OS Project 3 - Matrix Multiplication

CS307-Operating System, Chentao Wu, Spring 2018

∗ Name:Xuehan Sun   Email: Peter_suntain@outlook.com

∗ If there is any problem, please contact me on Wechat or my personal Email.

Contents

1 Project Introduction

Given two matrices A and B, where A is a matrix with M rows and K columns and matrix B contains K rows and N columns, the matrix product of A and B is matrix C, where C contains M rows and N columns. The entry in matrix C for row i column j ($C_{i:j}$) is the sum of the products of the elements for row i in matrix A and column j in matrix B. That is,

$$C_{i,j} = \sum_{n=1}^{K} A_{i,n} B_{n,j}$$

It is consisted by two steps:

1. Passing Parameters to Each Thread
2. Waiting for Threads to Complete

2 Project Environment

- Linux Ubuntu 16.04 and gcc;
- Intel Core i7-6700HQ CPU @2.60 GHz;
- NVIDIA GeForce GTX 970M;
- Windows 10 Education, 64 bit.

3 Project Realization

3.1 Thread Functions

The code for each thread to complete Matrix Multiplication is as follows:

```
inline void *run(void *data)
{
    int sum = 0;
    struct v *d = (struct v *) data;
// redefine data
    int j = d -> j;

    for (int m = j ; m < M ; m += NUMTHREADS)
    {
        for (int n = 0 ; n < N ; ++n)
        {
            sum = 0 ;
            for (int e = 0; e < K; ++e)
            {
                sum += A[m][e] * B[e][n] ;
            }
            C[m][n] = sum;
        }
    }
    printf("relative pid for %d is %g", j);
    pthread_exit(0);
}
```

1. Use the struct's pointer ***data** to pass parameters.
2. Define NUMTHREADS as the number of threads, three **for** loop to calculate results.
3. Use **printf (*"relative pid for %d is %g"*, j)** to print thread's identification;
4. Use **pthread_exit(0)** to exit the thread.

3.2 Create Thread

To generate a thread, we need to make use of function pthread_create(), this will pass the data pointer to our function above, and store its id in the "workers" array.

Originally, I use **M** $\times$ **N** threads but the result is not satisfying enough (see in Figure 7). So I then choose 10, 100 and n threads separately, which is shown in Part 4.2.2 and Part 5.2.2.

Here to avoid violating the original matrix data, I redefine data in the function "run()".

```
  for(int j=0;j<NUM_THREADS;++j)
  {
      struct v *data=(struct v*)malloc(sizeof(struct v));
      data->i=0;
      data->j=j;
      pthread_create(&workers[tmp],&attr,run,data);
      //create thread, run is the start_routine and data is the arg
   }
```

3.3 Waiting till Completion

Once all worker threads have completed, the main thread will output the product contained in matrix. This requires the main thread to wait for all worker threads to finish before it can output the value of the matrix product. Several different strategies can be used to enable a thread to wait for other threads to finish.

As is shown below, I use a **for** loop after the loop of **pthread_create()**.

And here I have met some problems, which will be further discussed in Part 6.1.

```
 for(int x=0;x<NUM_THREADS;++x)
 {
     pthread_join(workers[x],NULL);
  }
```

3.4 Calculating the Running Time

To compare the impact of different threads, I use Time and Throughput to assume their performance. Therefore, I need to calculate the time each program used. Here I use library **#include <sys/time.h>** for using function **gettimeofday()**.

Then I calculate their time and compare the original one with multi-threads ones, which is shown in 4.2.2 and 5.2.2.

```c
    gettimeofday(&start1, NULL);
    /*
    Thread Processing
    */
    gettimeofday(&end1, NULL);

    int timeuse1 = 1000000 * (end1.tv_sec - start1.tv_sec) + end1.
       tv_usec - start1.tv_usec;
    printf("The time of pthread is: %d usec \n", timeuse1);
```

3.5 Multi-thread for Windows

Since the main part of the code is similar on Windows, I also realize muti-thread programming on Windows.

1. **Create Thread.** On Windows, relevant API for this is **CreateThread** () function to create a new thread. Its codes are as follows:

```c
    thread_id[i * N + j] = CreateThread(NULL, 0, Proc, data, 0,
       NULL);
```

2. **Waiting till Completion.** Use Function **WaitForMultipleObjects** () to wait relevant threads coming to an end.

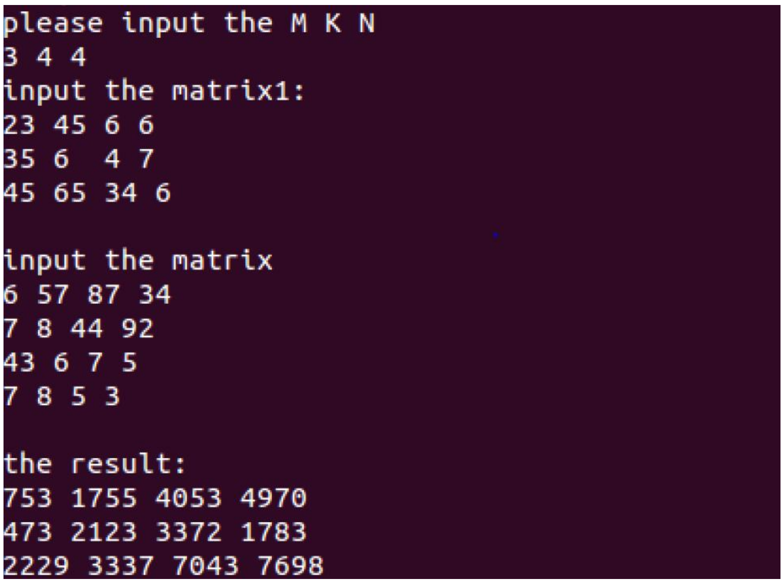
```c
    WaitForMultipleObjects(M*N, hthread, TRUE, INFINITE);
```

