CSC345 Project 2: Multithreaded Programming

DJ Landau
The College of New Jersey
landaud1@tcnj.edu

Jasmine Ocasio
The College of New Jersey
ocasioj1@tcnj.edu

I. Accepting Input

To create the Sudoku board for the Options outlined in the later sections, the following function was created:

```
/* Open input file and format into 2d array board */
int make_board(FILE* inputFile, int board[9][9], char line[80] ){
    if(inputFile == NULL) {
    perror("Error opening file");
    return 1;
    }

    for (int i = 0; i < 9; i++) {
        if (fgets(line, 80, inputFile) == NULL) {
            perror("Error getting line");
            fclose(inputFile);
            return 1;
        }

        /* tokenize after fgets successfully reads a line */
        char* num = strtok(line, " \n");
        for (int j = 0; j < 9 && num != NULL; j++) {
            board[i][j] = atoi(num);
            num = strtok(NULL, " \n");  /* get the next number */
        }
    }

    fclose(inputFile);

    /* print board to verify */
    printf("BOARD STATE IN input.txt: \n");
    for(int i =0; i<9; i++){
        for(int j=0; j<9; j++){
            printf("%d ",board[i][j]);
            if(j%3 ==2){
            printf(" ");
        }
        }
        printf("\n");
        if(i%3 ==2){
            printf("\n");
        }
    }
    return 0;

}
```

The above function takes a pointer to the input file containing the space-separated integers arranged in 9 columns and 9 rows, the global 2D array that will hold the Sudoku board, and the global line that is used to hold each line of the board from the input file. The function takes each line from the file, separates the integers into tokens, and stores each of them into the 2D board array. This process is done 9 times, once for each row, to create the full Sudoku board. The function then prints the contents of the 2D array to the user, verifying that it is the same as the input file.

To call this function, the following lines begin the main function to create the board, as well as get the input from the command line to select which option will validate the board:

```
/* Get option choice from user */
    int option = atoi(argv[1]);
    FILE* inputFile = fopen("input.txt", "r");

    make_board(inputFile, board, line);
```

II. Execution Option 1

A. Thread Structure

Option 1 requests 11 worker threads. 1 for checking rows, 1 for checking columns, and 9 for checking blocks. In our implementation, there is an additional thread which dispatches the multithreaded function to check blocks, leading to a total of 12 threads not including the main one. This is implemented as such:

```
/*1 thread for rows, 1 thread for cols, 9 threads for blocks. 11 worker threads, 1 dispatch thread*/
int option_1(){
    pthread_t tid_row, tid_col, tid_block;
    pthread_create(&tid_row, NULL, check_rows, NULL);
    pthread_create(&tid_col, NULL, check_cols, NULL);
    pthread_create(&tid_block, NULL, check_blocks_mt, NULL);
    /*Bring the boys back home*/
    pthread_join(tid_row, NULL);
    pthread_join(tid_col, NULL);
    pthread_join(tid_block, NULL);
    /*Check results*/
    for (int i = 0; i < 3; ++i){
        if (results_final[i] == 0){
            /*At least one result was invalid*/
            return 0;
        }
    }
    /*All were valid!*/
    return 1;
}
```

B. Validating Sudoku Board

A Sudoku board is considered valid if all digits 1 through 9 are present in every row, column, and 3x3 block. Starting at the baseline operations, functions for checking a single row, column, and block were implemented. Each of them receive input specifying where on the Sudoku board to validate, and first load each element into a 1-dimensional array of integers. This array is sorted using a bubble sort algorithm, then checked to see if the array is exactly [1, 2, 3, 4, 5, 6, 7, 8, 9]. The validation step for this 1-dimensional array is implemented in the following two functions:

```
/*Sorts the given integer array, assumes size of 9.*/
void sort_array(int *arr) {
    /*Bubble sort is chosen, n is confirmed to be pretty small so it
    doesn't really matter*/
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 8 - i; ++j) {
            if (arr[j] > arr[j + 1]) {
                /*swap*/
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

/*Given an array of numbers, checks if exactly [1, 2, 3, 4, 5, 6, 7, 8, 9]
after sorted*/
int check_valid(int *arr){
    /*Sort so numbers are in order*/
    sort_array(arr);
    for (int i = 0; i < 9; ++i){
        if (arr[i] != i + 1){
            /*Number is not expected, return false*/
            return 0;
        }
    }
    /*Array is correct, line has 1 of each digit*/
    return 1;
}
```

Option 1 demands all rows and columns are checked in a single thread, so the single-threaded check_rows() and check_cols() functions were implemented.

```
/*Checks if a row on the sudoku board is valid*/
int check_row(int y){
    if (y >= 9) {
        /*Invalid offset*/
        return -1;
    }
    int arr[9];
    for (int i = 0; i < 9; ++i){
        arr[i] = (board[y][i]);
    }
    return check_valid(arr);
}
/*Checks all rows in one thread*/
void *check_rows(){
    for (int i = 0; i < 9; ++i){
        if (check_row(i) == 0){
            set_results(results_lock, &results_final[0], 0);
            pthread_exit(0);
        }
        //printf("Row %d is valid\n", i); //DELETE ME-- JUST FOR CHECKS
    }
    set_results(results_lock, &results_final[0], 1);

    pthread_exit(0);
}
```

