

Threading in software

1. INTRODUCTION:

1.1 Overview of thread systems

Nowadays, the growth in the usage of advanced graphics performance and CPU has caused an increasing demand for threaded processes.

1.2 The structure of thread system

In computer architecture, threads are the smallest sequence of programmed instructions that is executed by a processor, and managed by a scheduler. It contains a unique ID, a stack, a program counter, and a set of registers.

1.3 Aim of the paper

This paper presents matrix manipulation in the threading process to discuss the following issues. Firstly, examining the reason why it is highly unlikely that the process of multithreading achieves $O(n)$ performance by using n^2 threads, and the deviation from $O(n)$ performance. Secondly, observing whether the change of thread number will affect the multiplication process. Thirdly, using smaller number of thread ($< n^2$) to see if it still achieves the same performance as n^2 . Finally, we want to know how the number of physical core and cache size of computer affects the computational performance. Based on these factors, the limitations and feasibility will be discussed in order to find if the threading system can bring a positive contribution to technologies.

2. METHODOLOGY

2.1 The structure of the designed program

This experiment primarily uses a C compiler as the background programming language and utilizing pthread library to achieve matrix multiplication. There are two different programs that have been carried out for this test where one of them uses an $n \times n$ matrix with n^2 threads, while the other analyzes the matrix with a different number of threads. For the designed structure, it mainly uses the so-called static thread allocation model where every thread's mission is assigned manually instead of being allocated automatically by the system. Also, the concept of "structure pointer" is applied into the program. Firstly, the matrix multiplication is achieved using the "for loop" concept and marked as a "void" function. In the main program, two dynamic matrix inputs were carried out first. Secondly, the main part of this experiment is to create the pthread function. This time, the instructions such as pthread_t, pthread_attr_t, pthread_attr_init, pthread_create and pthread_join from pthread library were applied to create the function where pthread_create is to create the threading system. Lastly, the output matrix is completed using the printing function.

2.2 The performance of the designed program

This experiment applied IOS system with an i5 processor, which has 4 cores in the computer used to run the program. **Figure 2** below shows the processing time with different number of threads under 1000 by 1000 matrix conditions:

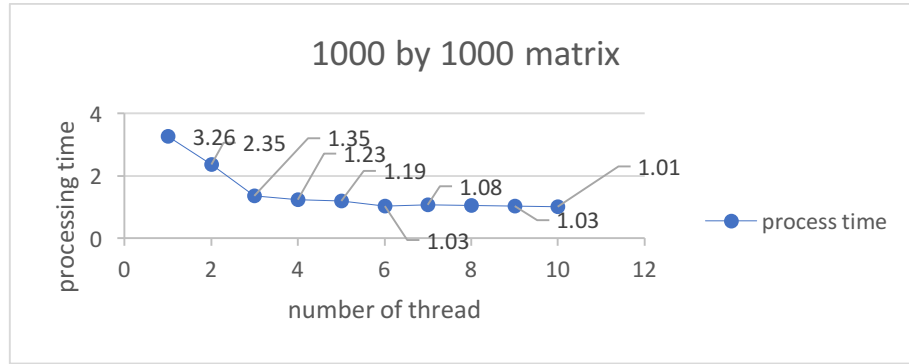


Figure 2. 1000 by 1000 matrix multiplication processing time with different number of threads

Figure 3 shows the multithreaded processes performance with different size of matrix using n^2 matrix:

