CSC 345 Report for Project: 02

Full Name: Robert Bjorklund,

Sam Blamo

Section: 01

0. Project Discussion Note

Sam: As we were working on this project, Bob and I talked greatly about the implementation of the project over the phone, and in person several times. However, most of the actual coding of this project was done separately, at different times. We were able to smoothly build on top of each other's code, with the help of Git and GitHub. A log of our commits was included in the zip file.

Bob: We discussed division of labor early on, and started a git to collaborate. Sam and I each took 2 parts to implement, I took the 3 thread implementation and took charge of testing / collecting statistical data.

Sam implemented the 27 thread and 3 process options, building off the 3 thread solution. We both worked through the stats individually with different tools, arriving at similar conclusions. We also both contributed to this file.

1. Sudoku Solution Validator

1.1. Source Code

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

char* solved = "YES";
int s[9][9];

You, 3 days ago | 1 author (You)
typedef struct {
    int x;
    int y;
} gridCoord;


void* checkRows(void* param){ //check all rows
    int c[9] = {0};
    for (int i = 0; i < 9; i++){
        for (int j = 0; j < 9; j++) {
            c[s[i][j]-1]++;
        }
        for (int j = 0; j < 9; j++){
            if (c[j] != i+1) {
                solved = "NO";
            }
        }
    }
    pthread_exit(0);
}

void* checkCols(void* param){ //check all columns
    int c[9] = {0};
    for (int i = 0; i < 9; i++){
        for (int j = 0; j < 9; j++) {
            c[s[j][i]-1]++;
        }
        for (int j = 0; j < 9; j++){
            if (c[j] != i+1) {
                solved = "NO";
            }
        }
    }
    pthread_exit(0);
}
```

```c
void* checkSquares(void* param){ //check all squares
    int c[9] = {0};
    gridCoord* sq = (gridCoord*) param;
    for(int i = 0; i < 9; i++){
        int startX = sq[i].x;
        int startY = sq[i].y;
        for(int x = startX; x < startX+3; x++){
            for(int y = startY; y < startY+3; y++){
                c[s[x][y]-1]++;
            }
        }

        for (int j = 0; j < 9; j++) {
            if (c[j] != i+1) {
                solved = "NO";
            }
        }
    }
    pthread_exit(0);
}

void* checkSingleRow(void* param){ //check a single row.  param is an integer pointer to the index of the row to check
    int c[9] = {0};
    int row = (*(int *)param);
    for (int i = 0; i < 9; i++) {
        c[s[row][i]-1]++;
    }
    for (int i = 0; i < 9; i++){
        if (c[i] != 1) {
            solved = "NO";
        }
    }
    pthread_exit(0);
}

void* checkSingleCol(void* param){ //check a single column.  param is an integer pointer to the index of the column to check
    int c[9] = {0};
    int col = (*(int *)param);
    for (int i = 0; i < 9; i++) {
        c[s[i][col]-1]++;
    }
    for (int i = 0; i < 9; i++){
        if (c[i] != 1) {
            solved = "NO";
        }
    }
    pthread_exit(0);
}
```

```c
}

void* checkSingleSquare(void* param){ //checks a single square.  param is a pointer to a gridCoord, which contains row and col index of the first number in the square
    int c[9] = {0};
    gridCoord* currPos = (gridCoord*) param;
    int startX = currPos->x;
    int startY = currPos->y;
    for(int x = startX; x < startX+3; x++){
        for(int y = startY; y < startY+3; y++){
            c[s[x][y]-1]++;
        }
    }

    for (int i = 0; i < 9; i++) {
        if (c[i] != 1) {
            solved = "NO";
        }
    }
    pthread_exit(0);
}
```

```c

void multiProcessCheck(){ //check all rows, columns, and squares, with a process for each category
    int SIZE = 4096;
    char* name = "solution?";
    char* ptr = "";
    int shm_fd;
    int c[9] = {0};
    pid_t pid1 = fork();
    if(pid1 == 0){
        shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
        ftruncate(shm_fd, SIZE);
        ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
        for (int i = 0; i < 9; i++){
            for (int j = 0; j < 9; j++) {
                c[s[i][j]-1]++;
            }
            for (int j = 0; j < 9; j++){
                if (c[j] != i+1) {
                    sprintf(ptr, "%s", "NO");
                }
            }
        }
        exit(0);
    }
    pid_t pid2 = fork();
    if(pid2 == 0){
        shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
        ftruncate(shm_fd, SIZE);
        ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
        for (int i = 0; i < 9; i++){
            for (int j = 0; j < 9; j++) {
                c[s[j][i]-1]++;
            }
            for (int j = 0; j < 9; j++){
                if (c[j] != i+1) {
                    sprintf(ptr, "%s", "NO");
                }
            }
        }
        exit(0);
    }
    pid_t pid3 = fork();
    if(pid3 == 0){
        shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
        ftruncate(shm_fd, SIZE);
        ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
        gridCoord* sqPos = malloc(9 * sizeof(gridCoord));
        int index = 0;
        for(int i = 0; i < 3; i++){
            for(int j = 0; j <= 6; j+=3){
                sqPos[index].x = i * 3;
                sqPos[index].y = j;
                index++;
            }
        }

        for(int i = 0; i < 9; i++){
            int startX = sqPos[i].x;
            int startY = sqPos[i].y;
            for(int x = startX; x < startX+3; x++){
                for(int y = startY; y < startY+3; y++){
                    c[s[x][y]-1]++;
                }
            }
```