The check_rows() function submits its results to a global array using the set_results() function. This function utilizes a mutex lock as a contingency, although it has been found to be not entirely necessary. This functioin is implemented as such:

```
/* Locks given location until result is given */
void set_results(pthread_mutex_t lock, int *result_location, int result) {
    pthread_mutex_lock(&lock);
    *result_location = result;
    pthread_mutex_unlock(&lock);
}
```

There is a separate results array for the rows, columns, and blocks, each with a size of 9, that is used when the multithreaded versions of those functions are called

C. *Handling multiple threads*

Option 1 requests each block of the Sudoku board is to be checked in a separate thread, so the blocks were only implemented in a multithreaded manner as such:

```
/*Checks if a 3x3 square is valid. This guy is always multithreaded based on assignment specs.*/
void *check_block(void *param){
    int arr[9];
    /*Input is a structure containing x and y offset*/
    block_offset *offset = (block_offset *)param;
    int x = offset->x;
    int y = offset->y;
    free(offset);
    /*Check in a square*/
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j){
            arr[i * 3 + j] = board[i + (y * 3)][j + (x * 3)];
        }
    }
    set_results(blocks_lock, &results_blocks[x + 3 * y], check_valid(arr));
    pthread_exit(0);
}
/*Checks all blocks, multithreaded*/
void *check_blocks_mt(){
    pthread_t tid[9];
    /*Dispatch threads*/
    for (int i = 0; i < 3; ++i){
        for (int j = 0; j < 3; ++j){
            /*i*3+j returns a nice little index number, same as 2d array to pointer offset*/
            block_offset *input = malloc(sizeof(block_offset));
            input->x = i;
            input->y = j;
            pthread_create(&tid[i * 3 + j], NULL, check_block, input);
        }
    }
    /*Wait for threads to conclude*/
    for (int i = 0; i < 9; ++i){
        pthread_join(tid[i], NULL);
    }
    /*Check result*/
    for (int i = 0; i < 9; ++i){
        if (results_blocks[i] == 0){
            /*At least one result was invalid*/
            set_results(results_lock, &results_final[2], 0);
            pthread_exit(0);
        }
    }
    /*All results were valid!*/
    set_results(results_lock, &results_final[2], 1);
    pthread_exit(0);
}
```

III. Execution Option 2

A. Thread Structure

Option 2 requests 27 worker threads, 9 for each validation type. This was implemented in a similar manner, as such:

```
/*9 threads for rows, 9 threads for cols, 9 threads for blocks. 27 worker threads, 3 dispatch thread*/
int option_2(){
    pthread_t tid_row, tid_col, tid_block;
    pthread_create(&tid_row, NULL, check_rows_mt, NULL);
    pthread_create(&tid_col, NULL, check_cols_mt, NULL);
    pthread_create(&tid_block, NULL, check_blocks_mt, NULL);
    /*Bring the boys back home*/
    pthread_join(tid_row, NULL);
    pthread_join(tid_col, NULL);
    pthread_join(tid_block, NULL);
    /*Check results*/
    for (int i = 0; i < 3; ++i){
        if (results_final[i] == 0){
            /*At least one result was invalid*/
            return 0;
        }
    }
    /*All were valid!*/
    return 1;
}
```

For this, multithreaded variations of the check_rows() and check_cols() functions were implemented as such:

```
/*Checks if a row on the sudoku board is valid, multithreaded*/
void *check_row_mt(void *param){
    /*Obtain parameter*/
    int y = *((int *)param);
    free(param);
    /*Check row*/
    int arr[9];
    for (int i = 0; i < 9; ++i){
        arr[i] = (board[y][i]);
    }
    set_results(rows_lock, &results_rows[y], check_valid(arr));
    pthread_exit(0);
}
/*Checks all rows in nine threads*/
void *check_rows_mt(){
    pthread_t tid[9];
    /*Dispatch threads*/
    for (int i = 0; i < 9; ++i){
        int *input = malloc(sizeof(*input));
        *input = i;
        pthread_create(&tid[i], NULL, check_row_mt, input);
    }
    /*Wait for threads to conclude*/
    for (int i = 0; i < 9; ++i){
        pthread_join(tid[i], NULL);
    }
    for (int i = 0; i < 9; ++i){
        if (results_rows[i] == 0){
            /*At least one result was invalid*/
            set_results(results_lock, &results_final[0], 0);
            pthread_exit(0);
        }
    }
    /*All results were valid!*/
    set_results(results_lock, &results_final[0], 1);
    pthread_exit(0);
}
```

IV. Execution Option 3

Option 3 requires three child-processes to each validate a different component of the board. The first child uses the following function to validate the rows similarly to Option 1:

```
    /* Check all rows in one thread for child process */
void *check_rows_child(void* mem){
    int* results = (int*)mem;
    /* assume row is valid */
    int valid = 1;
    for (int i = 0; i < 9; ++i){
        if (check_row(i) == 0){
            valid = 0;
```

```
                break;
            }
        }
        *results =  valid;

        pthread_exit(0);
}
```

