Homework 3: Heated Plate With Pthreads

Steven Stetzler

Pledge: On my honor as a student, I have neither given nor received unauthorized aid on this assignment.

1 Statement of Completion

I was able to successfully parallelize a simulation of heat spreading across a plate using `Pthreads`. I implemented three different ways of subproblem assignment to each thread to optimize the memory access pattern. I found the running time of the parallel solution as a function for the number of threads for each of these implementations.

2 Problem Description

The problem I set out to solve was to parallelize a simulation of heat diffusing across a conductive plate of size $10,000 \times 10,000$ for $10,000$ iterations. The edges of the plate were set to a specific constant temperature, and the following update rule was applied for each point at row i and column j at time t:

$$P_{ij}^{t} = \frac{1}{4}\left(P_{i-1,j}^{t-1} + P_{i+1,j}^{t-1} + P_{i,j-1}^{t-1} + P_{i,j+1}^{t-1}\right) \tag{1}$$

which simply says that the temperature at each point in the plate at the current time t should have the average temperature of its neighbors at the previous time $t-1$. This problem cannot be parallelized in time, as the state of the plate at any time depends on the state at the time one step in the past. However, this problem can be parallelized in space as long as we keep two copies of the plate: P^t and P^{t-1}.

3 Approach, Results, and Analysis

I used `Pthreads` to spawn many different threads that act independent of one another to update values in P^t using the values in P^{t-1}. For all of the tests discussed in this paper, I used the hermes nodes on the University of Virginia Computer Science cluster. The hardware specification for these nodes is:

Processor: 8× AMD Opteron(TM) Processor 6276

Cache: 512 KB L1 instruction, 256 KB L1 data, 16 MB L2, 16 MB L3

Memory: 252 GB of RAM

The use of the hermes nodes allows for up to 64 parallel threads to be run at the same time, more than are available using any of the other nodes in the cluster.

3.1 Approach

The entirety of the complexity of this parallelization comes from how we choose to divide our problem in space over the available threads we have. I came up with three ways to do this:

1. Distribute to each thread some range of rows
2. Distribute to each thread some range of columns

3. Distribute to each thread some range of rows and columns

In (1), we tell each thread to work on all of the columns for some subset of the rows. In (2), we tell each thread to work on all of the rows for some subset of columns. In (3), we break the grid into a power of 4 squares and assign to each thread one of the squares to work on. This also requires that we have a power of 4 number of threads to work with.[1] In the following section, I elaborate on the performance results of these three distribution methods as a function of the number of threads used.

3.2 Results

I ran the heat diffusion simulation on a $10,000 \times 10,000$ sized plate for $10,000$ iterations using each of the work distribution methods described above. In Fig. 1, I show the run time of these simulations as a function of the number of threads used for each of the three methods. I found that the row assignment distribution method achieved that best run times and showed the best scaling in the number of threads used. If one thinks about the memory access pattern during computation, this result makes sense. In the limit of having $10,000$ threads, we would be able to assign to each thread a single row of the plate to work on. Since memory is row major, that means that each thread will be able to work with adjacent memory and thus will have very few cache misses. In the limit that we could fit one row of the plate into the cache, then each thread would be able to work entirely within the cache on each update. If instead, we assign to each thread a single column to work on, then we could expect at the worst that we get a cache miss on every memory access, which would not be ideal. This is exactly what we see in the data: assigning rows to the threads is the best, assigning columns to the threads is the worst, and breaking the problem into squares (which assigns both rows and columns) lies somewhere in between in performance.

