Project 2

Multithreaded Programming

Alex Benasutti and Alex Van Orden

CSC345-01

October 18 2019

C Code:

```
/*
* Alex Benasutti and Alex Van Orden
* CSC345-01
* October 18 2019
* Project 2 - Multithreaded Programming
*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>

/* Results array carries the validation for each row, column, and grid on the board */
int result[27] = {0};

/* Create struct to pass into each thread as parameters */
typedef struct
{
    int row;
    int col;
    int (*board)[9];
} parameters;

/* Check grid in one thread */
void *check_grid(void *params)
{
    parameters *data = (parameters *) params;
    int startRow = data->row;
    int startCol = data->col;
    int valid[10] = {0};                    /* Zero array for each number in the grid */
    int i, j;
    for (i=startRow;i<startRow+3;i++)       /* Loop through each mini board aka grid */
    {
        for(j=startCol;j<startCol+3;j++)
        {
            int val = data->board[i][j];    /* Assign val to each number in the grid */
            if (valid[val] != 0) pthread_exit(NULL);
            else valid[val] = 1;            /* If there are no duplicates, place a 1 in val's
position */
        }
    }
    result[startRow+startCol/3] = 1;        /* Grid passed validation test */
    pthread_exit(NULL);                     /* Exit thread */
}

/* Check single row in one thread */
void *check_row(void *params)
{
    parameters *data = (parameters *) params;
    int row = data->row;
    int valid[10] = {0};                    /* Zero array for each number in the row */
    int j;
    for (j=0;j<9;j++)                       /* Loop through the contents of each row */
    {
        int val = data->board[row][j];      /* Assign val to each number in the row */
        if (valid[val] != 0) pthread_exit(NULL);
        else valid[val] = 1;                /* If there are no duplicates, place a 1 in val's
position */
    }
    result[9+row] = 1;                      /* Row passed validation test */
    pthread_exit(NULL);                     /* Exit thread */
}

/* Check single column in one thread */
void *check_col(void *params)
{
    parameters *data = (parameters *) params;
    int col = data->col;
    int valid[10] = {0};                    /* Zero array for each number in the column */
    int i;
    for (i=0;i<9;i++)                       /* Loop through the contents of each column */
    {
        int val = data->board[i][col];      /* Assign val to each number in the column */
        if (valid[val] != 0) pthread_exit(NULL);
        else valid[val] = 1;                /* If there are no duplicates, place a 1 in val's
position */
    }
    result[18+col] = 1;                     /* Column passed validation test */
    pthread_exit(NULL);                     /* Exit thread */
}

/* Check all rows using one thread */
void *check_rows(void *params)
{
    parameters *data = (parameters *) params;
    int j,k;
    for (j=0;j<9;j++)
    {
        int valid[10] = {0};                /* Zero array for each number per row */
        for (k=0;k<9;k++)                   /* Loop through the contents of every row */
        {
            int val = data->board[j][k];    /* Assign val to each number per row */
            if (valid[val] != 0)
            {
                /* 1 */ pthread_exit(NULL); /* Method 1 exits thread when invalid */
                ///* 2 */ result[9+j] = 2;  /* Method 2 breaks to next row when invalid */
                ///* 2 */ break;
            }
            else valid[val] = 1;            /* If no duplicates in that row, place a 1 in
val's position */
        }
        if (result[9+j] == 2) result[9+j] = 0;
        else result[9+j] = 1;               /* Row passed validation test */
    }
    pthread_exit(NULL);                     /* Exit thread */
}

/* Check all columns using one thread */
void *check_cols(void *params)
{
    parameters *data = (parameters *) params;
    int i,j,k;
    for (i=0;i<9;i++)
    {
        int col[9];
        for (j=0;j<9;j++) col[j] = data->board[j][i];   /* Assign values to column array */
        int valid[10] = {0};                 /* Zero array for each number per column */
        for (k=0;k<9;k++)                    /* Loop through the contents of every column */
        {
            int val = col[k];                /* Assign val to each number per column */
            if (valid[val] != 0)
            {
                /* 1 */ pthread_exit(NULL); /* Method 1 exits thread when invalid */
                ///* 2 */ result[18+i] = 2; /* Method 2 breaks to next column when invalid */