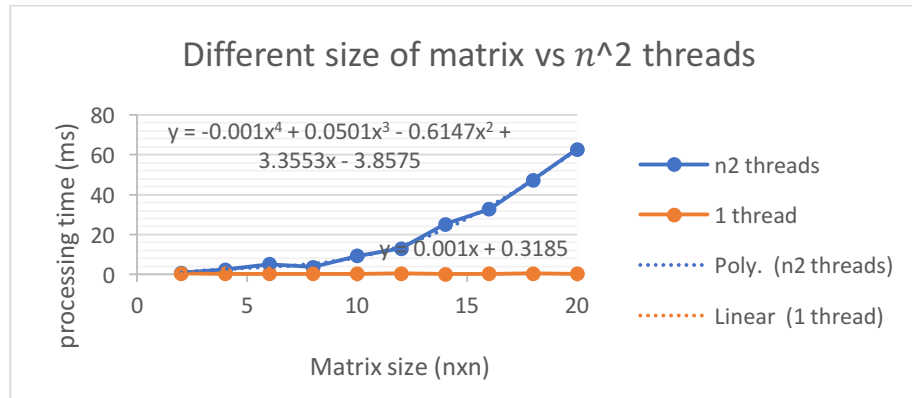


Figure 3. the performance of different size of matrix with n^2 threads

3. DISCUSSION

3.1 $O(n)$ performance by using n^2 threads

In theory, the benefit of threading system is to reduce the processing time of the tasks. Ideally, the processing time becomes faster as the number of threads increases. However, **Figure 2** shows that the processing time decreases dramatically from 1 thread to 3 threads, and remains roughly same performance for the next few samples. Hence, we observe that the thread is only able to make a major change for the first few ones, but it will not remain as efficient after the number of thread reach to a certain amount, known as the limitation of thread. Therefore, we can assume that it is unnecessary for the elements to have individual threads, because the processing time will not be able to speed up, as well as putting a greater load on the core. In addition, for algorithm is only suitable for linear equation. However, **Figure 3** shows the line is not linear ($y = -0.001x^4 + 0.0501x^3 - 0.6147x^2 + 3.3553x - 3.8575$), so it is not comparable with the algorithm.

3.2 the relationship varies with n and the number of threads

Figure 3 shows the processing time start to be slower while the matrix size increases and is executed by n^2 of thread. This phenomenon might be caused by a shortage of cache size and the number of cores. As

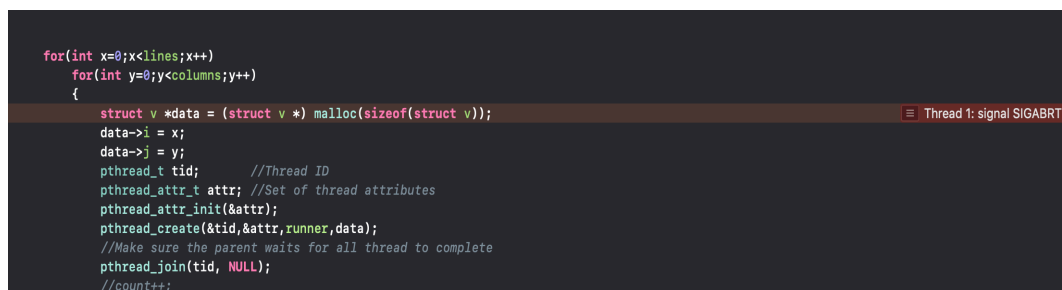
a result of the bigger matrix and larger number of threads, it occupied more memory to cache and increased the load on each core, where the cores were not able to finish all the tasks in one cycle.

3.3 Is it possible to use a smaller number of threads ($< n^2$) to achieve the same performance?

The performance of **Figure 2** shows that the processing time starts to drop slightly after the number of threads reaches 3, which means it is predictable that the processing time will remain approximately the same while the multiplication is within certain number of threads. For this experiment, the range of thread number maintains a similar performance from 3 to 12 threads. Especially, **Figure 3** presents that the performance of the multiplication, which used only one thread number is better than the one with multiple threads. Therefore, it is highly possible to achieve the same performance while using a smaller thread number.

3.4 Does the number of physical core and cache size of computer affect the performance?

According to the figures above, both of them show the same pattern as the thread was not able to speed up the multiplication process under a certain number of threads. Primarily, the tasks of the threads are allocated by cores in the computer. For example, this experiment is based on a quad-core computer, which means each core is able to process one thread per cycle, in addition to the threads usually being allocated evenly by the cores. However, if there are 5 threads that need to be assigned, the cores automatically divide the extra task of the thread equally into each core, but need more cycles to complete the tasks where this scenario will slow down the processing time of the thread. In addition, cache size also plays an important role in processing time: larger cache size enables each thread to contain a larger thread. Hence, it is able to run more data for each cycle. **Figure 4** shows an issue during the experiment, as the system was not able to operate the program because the matrix size was too big with using threads.



```
for(int x=0;x<lines;x++)
    for(int y=0;y<columns;y++)
    {
        struct v *data = (struct v *) malloc(sizeof(struct v));
        data->i = x;
        data->j = y;
        pthread_t tid;
        pthread_attr_t attr; //Set of thread attributes
        pthread_attr_init(&attr);
        pthread_create(&tid,&attr,runner,data);
        //Make sure the parent waits for all thread to complete
        pthread_join(tid, NULL);
        //count++;
    }
```

Thread 1: signal SIGABRT

Figure 4. the error happened during 1000 by 1000 matrix with n^2 thread

4. CONCLUSION

This paper presented the performance of threads in matrix multiplication under different conditions. Although the purpose of thread is to achieve the parallel computation, the results showed that there were some limitations on threads where a larger number of threads does not necessarily mean better performance. In addition, this experiment also found a strong variation between thread performance and number of cores, and the cache size of the computer. Although the result of the thread performance is corresponding to the theory of threads, there are some uncertainties that might have affected the experiment, such as background applications slowing down the processing time during the excitation. Therefore, the result might need to be tested a few more times to get a more accurate number. To conclude, the thread is able to notably speed up the processing time, but it is only applicable for a certain number of threads and the compatible specification of the computer. Hence, it is important to manage the threads wisely in order to achieve more efficient multi-thread processing.