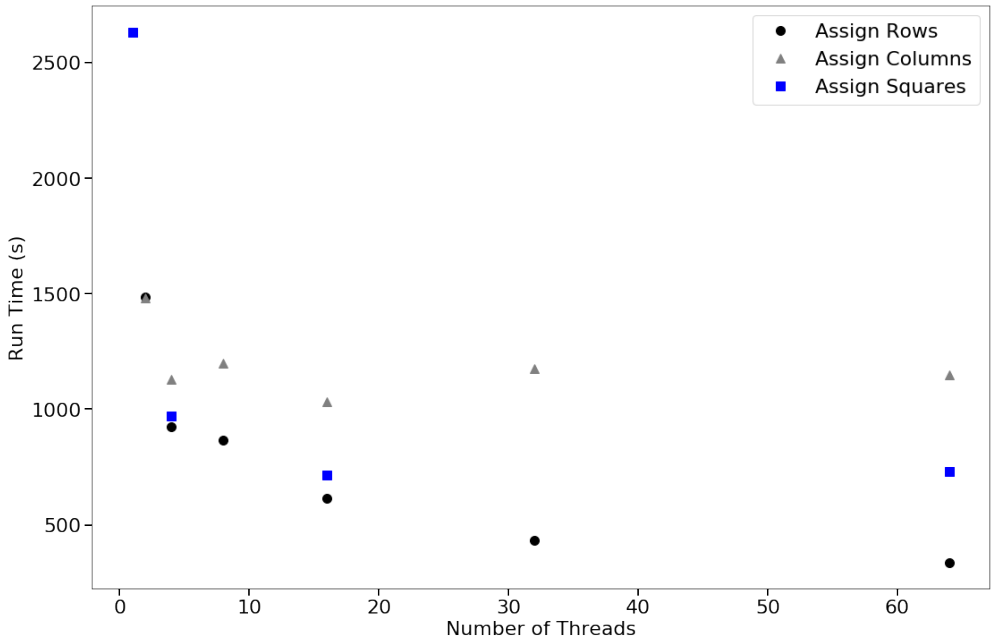

Figure 1: The run time of simulating heat diffusing across a $10,000 \times 10,000$ conductive plate for $10,000$ iterations for three different ways of distributing work to threads.

Plotted in Fig. 2 is the run time scaling on a log-log scale in an attempt to recover the power law scaling of the row assignment method (as that is the only method that behaves like a power law). The fit applied to the data in Fig. 2 gives the equation for the run time

$$T_{row} = 2200 \times N^{-0.47}$$

[1] As an example, consider a plate of size 16×16 which can be split into 4 smaller plates of size 4×4, which each can be assigned to one of 4 threads.

where T_{row} is the run time of the row distribution method and N is the number of threads. Note that had our code been completely optimized, we would expect a scaling like $T_{row} \sim N^{-1}$ (since we would expect the run time with 2 threads to be half as long as with 1 thread and twice as long as with 4 threads). This indicates that more can be done to attempt to optimize this code and achieve better performance scaling in the number of threads.

