OS Midsem Report

G S Santhosh Raghul
COE18B045

Question:

Develop a C program that checks if a matrix is orthogonal or not. An orthogonal matrix say A satisfies the property that AT=A-1. Have appropriate number of processes. Compare the efficiency of the multiprocessing version over its equivalent serial version.

Explanation :

The program takes a file as an input through the command line arguments, checks whether the given matrix is orthogonal or not serially and in a multithreaded fashion and displays the time taken for both.

The format of the file is :

- 1st 2 lines contain 2 integers m,n denoting the matrix is of order mxn
- The next m line contain n space separated floating point values, the m rows of the column

To check whether the given matrix is orthogonal, the program multiplies the given matrix with its transpose and checks whether the product obtained is an identity matrix.

Serial code is straight forward. Just 3 for loops, 2 to calculate each element, the inner for to calculate sum of a[i][k]*b[k][j]

For multithreading, the program first gets the number of processor cores in the system by using the `get_nprocs_conf()` function from the library. `<sys/sysinfo.h>`

Then the work load is divided by giving m/N + x rows to each thread where m is the number of rows to be calculated, N is the number of processor cores available and x for the nth thread is 1 if m%N>(n-1) (suppose N=4, m=2 then work is divided as 1st 2 rows for thread 1, 3rd and 4th rows for thread 2, 5th row for thread 3 and the 6th row for thread 4) . This is done because, if we create more than N threads, some of the threads will anyways be waiting for CPU time since there are only N cores and will only add to the overhead of thread creation.

In each thread, the range of rows to be calculated is specified and each thread calculates those rows allotted to it.

After this is over, it is checked if the resultant matrix is identity matrix or not to determine whether the given matrix is orthogonal.

