EEE6207 - Threading

Hamish Sams

December 2019

1 Introduction

Here we discuss the analysis vs real world effect of threading specifically on calculating the multiplication of matrices. Theoretically for a single thread matrix multiplication we have a $O(n^3)$ multiply-add algorithm due to the three loops needed. For larger matrix values these computation times increase drastically, theoretically with n^2 threads working in parallel the program becomes a $O(n)$ multiply add algorithm meaning the processing time is directly proportional to the size of the matrix reducing the exponential increase into a slow linear increase.

1.1 Single threaded multiplication

During single threaded multiplication the program works left to right through the matrix, the core of which are shown below in Figure :1. The output of which can be seen in Figure:8 along with full source code in the Appendix.

```
\\Input of two input matrices and one output: **first, **second and **out, of height and width n
int sum;
for (int i = 0; i < n; i++)
        for (int ii = 0; ii < n; ii++) {
            sum = 0;
            for (int iii = 0; iii < n; iii++)
                sum = sum + first[i][iii] * second[iii][ii];
            out[i][ii] = sum;
        }
```

Figure 1: Matrix multiplication code basis

2 N^2 threads

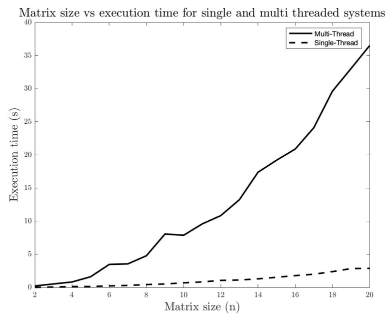
2.1 Multi threaded multiplication

A console output example of multi-threaded operation can be seen in the appendix(Figure:9). In N^2 thread mode a thread is created for each column vs row multiplication, this is represented in the iii for loop seen in Figure 1.

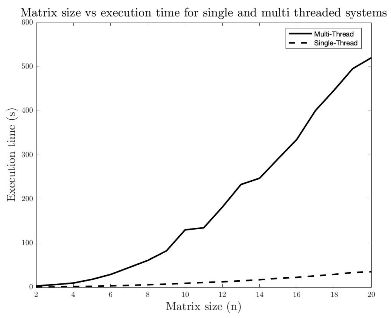
2.2 Analysis

Complete results can be found in Table :1 in the appendix.

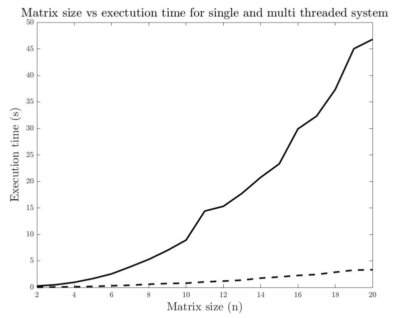
From the results in Figure:2 we see that the multi-threaded performance is worse in all cases with execution time rising non-linearly instantly disproving the Big-O theory. In Figure: 3 the ratio of execution time ($\frac{T_{Multi-thread}}{T_{Single-Thread}}$) from this we notice that the ratio is much lower in the region where there are less threads than processors (Less than n^2 threads). Above this point the execution time ratio rises quickly and then plateaus. These results would make theoretical sense as our threads are executed in parallel before quickly saturating the thread pool where more threads are ready than processors available, given each of these threads define their own memory more and more memory is required for the process at one time leading to higher level caching required and thus much longer execution time with less locality. The value of time ratio stays low for a few values above the $\sqrt{processors}$ point, this is likely as when IO operations occur the thread swaps out whilst the IO buffer loads. The one point that goes against these results is the first point in Figure: 3a, looking at the table of results (Table :1) the values measured are around the accuracy of the time measurement which could mean massive error margins. For this reason I believe these results are less trustworthy than those of the 32 core machine.