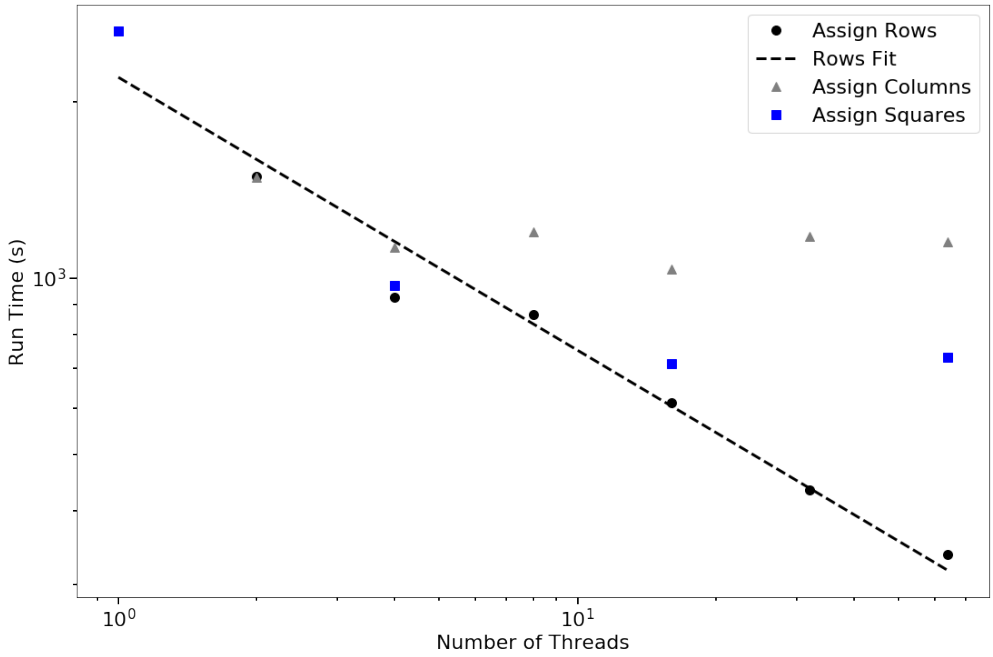

Figure 2: A version of Fig. 1 plotted on a log-log scale, revealing a possible power law pattern for the row distribution method (method (1) above).

4 Conclusions

I successfully parallelized the conductive plate problem using `Pthreads`. I found that the choice in how work is distributed over the threads is incredibly important in getting optimal performance out a `Pthreads` parallel implementation. I found that assigning a range of rows (and all of their columns) as work for each thread achieved the best performance as it had the most optimal memory access pattern. I found that the run time of this implementation scaled in a moderately favorable way with $T \sim N^{-0.47}$. More optimizations should be attempted in the future to try to recover the optimal scaling of $T \sim N^{-1}$.