                ///* 2 */ break;
            }
            else valid[val] = 1;             /* If no duplicates in that column, place a 1 in
val's position */
        }
        if (result[18+i] == 2) result[18+i] = 0;
        else result[18+i] = 1;               /* Column passed validation test */
    }
    pthread_exit(NULL);                      /* Exit thread */
}

int main(int argc, char** argv)
{
    int option = atoi(argv[1]);
    int num_threads;

    /*
        Process option argument
        option 1 -> 9 grid threads, 1 row thread, 1 column thread
        option 2 -> 9 grid threads, 1 row thread, 9 column threads
        option 3 -> 9 grid threads, 9 row threads, 9 column threads

        Note for options 1 and 2:
        the single row/column threads exit the thread by default when encountered by an invalid
row/column
        a second method is commented out and can be used to continue evaluating the threads
regardless of invalid row/column
    */
    if      (option == 1) num_threads = 11;
    else if (option == 2) num_threads = 19;
    else if (option == 3) num_threads = 27;

    FILE *input;
    input = fopen("input.txt","r");

    /* Initialize board */
    int sudoku[9][9], i, j;

    printf("BOARD STATE IN input.txt\n");

    /* Read and assign sudoku board from input.txt */
    for (i=0;i<9;i++)
    {
        for (j=0;j<9;j++)
```

```c
        {
            fscanf(input, "%1d", &sudoku[i][j]);
            printf("%d ", sudoku[i][j]);
        }
        printf("\n");
    }

    /* Start tracking time at the beginning of thread creation/processing */
    clock_t t;
    t = clock();

    /* Create amount of threads specified by the option */
    pthread_t threads[num_threads];

    int thread_idx = 0;

    /* Create threads */
    for (i=0;i<9;i++)
    {
        for (j=0;j<9;j++)
        {
            /* Check Grid Threads */
            if (i % 3 == 0 && j % 3 == 0) /* Created for every option */
            {
                /* Create and assign struct as parameter for thread */
                parameters *gridData = (parameters *) malloc(sizeof(parameters));
                gridData->row = i;
                gridData->col = j;
                gridData->board = sudoku;
                pthread_create(&threads[thread_idx++], NULL, check_grid, gridData);
            }
            /* Check Row Threads */
            if (j == 0 && option == 3) /* Only for the 27 thread option */
            {
                /* Create and assign struct as parameter for thread */
                parameters *rowData = (parameters *) malloc(sizeof(parameters));
                rowData->row = i;
                rowData->col = j;
                rowData->board = sudoku;
                pthread_create(&threads[thread_idx++], NULL, check_row, rowData);
            }
            /* Check Column Threads */
            if (i == 0 && (option == 2 || option == 3)) /* For 19 and 27 thread options */
            {
                /* Create and assign struct as parameter for thread */
                parameters *colData = (parameters *) malloc(sizeof(parameters));
                colData->row = i;
                colData->col = j;
                colData->board = sudoku;
                pthread_create(&threads[thread_idx++], NULL, check_col, colData);
            }
        }
    }

    if (option == 1 || option == 2) /* For 11 and 19 thread options */
    {
        /* Create and assign struct for referencing sudoku board in thread */
        parameters *rowsData = (parameters *) malloc(sizeof(parameters));
        rowsData->board = sudoku;
        /* Create thread to validate all rows on the board */
        pthread_create(&threads[thread_idx++], NULL, check_rows, rowsData);
    }
    if (option == 1) /* For 11 thread option only */
    {
        /* Create and assign struct for referencing sudoku board in thread */
        parameters *colsData = (parameters *) malloc(sizeof(parameters));
        colsData->board = sudoku;
        /* Create thread to validate all columns on the board */
        pthread_create(&threads[thread_idx++], NULL, check_cols, colsData);
    }

    /* Join all threads */
    for (i=0;i<num_threads;i++) pthread_join(threads[i], NULL);

    /* End clock */
    t = clock() - t;
    double time_taken = ((double)t/CLOCKS_PER_SEC);

    /* Evaluate results - if a 0 exists, validation failed */
    for (i=0;i<num_threads;i++)
    {
        if (result[i] == 0)
        {
            printf("SOLUTION: NO (%f seconds)\n", time_taken);
            return 1;
        }
    }
    printf("SOLUTION: YES (%f seconds)\n", time_taken);

    return 0;
}
```