Source:

```
#include<stdio.h>

#include<stdlib.h>

#include<pthread.h>

#include<sys/sysinfo.h>

#include<sys/time.h>



typedef struct par  // structure to send parameters

{

   float *A,*B,*C; // A, B are matrices to be multiplied, C is where product is
stored

   int m,n,start_row,end_row;  // m,n - order of matrix A

   // start_row - row from which product is to be calculated in the current thread

   // end_row - row till which product is to be calculated in the current thread

}par;



// check whether given matrix is orthogonal or not using multithreading and return
flag value

int check_orthogonal(char* filename);

// check whether given matrix is orthogonal or not serially and return flag value

int check_orthogonal_serial(char* filename);

// print message about the orthogonality of the matrix acc to the flag value
returned by orthogonality checking functions

void print_message(int flag,char* filename);

// utility function to multiply A, B and store it in product

void multiply_utility(float *A,float *B,float *product,int m,int n);
```

```
// runner function to calculate specified rows of the product matrix
void* matrix_product_runner(void* _param);

// to get time difference in milliseconds between 2 timeval variables
double time_diff(struct timeval begin,struct timeval end);

int main(int argc,char* argv[])
{
   if(argc!=2)
   {
       fprintf(stderr,"%s: missing file operand\nUsage: %s
filename\n",argv[0],argv[0]);
       exit(1);
   }

   struct timeval begin, end;
   double multithreaded,serial;

   printf("serial:\n");
   gettimeofday(&begin, 0);
   print_message(check_orthogonal_serial(argv[1]),argv[1]);
   gettimeofday(&end, 0);
   serial=time_diff(begin,end);

   printf("\nmultithreaded:\n");
   gettimeofday(&begin, 0);
```

```
    print_message(check_orthogonal(argv[1]),argv[1]);
    gettimeofday(&end, 0);
    multithreaded=time_diff(begin,end);

    printf("\ntime taken for:\n");
    printf("multithreaded: %lfms\n",multithreaded);
    printf("serial:        %lfms\n",serial);
    return 0;
}

int check(float x)
{
    return (0.999990<=x && x<=1.000001);
}

int check_orthogonal_serial(char* filename)
{
    FILE* file;
    if((file=fopen(filename,"r"))==NULL)
        return -2;  // if file is not found, return -2

    int m,n,flag=1;
    fscanf(file,"%d %d",&m,&n); // read matrix dimensions in m,n
    if(m!=n)
        return -1;  // if given matrix is not square, return -1
    // read matrix elements in M[][] and also store the transpose in M_t[][]
    float M[m][n],M_tr[n][m],pd[m][m],sum;
```

```
for(int i=0;i<m;i++)
    for(int j=0;j<n;j++)
    {
        fscanf(file,"%f",&(M[i][j]));
        M_tr[j][i]=M[i][j];
    }
fclose(file);


// calculate product matrix
for(int i=0;i<m;i++)
    for(int j=0;j<m;j++)
    {
        sum=0;
        for(int k=0;k<n;k++)
            sum+=M[i][k]*M_tr[k][j];
        pd[i][j]=sum;
    }


// print the product matrix
printf("the product is:\n");
for(int i=0;i<m;i++)
{
    printf("\t");
    for(int j=0;j<m;j++)
        printf("%f ",pd[i][j]);
    printf("\n");
}
```

```
    // check if product has 0 in non diagonal elements and same value in diagonal
elements
    for(int i=0;i<m && flag!=0;i++)
    {
        for(int j=0;j<m;j++)
            if((i==j && pd[i][j]!=pd[0][0])||(i!=j && pd[i][j]!=0))
            {
                flag=0;
                break;
            }
    }

    // check if product has 0 in non diagonal elements and same value in diagonal
element, check if the diagonal values are 1
    if(flag && pd[0][0]==1)
        flag=2; // if yes set flag to 2
    return flag;
}

int check_orthogonal(char* filename)
{
    FILE* file;
    if((file=fopen(filename,"r"))==NULL)
        return -2;  // if file is not found, return -2

    int m,n,flag=1;
    fscanf(file,"%d %d",&m,&n); // read matrix dimensions in m,n
```

```
    if(m!=n)