Here, I include the code I used to parallelize the conductive plate problem.

```
/*
 * CS 4444
 * Steven Stetzler
 * Homework 3: Heated Plate With Pthreads
 */

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <pthread.h>
#include <math.h>
```

```c
// Define the immutable boundary conditions and the inital cell value
#define TOP_BOUNDARY_VALUE 0.0
#define BOTTOM_BOUNDARY_VALUE 100.0
#define LEFT_BOUNDARY_VALUE 0.0
#define RIGHT_BOUNDARY_VALUE 100.0
#define INITIAL_CELL_VALUE 50.0
#define hotSpotRow 4500
#define hotSpotCol 6500
#define hotSpotTemp 100.0

// Function prototypes
void print_cells(float **cells, int n_x, int n_y);
void initialize_cells(float **cells, int n_x, int n_y);
void create_snapshot(float **cells, int n_x, int n_y, int id);
float **allocate_cells(int n_x, int n_y);
void die(const char *error);

// struct to store arguments that each thread should carry with it
// Each thread should know which rows and columns of the plate it should work on
// given by start_row, end_row, start_col, and end_col
// Each thread should also carry with it the location of the plate in memory
// and a barrier to know when to stop and wait for othe threads to catch up to it
typedef struct args {
    int start_row;
    int start_col;
    int end_row;
    int end_col;
    int num_rows;
    int num_cols;
    int thread_id;
    int iterations;
    float*** plate;
    pthread_barrier_t* update_barrier;
} args;

// Global variables that are used throughout
// make_movie tells us whether or not to print out a snapshot every few iterations
int make_movie = 0;
// curr_iter tells us which iteration should be snapshotted next
// curr_iter should only be modified by a single thread at a time
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int curr_iter = 0;

// This is the function containing the work that each thread will perform
void* thread_work(void* p) {
    // Get this threads parameters
    args* t_args = (struct args*) p;

    // Get the plate to work with
    float*** cells = t_args->plate;

    int t, y, x;
    int next_cells_index = 1;
    int curr_cells_index = 0;

    // Print this thread's parameters (as a sanity check)
    printf("Thread %d, start_row = %d, end_row = %d, start_col = %d, end_col = %d\n",
        t_args->thread_id, t_args->start_row, t_args->end_row, t_args->start_col, t_args->end_col);
```

```
    // Simulate the heat flow for the specified number of iterations
   for (t = 0; t < t_args->iterations; t++) {
     // Traverse the plate, computing the new value of each cell
     for (y = t_args->start_row; y <= t_args->end_row; y++) {
        for (x = t_args->start_col; x <= t_args->end_col; x++) {
          // The new value of this cell is the average of the old values of this cell's four
              neighbors
          // printf("Thread %d updating (%d, %d) = %f\n", t_args->thread_id, y, x,
              cells[next_cells_index][y][x]);
             cells[next_cells_index][y][x] = (cells[curr_cells_index][y][x - 1] +
                                      cells[curr_cells_index][y][x + 1] +
                                      cells[curr_cells_index][y - 1][x] +
                                      cells[curr_cells_index][y + 1][x]) * 0.25;
          // printf("Thread %d updated (%d, %d) to %f\n", t_args->thread_id, y, x,
              cells[next_cells_index][y][x]);
          }
     }

       // All threads should wait for the others to finish updating the t + 1 plate
       pthread_barrier_wait(t_args->update_barrier);

       // Swap the two arrays
     curr_cells_index = next_cells_index;
     next_cells_index = !curr_cells_index;

       // Update the hot spot temperature
       if (hotSpotRow <= t_args->num_rows && hotSpotCol <= t_args->num_cols) {
         cells[curr_cells_index][hotSpotRow][hotSpotCol] = hotSpotTemp;
       }

       // Check if we're supposed to make a movie
       if (make_movie == 1) {
          // First thread that gets to this creates a snapshot at current iter and then
              increments iter
          pthread_mutex_lock(&mutex1);

          // Only snapshot on certain frames (we will make at most 100 snapshots of the system)
          if (curr_iter == t) {
             create_snapshot(cells[curr_cells_index], t_args->num_cols, t_args->num_rows,
                 curr_iter);
             curr_iter += (t_args->iterations > 100) ? t_args->iterations / 100 : 1;
          }

          pthread_mutex_unlock(&mutex1);
       }

       // All threads should wait to make sure everyone switched their arrays and that the movie
           finished processing