Implementation

Highlighted in **red** is the implementation for reading in a text document “input.txt”, where the contents of the text file contains a 9x9 matrix of space-separated numbers to be validated by the program. Upon execution, the program obtains a file pointer to input.txt and scans its contents into the 2D sudoku array using the fscanf() method.

Highlighted in **blue** shows the code that the program prints to the console upon execution. The 9x9 sudoku puzzle board state read from “input.txt” is printed out in addition to the validity of the solution and the execution time. The time taken for execution is measured from the beginning of thread creation to after the threads are joined.

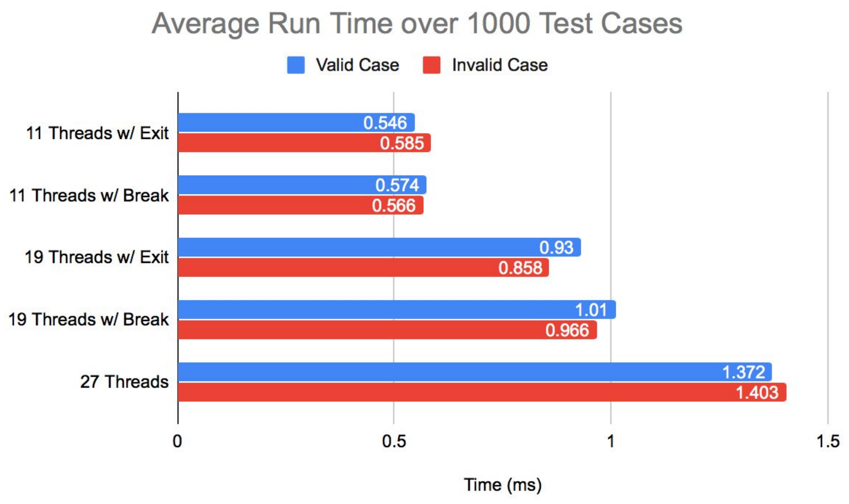
Highlighted in **orange** is the code of various threading implementations used by the program. We compare these implementations later to evaluate the effectiveness of multithreading on this simple sudoku program. The number of threads ran is determined the “option” argument the user provides when running the program on the command line. Option 1 creates 11 threads, where 9 threads are used to check each 3x3 grid on the board, and 1 thread each to check the 9 rows and 9 columns of the board. Option 2 creates 19 threads, where instead we use 9 threads to check each column of the board. Option 3 creates 27 threads where we finally use 9 threads to check each row as well.

A *struct* is created for each thread containing variables for the row and column index as well as a pointer to the sudoku board contents. In cases for threads that check all 9 rows or columns, only the board pointer is declared. These structs are passed through as parameters to the functions the worker threads run (i.e. check_col(), check_rows(), check_grid(), etc.) and are evaluated for their validity on the board. Each of these functions evaluate a grid, row(s), or column(s) for whether they contain unique entries 1-9. If so, a global array *result[]* will place a 1 in one of its 27 indices - showing that row, column, or grid was validated. At the end of the program, a loop checks this array to see whether every index contains a 1, outputting “SOLUTION: YES” if true and “SOLUTION: NO” if false.

Performance Analysis

To determine which threading method was best, we ran a statistical analysis on the run times of 1000 test cases for each method. The 11 and 19 thread methods have two variants that change how the 1-thread-9-columns and 1-thread-9-rows threads are handled. One variant causes the thread to exit upon finding an invalid row/column, while the other validates all rows and columns while only breaking out of ones that are invalid. These variants are denoted as "Exit" and "Break", respectively, as shown in the following graphs.