4 Project Result

4.1 Linux

4.1.1 Result

The Matrix Multiplication result on Linux is as follows:

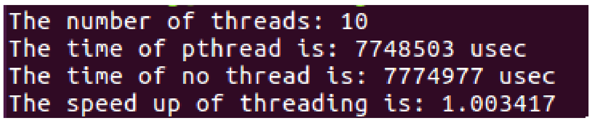


```
please input the M K N
3 4 4
input the matrix1:
23 45 6 6
35 6  4 7
45 65 34 6

input the matrix
6 57 87 34
7 8 44 92
43 6 7 5
7 8 5 3

the result:
753 1755 4053 4970
473 2123 3372 1783
2229 3337 7043 7698
```

Figure 1: Result on Linux

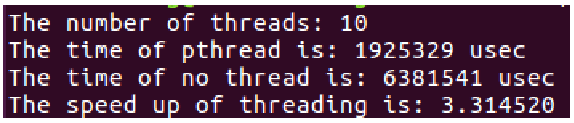
4.1.2 Acceleration of different core

When I'm working on this project, problems happen that I can hardly improve the acceleration with different numbers of threads. The problem occurs from the CPU cores I used is restricted, such that improvement with more threads is equally running on a single core.



```
The number of threads: 10
The time of pthread is: 7748503 usec
The time of no thread is: 7774977 usec
The speed up of threading is: 1.003417
```

Figure 2: Accelerate with 1 core



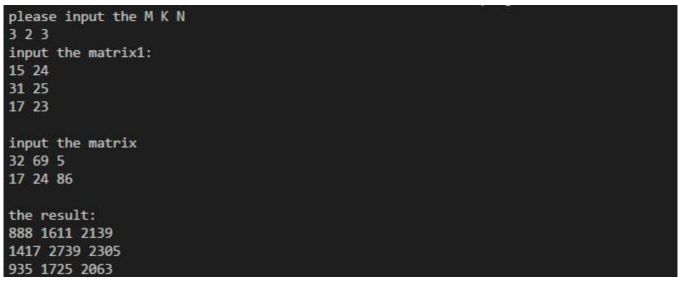
```
The number of threads: 10
The time of pthread is: 1925329 usec
The time of no thread is: 6381541 usec
The speed up of threading is: 3.314520
```

Figure 3: Accelerate with multi-cores

4.2 Windows

4.2.1 Result

The Matrix Multiplication result on Windows is as follows:

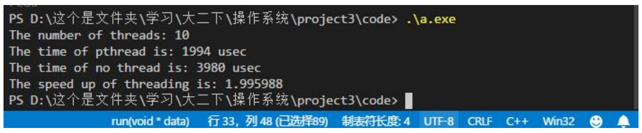


```
please input the M K N
3 2 3
input the matrix1:
15 24
31 25
17 23

input the matrix
32 69 5
17 24 86

the result:
888 1611 2139
1417 2739 2305
935 1725 2063
```

Figure 4: Result on Windows

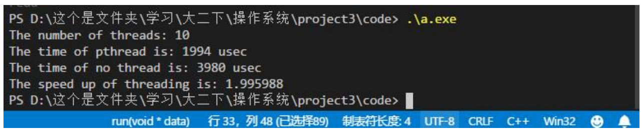
4.2.2 Acceleration of different threads

If we want to fully employ the acceleration, we must make relevant matrix large enough to avoid that creating pthread consumes too much time. Therefore, I choose 10 and 100 threads for a matrix whose size is 1000×1000.



```
PS D:\这个是文件夹\学习\大二下\操作系统\project3\code> .\a.exe
The number of threads: 10
The time of pthread is: 1994 usec
The time of no thread is: 3980 usec
The speed up of threading is: 1.995988
PS D:\这个是文件夹\学习\大二下\操作系统\project3\code>
```

Figure 5: Accelerate with 10 threads



```
PS D:\这个是文件夹\学习\大二下\操作系统\project3\code> .\a.exe
The number of threads: 10
The time of pthread is: 1994 usec
The time of no thread is: 3980 usec
The speed up of threading is: 1.995988
PS D:\这个是文件夹\学习\大二下\操作系统\project3\code>
```

Figure 6: Accelerate with 100 threads

It is easy to see that in such condition, a relatively-more-threads methods trades off for a more ideal acceleration. But will more threads always result in better performance? And how will the size of the matrix affect the performance? We will discuss this at Part 5.