       pthread_barrier_wait(t_args->update_barrier);
   }

}

// This helper function checks if a number is a power of 4
int power_of_four(unsigned int x) {
    if (x == 0)
        return 0;
    if (x & (x - 1))
```

```
        return 0;
    return x & 0x55555555;
}

// Main function
int main(int argc, char **argv) {
   // Record the start time of the program
   time_t start_time = time(NULL);

   // Extract the input parameters from the command line arguments
   // Number of columns in the grid (default = 1,000)
   int num_cols = (argc > 1) ? atoi(argv[1]) : 1000;
   // Number of rows in the grid (default = 1,000)
   int num_rows = (argc > 2) ? atoi(argv[2]) : 1000;
   // Number of iterations to simulate (default = 100)
   int iterations = (argc > 3) ? atoi(argv[3]) : 100;
    // Number of threads to use
    unsigned int num_threads = (argc > 4) ? atoi(argv[4]) : 1;
    // Specify the method of work distribution for the threads. See below for specific details
    int process_layout = (argc > 5) ? atoi(argv[5]) : 0;
    // Whether or not to make a movie
    make_movie = (argc > 6) ? atoi(argv[6]) : 0;

   // Output the simulation parameters
   printf("Grid: %dx%d, Iterations: %d, Threads: %d, Movie: %d \n", num_cols, num_rows,
       iterations, num_threads, make_movie);

   // We allocate two arrays: one for the current time step and one for the next time step.
   // At the end of each iteration, we switch the arrays in order to avoid copying.
   // The arrays are allocated with an extra surrounding layer which contains
   //  the immutable boundary conditions (this simplifies the logic in the inner loop).
   float **cells[2];
   cells[0] = allocate_cells(num_cols + 2, num_rows + 2);
   cells[1] = allocate_cells(num_cols + 2, num_rows + 2);
   int curr_cells_index = 0, next_cells_index = 1;

   // Initialize the interior (non-boundary) cells to their initial value.
   // Note that we only need to initialize the array for the current time
   //  step, since we will write to the array for the next time step
   //  during the first iteration.
   initialize_cells(cells[0], num_cols, num_rows);

   // Set the immutable boundary conditions in both copies of the array
   int x, y, i;
   for (x = 1; x <= num_cols; x++) cells[0][0][x] = cells[1][0][x] = TOP_BOUNDARY_VALUE;
   for (x = 1; x <= num_cols; x++) cells[0][num_rows + 1][x] = cells[1][num_rows + 1][x] =
       BOTTOM_BOUNDARY_VALUE;
   for (y = 1; y <= num_rows; y++) cells[0][y][0] = cells[1][y][0] = LEFT_BOUNDARY_VALUE;
   for (y = 1; y <= num_rows; y++) cells[0][y][num_cols + 1] = cells[1][y][num_cols + 1] =
       RIGHT_BOUNDARY_VALUE;

    // Create a set of threads
    pthread_t threadpool[num_threads];
    args arguments[num_threads];

    // Initialize thread attributes
    pthread_attr_t attr;
    pthread_attr_init(&attr);
```

```
// Make a barrier that waits for all of the threads
pthread_barrier_t update_barrier;
pthread_barrier_init(&update_barrier, NULL, num_threads);

// The number of squares we can divide the plate into
int divisions = (int) sqrt(num_threads);
// Counter to help specify row, column assignment
int j = 0;

// Define scheme for assigning which work each thread will take
for (i = 0; i < num_threads; i++) {
    // Check which distribution method we are using
    // Assign rows and columns that each thread should work on accordingly
    switch (process_layout) {
        case 0 :
            // Assign each thread an equal number of rows to work with
            arguments[i].start_row = (int) (num_rows / (float) num_threads * i) + 1;
            arguments[i].end_row = (int) (num_rows / (float) num_threads * (i + 1));
            arguments[i].start_col = 1;
            arguments[i].end_col = num_cols;
            break;
        case 1 :
            // Assign each thread an equal number of columns to work with
            arguments[i].start_row = 1;
            arguments[i].end_row = num_rows;
            arguments[i].start_col = (int) (num_cols / (float) num_threads * i) + 1;
            arguments[i].end_col = (int) (num_cols / (float) num_threads * (i + 1));
            break;
        case 2 :
            // Block problem into squares of equal size
            // We need to be using a power of 4 threads for this to work
            if (!power_of_four(num_threads)) {
                printf("num_threads = %d is not a power of four. Cannot block into squares.\n",
                    num_threads);
                return 1;
            }
            if (i % divisions == 0 && i != 0) {
                j += 1;
            }