```c
                for (int j = 0; j < 9; j++) {
                    if (c[j] != i+1) {
                        sprintf(ptr, "%s", "NO");
                    }
                }
            }
            exit(0);
        }

        if(pid1 > 0 && pid2 > 0 && pid3 > 0){
            waitpid(pid1, NULL, 0);
            waitpid(pid2, NULL, 0);
            waitpid(pid3, NULL, 0);
            shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
            ftruncate(shm_fd, SIZE);
            ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
            if(strlen(ptr) != 0){
                solved = ptr;
            }
            shm_unlink(name);
        }
}
```

```c
int main(int argc, char **argv){
    FILE *fp = fopen("input.txt","r");
    time_t st = clock();
    time_t msPerS = CLOCKS_PER_SEC/100000;
    int option = atoi(argv[1]);
    for (int i = 0; i < 9; i++) {
        fscanf(fp,"%d %d %d %d %d %d %d %d %d",&s[i][0],&s[i][1],&s[i][2],&s[i][3],&s[i][4],&s[i][5],&s[i][6],&s[i][7],&s[i][8]);
        printf("%d %d %d %d %d %d %d %d %d\n",s[i][0],s[i][1],s[i][2],s[i][3],s[i][4],s[i][5],s[i][6],s[i][7],s[i][8]);
    }

    if(option == 1){       You, 3 days ago • Organized Branch …
        // for(int att = 0; att < 10; att++){
            pthread_t threads[3];
            gridCoord* sqPos = malloc(9 * sizeof(gridCoord));
            int index = 0;
            for(int i = 0; i < 3; i++){
                for(int j = 0; j <= 6; j+=3){
                    sqPos[index].x = i * 3;
                    sqPos[index].y = j;
                    index++;
                }
            }
            pthread_create(&threads[0], NULL, checkRows, NULL);
            pthread_create(&threads[1], NULL, checkCols, NULL);
            pthread_create(&threads[2], NULL, checkSquares, sqPos);

            pthread_join(threads[0], NULL);
            pthread_join(threads[1], NULL);
            pthread_join(threads[2], NULL);
        // }
        // time_t ft = clock() - st;
        // printf("SOLUTION: %s (%ld ns)\n",solved, ft/msPerS);
    }

    if(option == 2){
        // for(int att = 0; att < 10; att++){
            pthread_t threads[27];
            gridCoord* sqPos = malloc(9 * sizeof(gridCoord));
            int index = 0;

            for(int i = 0; i < 3; i++){
                for(int j = 0; j <= 6; j+=3){
                    sqPos[index].x = i * 3;
                    sqPos[index].y = j;
                    index++;
                }
            }
            int params[9] = {0,1,2,3,4,5,6,7,8};
            for(int i = 0; i < 9; i++) {
                pthread_create(&threads[i], NULL, checkSingleRow, &params[i]);
                pthread_create(&threads[i+9], NULL, checkSingleCol, &params[i]);
                pthread_create(&threads[i+18], NULL, checkSingleSquare, &sqPos[i]);
            }
            for(int i = 0; i < 27; i++) {
                pthread_join(threads[i], NULL);
            }
        // }

        // if(att % 1000 == 0){
            //     printf("Here is the index:%d, (%ld ms)\n", att, (clock() - st));
            // }
        // time_t ft = clock() - st;
        // printf("SOLUTION: %s (%ld ns)\n",solved, ft/msPerS);
    }
```

```c
    if(option == 3){
        // for(int att = 0; att < 10; att++){
            multiProcessCheck();
        // }

        // time_t ft = clock() - st;
        // printf("SOLUTION: %s (%ld ns)\n",solved, ft/msPerS);
    }

    fclose(fp);
    time_t ft = clock() - st;
    printf("SOLUTION: %s (%ld ns)\n",solved, ft/msPerS);

    return 0;
}
```

1.2. Output

```
phoenix@thephoenix:~/Documents/sudoku$ ./main 3
1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 1
3 4 5 6 7 8 9 1 2
4 5 6 7 8 9 1 2 3
5 6 7 8 9 1 2 3 4
6 7 8 9 1 2 3 4 5
7 8 9 1 2 3 4 5 6
8 9 1 2 3 4 5 6 7
9 1 2 3 4 5 6 7 8
SOLUTION: NO (29 ns)
```