5 Project Analysis

5.1 Thread Mountain

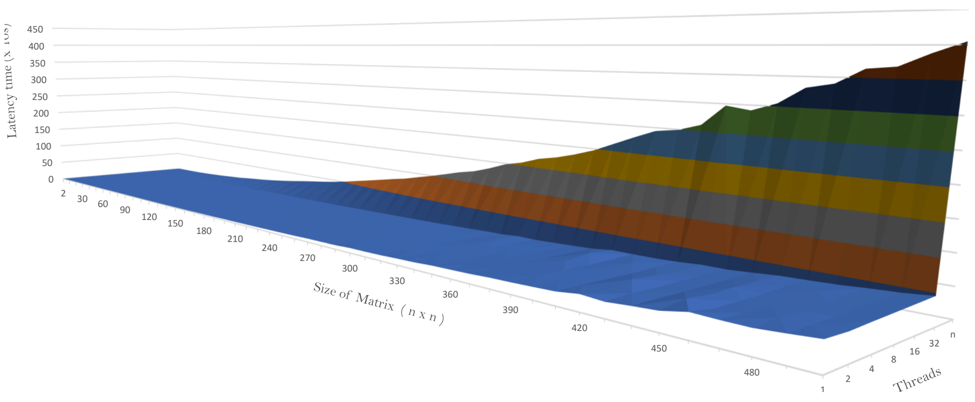


Figure 7: Thread Mountain

Similar to the idea from famous "Memory Mountain" introduce on «Computer Systems: A Programmer's Perspective»[1], I change the size of the matrix and the number of thread to examine how such things affect our performance.

If we continuously run this program with different sizes and threads, then we can recover a fascinating two-dimensional function of running time versus size and threads, which I named it as **"Thread Mountain"** Figure 7.

It is intuitive to see that $n \times n$ threads consumes much more time than others. So in the analysis follows, I change $n \times n$ threads into n threads to make them under the same magnitude.

From a brief observation, we can easily conclude that more threads takes fewer time (for the descending slope from 1 to 32 threads). Considering small size matrices, $n \times n$ threads is not severely bad performed. However, at the turning point around 256×256 (actual size 4 KB in cache), its performance gets worse and worse for a deep cache miss.

Also, we can see small "Wrinkles" in the **Blue Parts** of this mountain, because our number is generated randomly, so it is normally to see small variance.

5.2 Analysis of Thread Mountain

5.2.1 At the Foot of Thread Mountain

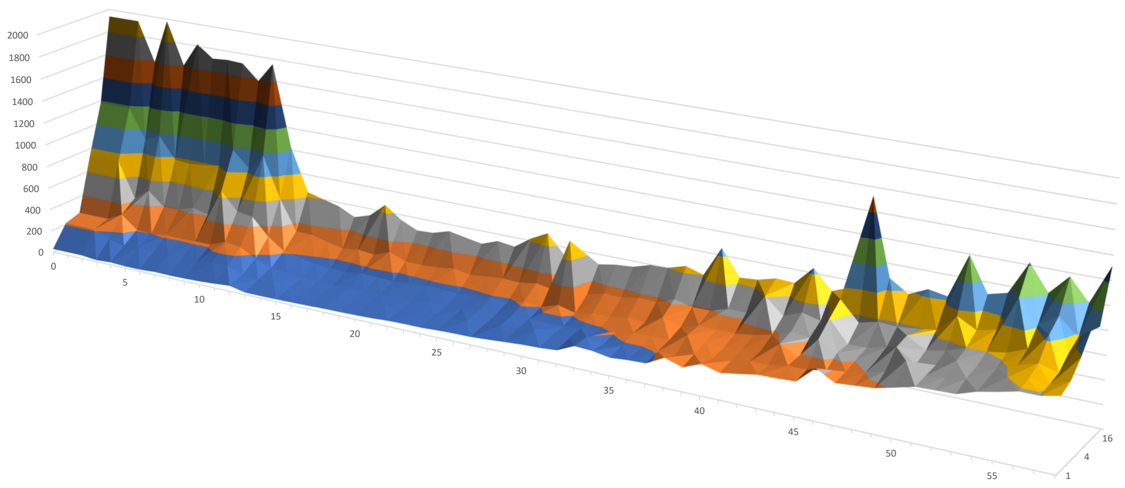
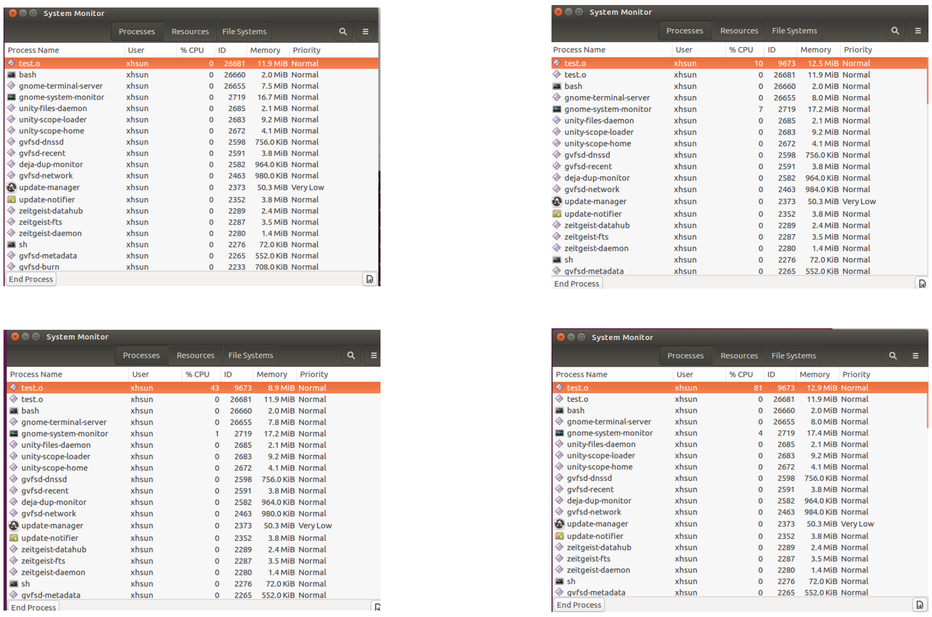


Figure 8: The Bottom of Thread Mountain

When we focus on the bottom part of Thread Mountain, as is shown in Figure 8, a strange phenomenon appears. We find that the starting part, from 1×1 to 10×10, consumes relatively 10 times of the following part. It is even longer than the matrix of 50×50!