        return -1;  // if given matrix is not square, return -1
    // read matrix elements in M[][] and also store the transpose in M_t[][]
    float M[m][n],M_tr[n][m],pd[m][m];
    for(int i=0;i<m;i++)
        for(int j=0;j<n;j++)
        {
            fscanf(file,"%f",&(M[i][j]));
            M_tr[j][i]=M[i][j];
        }
    fclose(file);


    // calculate product matrix
    multiply_utility((float*)M,(float*)M_tr,(float*)pd,m,n);
    // print the product matrix
    printf("the product is:\n");
    for(int i=0;i<m;i++)
    {
        printf("\t");
        for(int j=0;j<m;j++)
            printf("%f ",pd[i][j]);
        printf("\n");
    }
    // check if product has 0 in non diagonal elements and same value in diagonal
elements
    for(int i=0;i<m;i++)
    {
```

```
        for(int j=0;j<m;j++)
            if((i==j && pd[i][j]!=pd[0][0])||(i!=j && pd[i][j]!=0))
            {
                flag=0;
                break;
            }
    }
    // check if product has 0 in non diagonal elements and same value in diagonal element, check if the diagonal values are 1
    if(flag && pd[0][0]==1)
        flag=2; // if yes set flag to 2
    return flag;
}

void multiply_utility(float *A,float *B,float *product,int m,int n)
{
    int N=get_nprocs_conf(),    // get number of processor cores in N to create N number of threads
        start_row=0,
        start_col=0,
        max=-1;
    par param[N];
    pthread_t tid[N];
    pthread_attr_t attr[N];
    printf("number of CPU cores was found to be %d so creating %d threads\n",N,N);
    for(int i=0;i<N;i++)
    {
        // setting up parameters
```

```
        param[i].A=A; param[i].B=B; param[i].C=product;
        param[i].m=m; param[i].n=n;
        param[i].start_row=start_row;
        start_row+=m/N+(m%N>i?1:0); // setting up for next thread
        param[i].end_row=start_row-1;
        // refer printf message also
        if(param[i].start_row>param[i].end_row)
        {
            printf("\tthread %d will not calculate anything \n",i+1);
            if(max==-1) // if ith thread is useless (due to small input size), set
max to i
                max=i;
        }
        else
        {
            if(param[i].start_row<param[i].end_row)
                printf("\tthread %d will calculate row %d to row %d of the product
\n",i+1,param[i].start_row,param[i].end_row);
            else
                printf("\tthread %d will calculate row %d of the product
\n",i+1,param[i].start_row);
            pthread_attr_init(&attr[i]);
            pthread_create(&tid[i],&attr[i],matrix_product_runner,&param[i]);
        }
    }
    if(max==-1) // if all threads were useful, set max to N
        max=N;
    for(int i=0;i<max;i++)
```

```
        pthread_join(tid[i],NULL);  // join all the threads
}


void* matrix_product_runner(void* _param)
{
    float sum=0;
    par param=*(par*)_param;
    for(int i=param.start_row;i<=param.end_row;i++) // each row from start to end,
calculate product
        for(int j=0;j<param.m;j++)
        {
            for(int k=0;k<param.n;k++)
                sum+=(*((param.A+i*param.n)+k))*(*((param.B+k*param.m)+j));
            *((param.C+i*param.m)+j)=sum;
            sum=0;
        }
    pthread_exit(0);
}
```

```
void print_message(int flag,char* filename)

{

   switch(flag)

       {

           case -2: perror(filename); break;

           case -1: printf("not square matrix. so not orthogonal\n"); break;

           case 0: printf("not orthogonal\n"); break;

           case 1: printf("not orthogonal but can be made orthogonal by dividing
the entire matrix by a constant\n"); break;

           case 2: printf("orthogonal\n"); break;

       }

}



double time_diff(struct timeval begin,struct timeval end)

{

   long seconds = end.tv_sec - begin.tv_sec;

   long microseconds = end.tv_usec - begin.tv_usec;

   double elapsed = seconds*1000 + microseconds*1e-6;

   return elapsed;

}
```