```
phoenix@thephoenix:~/Documents/sudoku$ ./main 2
1 2 3 4 5 6 7 8 9
4 5 6 7 8 9 1 2 3
7 8 9 1 2 3 4 5 6
8 9 7 2 3 1 5 6 4
5 6 4 8 9 7 2 3 1
2 3 1 5 6 4 8 9 7
3 1 2 6 4 5 9 7 8
9 7 8 3 1 2 6 4 5
6 4 5 9 7 8 3 1 2
SOLUTION: YES (170 ns)
```

2. Breakdown of Sudoku Solution Validator Implementation

2.1. int main(int argc, char** argv)

The main function's primary responsibility is interpreting the *option* parameter given by the user and calling the other functions from within the code.

Option 1 uses the strategy of making three separate threads to validate the sudoku within input.txt; One thread to check every row, one to check every column, and one to check every square.

Option 2 uses 27 separate threads to check each column, row, and square for the digits 1-9 respectively. The 27 threads are stored within a thread array named *threads*.

Option 3 uses the multi-process approach, having three children processes to check the given input's rows, columns, and squares to validate whether it's solved.

It is also important to note that when the program begins execution, the time is recorded in the *st* variable until the given option finishes executing, and we return the time taken to execute along with whether or not the sudoku puzzle given was a valid solution.

2.2. void* checkRows(void* param)

This thread function is called in the *(option == 1)* if-statement block in *main()*. The *param* argument passed is NULL.

This function's responsibility is to check every row within input.txt to validate that each row only contains one occurrence of a value within the range of 1-9. If there is more than one occurrence of a specific value at any row, the global variable *solved* will be set to *NO*.

2.3. void* checkCols(void* param)

This thread function is called in the *(option == 1)* if-statement block in *main()*. The *param* argument passed is NULL.

This function's responsibility is to check every column within input.txt to validate that each column only contains one occurrence of a value within the range of 1-9. If there is more than one occurrence of a specific value at any column, the global variable *solved* will be set to *NO*.

2.4. void* checkSquares(void* param)

This thread function is called within the *(option == 1)* if-statement block in *main()*. The *param* argument passed is the memory address to an array of user-defined structs *gridCoord*. A gridCoord struct has attributes int *x* and int *y* which are used to properly loop through non-overlapping squares in the given sudoku input.

This function's responsibility is to check every non-overlapping 3x3 square within input.txt to validate that each square only contains one occurrence of a value within the range of 1-9. If there is more than one occurrence of a specific value at any square, the global variable *solved* will be set to *NO*.

2.5. void* checkSingleRow(void* param)

This thread function is called in the *(option == 2)* if-statement block in *main()*. The *param* argument passed is the memory address of the corresponding row index in the *params* array initialized in the (*option* == *2*) if-statement block.

This function's responsibility is to check the row that corresponds with the index passed in the function's parameter and validate that the row only contains one occurrence of a value within the range of 1-9. If there is more than one occurrence of a specific value in the given row, the global variable *solved* will be set to *NO*.

2.6. void* checkSingleColumn(void* param)

This thread function is called in the *(option == 2)* if-statement block in *main()*. The *param* argument passed is the memory address of the corresponding column index in the *params* array initialized in the (*option* == *2*) if-statement block.

This function's responsibility is to check the column that corresponds with the index passed in the function's parameter and validate that the column only contains one occurrence of a value within the range of 1-9. If there is more than one occurrence of a specific value in the given column, the global variable *solved* will be set to *NO*.

2.7. void* checkSingleSquare(void* param)

This thread function is called in the *(option == 2)* if-statement block in *main()*. The *param* argument passed is the memory address to an element within an array of user-defined structs *gridCoord*. A gridCoord struct has attributes int *x* and int *y* which are used to properly loop through non-overlapping squares in the given sudoku input.

This function's responsibility is to check the square that corresponds with the index passed in the function's parameter and validate that the square only contains one occurrence of a value within the range of 1-9. If there is more than one occurrence of a specific value in the given square, the global variable *solved* will be set to *NO*.

2.8. void multiProcessCheck()