This can derive from what we called **Cold Miss** in our cache. After putting into L1 Cache, the time wasted on Cache Miss is much shorter. Besides, I think that when size is small, program will not be allocated much space in CPU according to relative optimization. When I open System Monitor, I find these program will be allocated much more space when getting larger as in Figure 9.



Process Name	User	% CPU	ID	Memory	Priority
test.o	xhsun	0	26681	11.9 MiB	Normal
bash	xhsun	0	26660	2.0 MiB	Normal
gnome-terminal-server	xhsun	0	26655	7.5 MiB	Normal
gnome-system-monitor	xhsun	0	2719	16.7 MiB	Normal
unity-files-daemon	xhsun	0	2685	2.1 MiB	Normal
unity-scope-loader	xhsun	0	2683	9.2 MiB	Normal
unity-scope-home	xhsun	0	2672	4.1 MiB	Normal
gvfsd-dnssd	xhsun	0	2598	756.0 KiB	Normal
gvfsd-recent	xhsun	0	2591	3.8 MiB	Normal
deja-dup-monitor	xhsun	0	2582	964.0 KiB	Normal
gvfsd-network	xhsun	0	2463	980.0 KiB	Normal
update-manager	xhsun	0	2373	50.3 MiB	Very Low
update-notifier	xhsun	0	2352	3.8 MiB	Normal
zeitgeist-datahub	xhsun	0	2289	2.4 MiB	Normal
zeitgeist-fts	xhsun	0	2287	3.5 MiB	Normal
zeitgeist-daemon	xhsun	0	2280	1.4 MiB	Normal
sh	xhsun	0	2276	72.0 KiB	Normal
gvfsd-metadata	xhsun	0	2265	552.0 KiB	Normal
gvfsd-burn	xhsun	0	2233	708.0 KiB	Normal

Process Name	User	% CPU	ID	Memory	Priority
test.o	xhsun	10	9673	12.5 MiB	Normal
test.o	xhsun	0	26681	11.9 MiB	Normal
bash	xhsun	0	26660	2.0 MiB	Normal
gnome-terminal-server	xhsun	0	26655	8.0 MiB	Normal
gnome-system-monitor	xhsun	7	2719	17.2 MiB	Normal
unity-files-daemon	xhsun	0	2685	2.1 MiB	Normal
unity-scope-loader	xhsun	0	2683	9.2 MiB	Normal
unity-scope-home	xhsun	0	2672	4.1 MiB	Normal
gvfsd-dnssd	xhsun	0	2598	756.0 KiB	Normal
gvfsd-recent	xhsun	0	2591	3.8 MiB	Normal
deja-dup-monitor	xhsun	0	2582	964.0 KiB	Normal
gvfsd-network	xhsun	0	2463	984.0 KiB	Normal
update-manager	xhsun	0	2373	50.3 MiB	Very Low
update-notifier	xhsun	0	2352	3.8 MiB	Normal
zeitgeist-datahub	xhsun	0	2289	2.4 MiB	Normal
zeitgeist-fts	xhsun	0	2287	3.5 MiB	Normal
zeitgeist-daemon	xhsun	0	2280	1.4 MiB	Normal
sh	xhsun	0	2276	72.0 KiB	Normal
gvfsd-metadata	xhsun	0	2265	552.0 KiB	Normal

Process Name	User	% CPU	ID	Memory	Priority
test.o	xhsun	43	9673	8.9 MiB	Normal
test.o	xhsun	0	26681	11.9 MiB	Normal
bash	xhsun	0	26660	2.0 MiB	Normal
gnome-terminal-server	xhsun	0	26655	7.8 MiB	Normal
gnome-system-monitor	xhsun	1	2719	17.2 MiB	Normal
unity-files-daemon	xhsun	0	2685	2.1 MiB	Normal
unity-scope-loader	xhsun	0	2683	9.2 MiB	Normal
unity-scope-home	xhsun	0	2672	4.1 MiB	Normal
gvfsd-dnssd	xhsun	0	2598	756.0 KiB	Normal
gvfsd-recent	xhsun	0	2591	3.8 MiB	Normal
deja-dup-monitor	xhsun	0	2582	964.0 KiB	Normal
gvfsd-network	xhsun	0	2463	980.0 KiB	Normal
update-manager	xhsun	0	2373	50.3 MiB	Very Low
update-notifier	xhsun	0	2352	3.8 MiB	Normal
zeitgeist-datahub	xhsun	0	2289	2.4 MiB	Normal
zeitgeist-fts	xhsun	0	2287	3.5 MiB	Normal
zeitgeist-daemon	xhsun	0	2280	1.4 MiB	Normal
sh	xhsun	0	2276	72.0 KiB	Normal
gvfsd-metadata	xhsun	0	2265	552.0 KiB	Normal