(a) 8 cores 10k iterations



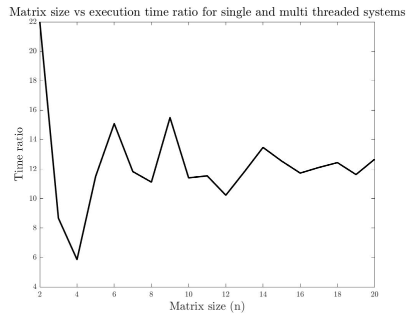
(b) 32 cores 100k iterations



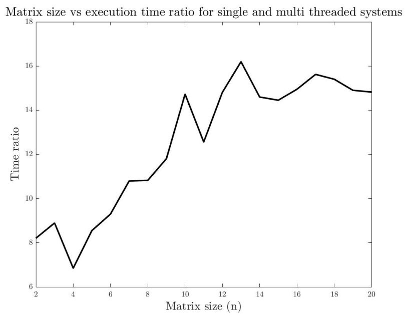
(c) 32 cores 10k iterations

Figure 2: Execution time vs matrix size comparisons using different core systems and iterations

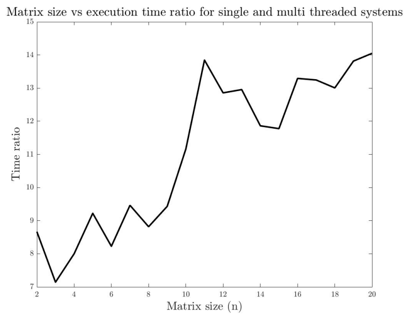
Overall it appears that for a low value of n each core is doing very little and for large values of n the number of threads saturates the number of system cores. Each thread also has very little processing to do meaning the thread setup and joining will likely much longer than the single threaded performance. For optimal performance it would be best to have each thread doing many calculations and have as many threads as processor cores available.



(a) 8 cores 10k iterations



(b) 32 cores 100k iterations



(c) 32 cores 10k iterations

Figure 3: Execution time ratio vs matrix size comparisons using different core systems and iterations

3 N threads

3.1 Multi threaded multiplication

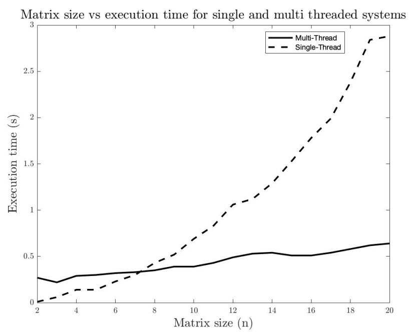
To modify the source code seen in the appendix (Figure:7) to make the code instead create n threads with n times more processing to do, we move one of the loops out of the thread creation into a task for the thread to complete, the changes of which can be seen below in Figure:4.

```
void *vectorMultiply(void *arguments){
    struct vectorV *args = arguments;
    for (int ii = 0; ii < args->n; ii++) {
        int sum = 0;
        for (int iii = 0; iii < args->n; iii++)
            sum = sum + args->first[args->i][iii] * args->second[iii][ii];
        args->out[args->i][ii] = sum;
    }
}
```

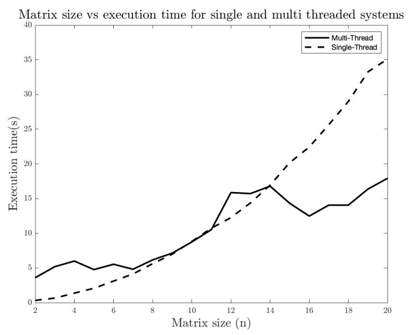
Figure 4: Caption

Now with N threads each thread has much more data to process and return, this means much more info can be processed on the processor caches before and IO operation is required. This means for larger matrices many threads can run at once and swap during IO operations for high processor task efficiency. Looking at Figure: 5a, the multi processor speed begins to be faster around where there are as many threads as processes, after this the multi-processing execution time is much lower than single processor as each process has tasks to swap to when waiting for IO accesses. For the 32 core system the multi processing becomes faster after around half the core count of threads is reached, this may be an artifact of the computer architecture as two 16 core processors are utilised. This again points to a useful region where there are more processes than processors to allow for task swapping during IO cycles. Although un-measured it's possible to theorise that in the future these speeds will not extrapolate to tiny processing speeds vs single threaded but instead the threads will again saturate and cause more (and thus slower) memory to be used whilst stagnant threads would exist that never get processed without a previous thread finishing.