            arguments[i].start_row = (int) (num_rows / (float) divisions * (i % divisions)) + 1;
            arguments[i].end_row = (int) (num_rows / (float) divisions * ((i % divisions) + 1));
            arguments[i].start_col = (int) (num_cols / (float) divisions * j) + 1;
            arguments[i].end_col = (int) (num_cols / (float) divisions * (j + 1));

            break;
    }

    // Assign other parameter values
    arguments[i].thread_id = i;
    arguments[i].iterations = iterations;
    arguments[i].plate = cells;
    arguments[i].update_barrier = &update_barrier;
    arguments[i].num_rows = num_rows;
    arguments[i].num_cols = num_cols;
}

for (i = 0; i < num_threads; i++) {
    // Create each thread, passing thread attributes, the function to run, and the arguments to
        use
```

```
        pthread_create(&threadpool[i], &attr, &thread_work, (void *) &arguments[i]);
    }

    for (i = 0; i < num_threads; i++) {
        pthread_join(threadpool[i], NULL);
    }

   // Compute and output the execution time
   time_t end_time = time(NULL);
   printf("\nExecution time: %f seconds\n", difftime(end_time, start_time));

   return 0;
}


// Allocates and returns a pointer to a 2D array of floats
float **allocate_cells(int num_cols, int num_rows) {
   // Initialize array of float pointers -- array of arrays
    float **array = (float **) malloc(num_rows * sizeof(float *));
   if (array == NULL) die("Error allocating array!\n");

    // Initialize first pointer to point to beginning of flattened array of size num_rows x
        num_cols
   array[0] = (float *) malloc(num_rows * num_cols * sizeof(float));
   if (array[0] == NULL) die("Error allocating array!\n");

    // Initialize each row to point to a portion of the flattened array that is of size num_cols
   int i;
   for (i = 1; i < num_rows; i++) {
      array[i] = array[0] + (i * num_cols);
   }

   return array;
}


// Sets all of the specified cells to their initial value.
// Assumes the existence of a one-cell thick boundary layer.
void initialize_cells(float **cells, int num_cols, int num_rows) {
   int x, y;
   for (y = 1; y <= num_rows; y++) {
      for (x = 1; x <= num_cols; x++) {
         cells[y][x] = INITIAL_CELL_VALUE;
      }
   }
}


// Creates a snapshot of the current state of the cells in PPM format.
// The plate is scaled down so the image is at most 1,000 x 1,000 pixels.
// This function assumes the existence of a boundary layer, which is not
//  included in the snapshot (i.e., it assumes that valid array indices
//  are [1..num_rows][1..num_cols]).
void create_snapshot(float **cells, int num_cols, int num_rows, int id) {
   int scale_x, scale_y;
   scale_x = scale_y = 1;

   // Figure out if we need to scale down the snapshot (to 1,000 x 1,000)
   //  and, if so, how much to scale down
```

```
  if (num_cols > 1000) {
    if ((num_cols % 1000) == 0) scale_x = num_cols / 1000;
    else {
      die("Cannot create snapshot for x-dimensions >1,000 that are not multiples of 1,000!\n");
      return;
    }
  }
  if (num_rows > 1000) {
    if ((num_rows % 1000) == 0) scale_y = num_rows / 1000;
    else {
      printf("Cannot create snapshot for y-dimensions >1,000 that are not multiples of
          1,000!\n");
      return;
    }
  }

  // Open/create the file
  char text[255];
  sprintf(text, "snapshot_parallel.%06d.ppm", id);
  FILE *out = fopen(text, "w");
  // Make sure the file was created
  if (out == NULL) {
    printf("Error creating snapshot file!\n");
    return;
  }

  // Write header information to file
  // P3 = RGB values in decimal (P6 = RGB values in binary)
  fprintf(out, "P3 %d %d 100\n", num_cols / scale_x, num_rows / scale_y);

  // Precompute the value needed to scale down the cells
  float inverse_cells_per_pixel = 1.0 / ((float) scale_x * scale_y);

  // Write the values of the cells to the file
  int x, y, i, j;
  for (y = 1; y <= num_rows; y += scale_y) {
    for (x = 1; x <= num_cols; x += scale_x) {
      float sum = 0.0;
      for (j = y; j < y + scale_y; j++) {
        for (i = x; i < x + scale_x; i++) {
          sum += cells[j][i];
        }
      }
      // Write out the average value of the cells we just visited
      int average = (int) (sum * inverse_cells_per_pixel);
      // printf("%d 0 %d\t", average, 100 - average);
      fprintf(out, "%d 0 %d\t", average, 100 - average);
    }
     // printf("\n");
    fwrite("\n", sizeof(char), 1, out);
  }

  // Close the file
  fclose(out);
}


// Prints the specified error message and then exits
void die(const char *error) {
```

```
   printf("%s", error);
   exit(1);
}
```