Process Name	User	% CPU	ID	Memory	Priority
test.o	xhsun	81	9673	12.9 MiB	Normal
test.o	xhsun	0	26681	11.9 MiB	Normal
bash	xhsun	0	26660	2.0 MiB	Normal
gnome-terminal-server	xhsun	0	26655	8.0 MiB	Normal
gnome-system-monitor	xhsun	4	2719	17.4 MiB	Normal
unity-files-daemon	xhsun	0	2685	2.1 MiB	Normal
unity-scope-loader	xhsun	0	2683	9.2 MiB	Normal
unity-scope-home	xhsun	0	2672	4.1 MiB	Normal
gvfsd-dnssd	xhsun	0	2598	756.0 KiB	Normal
gvfsd-recent	xhsun	0	2591	3.8 MiB	Normal
deja-dup-monitor	xhsun	0	2582	964.0 KiB	Normal
gvfsd-network	xhsun	0	2463	984.0 KiB	Normal
update-manager	xhsun	0	2373	50.3 MiB	Very Low
update-notifier	xhsun	0	2352	3.8 MiB	Normal
zeitgeist-datahub	xhsun	0	2289	2.4 MiB	Normal
zeitgeist-fts	xhsun	0	2287	3.5 MiB	Normal
zeitgeist-daemon	xhsun	0	2280	1.4 MiB	Normal
sh	xhsun	0	2276	72.0 KiB	Normal
gvfsd-metadata	xhsun	0	2265	552.0 KiB	Normal

Figure 9: CPU Allocated to this program growing from 0%, 10%, 43%, 81%

From a Larger size varying from 1×1 to 90×90, the time for single thread is fastest. There also exist several "Peaks" along the ridge, causing by difference of random number and the current CPU allocated to it. Interestingly, in such case, until around 100×100 multi-threads programming does not performs better, and the expense paid for creating new threads is too large.

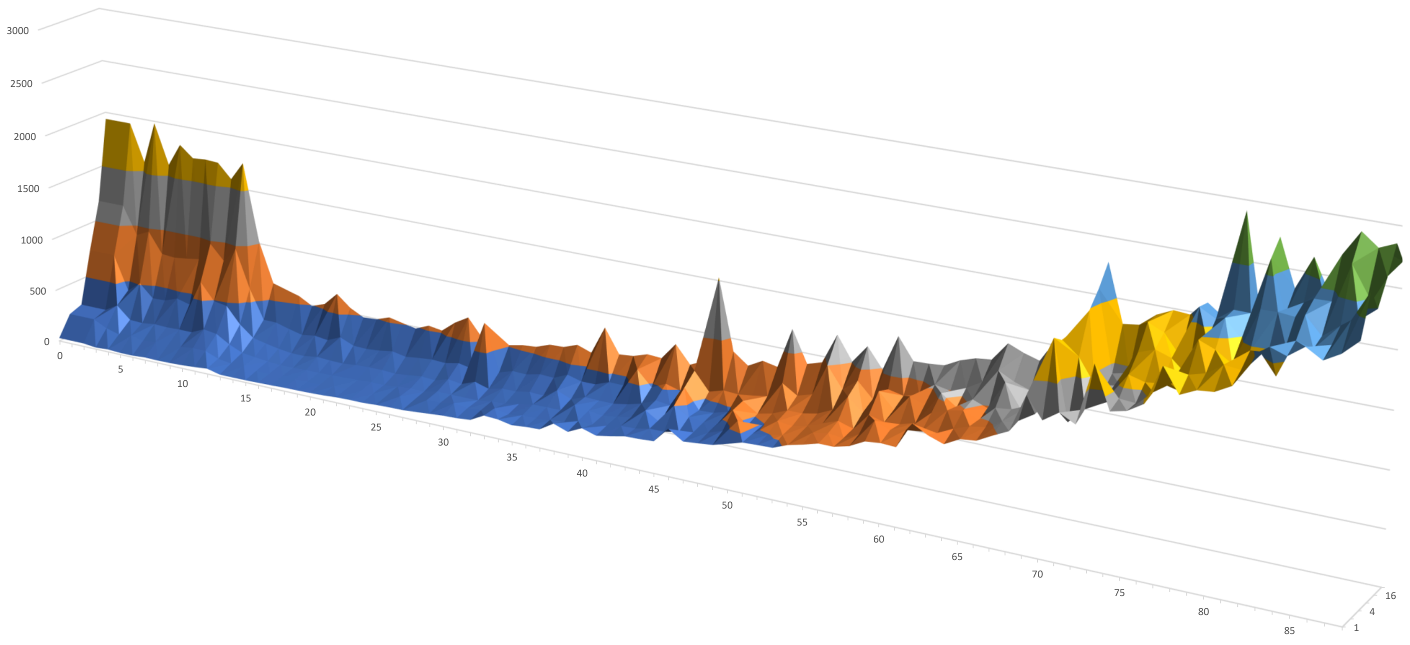


Figure 10: A Larger Bottom of Thread Mountain

Hence, when we need to conduct operations on some small matrices, we can use single threads as an optimal solution. And here we shall try our best to eliminate the side effects from the "Cold Start" by earlier warm-up before executing our programs.

5.2.2 Sharp Dips of the Throughput: Difference in Threads

This part we discuss the relation of different threads, and use **Throughput** to measure their behaviour.

At small size, different threads do not appear much difference. After about 75, the First Dip comes out between 2 threads and 4 threads with throughput double. The corresponding acceleration rate is about 37%-72%. After about 110, the Second Dips between 32 threads and n threads, speeding up about 50%-82%.