Appendix

First program: $n \times n$ matrix with n^2 threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <sys/types.h>

int e, f, lines, columns, *Matrix1, *Matrix2;

unsigned int c[20][20] = {};

struct v {
    int i; /* row */
    int j; /* column */
};

void *opreation(void *param)
{
    struct v *data = param;
    for (int i = 0; i < data->i+1; i++)
    {
        for (int j = 0; j < data->j+1; j++)
        {
            c[i][j] = 0;
            for (int k = 0; k < lines; k++)
            {
                c[i][j] = c[i][j] + (*(Matrix1 + (lines*i) + k)) * (*(Matrix1 + j + (columns*k)));
            }
        }
    }
    // printf("This thread execute [%d][%d]", data->i, data->j);
    pthread_exit(0);
}

int main(void){
    // dynamic matrix
    Matrix1 = (int *)malloc(lines * columns * sizeof(int));

    printf("Type the matrix lines:\t");
    scanf("%d", &lines);
    printf("Type the matrix columns:\t");
    scanf("%d", &columns);
    for (e = 0; e < lines; e++)
    {
        for (f = 0; f < columns; f++)
        {
            *(Matrix1 + e*columns + f) = e*columns + f;
            //printf("%4d ", *(Matrix1 + e*columns + f));
            //printf("%4d ", *(Matrix2 + e*f + f));
        }
    }
}
```

```

        printf("\n");
    }

    // dynamic matrix2
    /*
    Matrix2 = (int *)malloc(lines * columns * sizeof(int));

    printf ("Type the matrix lines:\t");
    scanf ("%d", &lines);
    printf ("Type the matrix columns:\t");
    scanf ("%d", &columns);
    for (e = 0; e < lines; e++)
    {
        for (f =0 ; f < columns; f++)
        {
            *(Matrix2+e*columns+f)=e*columns+f;
            // printf("%4d ",*(Matrix2+e*columns+f));
        }
        printf("\n");
    }

    */

    time_t time= clock();
    for(int x=0;x<lines;x++)
        for(int y=0;y<columns;y++)
        {
            struct v *data = (struct v *) malloc(sizeof(struct v));
            data->i = x;
            data->j = y;
            pthread_t tid;          //Thread ID
            pthread_attr_t attr; //Set of thread attributes
            pthread_attr_init(&attr);
            pthread_create(&tid,&attr,opreation,data);
            //Make sure the parent waits for all thread to complete
            pthread_join(tid, NULL);
            //count++;
            //printf("Thread  number %4d\n ",x*lines+y);
        }

    time = clock() - time;
    printf("Exetution time: %4f Sec \n", ((double)time) / CLOCKS_PER_SEC);

    /*
    for (int i = 0; i <lines; i++)
    {
        for(int j=0; j< columns; j++)
            //printf("%4d ",c[i][j]);
        printf("\n");
    }
    */
    free(Matrix1);
    free(Matrix2);
}

```

Second program: Matrix with a different number of threads

```
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#define NUM_THREADS 2

int e, f, lines, columns, *Matrix1;

unsigned int c[20][20]={};

void *hello1(void *arg)
{
    for (int i = 0; i < 20; i++)
    {
        for (int j = 0; j < 20; j++)
        {
            c[i][j]= 0;
            for (int k = 0; k <20; k++)
            {
                c[i][j]=c[i][j]+ (*(Matrix1+20*i+k))*(*(Matrix1+j+20*k));
            }
            //printf("%4d ",c[i][j]);
        }
        //printf("\n");
    }
    return 0;
}

