Adhesh Reghu Kumar
COE18B001

Operating Systems Lab

Mid-Sem Exam

Develop a package LATIN in a multithreaded fashion that generates a LATIN square of order n by n input by the user. The generation should be done in a multithreaded fashion. Extend the package to check if two input square matrices (latin matrices) are orthogonal. Refer to the internet to explore on latin squares and orthogonality checking. Compare the performance of the multithreaded version with their serial counterparts and the effect of no of processes on the computation time.

Logic Used

In this question, there are two operations or functions we wish to perform.

Part-1: Given an input N, we wish to generate a N*N Latin Square Matrix.

Part-2: Given two Latin Square matrices of a given order we wish to test the orthogonality of the matrices.

We are also required to develop each of these parts in a multi-threaded fashion. We are also required to compare their performance with their serial counterparts.

Let us take a look at each part separately.

PART - 1

Input: Dimension of latin square matrix to be generated.

Algorithm Used:

Let us consider a N*N matrix.

We fill this matrix following the given procedure:

- In the first row of the matrix, we fill the cells with values from 1 to N.

- In the second row of the matrix the numbers are shifted to the right by one column - we fill 1 in the 2nd column of the matrix and so on. If while filling the matrix we reach the Nth column, we then move back to the first column (circular fashion). We keep filling until we have filled in every cell.

- In this manner the remaining rows are filled.

If we notice, we can observe that filling of each row is an independent process and can be done parallelly with the help of N threads.
So we create N threads, on for each row and fill in the values of the matrix. We pass the row number as an argument to each thread runner - `latin_runner`.

Let us take a look at that thread runner function

```
void *latin_runner(void *arg)
{
    // Get the argument in appropriate type
    int row = *((int *)arg);

    // Fill in the row of the matrix
    int i = row;                                        // define
start position in the matrix
    int fill = 1;                                      // define the
value to be filled in
    while(fill <= N)
    {
       arr[row][i] = fill;
       i = (i + 1)%N;
       ++fill;
    }

    pthread_exit(NULL);
}
```

PART - 2

Input: Given two Latin square matrices A and B of given order, check if they are orthogonal

Algorithm Used:

- To check if the two Latin squares are orthogonal, we can follow the following algorithm:
- We superimpose the cell values of A and B to get the cell values of C .
- Once we have the matrix C computed, we can then check if each entry in that matrix is unique.
    - If all elements are unique then the matrices are orthogonal.
    - Else not orthogonal.

We can notice that checking the uniqueness of each element in the matrix C can be done in a multi-threaded manner by calling order*order threads - one for each element in the matrix. The thread runner ortho_runner(void *arg) will take a struct Position (which stores row and col) as argument and checks the uniqueness of the element C[row][col] by comparing against all cell values following that given cell.

Look at the corresponding functions.

```
int isOrthogonal(int order)
{
    // Let us first eliminate the trivial cases

    // Based on Eulers conjecture
    if(order == 6)
       return -1;

    // Now for the remaining orders - let us check the the orthogonality

    /* -- Step - 1: Superimpose Matrix A and Matrix B to form Matrix C -- */
    int i,j;
    int num_digits_B = 0;
    int exp = 0;
    for(i=0;i<order;++i)
    {
```

```
       for(j=0;j<order;++j)
       {
              num_digits_B = floor(log10(B[i][j]) + 1);
              exp = pow(10,num_digits_B);
              C[i][j] = A[i][j] * exp + B[i][j];
       }
    }

    /* --  Step -2: Check uniqueness of each element in the Matrix C in
multi-threaded fashion. -- */

    // Create (order * order) threads for each entry in the C matrix
    pthread_t tid[order * order];
    int thread_no = 0;

    // Create and init attr with default values
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    // Call the runner function
    struct Position pos[order * order];
    for(i=0;i<order;++i)
    {
       for(j=0;j<order;++j)
       {
              pos[thread_no].row = i;
              pos[thread_no].col = j;
              pthread_create(&tid[thread_no],&attr,ortho_runner,(void
*)&pos[thread_no]);
              ++thread_no;
       }
    }

    // Join all the threads
    thread_no =0;
    for(i=0;i<order;++i)
       for(j=0;j<order;++j)
              pthread_join(tid[thread_no++],NULL);

    /* --  Step -3: Evaluate the status of global variable orthogonalFlag and
return status -- */
    if(orthogonalFlag == 0)
       return 1;
```

```
    return -1;
}

// Define the thread runner to check the orthogonality of the input matrix
void *ortho_runner(void *arg)
{
    struct Position *position = (struct Position*)arg;
    int row = position->row;
    int col = position->col;

    int i,j;
    int flag = 0;
    for(i=row;i<order;++i)
    {
       for(j=col;j<order;++j)
       {
             // Skip for that position
             if(i == row && j == col)
                   continue;

             if(C[i][j] == C[row][col])
             {
                   ++flag;
                   break;
             }
       }

       if(flag!=0)
             break;
    }

    if(flag != 0)
       ++orthogonalFlag;

    // Exit the thread
    pthread_exit(NULL);
}
```