The second child uses the following function to validate the columns similarly to Option 1:

```
    /*Checks all columns in one thread for child process*/
void *check_cols_child(void* mem){
    int* results = (int*)mem;
    /* assume col is valid */
    int valid = 1;
    for (int i = 0; i < 9; ++i){
        if (check_col(i) == 0){
            valid = 0;
            break;
        }
    }
    *results = valid;
    pthread_exit(0);
}
```

The third child uses the following function to validate the blocks similarly to Option 1:

```
    /*Checks all blocks, multithreaded for child process */
void *check_blocks_child(void* mem){
    int* results = (int*)mem;
    /* assume col is valid */
    int valid = 1;
    pthread_t tid[9];
    /*Dispatch threads*/
    for (int i = 0; i < 3; ++i){
        for (int j = 0; j < 3; ++j){
            /*i*3+j returns a nice little index number, same as 2d array to pointer offset*/
            block_offset *input = malloc(sizeof(block_offset));
            input->x = i;
            input->y = j;
            pthread_create(&tid[i * 3 + j], NULL, check_block, input);
        }
    }
    /*Wait for threads to conclude*/
    for (int i = 0; i < 9; ++i){
        pthread_join(tid[i], NULL);
    }
    /*Check result*/
    for (int i = 0; i < 9; ++i){
        if (results_blocks[i] == 0){
            /*At least one result was invalid*/
            valid = 0;
            break;
        }
    }
    *results = valid;
    pthread_exit(0);
}
```

These three methods were combined to be properly implemented within their respective child processes in the following function:

```
    int option_3(){

    pid_t pid;
    pthread_t tid_row, tid_col, tid_block;
    /* pass this as argument for check- functions */
    int* memory = shared_mem();

    for(int i = 0; i<3; i++){
        pid = fork();

        if(pid == 0){ /* Child */
            /* check rows */
            if(i==1){
                pthread_create(&tid_row, NULL, check_rows_child, &memory[0]);
                pthread_join(tid_row, NULL);
                exit(0);
            }
            /* check cols */
            else if(i==2){
                pthread_create(&tid_col, NULL, check_cols_child, &memory[1]);
                pthread_join(tid_col, NULL);
                exit(0);
            }
            /* check blocks */
            else{
                pthread_create(&tid_block, NULL, check_blocks_child, &memory[2]);
                pthread_join(tid_block, NULL);
                exit(0);
            }

        }
        else if(pid<0){
            if(i==1){
                perror("Error in checking rows\n");
            }
            else if(i==2){
                perror("Error in checking columns\n");
            }
            else{
                perror("Error in checking blocks\n");
            }
            exit(1);
        }
    }
    for(int i = 0 ; i<3 ; i++){ /* Parent process*/
        wait(NULL);
    }
    for(int i = 0; i < 3; ++i){
        if (memory[i] == 0){
            /*At least one result was invalid*/
            munmap(memory, sizeof(int)*3); /* clean up shared mem*/
            return 0;
        }
    }

    /*All were valid!*/
    munmap(memory, sizeof(int)*3); /* clean up shared mem*/
    return 1;
}
```

This code implements Option 3 so that the first child validates the rows, the second validates the columns, and the third validates the blocks. There is shared memory initialized between the child processes and the parent so that the results of each of their validations can be returned to the parent. The function that creates the shared memory is below:

```
int* shared_mem(){
        /* Memory map the pointer to shared memory object */
    int* mem = mmap(NULL, sizeof(int) * 3, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    /* initialize to all zeros */
    for(int i=0; i<3; i++){
        mem[i]=0;
    }
    return mem;
}
```

V. Collecting Time Data

To record the timing of each Option to output it to the command line, the clock() function was used to capture the fine difference in timing in the execution. This was implemented in the main method as follows:

```
 /* Execute chosen option and get timing */
    int verdict = 2;
    clock_t begin,end;
    begin = clock();
    if(option == 1){
        verdict = option_1();
    }
    else if(option == 2){
        verdict = option_2();
    }
    else if(option == 3){
        verdict = option_3();
    }
    else{
        printf("Invalid option selected. Please select 1,2 or 3.\n");
    }
    end = clock();

    /* Printing verdict with timing*/
    double total_time = ((double) (end - begin)) / CLOCKS_PER_SEC;
    if(verdict == 1){
        printf("SOLUTION: YES(%lf seconds)\n", total_time);
    }
    else if(verdict == 0){
        printf("SOLUTION: NO(%lf seconds)\n", total_time);
    }
    else{
        printf("There was an error in determining the solution. \n");
    }
```

The above code also prints to the command line the verdict of whether or not the board is a valid solution. If the chosen Option returns 1, it is a solution, if it returns 0, it is not.