3.2 Analysis

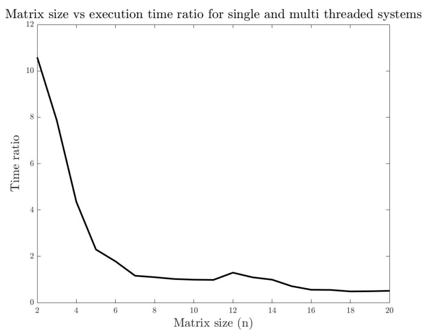


(a) 8 cores 10k iterations

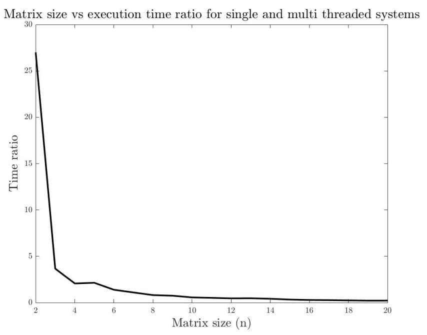


(b) 32 cores 100k iterations

Figure 5: Execution time vs matrix size comparisons using different core systems and iterations



(a) 8 cores 10k iterations



(b) 32 cores 100k iterations

Figure 6: Execution time ratio vs matrix size comparisons using different core systems and iterations

4 Overall analysis

Generically we see that not all situations are improved with threading, in particular systems that have little processing, threading situations with many more threads than cores. Due to how threads work, a system that requires sequential data would struggle to synchronise and would inevitably slow the system to single threaded speeds, if not worse. It would also appear that creating less threads than processors leads to under-performance, at least for the small amount of processing used for matrix multiplication. For much heavier loads threading may reap much larger rewards.

4.1 Conclusion

Overall it appears that the higher a threads load/processing required the better allowing for more local core cache data to be used for fast IO as well as to reduce any thread creation overheads for an overall faster process. This may mean that for the fastest processing speeds in a matrix calculator (such as MatLab) may be to assign a thread for each matrix to be computed reducing any low level memory accesses as much as possible. It it also desirable to have around double the threads created for each core available on a machine allowing for any threads to be swapped in and out when waiting for IO operations reducing CPU down-time without saturating the cache memory increasing IO operation delay.