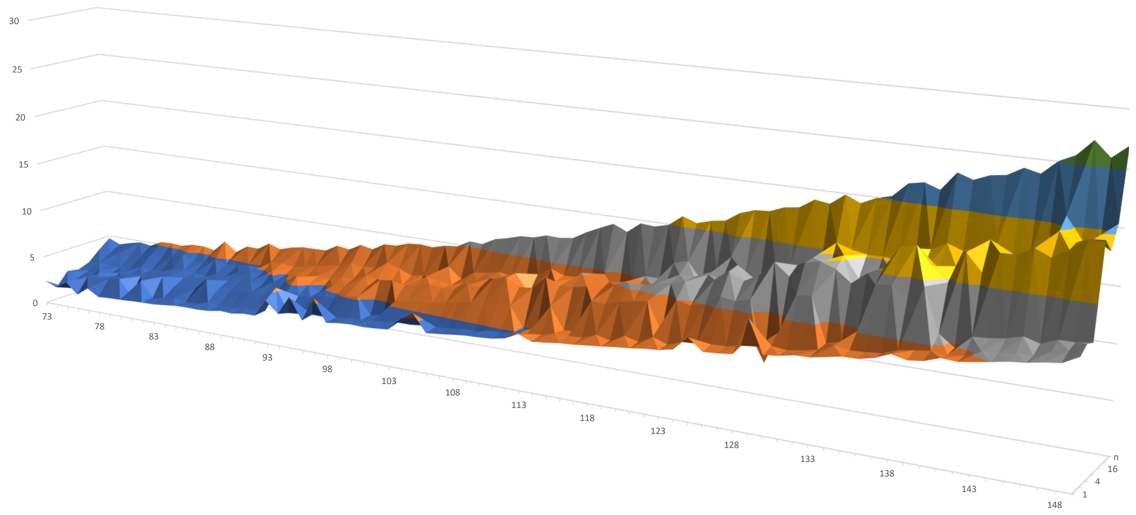


Figure 11: Throughput of $1, 2, 4, 8, 16, 32, n$ threads from size 70 to size 150

Things get different when size turns larger. Dips between 4 threads and 8 threads extend extremely large, then here is a platform between 8 to 32, and a small relative small dip between 32 and n. This means that the difference on performance varies much among 4 and 8, probably because of the constraint of cache's size. And we can see the trade off between calculating new matrix and creating new threads is fairly acceptable. But when the matrix grows too large, e.g. ≥ 300 as in Figure 13 because n will get too large, so the corresponding creating and scheduling so much threads in CPU consume unbearable time and space.

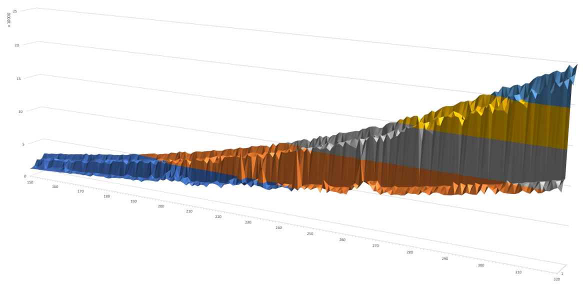


Figure 12: Throughput from size 150 to size 320

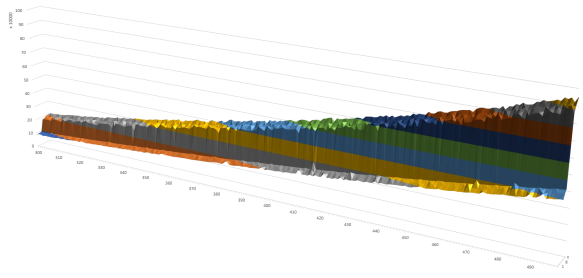


Figure 13: Throughput from size 300 to size 500

When viewing from a higher perspective, the difference between 4 and 8 threads gets more severe, and other differences become not that obvious. Besides, as is shown in Figure 14, the **Robustness** for multi-threads programming is better, for the wrinkles change severe on $1, 2, 4$-threads programming but remain almost unchanged on 4 n-threads programming.

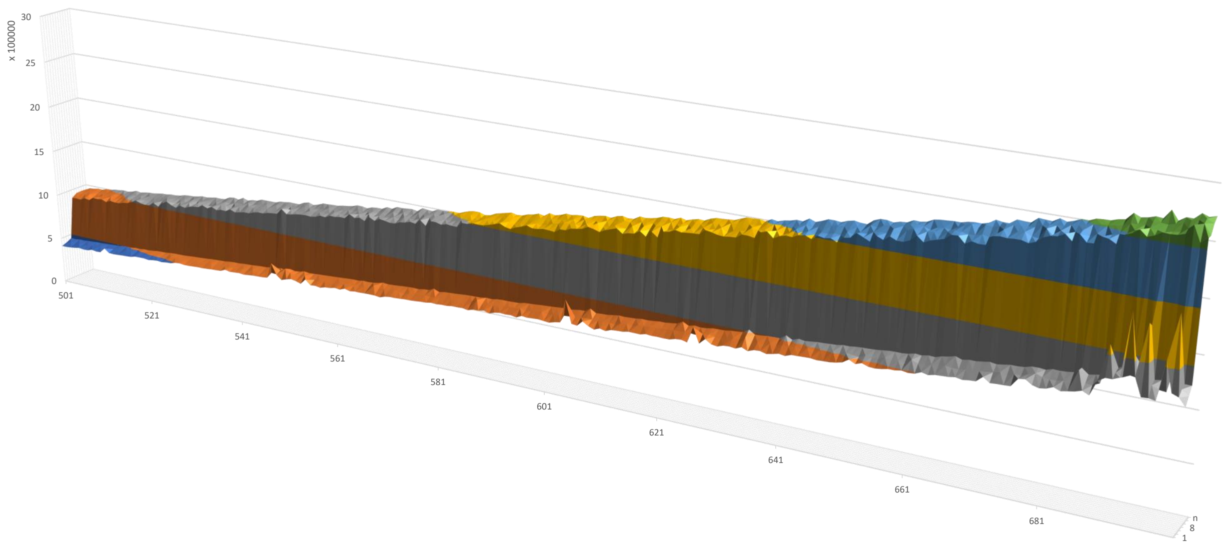


Figure 14: Throughput from size 500 to size 700

5.2.3 Ridge Line

When consider a special number of threads, e.g. 2 as is shown in Figure 15 and Figure 16, the trends for this goes approximately up with small wrinkles somewhere.