Look at the entire multi-threaded code

Code

```
/*****************************************
* AUTHOR : AdheshR *
* Problem Description:     Part - I: Multi-threaded version of Latin Matrix Generator
                         Part - II: Given two input latin squares - check their
orthogonality*
*****************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <pthread.h>
#include <math.h>
#define MAX 1000

// Define the array and length as global variable
int arr[MAX][MAX];
int N;

// Define global variables for PART II - Checking orthogonality
int A[MAX][MAX];
int B[MAX][MAX];
int C[MAX][MAX];
int order;
int orthogonalFlag = 0;

void orthogonalityCheck();

// Create a struct to pass the Positional arguments to the thread runner
struct Position
{
    int row;
    int col;
};

void *latin_runner(void *arg);
void *ortho_runner(void *arg);
```

```c
int main()
{
    /* --  PART-I: Generate the Latin square in multi-threaded manner
            Each row is generated in a different thread.    -- */

    // Input the Dimension of the Latin Square Matrix
    printf("\n************************ PART-I: Latin Square Generator.
**************************\n\n");
    printf("Enter the Dimension of the Latin Square Matrix to be generated.\n");
    scanf("%d",&N);

    // Create N threads
    pthread_t tid[N];

    // Create and init attr with default values
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    // Call the thread runner for each thread
    int i;
    for(i=0;i<N;++i)
    {
       // Define the argument to be passed to each thread runner
       int *row = malloc(sizeof(int));
       *row = i;
       pthread_create(&tid[i],&attr,latin_runner,row);
    }

    // Join all the N threads
    for(i=0;i<N;++i)
       pthread_join(tid[i],NULL);

    // Print the matrix
    int j;
    printf("\nA Latin Square of order %d is:\n",N);
    for(i=0;i<N;++i)
    {
       for(j=0;j<N;++j)
             printf("%d ",arr[i][j]);
       puts("");
    }
```

```
    /* -- PART II - Checking Orthogonality of two Input Latin Square matrices -- */
    orthogonalityCheck();

    return 0;
}

void *latin_runner(void *arg)
{
    // Get the argument in appropriate type
    int row = *((int *)arg);

    // Fill in the row of the matrix
    int i = row;                                        // define start position
in the matrix
    int fill = 1;                                      // define the value to be
filled in
    while(fill <= N)
    {
       arr[row][i] = fill;
       i = (i + 1)%N;
       ++fill;
    }

    pthread_exit(NULL);
}

// Define the function to perform the PART II of the question
void orthogonalityCheck()
{

    printf("\n********************* PART-II: Orthogonality Check of Two Latin Square
Matrices. **********************\n\n");

    // Get the inputs from the user
    printf("Enter the order of Latin Squares.\n");
    scanf("%d",&order);

    // Loop variables
    int i,j;

    // Accept Square A
    printf("\nEnter the elements of Latin Square A.\n");
    for(i=0;i<order;++i)
```

```
       for(j=0;j<order;++j)
             scanf("%d",&A[i][j]);

    // Accept Square B
    printf("\nEnter the elements of Latin Square B.\n");
    for(i=0;i<order;++i)
       for(j=0;j<order;++j)
             scanf("%d",&B[i][j]);

    int status = isOrthogonal(order);

    if(status == -1)
       printf("\nGiven Latin Square Matrices are NOT Mutually Orthogonal.\n");
    else
       printf("\nGiven Latin Square Matrices are Mutually Orthogonal.\n");
}

int isOrthogonal(int order)
{
    // Let us first eliminate the trivial cases

    // Based on Eulers conjecture
    if(order == 6)
       return -1;

    // Now for the remaining orders - let us check the the orthogonality

    /* -- Step - 1: Superimpose Matrix A and Matrix B to form Matrix C -- */
    int i,j;
    int num_digits_B = 0;
    int exp = 0;
    for(i=0;i<order;++i)
    {
       for(j=0;j<order;++j)
       {
             num_digits_B = floor(log10(B[i][j]) + 1);
             exp = pow(10,num_digits_B);
             C[i][j] = A[i][j] * exp + B[i][j];
       }
    }

    /* --  Step -2: Check uniqueness of each element in the Matrix C in multi-threaded
fashion. -- */
```

```
    // Create (order * order) threads for each entry in the C matrix
    pthread_t tid[order * order];
    int thread_no = 0;

    // Create and init attr with default values
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    // Call the runner function
    struct Position pos[order * order];
    for(i=0;i<order;++i)
    {
       for(j=0;j<order;++j)
       {
             pos[thread_no].row = i;
             pos[thread_no].col = j;
             pthread_create(&tid[thread_no],&attr,ortho_runner,(void
*)&pos[thread_no]);
             ++thread_no;
       }
    }

    // Join all the threads
    thread_no =0;
    for(i=0;i<order;++i)
       for(j=0;j<order;++j)
             pthread_join(tid[thread_no++],NULL);

    /* --  Step -3: Evaluate the status of global variable orthogonalFlag and return
status -- */
    if(orthogonalFlag == 0)
       return 1;
    return -1;
}

// Define the thread runner to check the orthogonality of the input matrix
void *ortho_runner(void *arg)
{
    struct Position *position = (struct Position*)arg;
    int row = position->row;
    int col = position->col;
```

```
    int i,j;
    int flag = 0;
    for(i=row;i<order;++i)
    {
       for(j=col;j<order;++j)
       {
             // Skip for that position
             if(i == row && j == col)
                   continue;

             if(C[i][j] == C[row][col])
             {
                   ++flag;
                   break;
             }
       }

       if(flag!=0)
             break;
    }

    if(flag != 0)
       ++orthogonalFlag;

    // Exit the thread
    pthread_exit(NULL);
}
```