Appendices

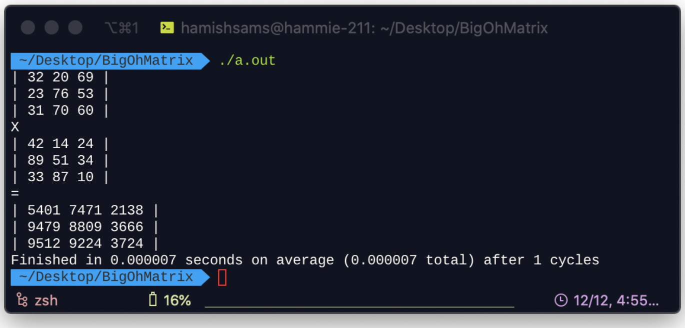
```c
// main.c
#define AUTO 1 // bool for automatic code running
#define CYCLES 10000 // Cycles of operation to average compute time
#define VERBOSE 0 // bool for console output
#define SETSIZE 2 //Matrix size
#define SINGLETHREAD 0 //bool for using single thread

#ifndef SINGLETHREAD
#define SINGLETHREAD 0
#endif
#ifndef AUTO
#define AUTO 0
#endif
#ifndef CYCLES
#define CYCLES 1
#endif
#ifndef VERBOSE
#define VERBOSE 1
#endif
#ifndef SETSIZE
#define SETSIZE 0
#endif

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

double totalTime;

struct vectorV {
    int n;
    int i;
    int ii;
    int** first;
    int** second;
    int** out;
};

int defineSize(){
    int n;
    if(SETSIZE){
        n=SETSIZE;
    }
    else if(AUTO){
        n=(rand() % 11);
    }
    else{
        printf("Size of NxN array (max 100):\n\r");
        scanf("%d",&n);
        if((n<=0) || (n>100)){
            printf("Matrix size invalid\n\r");
            exit(1);
        }
    }
    return n;
}

int** defineMatrix(int n)
{
    int* values = calloc(n*n, sizeof(int));
    int** rows = malloc(n*sizeof(int*));
    for (int i=0; i<n; ++i)
    {
        rows[i] = values + i*n;
    }
    return rows;
}

void killMatrix(int** arr)
{
    free(*arr);
    free(arr);
}

void enterValues(int len, int** arr){
    for (int i = 0; i < len; i++)
        for (int ii = 0; ii < len; ii++){
            if(AUTO){
                arr[i][ii]=(rand() % 100);
            }
            else{
                printf("Enter matrix values:\n\r");
                printf("%d:",ii+(i*len));
                scanf("%d", &arr[i][ii]);
            }
        }
}

void printMatrix(int len, int** arr){
    int max=-2147483647;
    for (int i = 0; i < len; i++)
        for (int ii = 0; ii < len; ii++){
            if(arr[i][ii]>max)
                max=arr[i][ii];
        }

    int ret = ceil(log10((double)max+0.001))+1;

    for (int i = 0; i < len; i++){
        printf("| ");
        for (int ii = 0; ii < len; ii++)
            printf("%-*d",ret,arr[i][ii]);
        printf("|\n\r");
    }

}

void *vectorMultiply(void *arguments){
    struct vectorV *args = arguments;

    int sum = 0;
    for (int iii = 0; iii < args->n; iii++) {
        sum = sum + args->first[args->i][iii] * args->second[iii][args->ii];
    }
    args->out[args->i][args->ii] = sum;

}

void singleThreadVectorMultiply(int n,int i,int ii, int **first, int **second, int **out){

    int sum = 0;
    for (int iii = 0; iii < n; iii++) {
        sum = sum + first[i][iii] * second[iii][ii];
    }
    out[i][ii] = sum;
```

```
}

void multiplyMatrix(int n, int** first, int** second, int** out){
    pthread_t tid[n][n];
    struct vectorV values[n][n];
    for (int i = 0; i < n; i++) {
        for (int ii = 0; ii < n; ii++) {
            if(!SINGLETHREAD){
                values[i][ii].n=n;
                values[i][ii].i=i;
                values[i][ii].ii=ii;
                values[i][ii].first = first;
                values[i][ii].second=second;
                values[i][ii].out=out;
                pthread_create(&tid[i][ii], NULL, vectorMultiply, (void *)&values[i][ii]);
            }
            else{
                clock_t time;
                time = clock();
                singleThreadVectorMultiply(n,i,ii,first,second,out);
                time = clock() - time;
                totalTime+=time;
            }

        }
    }
    if(!SINGLETHREAD){
        clock_t time;
        time = clock();
        for(int i=0; i < n; i++)
            for(int ii=0; ii < n; ii++)
                pthread_join( tid[i][ii], NULL);
        time = clock() - time;
        totalTime+=time;
    }
}

int main(void){
    for(int i=0;i<CYCLES;i++){
    srand(time(NULL));  //Start random seed

    int n;
    n=defineSize();

    int** first = defineMatrix(n);
    enterValues(n,first);

    int** second = defineMatrix(n);
    enterValues(n,second);

    if(VERBOSE){
        printMatrix(n,first);
        printf("X\n\r");
        printMatrix(n,second);
    }

    int** out = defineMatrix(n);

    multiplyMatrix(n,first,second,out);

    if(VERBOSE){
        printf("=\n\r");
        printMatrix(n,out);
    }
    killMatrix(first);
    killMatrix(second);
    killMatrix(out);
    }

    double timeTaken = ((double)totalTime)/((long)CLOCKS_PER_SEC);
    printf("Finished in %f seconds on average (%f total) after %d cycles \n\r",timeTaken/CYCLES ,timeTaken,
         CYCLES);

  return 0;
}
```