/*
void *hello2(void *arg)
{
    for (int i = 2; i <4 ; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            c[i][j]= 0;
            for (int k = 0; k <4; k++)
            {
                c[i][j]=c[i][j]+ (*(Matrix1+4*i+k))*(*(Matrix1+j+4*k));
            }
            //printf("%4d ",c[i][j]);
        }
        //printf("\n");
    }
    return 0;
}

void *hello3(void *arg)
{
    for (int i = 2; i <3 ; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            c[i][j]= 0;
            for (int k = 0; k <4; k++)
            {
                c[i][j]=c[i][j]+ (*(Matrix1+4*i+k))*(*(Matrix1+j+4*k));
            }
        }
    }
}
```

```

        }
        //printf("%4d ",c[i][j]);
    }
    //printf("\n");
}
return 0;
}

void *hello4(void *arg)
{
    for (int i = 3; i <4 ; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            c [i][j]= 0;
            for (int k = 0; k <4; k++)
            {
                c[i][j]=c[i][j]+ (*(Matrix1+4*i+k))*(*(Matrix1+j+4*k));
            }
            printf("%4d ",c[i][j]);
        }
        printf("\n");
    }
    return 0;
}

void *hello5(void *arg)
{
    for (int i = 560; i <700 ; i++)
    {
        for (int j = 0; j < 1000; j++)
        {
            c [i][j]= 0;
            for (int k = 0; k <1000; k++)
            {
                c[i][j]=c[i][j]+ (*(Matrix1+4*i+k))*(*(Matrix1+j+4*k));
            }
            printf("%4d ",c[i][j]);
        }
        printf("\n");
    }
    return 0;
}

void *hello6(void *arg)
{
    for (int i = 700; i <840 ; i++)
    {
        for (int j = 0; j < 1000; j++)
        {
            c [i][j]= 0;
            for (int k = 0; k <1000; k++)
            {
                c[i][j]=c[i][j]+ (*(Matrix1+4*i+k))*(*(Matrix1+j+4*k));
            }
            printf("%4d ",c[i][j]);
        }
        printf("\n");
    }
}

```

```

        return 0;
    }

void *hello7(void *arg)
{
    for (int i = 840; i < 1000 ; i++)
    {
        for (int j = 0; j < 1000; j++)
        {
            c[i][j]= 0;
            for (int k = 0; k < 1000; k++)
            {
                c[i][j]=c[i][j]+ (*(Matrix1+4*i+k))*(*(Matrix1+j+4*k));
            }
            printf("%4d ",c[i][j]);
        }
        printf("\n");
    }
    return 0;
}
*/

int main(void)
{

    //Matrix 1
    printf ("Type the matrix lines:\t");
    scanf ("%d", &lines);
    printf ("Type the matrix columns:\t");
    scanf ("%d", &columns);
    Matrix1 = (int *)malloc(lines * columns * sizeof(int));
    //Matrix2 = (int *)malloc(lines * columns * sizeof(int));

    for (e = 0; e < lines; e++)
    {
        for (f =0 ; f < columns; f++)
        {
            *(Matrix1+e*columns+f)=e*columns+f;
            //printf("%4d ",*(Matrix1+e*columns+f));
        }
        //printf("\n");
    }


    //Matrix 2
    /*
    printf ("Type the matrix lines:\t");
    scanf ("%d", &lines);
    printf ("Type the matrix columns:\t");
    scanf ("%d", &columns);
    for (e = 0; e < lines; e++)
    {
        for (f =0 ; f < columns; f++)
        {
            *(Matrix2+e*columns+f)=e*columns+f;
            printf("%4d ",*(Matrix2+e*columns+f));
        }
    }
    */
}

```



```

        printf("\n");
    }
    */

    printf("\n");

    time_t time= clock();
    //threads
    pthread_t tid[NUM_THREADS];

    pthread_create(&tid[0],NULL,hello1,NULL);

    //pthread_create(&tid[1],NULL,hello2,NULL);
    //pthread_create(&tid[2],NULL,hello3,NULL);
    //pthread_create(&tid[3],NULL,hello4,NULL);
    //pthread_create(&tid[4],NULL,hello5,NULL);
    //pthread_create(&tid[5],NULL,hello6,NULL);

    for(int i=0;i<NUM_THREADS;i++)
        pthread_join(tid[i],NULL);

    time = clock() - time;
    printf("Exetution time: %4f Sec \n", ((double)time) / CLOCKS_PER_SEC);
    //printf("%.4f\n", (double)(time(NULL) - start));
    //pthread_exit(NULL);
    //printf("\n");

    /*
    //print matrix
    for (int i = 0; i <1000; i++)

    {
        for(int j=0; j< 1000; j++)
            printf("%4d    ",c[i][j]);
        printf("\n");
    }
    */

    //return 0;
}

```