Project 3: Matrix Multiplication

Matrix multiplication is widely used in many fields. However, it may become rather time-consuming when the size of the input matrix is large. In this project, we will solve this problem to a certain degree. We calculate each element in a separate *worker* thread. This will involve creating $M \times N$ worker threads. For convenience, the input matrix and the result matrix will be declared as global data so that each worker thread has access to A, B, and C.

Getting Started

Since I have constituted the linux environment in previous project, this part is skimmed. In this project, *Ubuntu-16.04.1* for linux is used, and its kernel version is *4.4.0*.

Passing Parameters to Each Thread

The parent thread should create $M \times N$ worker threads, passing each worker the values of row i and column j that it is to use in calculating the matrix product. Notice that we have two parameters, i and j will be passed at the same time. To make this process easier, we can define a date structure:

```
//structure for passing data to threads
struct v
{
    int vi;
    int vj;
};
```

Code 1: Defined data structure

Then we need to create the worker threads. Since each element is calculated in a separate worker thread, $M \times N$ worker threads should be created. After that, we will call the `pthread_create()` function to pass the data pointer as a parameter to the function `runner()` that is to run as a separate thread.

```
for(i = 0; i < M; ++i) //create the worker threads
    for(j = 0; j < N; ++j)
    {
        data = (struct v*) malloc(sizeof(struct v));
        data -> vi = i;
        data -> vj = j;

        //create the thread passing it data as a parameter
        rc = pthread_create(&thread[i][j], NULL, runner, data);

        if(rc)
```

```
        {
            printf("create error!\n");
            return 0;
        }
    }
```

Code 2: Creating worker threads

```
void *runner(void *param)
{
    struct v *args;
    args = (struct v*)param;
    int i = args -> vi, j = args -> vj;
    int m, res = 0;
    for(m = 0; m < K; ++m)
    {
        res += A[i][m] * B[m][j];
    }
    C[i][j] = res;
    free(param);
    return NULL;
}
```

Code 3: Runner

Waiting for Threads to Complete

Since a large number of threads are creating, and they may run differently in time. Our final job is to find the result matrix, that is to say, we must get all the elements. As a result, we have to wait until all the threads come to an end. `pthread_join()` function meets our requirement - the parent thread mush wait for child threads.

```
for(i = 0; i < M; ++i) //wait until the working thread comes to an end
    for(j = 0 ; j < N; ++j)
    {
        pthread_join(thread[i][j], NULL);
    }
```

Code 4: Pthread_join

Running Result

Figure 1 shows the running result.

In order to find the difference between multithread and single thread, I write a program to test it. The code is available in the APPENDIX.

```
please input the M K N
2 2 2
input the matrix1 (M*K):
1 2
3 4

input the matrix2 (K*N)
5 6
7 8
the running time using multithread is : 0.000072

the result:
19 22
43 50
```