Output

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Midsem$ gcc -o out latinMatrix.c -lpthread -lm
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Midsem$ ./out

************************ PART-I: Latin Square Generator. ************************

Enter the Dimension of the Latin Square Matrix to be generated.
4

A Latin Square of order 4 is:
1 2 3 4
4 1 2 3
3 4 1 2
2 3 4 1

******************** PART-II: Orthogonality Check of Two Latin Sqaure Matrices. **********************

Enter the order of Latin Squares.
2

Enter the elements of Latin Sqaure A.
1 2
2 1

Enter the elements of Latin Sqaure B.
2 1
1 2

Given Latin Square Matrices are NOT Mutually Orthogonal.
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Midsem$
```

```
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Midsem$ gcc -o out latinMatrix.c -lpthread -lm
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Midsem$ ./out

************************ PART-I: Latin Square Generator. ************************

Enter the Dimension of the Latin Square Matrix to be generated.
3

A Latin Square of order 3 is:
1 2 3
3 1 2
2 3 1

******************** PART-II: Orthogonality Check of Two Latin Sqaure Matrices. **********************

Enter the order of Latin Squares.
3

Enter the elements of Latin Sqaure A.
1 2 3
3 1 2
2 3 1

Enter the elements of Latin Sqaure B.
1 2 3
2 3 1
3 1 2

Given Latin Square Matrices are Mutually Orthogonal.
adheshreghu@adheshreghu-Inspiron-5570:~/Documents/SEM5/OS/Lab/Midsem$
```

Performance Comparison:

Let us compare this with the serial version of the same code.

Serial Version of the Program

```
/*****************************************
* AUTHOR : AdheshR *
* Problem Description:     Part - I:  Serial version of Latin Matrix Generator
                               Part - II: Given two input latin squares - check their
orthogonality*
*****************************************/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <pthread.h>
#include <math.h>
#define MAX 1000

// Define the array and length as global variable
int arr[MAX][MAX];
int N;

// Define global variables for PART II - Checking orthogonality
int A[MAX][MAX];
int B[MAX][MAX];
int C[MAX][MAX];
int order;
int orthogonalFlag = 0;

void latinUtils(int row);
void orthoUtils(int row, int col);
void orthogonalityCheck();

int main()
{
    /* --  PART-I: Generate the Latin square in seria -- */
```

```
    // Input the Dimension of the Latin Square Matrix
    printf("\n************************ PART-I: Latin Square Generator.
**************************\n\n");
    printf("Enter the Dimension of the Latin Square Matrix to be generated.\n");
    scanf("%d",&N);

    // Call the util function
    int i;
    for(i=0;i<N;++i)
       latinUtils(i);

    // Print the matrix
    int j;
    printf("\nA Latin Square of order %d is:\n",N);
    for(i=0;i<N;++i)
    {
       for(j=0;j<N;++j)
             printf("%d ",arr[i][j]);
       puts("");
    }

    /* -- PART II - Checking Orthogonality of two Input Latin Square matrices -- */
    orthogonalityCheck();

    return 0;
}

void latinUtils(int row)
{
    // Fill in the row of the matrix
    int i = row;                                          // define start position
in the matrix
    int fill = 1;                                        // define the value to be
filled in
    while(fill <= N)
    {
       arr[row][i] = fill;
       i = (i + 1)%N;
       ++fill;
    }
}

