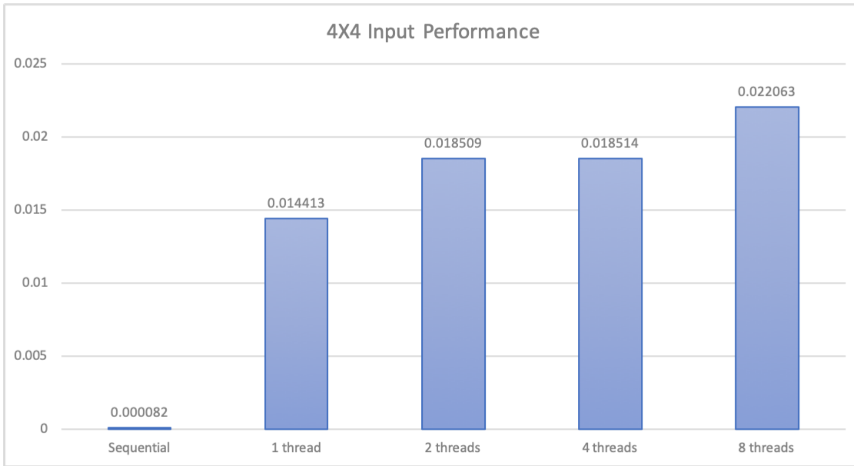


Figure 5: Performances with varying number of threads for 4X4 input

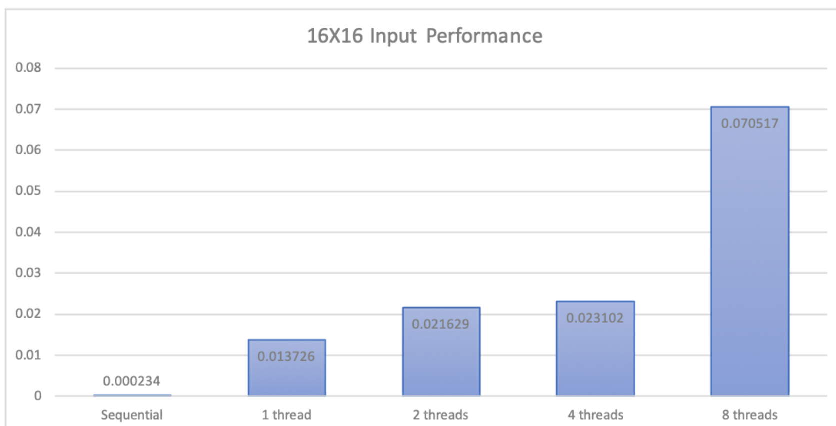


Figure 6: Performances with varying number of threads for 16X16 input

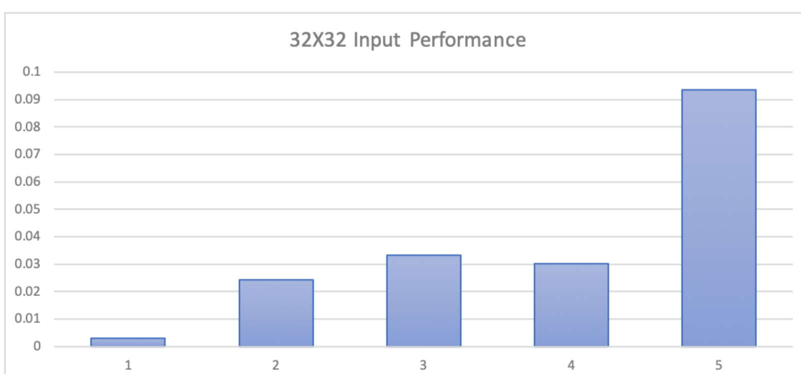


Figure 7: Performances with varying number of threads for 32X32 input

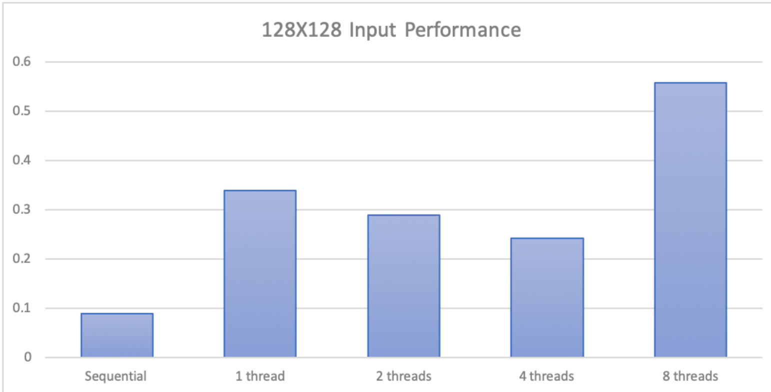


Figure 8: Performances with varying number of threads for 128X128 input

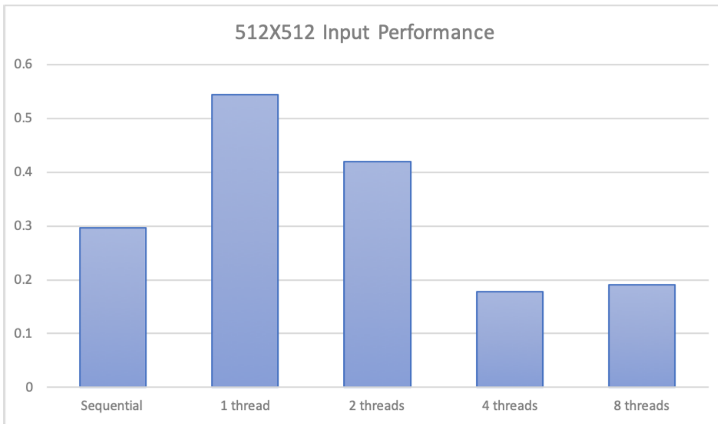


Figure 9: Performances with varying number of threads for 512X512 input

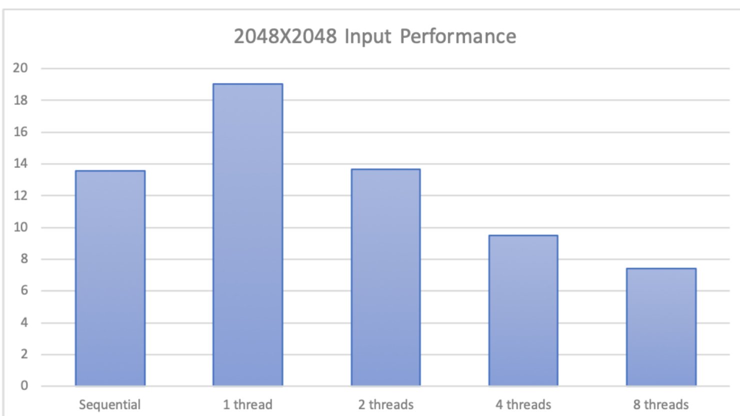


Figure 10: Performances with varying number of threads for 2048X2048 input