Figure 1: Matrix Multiplication using multi-threads

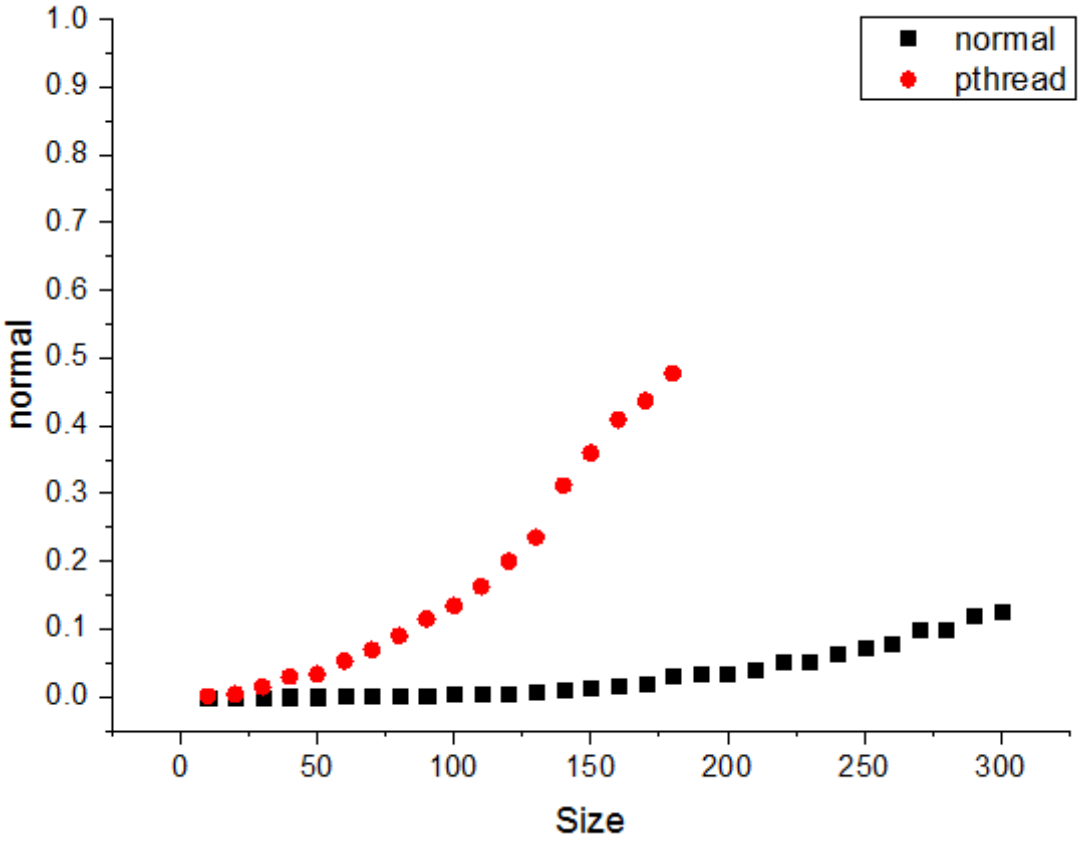

Figure 2: Comparing result

However, the result seems far from satisfactory. Figure 2 shows the increment speed of both methods. The X axis represents the size of the input matrix while the Y axis is the time taken to get the correct answer. To my surprise, multithread programming takes even more time! After consideration, I think that the following factors may explain:

- The virtual machine is to blame. Although I have assign 2GB memory and 2 cores to improve the efficiency, I doubt that the power of multithread are not made full use of.
- The creation of $M \times N$ worker threads may take a majority of time.

However, as can be seen in the graph, when the size increases, the running time of normal operation will increase drastically since the algorithm is $O(n^3)$. Notice the curve of pthread, when the size is greater than 150, the increment speed generally slows down.

We can infer that when the size of input matrixes is large enough, the application of multi-threads will gain the advantage over the normal single thread method.

Summary

This project is relatively easy, but really meaningful. Multithread is widely used nowadays, raising the efficiency and saving time. The project helps reinforce the understanding of how to program using multithread. But when the size of matrixes is not large enough, it may takes more time. Maybe we can use other methods such as divide and conquer to solve matrix multiplication when the size is small. I also learned that we should pick the best method after the final evaluation rather than act on assumptions (at first I think that on no account will multithread take more time, but it turns out to be wrong).

Student name: Han Kunyan
Student No.:5140309534

Appendix

Full code of Compare.c

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

int M;
int K;
int N;

#define MAX 500
int A[MAX][MAX];
int B[MAX][MAX];
int C[MAX][MAX];

struct v
{
    int vi;
    int vj;
};

void *runner(void *param)
{
    struct v *args;
    args = (struct v*)param;
    int i = args -> vi, j = args -> vj;
    int m, res = 0;
    for(m = 0; m < K; ++m)
    {
        res += A[i][m] * B[m][j];
    }
    C[i][j] = res;
    free(param);
    return NULL;

}

int main()
{
    pthread_t thread[MAX][MAX];

    int i, j, k;

    int rc;

    time_t start1, end1, start2, end2;

    struct v *data;

    srand((unsigned)time(NULL));


    for (int n=10;n<=200;n+=10){ //increase the size of input matrixes

        printf("size: %d\n", n);
        M = N = K = n;
        //printf("input the matrix1 (M*K):\n");
        for(i = 0; i < M; ++i)
            for(j = 0; j < K;++j)
            {
```

```
            A[i][j] = rand();
        }
    //printf("\ninput the matrix2 (K*N)\n");
    for(i = 0; i < K; ++i)
        for(j = 0; j < N; ++j)
        {
            B[i][j] = rand();
        }


    start1 = clock();

    int res;

    for (i=0; i<M; ++i)
    {
        for (j=0; j<N; ++j)
        {
            C[i][j] = 0;
            for (int k=0; k<K; ++k)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    end1 = clock();
    printf("the running time is : %f\n", (double)(end1-start1)/CLOCKS_PER_SEC );


    start2 = clock();

    for(i = 0; i < M; ++i) //create the worker threads
        for(j = 0; j < N; ++j)
        {
            data = (struct v*) malloc(sizeof(struct v));
            data -> vi = i;
            data -> vj = j;

            //create the thread passing it data as a parameter
            rc = pthread_create(&thread[i][j], NULL, runner, data);

            if(rc)
            {
                printf("create error!\n");
                return 0;
            }


        }


    for(i = 0; i < M; ++i) //wait until the working thread comes to an end
        for(j = 0 ; j < N; ++j)
        {
            pthread_join(thread[i][j], NULL);
        }

    end2 = clock();
    printf("the running time using multithread is : %f\n", (double)(end2-start2)/CLOCKS_PER_SEC

}
```

```
    return 0;
}
```