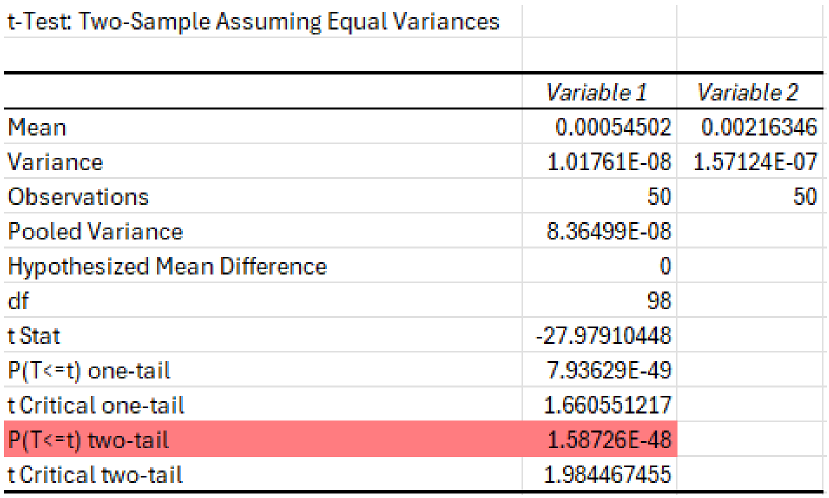
VI. Results

To collect the data necessary to run a two-sample t-test, the following function was created:

```
/* Runs stats for t test and stores in csv to be used in Excel file for results */
int run_stats(){
    FILE* fp = fopen("timing_results.csv", "w"); /* making data collection file*/
    if (!fp) {
        perror("Failed to open file");
        return 1;
    }

    fprintf(fp, "Option,Run,Time(seconds),Mean Time(seconds), %d\n", getpid());
```

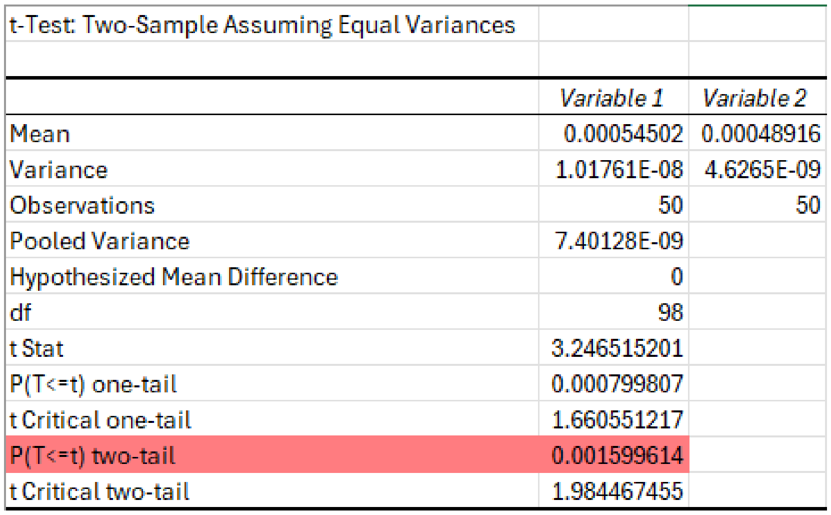
```c
    clock_t begin,end;
    double full_time = 0;
    double timings[50];

    for(int run =0; run<50; run++){
        begin = clock();
        option_1();
        end = clock();

        double run_time = ((double)(end - begin)) / CLOCKS_PER_SEC;

        timings[run] = run_time;
        fprintf(fp, "%d,%d,%f\n", 1, run + 1, timings[run]);
    }
    for(int run =0; run<50; run++){
        begin = clock();
        option_2();
        end = clock();


        double run_time = ((double)(end - begin)) / CLOCKS_PER_SEC;

        timings[run] = run_time;
        fprintf(fp, "%d,%d,%f\n", 2, run + 1, timings[run]);
    }
    fclose(fp);

    for(int run =0; run<50; run++){
        begin = clock();
        option_3();
        end = clock();

        double run_time = ((double)(end - begin)) / CLOCKS_PER_SEC;

        timings[run] = run_time;
        fp = fopen("timing_results.csv", "a");
        fprintf(fp, "%d,%d,%f\n", 3, run + 1, timings[run]);
        fclose(fp);
    }
    return 0;
}
}
```

The function runs each of the options 50 times, writing the individual run times to a csv file. The result is 150 times associated with their respective option. Each group of 50 times was used as one of the samples for the two-sample t-test, resulting in three tests total to compare each of the tests. The tests were completed in Excel, resulting in the following tables:



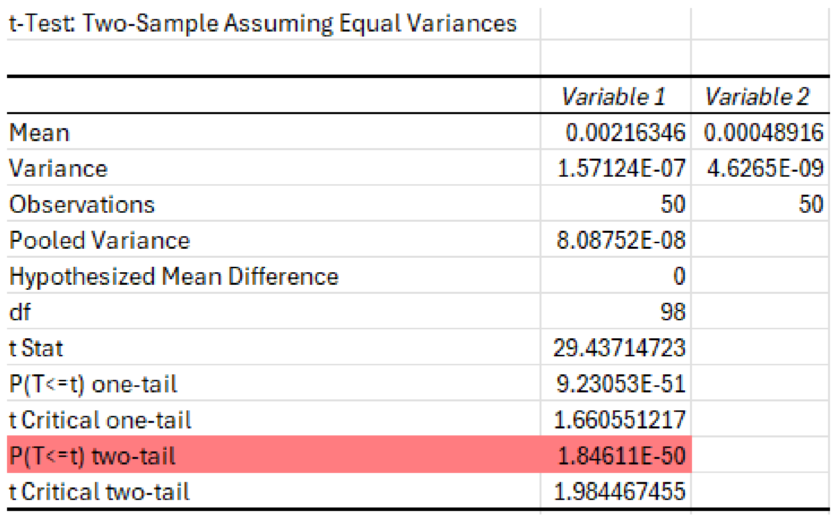
t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.00054502	0.00216346
Variance	1.01761E-08	1.57124E-07
Observations	50	50
Pooled Variance	8.36499E-08	
Hypothesized Mean Difference	0	
df	98	
t Stat	-27.97910448	
P(T<=t) one-tail	7.93629E-49	
t Critical one-tail	1.660551217	
P(T<=t) two-tail	1.58726E-48	
t Critical two-tail	1.984467455	