When the size is small, e.g. 90 to 150 in Figure 15, wrinkles not vary much. Using Matlab, the variance in performance is restricted in 28.7.

As size goes larger, e.g. 300 to 500 in Figure 16, wrinkles vary fairly much. The larger the size goes, the worse the variance becomes. At around 460 the variance is up to 174.3. The average for 300-500 is about 1.76 times of 90-150.

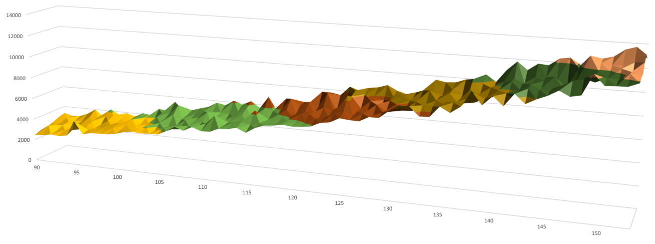


Figure 15: Ridge Line From 90 to 150

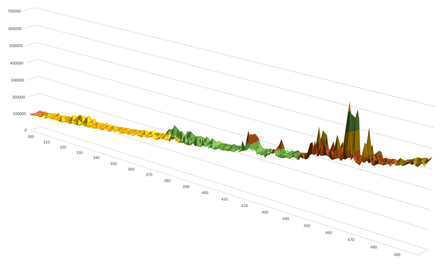


Figure 16: Ridge Line From 300 to 500

5.3 $n \times n$ Threads Performance

If we take a slice though the mountain, holding the stride constant as in Figure 7, we can see the impact of size on $n \times n$ threads. Since we create too many threads, its time latency can be badly frustrating. View from Figure 17, it is interesting to see two sharp dips at around size $2^6 = 64$ and $2^8 = 256$.

Such dips occur may result from the size constrained by the cache. Besides, around the platform and the edges, there appear some small drops but not violating the magnitude. This may result from conflicts with code and data lines, especially in such a busy program where threads' instructions interact with matrix data quite often.

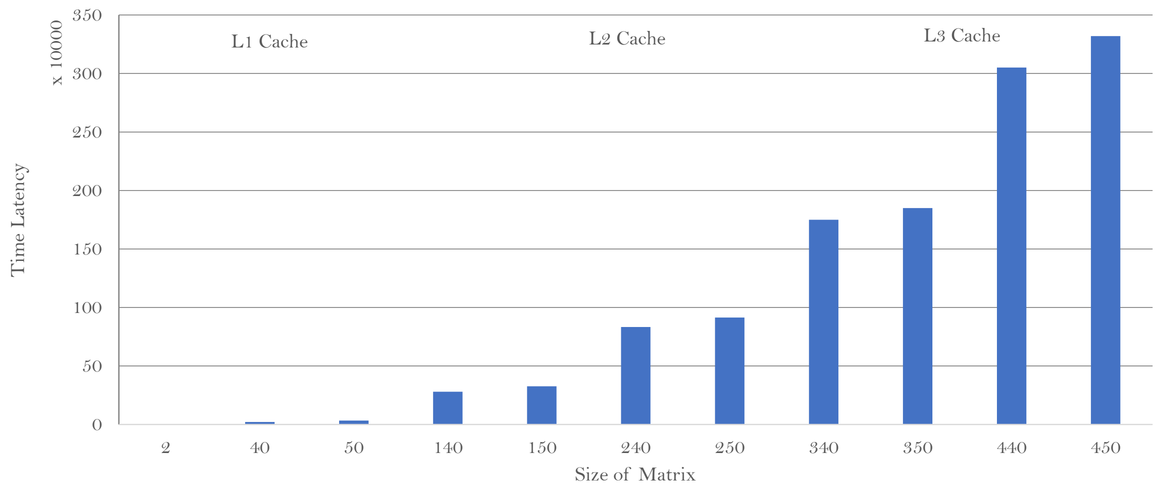


Figure 17: $n \times n$ Time Latency

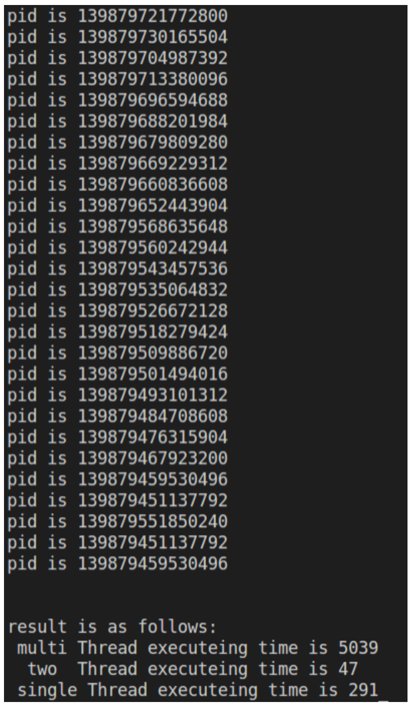
6 Problems Conquered

6.1 Pid is the Same

In my very first attempt, I output my Pid but all threads' id is the same, which means I didn't actually use parallel programming at all.