- I wrote one more program which does the following:

./a.out m n min max filename

to generate a mxn matrix with entries ranging from min to max and store it in the specified file.

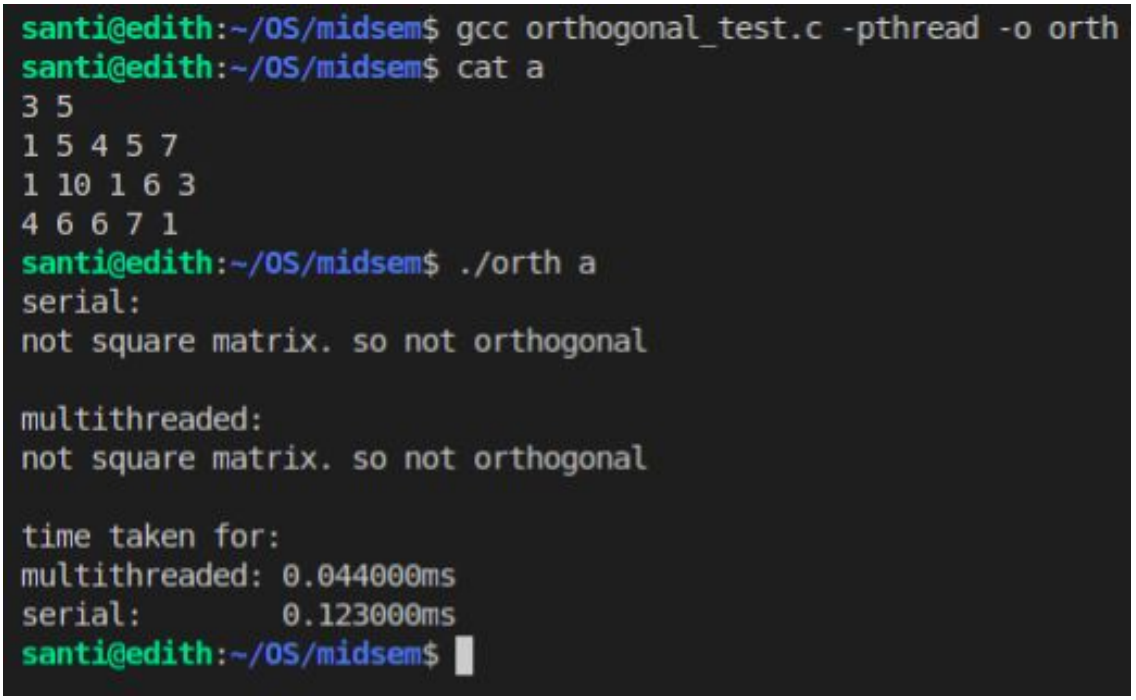
But this generates integer matrix only

Source:

```c
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<time.h>

#include<fcntl.h>



int main(int argc,char* argv[])

{

   if(argc!=6)

   {

       fprintf(stderr,"%s: invalid usage\nUsage: %s m n min max filename\nto generate a mxn matrix with entries ranging from min to max and store it in the specified file\n",argv[0],argv[0]);

       exit(1);

   }



   int m=atoi(argv[1]),

       n=atoi(argv[2]),

       min=atoi(argv[3]),

       max=atoi(argv[4]),

       file;



   fclose(fopen(argv[5],"w"));    // create file if it doesn't already exist

   file=open(argv[5],O_WRONLY);

   dup2(file,1);

   printf("%d %d",m,n);
```

```
    srand(time(0));

    for(int i=0;i<m;i++)

    {

        printf("\n");

        for(int j=0;j<n;j++)

            printf("%d ",min+rand()%(max-min+1));

    }



    return 0;

}
```

Output screenshots :



```
santi@edith:~/OS/midsem$ cat b
5 3
1 1 2
9 2 2
10 4 10
1 4 3
2 6 5
santi@edith:~/OS/midsem$ ./orth b
serial:
not square matrix. so not orthogonal

multithreaded:
not square matrix. so not orthogonal

time taken for:
multithreaded: 0.057000ms
serial:        0.139000ms
santi@edith:~/OS/midsem$
```

```
santi@edith:~/OS/midsem$ cat c
3 3
0.666666666666 0.333333333333 0.666666666666
-0.666666666666 0.666666666666 0.333333333333
0.333333333333 0.666666666666 -0.666666666666
santi@edith:~/OS/midsem$
```

```
santi@edith:~/OS/midsem$ ./orth c
serial:
the product is:
        1.000000 0.000000 0.000000
        0.000000 1.000000 0.000000
        0.000000 0.000000 1.000000
not orthogonal

multithreaded:
number of CPU cores was found to be 4 so creating 4 threads
        thread 1 will calculate row 0 of the product
        thread 2 will calculate row 1 of the product
        thread 3 will calculate row 2 of the product
        thread 4 will not calculate anything
the product is:
        1.000000 0.000000 0.000000
        0.000000 1.000000 0.000000
        0.000000 0.000000 1.000000
not orthogonal

time taken for:
multithreaded: 0.356000ms
serial:        0.068000ms
santi@edith:~/OS/midsem$
```

Here, the output says there are 4 processes so 4 threads are being created. This can vary depending on the system which this program is being run. For example, in an octa core machine, 8 threads will be created at max.

The below output says it is not orthogonal because the values aren't precise enough.

```
santi@edith:~/OS/midsem$ cat d
3 3
0.666666 0.333333 0.666666
-0.666666 0.666666 0.333333
0.333333 0.666666 -0.666666
santi@edith:~/OS/midsem$
```

```
santi@edith:~/OS/midsem$ ./orth d
serial:
the product is:
        0.999998 0.000000 0.000000
        0.000000 0.999998 0.000000
        0.000000 0.000000 0.999998
not orthogonal but can be made orthogonal by dividing the entire matrix by a constant

multithreaded:
number of CPU cores was found to be 4 so creating 4 threads
        thread 1 will calculate row 0 of the product
        thread 2 will calculate row 1 of the product
        thread 3 will calculate row 2 of the product
        thread 4 will not calculate anything
the product is:
        0.999998 0.000000 0.000000
        0.000000 0.999998 0.000000
        0.000000 0.000000 0.999998
not orthogonal but can be made orthogonal by dividing the entire matrix by a constant

time taken for:
multithreaded: 1.727000ms
serial:        0.316000ms
santi@edith:~/OS/midsem$
```