Figure 1: T-Test Results Comparing Options 1 and 2



t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.00054502	0.00048916
Variance	1.01761E-08	4.6265E-09
Observations	50	50
Pooled Variance	7.40128E-09	
Hypothesized Mean Difference	0	
df	98	
t Stat	3.246515201	
P(T<=t) one-tail	0.000799807	
t Critical one-tail	1.660551217	
P(T<=t) two-tail	0.001599614	
t Critical two-tail	1.984467455	

Figure 2: T-Test Results Comparing Options 1 and 3



t-Test: Two-Sample Assuming Equal Variances		
	Variable 1	*Variable 2*
Mean	0.00216346	0.00048916
Variance	1.57124E-07	4.6265E-09
Observations	50	50
Pooled Variance	8.08752E-08	
Hypothesized Mean Difference	0	
df	98	
t Stat	29.43714723	
P(T<=t) one-tail	9.23053E-51	
t Critical one-tail	1.660551217	
P(T<=t) two-tail	1.84611E-50	
t Critical two-tail	1.984467455	

Figure 3: T-Test Results Comparing Options 2 and 3

Through these tests, it is the goal to reject the Null Hypothesis of "There is no statistically significant difference between two methods." Using an α value of 0.05, it is clear that in all three tests, the Null Hypothesis is rejected. The p-values for the two tailed t-tests, as seen highlighted in red, all fall below the α value, signifying that there is significant statistical evidence that disproves the Null. This result is even clearer to see in the following graph:

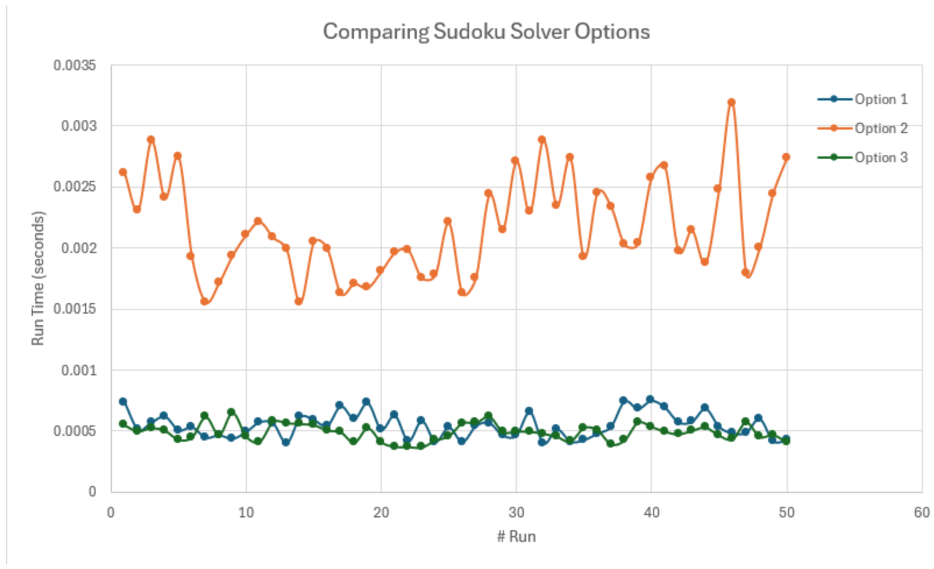


Figure 4: Graph of All 50 Recorded Timings for Each Option

As observed in the graph above, Option 2 is much slower in all 50 runs compared to Options 1 and 3. This explains the extremely small p-value for the tests including Option 2 and the larger p-value comparing 1 and 3. As the graph shows, Options 1 and 3 are fairly similar; however, Option 3 is more often faster than Option 1.

The conclusion of this statistical experiment is that there is, in fact, statistical difference between all three methods of solving this problem. Utilizing only this test, it seems that the best method to use for this problem is Option 3.

VII. Appendix

A. main.c

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <time.h>
#include <sys/stat.h>


/* Global variables for board and line in board */
char line[80];  /* Buffer to hold each line */
int board[9][9]; /* Empty sudoku board*/


/*Results for multithreaded functions are recorded in these global arrays*/
int results_rows[9];
int results_cols[9];
int results_blocks[9];

/*Rows is index 0, columns is index 1, blocks is index 2*/
int results_final[3];

/*Locks for the result arrays*/
pthread_mutex_t rows_lock;
pthread_mutex_t cols_lock;
pthread_mutex_t blocks_lock;
pthread_mutex_t results_lock;

/*Structure to send values to check_block()*/
typedef struct {
    int x;
    int y;
} block_offset;