After searching on Google, I find my **Thread-join()** function comes exactly after each **CreateThread()** function. This means, every time I create a thread I exactly create it and wait for its end, but not create many threads, running them concurrently and end them at the same time.

Interestingly enough, when on Virtual Box, up-speeding is much faster than on dual-systems. The reason may be different version of Ubuntu, one is 14.04 and another is 16.04. Corresponding optimization is continuously improved.



```
pid is 139879721772800
pid is 139879730165504
pid is 139879704987392
pid is 139879713380096
pid is 139879696594688
pid is 139879688201984
pid is 139879679809280
pid is 139879669229312
pid is 139879660836608
pid is 139879652443904
pid is 139879568635648
pid is 139879560242944
pid is 139879543457536
pid is 139879535064832
pid is 139879526672128
pid is 139879518279424
pid is 139879509886720
pid is 139879501494016
pid is 139879493101312
pid is 139879484708608
pid is 139879476315904
pid is 139879467923200
pid is 139879459530496
pid is 139879451137792
pid is 139879551850240
pid is 139879451137792
pid is 139879459530496


result is as follows:
 multi Thread executeing time is 5039
  two  Thread executeing time is 47
 single Thread executeing time is 291
```

Figure 18: Different Pid Under Multi-Threads Programming

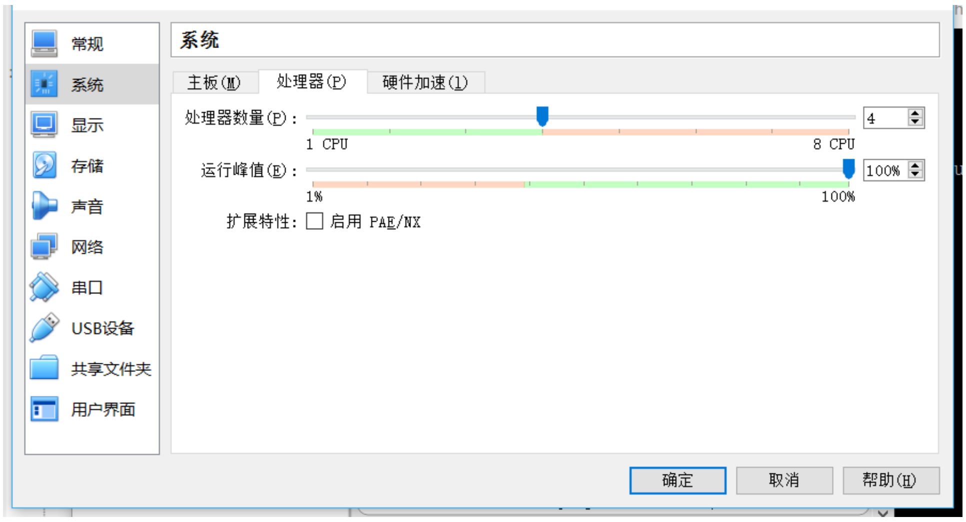
6.2 CPU Dumped Encountered with too Much Threads

When running simulation for Part5, my CPU always dumped when the matrix is too large. As I take a closer look into this, I find that when encountered with $n \times n$ threads, this will consume too much space allocated to it. So I change it into several parts and use **Divide and Conquer** methods to complete the simulation.

6.3 Running with Multi-Core

During the process of writing my report, I running my code on Virtual Box on my Windows instead of dual-boot operating systems to make it easier for me to compare it with my previous results. But although the program runs with pretty satisfying acceleration rate on Linux, it fails to accelerate on Virtual Box.

This result is annoying but strange, after I call the system performance monitor, I find my CPU is always fully employed. So I instantly think whether on Virtual Box I have not use multi-cores, the result is obvious as Figure 19 shows.



处理器数量(P): 4

运行峰值(E): 100%

扩展特性: ☐ 启用 PAE/NX

Figure 19: Setting Virtual Box with Multi-Core

7 Harvest

In this project, I have harvested pretty much. This project is easy to complete, but fully understanding it is another thing. By referring to what I have learned on Computer Architecture.

1. **A better understanding of multi-thread programming and its completion.** I used to write several multi-thread python crawlers with "Threading" library. Corresponding principles are similar but varies on details, such that I make a better progress on multi-thread programming.

2. **A better understanding of CPU and Threads.** As I have shown in Part 5, after creating and analysing such a "Thread Mountain" helps me better to make sense of CPU and its performance. With methods similar to CSAPP, I find such a mountain is quite intuitive and straightforward.

3. **A better understanding of Virtual Box compared with Dual-Boot Operating Systems.** The optimization used on Virtual Box reduces relevant performance to make two operating system can run at the same time.

8 Further Work

Here is some Further Work which I feel needs a try:

1. As is known, CPU may not be the best choice for Matrix Multiplication, so the performance on GPU and TPU is fascinating.

2. I have not explore the scheduling of CPU, so there is a chance that because of other programs running at the same time, my simulation Thread Mountain is influenced without my awareness. So how to make the CPU unchanged during my simulation may need further exploration.

3. The difference for threads programming on Windows and Linux is not clear, so maybe some tutorials can help figure out this and reconsider relevant performance on these two Operating Systems.

References

[1] Randal E Bryant and David R O'Hallaron. Computer systems: A programmer's perspective. pages 639–642, 2015.