This function is called in the *(option == 3)* if-statement block in *main()*. Its responsibility is to create three child processes and use each child process to validate if the rows, columns, and squares are solved respectively. If the given sudoku input is not a valid solution, the parent will change the global variable *solved* to reflect that.

3. Breakdown of Statistical Experimentation.

3.1. Preface

Since we were tasked to provide an analytical conclusion if there was any statistical difference between any of the methods, we used the *t-test* to determine whether our hypothesis was correct.

method 1 results:	method 2 results:	method 3 results:
81	312	39
65	272	42
66	327	37
61	317	35
59	325	38
56	218	42
62	251	38
57	275	38
64	308	45
59	210	37
53	248	34
62	284	35
70	280	37
63	310	36
53	244	35
60	259	37
48	278	44
68	251	44
59	301	40
64	292	45
61	280	39
48	313	39
46	268	39
49	301	43
50	271	44
61	233	39
60	257	37
57	278	38
47	261	36
44	250	36
66	252	38
60	249	34
63	242	37
61	237	35
61	257	36
44	182	39
42	241	38
45	293	37
48	260	37
50	254	35
60	266	37
60	294	41
60	365	39
59	403	39
58	331	39
68	258	39
63	257	40
68	275	38
59	289	38
60	250	35

method 1 results:	
Mean	58.16
Standard Error	1.116247
Median	60
Mode	60
Standard Deviation	7.893061
Sample Variance	62.30041
Kurtosis	0.371706
Skewness	-0.08324
Range	39
Minimum	42
Maximum	81
Sum	2908
Count	50

method 2 results:	
Mean	274.58
Standard Error	5.412917
Median	269.5
Mode	257
Standard Deviation	38.2751
Sample Variance	1464.983
Kurtosis	2.042952
Skewness	0.767066
Range	221
Minimum	182
Maximum	403
Sum	13729
Count	50

method 3 results:	
Mean	38.38
Standard Error	0.400907
Median	38
Mode	39
Standard Deviation	2.834842
Sample Variance	8.036327
Kurtosis	0.155985
Skewness	0.800024
Range	11
Minimum	34
Maximum	45
Sum	1919
Count	50

t-Test: Two-Sample Assuming Unequal Variances

	method 1 results:	*method 2 results:*
Mean	58.16	274.58
Variance	62.30040816	1464.983265
Observations	50	50
Hypothesized Mean Difference	0	
df	53	
t Stat	-39.15818378	
P(T<=t) one-tail	4.23126E-41	
t Critical one-tail	1.674116237	
P(T<=t) two-tail	8.46E-41	
t Critical two-tail	2.005745995	

t-Test: Two-Sample Assuming Unequal Variances

	method 2 results:	*method 3 results:*
Mean	274.58	38.38
Variance	1464.983265	8.036326531
Observations	50	50
Hypothesized Mean Difference	0	
df	50	
t Stat	43.51716892	
P(T<=t) one-tail	1.03031E-41	
t Critical one-tail	1.675905025	
P(T<=t) two-tail	2.06061E-41	
t Critical two-tail	2.008559112	

t-Test: Two-Sample Assuming Unequal Variances

	method 1 results:	*method 3 results:*
Mean	58.16	38.38
Variance	62.30040816	8.036326531
Observations	50	50
Hypothesized Mean Difference	0	
df	61	
t Stat	16.67708676	
P(T<=t) one-tail	1.05829E-24	
t Critical one-tail	1.670219484	
P(T<=t) two-tail	2.11657E-24	
t Critical two-tail	1.999623585	

3.2. Results

3.2.1. Method 1 vs Method 2

Null hypothesis: There is no statistically significant difference between the means of Method 1 and Method 2.

Alternative Hypothesis: There is a statistically significant difference between the means of Method 1 and Method 2.

Assumptions:

- Observations are independent? **Yes.**
- Number of samples is greater than 30 in each group? **Yes.**

Conclusion: After performing a t-test between the two sample groups, method 1 and method 2, a p-value that was closely estimated to 0 was returned (8.46e-41). This is significantly less than alpha, and therefore concludes that we have enough evidence to prove that there is a statistically significant difference between the means of Method 1 and Method 2.

3.2.2. Method 3 vs Method 2

Null hypothesis: There is no statistically significant difference between the means of Method 3 and Method 2.

Alternative Hypothesis: There is a statistically significant difference between the means of Method 3 and Method 2.

Assumptions:

- Observations are independent? **Yes.**
- Number of samples is greater than 30 in each group? **Yes.**

Conclusion: After performing a t-test between the two sample groups, method 3 and method 2, a p-value that was closely estimated to 0 was returned (2.06061e-41). This is significantly less than alpha, and therefore concludes that we have enough evidence to prove that there is a statistically significant difference between the means of Method 3 and Method 2.

3.2.3. Method 3 vs Method 1

Null hypothesis: There is no statistically significant difference between the means of Method 3 and Method 1.

Alternative Hypothesis: There is a statistically significant difference between the means of Method 3 and Method 1.

Assumptions:
- Observations are independent? **Yes.**
- Number of samples is greater than 30 in each group? **Yes.**

Conclusion: After performing a t-test between the two sample groups, method 3 and method 1, a p-value that was closely estimated to 0 was returned (2.11657e-24). This is significantly less than alpha, and therefore concludes that we have enough evidence to prove that there is a statistically significant difference between the means of Method 3 and Method 1.