It says it is not orthogonal but can be made orthogonal by dividing the entire matrix by a constant, which is true. The given matrix divided by 3 will result in an orthogonal matrix

```
santi@edith:~/OS/midsem$ cat e
3 3
2 1 2
-2 2 1
1 2 -2
santi@edith:~/OS/midsem$ ./orth e
serial:
the product is:
        9.000000 0.000000 0.000000
        0.000000 9.000000 0.000000
        0.000000 0.000000 9.000000
not orthogonal but can be made orthogonal by dividing the entire matrix by a constant

multithreaded:
number of CPU cores was found to be 4 so creating 4 threads
        thread 1 will calculate row 0 of the product
        thread 2 will calculate row 1 of the product
        thread 3 will calculate row 2 of the product
        thread 4 will not calculate anything
the product is:
        9.000000 0.000000 0.000000
        0.000000 9.000000 0.000000
        0.000000 0.000000 9.000000
not orthogonal but can be made orthogonal by dividing the entire matrix by a constant

time taken for:
multithreaded: 1.752000ms
serial:        0.333000ms
santi@edith:~/OS/midsem$
```

As we can see in all the above examples, the time for multithreaded is more than the time for serial. This happens because the input size is very small that the overhead of thread creation is more than the time taken for serial calculation itself. Let us now try large matrices.

Note: I have now commented out the printf statements that print the matrices.

We can see that for input size 50x50, both times are almost the same, just 1 millisecond apart.

```
santi@edith:~/OS/midsem$ gcc orthogonal_test.c -pthread -o orth
santi@edith:~/OS/midsem$ gcc random_matrix_generator.c -o rand
santi@edith:~/OS/midsem$ ./rand 50 50 1 1000 w
santi@edith:~/OS/midsem$ ./orth w
serial:
not orthogonal

multithreaded:
number of CPU cores was found to be 4 so creating 4 threads
        thread 1 will calculate row 0 to row 12 of the product
        thread 2 will calculate row 13 to row 25 of the product
        thread 3 will calculate row 26 to row 37 of the product
        thread 4 will calculate row 38 to row 49 of the product
not orthogonal

time taken for:
multithreaded: 5.699000ms
serial:        4.529000ms
santi@edith:~/OS/midsem$
```

Lets try larger sizes

```
santi@edith:~/OS/midsem$ ./rand 100 100 1 1000 x
santi@edith:~/OS/midsem$ ./orth x
serial:
not orthogonal

multithreaded:
number of CPU cores was found to be 4 so creating 4 threads
        thread 1 will calculate row 0 to row 24 of the product
        thread 2 will calculate row 25 to row 49 of the product
        thread 3 will calculate row 50 to row 74 of the product
        thread 4 will calculate row 75 to row 99 of the product
not orthogonal

time taken for:
multithreaded: 4.698000ms
serial:        7.672000ms
santi@edith:~/OS/midsem$
```

Now the multithreaded version is running faster. Lets try 2 more big sizes.

```
santi@edith:~/OS/midsem$ ./rand 200 200 1 1000 y
santi@edith:~/OS/midsem$ ./orth y
serial:
not orthogonal

multithreaded:
number of CPU cores was found to be 4 so creating 4 threads
        thread 1 will calculate row 0 to row 49 of the product
        thread 2 will calculate row 50 to row 99 of the product
        thread 3 will calculate row 100 to row 149 of the product
        thread 4 will calculate row 150 to row 199 of the product
not orthogonal

time taken for:
multithreaded: 19.804000ms
serial:        64.664000ms
santi@edith:~/OS/midsem$
```

```
santi@edith:~/OS/midsem$ ./rand 300 300 1 1000 z
santi@edith:~/OS/midsem$ ./orth z
serial:
not orthogonal

multithreaded:
number of CPU cores was found to be 4 so creating 4 threads
        thread 1 will calculate row 0 to row 74 of the product
        thread 2 will calculate row 75 to row 149 of the product
        thread 3 will calculate row 150 to row 224 of the product
        thread 4 will calculate row 225 to row 299 of the product
not orthogonal

time taken for:
multithreaded: 63.888000ms
serial:        102.255000ms
santi@edith:~/OS/midsem$
```

So, we can see that parallel code runs faster for larger inputs