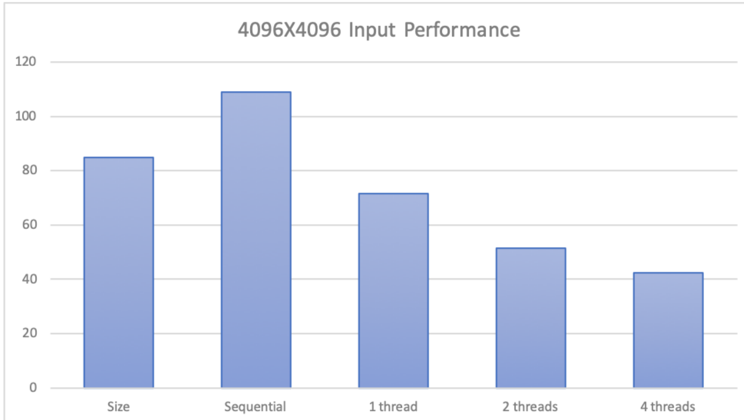


Figure 11: Performances with varying number of threads for 4096X4096 input

As seen in the results, the parallel version does worse on all small files but there is upto 2x speedup on large files such as 2048x2048 and 4096x4096. This is somewhat expected due to the overhead introduced by the parallelism in the code in terms of thread, lock initialization, synchronization, etc.

The overhead of the parallel code is very apparent when comparing the sequential code performance to the performance of the parallel code run with only one thread where the parallel version shows upto 30% overhead.

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