/* Locks given location until result is given */
void set_results(pthread_mutex_t lock, int *result_location, int result) {
    pthread_mutex_lock(&lock);
    *result_location = result;
    pthread_mutex_unlock(&lock);
}

/*Sorts the given integer array, assumes size of 9.*/
void sort_array(int *arr) {
    /*Bubble sort is chosen, n is confirmed to be pretty small so it doesn't really matter*/
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 8 - i; ++j) {
            if (arr[j] > arr[j + 1]) {
                /*swap*/
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

```c
/*Given an array of numbers, checks if exactly [1, 2, 3, 4, 5, 6, 7, 8, 9] after sorted*/
int check_valid(int *arr){
    /*Sort so numbers are in order*/
    sort_array(arr);
    for (int i = 0; i < 9; ++i){
        if (arr[i] != i + 1){
            /*Number is not expected, return false*/
            return 0;
        }
    }
    /*Array is correct, line has 1 of each digit*/
    return 1;
}

/*Checks if a row on the sudoku board is valid*/
int check_row(int y){
    if (y >= 9) {
        /*Invalid offset*/
        return -1;
    }
    int arr[9];
    for (int i = 0; i < 9; ++i){
        arr[i] = (board[y][i]);
    }
    return check_valid(arr);
}

/*Checks if a column on the sudoku board is valid*/
int check_col(int x){
    if (x >= 9) {
        /*Invalid offset*/
        return -1;
    }
    int arr[9];
    for (int i = 0; i < 9; ++i){
        arr[i] = (board[i][x]);
    }
    return check_valid(arr);
}

/*Checks if a row on the sudoku board is valid, multithreaded*/
void *check_row_mt(void *param){
    /*Obtain parameter*/
    int y = *((int *)param);
    free(param);
    /*Check row*/
    int arr[9];
    for (int i = 0; i < 9; ++i){
        arr[i] = (board[y][i]);
    }
    set_results(rows_lock, &results_rows[y], check_valid(arr));
    pthread_exit(0);
}

/*Checks if a column on the sudoku board is valid, multithreaded*/
void *check_col_mt(void *param){
    /*Obtain parameter*/
    int x = *((int *)param);
    free(param);
    /*Check column*/
    int arr[9];
    for (int i = 0; i < 9; ++i){
        arr[i] = (board[i][x]);
    }
    set_results(cols_lock, &results_cols[x], check_valid(arr));
    pthread_exit(0);
}

/*Checks if a 3x3 square is valid. This guy is always multithreaded based on assignment specs.*/
void *check_block(void *param){
    int arr[9];
    /*Input is a structure containing x and y offset*/
    block_offset *offset = (block_offset *)param;
    int x = offset->x;
    int y = offset->y;
    free(offset);
    /*Check in a square*/
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j){
            arr[i * 3 + j] = board[i + (y * 3)][j + (x * 3)];
        }
    }
    set_results(blocks_lock, &results_blocks[x + 3 * y], check_valid(arr));
    pthread_exit(0);
}

/*Checks all rows in one thread*/
void *check_rows(){
    for (int i = 0; i < 9; ++i){
        if (check_row(i) == 0){
            set_results(results_lock, &results_final[0], 0);
            pthread_exit(0);
        }
    }
    set_results(results_lock, &results_final[0], 1);

    pthread_exit(0);
}

/*Checks all columns in one thread*/
void *check_cols(){
    for (int i = 0; i < 9; ++i){
        if (check_col(i) == 0){
            set_results(results_lock, &results_final[1], 0);
            pthread_exit(0);
        }
    }
    set_results(results_lock, &results_final[1], 1);
    pthread_exit(0);
}

/*Checks all rows in nine threads*/
void *check_rows_mt(){
    pthread_t tid[9];
    /*Dispatch threads*/
    for (int i = 0; i < 9; ++i){
        int *input = malloc(sizeof(*input));
        *input = i;
        pthread_create(&tid[i], NULL, check_row_mt, input);
    }
    /*Wait for threads to conclude*/
    for (int i = 0; i < 9; ++i){
        pthread_join(tid[i], NULL);
    }
    for (int i = 0; i < 9; ++i){
        if (results_rows[i] == 0){
            /*At least one result was invalid*/
            set_results(results_lock, &results_final[0], 0);
            pthread_exit(0);
        }
    }
    /*All results were valid!*/
    set_results(results_lock, &results_final[0], 1);
    pthread_exit(0);
}

/*Checks all rows in nine threads*/
void *check_cols_mt(){
    pthread_t tid[9];
    /*Dispatch threads*/
    for (int i = 0; i < 9; ++i){
        int *input = malloc(sizeof(*input));
        *input = i;
        pthread_create(&tid[i], NULL, check_col_mt, input);
    }
    /*Wait for threads to conclude*/
    for (int i = 0; i < 9; ++i){
        pthread_join(tid[i], NULL);
    }
    for (int i = 0; i < 9; ++i){
        if (results_cols[i] == 0){
            /*At least one result was invalid*/
            set_results(results_lock, &results_final[1], 0);
            pthread_exit(0);
        }
    }
    /*All results were valid!*/
    set_results(results_lock, &results_final[1], 1);