Figure 7: Complete source code

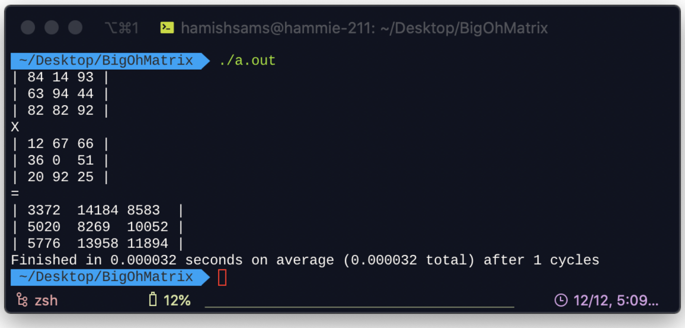
	8 cores 10k iterations			**32 cores 100k iterations**			**32 cores 10k iterations**			**n threads**	
N (matrix size)	n^2 Threads Time (s)	Single thread Time (s)	T_{mt}/T_{st}	n^2 Threads Time (s)	Single thread Time (s)	T_{mt}/T_{st}	n^2 Threads Time (s)	Single thread Time (s)	T_{mt}/T_{st}	8 core 10k iterations	32 core 100k iterations
2	0.22	0.01	22.00	2.79	0.34	8.21	0.26	0.03	8.67	0.27	3.6
3	0.52	0.06	8.67	5.87	0.66	8.89	0.50	0.07	7.14	0.22	5.19
4	0.82	0.14	5.86	9.45	1.38	6.85	0.96	0.12	8.00	0.29	6
5	1.61	0.14	11.50	17.78	2.08	8.55	1.66	0.18	9.22	0.3	4.77
6	3.47	0.23	15.09	28.92	3.11	9.30	2.55	0.31	8.23	0.32	5.54
7	3.55	0.30	11.83	44.80	4.15	10.80	3.88	0.41	9.46	0.33	4.82
8	4.78	0.43	11.12	60.83	5.62	10.82	5.29	0.60	8.81	0.35	6.17
9	8.06	0.52	15.50	82.53	6.99	11.81	6.98	0.74	9.43	0.39	7.14
10	7.87	0.69	11.46	130.03	8.83	14.73	8.93	0.80	11.16	0.39	8.75
11	9.58	0.83	11.54	134.79	10.73	12.56	14.40	1.04	13.85	0.43	10.53
12	10.84	1.06	10.26	181.55	12.26	14.81	15.30	1.19	12.86	0.49	15.86
13	13.24	1.12	11.82	232.89	14.38	16.20	17.75	1.37	12.96	0.53	15.72
14	17.38	1.29	13.47	247.12	16.93	14.60	20.76	1.75	11.86	0.54	16.79
15	19.20	1.53	12.55	291.51	20.17	14.45	23.32	1.98	11.78	0.51	14.33
16	20.88	1.78	11.73	335.16	22.42	14.95	29.91	2.25	13.29	0.51	12.48
17	24.10	1.99	12.11	400.56	25.64	15.62	32.32	2.44	13.25	0.54	14.06
18	29.61	2.38	12.44	446.68	29.00	15.40	37.33	2.87	13.01	0.58	14.06
19	33.03	2.84	11.63	495.51	33.25	14.90	45.04	3.26	13.82	0.62	16.37
20	36.49	2.88	12.67	520.25	35.10	14.82	46.79	3.33	14.05	0.64	17.92

Table 1: Table of multi-threaded vs single-threaded execution time



```
~/Desktop/BigOhMatrix  ./a.out
| 32 20 69 |
| 23 76 53 |
| 31 70 60 |
X
| 42 14 24 |
| 89 51 34 |
| 33 87 10 |
=
| 5401 7471 2138 |
| 9479 8809 3666 |
| 9512 9224 3724 |
Finished in 0.000007 seconds on average (0.000007 total) after 1 cycles
~/Desktop/BigOhMatrix
```

Figure 8: Console output of single 3x3 single threaded matrix multiplication (Auto = 1, Cycles = 1, Verbose = 1, Setsize = 3, SingleThread=1)



```
~/Desktop/BigOhMatrix  ./a.out
| 84 14 93 |
| 63 94 44 |
| 82 82 92 |
X
| 12 67 66 |
| 36 0  51 |
| 20 92 25 |
=
| 3372  14184 8583  |
| 5020  8269  10052 |
| 5776  13958 11894 |
Finished in 0.000032 seconds on average (0.000032 total) after 1 cycles
~/Desktop/BigOhMatrix
```

Figure 9: Console output of single 3x3 multi threaded matrix multiplication (Auto = 1, Cycles = 1, Verbose = 1, Setsize = 3, SingleThread=0)