// Define the function to perform the PART II of the question
```

```
void orthogonalityCheck()
{

    printf("\n******************** PART-II: Orthogonality Check of Two Latin Sqaure
Matrices. **********************\n\n");

    // Get the inputs from the user
    printf("Enter the order of Latin Squares.\n");
    scanf("%d",&order);

    // Loop variables
    int i,j;

    // Accept Square A
    printf("\nEnter the elements of Latin Square A.\n");
    for(i=0;i<order;++i)
       for(j=0;j<order;++j)
             scanf("%d",&A[i][j]);

    // Accept Square B
    printf("\nEnter the elements of Latin Square B.\n");
    for(i=0;i<order;++i)
       for(j=0;j<order;++j)
             scanf("%d",&B[i][j]);

    int status = isOrthogonal(order);

    if(status == -1)
       printf("\nGiven Latin Square Matrices are NOT Mutually Orthogonal.\n");
    else
       printf("\nGiven Latin Square Matrices are Mutually Orthogonal.\n");
}

int isOrthogonal(int order)
{
    // Let us first eliminate the trivial cases

    // Based on Eulers conjecture
    if(order == 6)
       return -1;

    // Now for the remaining orders - let us check the the orthogonality
```

```
    /* -- Step - 1: Superimpose Matrix A and Matrix B to form Matrix C  -- */
    int i,j;
    int num_digits_B = 0;
    int exp = 0;
    for(i=0;i<order;++i)
    {
       for(j=0;j<order;++j)
       {
             num_digits_B = floor(log10(B[i][j]) + 1);
             exp = pow(10,num_digits_B);
             C[i][j] = A[i][j] * exp + B[i][j];
       }
    }

    /* --  Step -2: Check uniqueness of each element in the Matrix C. -- */
    for(i=0;i<order;++i)
       for(j=0;j<order;++j)
             orthoUtils(i,j);

    /* --  Step -3: Evaluate the status of global variable orthogonalFlag and return
status -- */
    if(orthogonalFlag == 0)
       return 1;
    return -1;
}

// Define the utility function to check uniqueness of the
void orthoUtils(int row, int col)
{
    int i,j;
    int flag = 0;
    for(i=row;i<order;++i)
    {
       for(j=col;j<order;++j)
       {
             // Skip for that position
             if(i == row && j == col)
                   continue;

             if(C[i][j] == C[row][col])
             {
                   ++flag;
                   break;
```

```
            }
        }

        if(flag!=0)
            break;
    }

    if(flag != 0)
        ++orthogonalFlag;
}
```

Let us time both the programs.

The time has been computed by clock() functions of time.h

Parallel Version:

- Latin Matrix Generation = 2755 units of time (Avgd over 6 different iterations)
- Checking Orthogonality = 3273 units of time (Avgd over 3 different iterations)

Serial Version:

- Latin Matrix Generation = 3461 units of time (Avgd over 6 same iterations as above)
- Checking Orthogonality = 806.6 units of time (Avgd over 3 same iterations as above)

Almost all iterations have been tried on relatively small size inputs. On a small size input it can be noticed that there is hardly any difference or sometimes even the serial version performs better than the parallel ones. This is probably due to the overheads involved in creating threads and switching between them.

I assume that for a relatively larger input the parallel version would perform better than the serial version.

Let us take a look at the effect of no of processes on the computation time.

Theoretically
Amdahl's law gives us an insight on the effect of the number of threads on speed up (computation time). According to Amdahl's law

Speed Up = $1/((1-P) * (P/N))$

Where P stands for the amount of tasks that can be parallelized and N stands for the number of threads.

If we increase the number of threads, the speed up factor increases theoretically. However the number of threads that we can deploy is limited by the amount of tasks that can be parallelized as well. So beyond a certain limit, we may not be able to parallelize tasks by introducing newer threads as there may be no more parallelizable tasks.

Practically
Even though Amdahl's law states that speed up should increase with increase in number of threads, it does not take into account the overheads involved with threads such as time taken to create and exit a thread, time taken to switch between threads and so on.
All the overheads can cause the program to run a bit slower than expected. Amdahl's law just gives the theoretical upper bound on the speed up possible.

So a threaded program for smaller inputs may run slower than a sequential program. However I believe when the input size is fairly large and the system has sufficient number of cores, a threaded code can run much faster than the serial code.

In my program, I have parallelized most of the parallelizable tasks. So I assume for a large input, it should show considerably good speed up.