    pthread_exit(0);
}

/*Checks all blocks, multithreaded*/
void *check_blocks_mt(){
    pthread_t tid[9];
    /*Dispatch threads*/
    for (int i = 0; i < 3; ++i){
        for (int j = 0; j < 3; ++j){
            /*i*3+j returns a nice little index number, same as 2d array to pointer offset*/
            block_offset *input = malloc(sizeof(block_offset));
            input->x = i;
            input->y = j;
            pthread_create(&tid[i * 3 + j], NULL, check_block, input);
        }
    }
    /*Wait for threads to conclude*/
    for (int i = 0; i < 9; ++i){
        pthread_join(tid[i], NULL);
    }
    /*Check result*/
    for (int i = 0; i < 9; ++i){
        if (results_blocks[i] == 0){
            /*At least one result was invalid*/
            set_results(results_lock, &results_final[2], 0);
            pthread_exit(0);
        }
    }
    /*All results were valid!*/
    set_results(results_lock, &results_final[2], 1);
    pthread_exit(0);
```

```
}


/* Check all rows in one thread for child process */
void *check_rows_child(void* mem){
    int* results = (int*)mem;
    /* assume row is valid */
    int valid = 1;
    for (int i = 0; i < 9; ++i){
        if (check_row(i) == 0){
            valid = 0;
            break;
        }
    }
    *results =  valid;

    pthread_exit(0);
}

/*Checks all columns in one thread for child process*/
void *check_cols_child(void* mem){
    int* results = (int*)mem;
    /* assume col is valid */
    int valid = 1;
    for (int i = 0; i < 9; ++i){
        if (check_col(i) == 0){
            valid = 0;
            break;
        }
    }
    *results = valid;
    pthread_exit(0);
}


/*Checks all blocks, multithreaded for child process */
void *check_blocks_child(void* mem){
    int* results = (int*)mem;
    /* assume col is valid */
    int valid = 1;
    pthread_t tid[9];
    /*Dispatch threads*/
    for (int i = 0; i < 3; ++i){
        for (int j = 0; j < 3; ++j){
            /*i*3+j returns a nice little index number, same as 2d array to pointer offset*/
            block_offset *input = malloc(sizeof(block_offset));
            input->x = i;
            input->y = j;
            pthread_create(&tid[i * 3 + j], NULL, check_block, input);
        }
    }
    /*Wait for threads to conclude*/
    for (int i = 0; i < 9; ++i){
        pthread_join(tid[i], NULL);
    }
    /*Check result*/
    for (int i = 0; i < 9; ++i){
        if (results_blocks[i] == 0){
            /*At least one result was invalid*/
            valid = 0;
            break;
        }
    }
```

```c
    *results = valid;
    pthread_exit(0);
}

/*1 thread for rows, 1 thread for cols, 9 threads for blocks. 11 worker threads, 1 dispatch thread*/
int option_1(){
    pthread_t tid_row, tid_col, tid_block;
    pthread_create(&tid_row, NULL, check_rows, NULL);
    pthread_create(&tid_col, NULL, check_cols, NULL);
    pthread_create(&tid_block, NULL, check_blocks_mt, NULL);
    /*Bring the boys back home*/
    pthread_join(tid_row, NULL);
    pthread_join(tid_col, NULL);
    pthread_join(tid_block, NULL);
    /*Check results*/
    for (int i = 0; i < 3; ++i){
        if (results_final[i] == 0){
            /*At least one result was invalid*/
            return 0;
        }
    }
    /*All were valid!*/
    return 1;
}

/*9 threads for rows, 9 threads for cols, 9 threads for blocks. 27 worker threads, 3 dispatch thread*/
int option_2(){
    pthread_t tid_row, tid_col, tid_block;
    pthread_create(&tid_row, NULL, check_rows_mt, NULL);
    pthread_create(&tid_col, NULL, check_cols_mt, NULL);
    pthread_create(&tid_block, NULL, check_blocks_mt, NULL);
    /*Bring the boys back home*/
    pthread_join(tid_row, NULL);
    pthread_join(tid_col, NULL);
    pthread_join(tid_block, NULL);
    /*Check results*/
    for (int i = 0; i < 3; ++i){
        if (results_final[i] == 0){
            /*At least one result was invalid*/
            return 0;
        }
    }
    /*All were valid!*/
    return 1;
}

int* shared_mem(){

        /* Memory map the pointer to shared memory object */
    int* mem = mmap(NULL, sizeof(int) * 3, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    /* initialize to all zeros */
    for(int i=0; i<3; i++){
        mem[i]=0;
    }
    return mem;
}

int option_3(){

    pid_t pid;
    pthread_t tid_row, tid_col, tid_block;
    /* pass this as argument for check- functions */
    int* memory = shared_mem();

    for(int i = 0; i<3; i++){
        pid = fork();

        if(pid == 0){ /* Child */
            /* check rows */
            if(i==1){
                pthread_create(&tid_row, NULL, check_rows_child, &memory[0]);
                pthread_join(tid_row, NULL);
                exit(0);
            }
            /* check cols */
            else if(i==2){
                pthread_create(&tid_col, NULL, check_cols_child, &memory[1]);