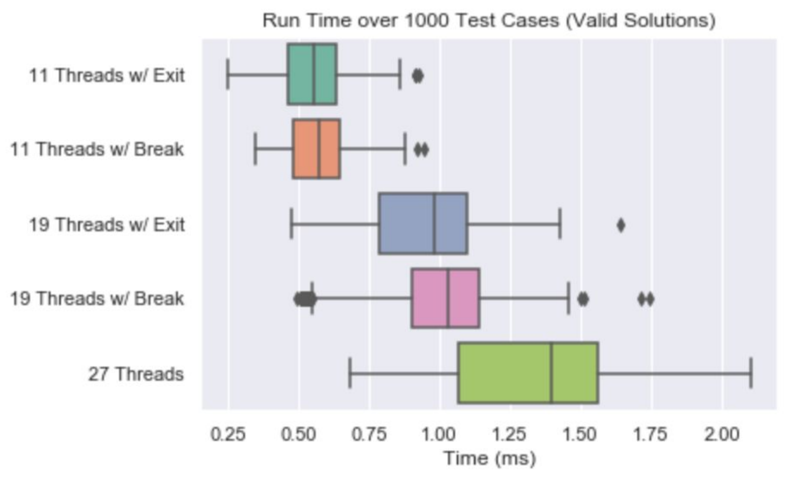
Graph 1 is a bar plot comparing the average times of valid and invalid solutions with each of the threading options. It is apparent that the number of threads is proportional to the average run time. Three out of five options also proved to show that the average run time for an invalid case is lower than a valid one. This should actually be expected for all cases utilizing a single thread for 9 rows/columns considering the thread is either breaking out of loops per invalid case or exiting the thread early. However, the average run time does not help us confirm whether breaking or exiting is more efficient.



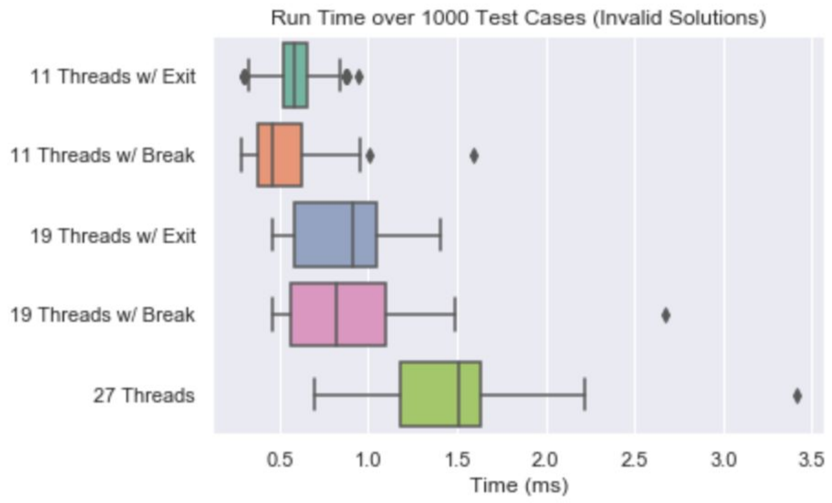
Graph 1. Average Run Time over 1000 Test Cases

Graphs 2 and 3 show boxplots for the run times for each of the 1000 test cases, giving a more accurate representation of the test data. Again, run time proportionally increased with thread, further confirming the best threading option.

To determine whether "Exit" or "Break" was more efficient, we want to look at Graph 3, as invalid cases are the only times when "exit" or "break" instructions would be executed. Observation of the graph shows that "Exit" shows a lower interquartile range and lower min-max range than that of "Break", showing that there is a slightly better consistency of run times for this method. However, in the case of a simple sudoku validation program, the slight time advantage is almost negligible and either method could be paired with the 11 thread option to achieve an optimal run time.



Graph 2. Run Time over 1000 Test Cases (Valid Solutions)



Graph 3. Run Time over 1000 Test Cases (Invalid Solutions)

In the case of lower complexity programs like the sudoku validator, less threads proves to be a faster option when trying to achieve an optimal run time. This is due to the computational expenses of thread creation and handling overhead becoming overkill for simpler programs. However, multi-threading will definitely have advantages and performance improvements over single threads for more complex programs.