                pthread_join(tid_col, NULL);
                exit(0);
            }
            /* check blocks */
            else{
                pthread_create(&tid_block, NULL, check_blocks_child, &memory[2]);
                pthread_join(tid_block, NULL);
                exit(0);
            }

        }
        else if(pid<0){
            if(i==1){
                perror("Error in checking rows\n");
            }
            else if(i==2){
                perror("Error in checking columns\n");
            }
            else{
                perror("Error in checking blocks\n");
            }
            exit(1);
        }
    }
    for(int i = 0 ; i<3 ; i++){ /* Parent process*/
        wait(NULL);
    }
    for(int i = 0; i < 3; ++i){
        if (memory[i] == 0){
            /*At least one result was invalid*/
            munmap(memory, sizeof(int)*3); /* clean up shared mem*/
            return 0;
        }
    }

    /*All were valid!*/
    munmap(memory, sizeof(int)*3); /* clean up shared mem*/
    return 1;
}


/* Open input file and format into 2d array board */
int make_board(FILE* inputFile, int board[9][9], char line[80] ){
    if(inputFile == NULL) {
    perror("Error opening file");
    return 1;
    }
    for (int i = 0; i < 9; i++) {
        if (fgets(line, 80, inputFile) == NULL) {
            perror("Error getting line");
            fclose(inputFile);
            return 1;
        }

        /* tokenize after fgets successfully reads a line */
        char* num = strtok(line, " \n");
        for (int j = 0; j < 9 && num != NULL; j++) {
            board[i][j] = atoi(num);
            num = strtok(NULL, " \n");  /* get the next number */
        }
    }

    fclose(inputFile);

    /* print board to verify */
    printf("BOARD STATE IN input.txt: \n");
    for(int i =0; i<9; i++){
        for(int j=0; j<9; j++){
            printf("%d ",board[i][j]);
            if(j%3 ==2){
            printf(" ");
        }
        }
        printf("\n");
        if(i%3 ==2){
            printf("\n");
        }
    }
    return 0;

}

/* Runs stats for t test and stores in csv to be used in Excel file for results */
int run_stats(){
    FILE* fp = fopen("timing_results.csv", "w"); /* making 3 data collection files-- seeing if running ea
    if (!fp) {
        perror("Failed to open file");
        return 1;
    }

    fprintf(fp, "Option,Run,Time(seconds),Mean Time(seconds), %d\n", getpid());
    clock_t begin,end;
    double full_time = 0;
    double timings[50];

    for(int run =0; run<50; run++){
        begin = clock();
        option_1();
        end = clock();

        double run_time = ((double)(end - begin)) / CLOCKS_PER_SEC;

        timings[run] = run_time;
        fprintf(fp, "%d,%d,%f\n", 1, run + 1, timings[run]);
    }
    for(int run =0; run<50; run++){
        begin = clock();
        option_2();
        end = clock();
        double run_time = ((double)(end - begin)) / CLOCKS_PER_SEC;

        timings[run] = run_time;
        fprintf(fp, "%d,%d,%f\n", 2, run + 1, timings[run]);
    }
    fclose(fp);

    for(int run =0; run<50; run++){
        begin = clock();
        option_3();
        end = clock();

        double run_time = ((double)(end - begin)) / CLOCKS_PER_SEC;

        timings[run] = run_time;
        fp = fopen("timing_results.csv", "a");
        fprintf(fp, "%d,%d,%f\n", 3, run + 1, timings[run]);
        fclose(fp);
    }
    return 0;
}



int main(int argc, char** argv){
    /* Get option choice from user */
    int option = atoi(argv[1]);
    FILE* inputFile = fopen("input.txt", "r");

    make_board(inputFile, board, line);

    /* Execute chosen option and get timing */
    int verdict = 2;
    clock_t begin,end;
    begin = clock();
    if(option == 1){
        verdict = option_1();
    }
    else if(option == 2){
        verdict = option_2();
    }
    else if(option == 3){
        verdict = option_3();
    }
    else{
        printf("Invalid option selected. Please select 1,2 or 3.\n");
    }
    end = clock();

    /* Printing verdict with timing*/
    double total_time = ((double) (end - begin)) / CLOCKS_PER_SEC;
    if(verdict == 1){
        printf("SOLUTION: YES(%lf seconds)\n", total_time);
    }
    else if(verdict == 0){
        printf("SOLUTION: NO(%lf seconds)\n", total_time);
    }
    else{
        printf("There was an error in determining the solution. \n");
    }
    run_stats();

    return 0;
}

/* BAD BOARD
1 2 3 4 5 6 8 9 7
1 2 3 4 5 6 8 9 7
1 2 3 4 5 6 8 9 5
1 2 3 4 5 6 8 9 5
1 2 3 4 5 6 8 9 5
1 2 3 4 5 6 8 9 5
1 2 3 4 5 6 8 9 5
1 2 3 4 5 6 8 9 5
1 2 3 4 5 6 8 9 5
*/

/* SOLUTION BOARD
4 3 5 2 6 9 7 8 1
6 8 2 5 7 1 4 9 3
1 9 7 8 3 4 5 6 2
8 2 6 1 9 5 3 4 7
3 7 4 6 8 2 9 1 5
9 5 1 7 4 3 6 2 8
5 1 9 3 2 6 8 7 4
2 4 8 9 5 7 1 3 6
7 6 3 4 1 8 2 5 9
*/
```