CS 124, Programming Assignment 2

March 2017

1 Mathematical Analysis

Finding a reccurence for Strassen:

Strassen does 7 multiplications and 18 additions. It divides the problem into half each time. Hence, the recurrence is S(n) = 7S(n/2) + $18n^2$.

The conventional method does n^3 multiplications and does n^2(n-1) additions. Hence, the recurrence is T(n) = n^3 + n^2(n-1)

To find the cross over point, find the n at which switching from Strassen to conventional is equal to just doing conventional all along. Hence, replace the S(n/2) within S(n) by T(n) and set this equal equal to T(n).

$7[(n/2)^3 + (n/2)^2(n/2-1)] + 18n^2 = n^3 + n^3 - n^2$

$-1/4\ n^3 + 69/4n^2 = 0$

n = 0 or n = 69

Hence cross over point is 69.

2 Choosing A Language

I know Java, C and Python. I did not use Python because of the TF's recommendation to not use it. I chose Java over C for a few reasons: (a) Java has a garbage collector unlike C, making it convenient (b) In Java, I wouldn't have to worry about memory allocation, unlike C, especially as the main point of the assignment was not memory management. I modified and debated between data structures to modify my algorithm. (c) I could create classes to better organize my code while I tried to optimize it.

3 Overview of Algorithm

Let me quickly explain how my Strassen algorithm works. There are 4 new functions:

1. an add function that adds two matrices
2. a subtract function that subtracts two matrices
3. a divide function that splits my larger input matrices into four parts
4. a combine function that recombines the four submatrices into one larger matrix

When Strassen is called, it intializes 8 submatrices {A11, A12, A21, A22, B11, B12, B21, B22 } and then calls the divide function to fill these submatrices. It then calculates matrices P1 - P7 recursively, based on Strassen's formulas mentioned in the textbook using the add and subtract functions. It then uses the values in P1 - P7 to calucate the four submatrices (R1 - R4) that are combined using the combine function to create the result matrix.

4 Optimizations

1. First, I tried to reduce the number of intermediate matrices produced in Strassen by creating a class in Java that stored 4 variables (where row starts, where row ends, where column starts and where column ends) to keep track of the 8 submatrices. This meant I did not need the 8 submatrices and I saved space. During the running of Strassen, I indexed into the original matrices.

I reduced the number of intermediate matrices even further by not creating matrices R1 - R4 but by inputting the required calculations involving the P1-P7 matrices directly into my combine function.

However, after doing this I realized that reducing all these intermediate matrices had made my code hard to understand and did not improve Strassen's speed much anyways. Hence, I reverted to the use of intermediate matrices. (The un-used code is in the file I submitted, just commented out).

This was the classic case of ease of understanding and debugging vs a slight increase in efficiency. I chose the former as the increase in efficiency was very slight.

2. Initially, I used 2D arrays to keep track of all the matrices. However, I realized that multidimensional arrays are slower than one dimensional arrays so I changed my code to one-dimensional arrays (basically by replacing matrix[i][j] by matrix[i*dimension+j]. This improved my performance.

5 Further Possible Optimizations: Improving Efficiency of Conventional Multiplication

In simple matrix multiplication, we multiply elements in ith row of Matrix 1 with elements in jth column of Matrix 2 and then add these products up. As the additions are commutative, the order of multiplications does not matter. Let's just ensure that all the multiplications take place and are added up (the order this is done in doesn't matter). This will be most efficient when the j'th loop is the innermost loop and ith loop is the outermost loop. The cache line can be visualized as a grid running through the matrix. As we are using ints (4 bytes) and cache lines are ≥ 64 so there will be a lot of columns in any given row. Hence, this means that if we access multiple columns in the same row, it is likely to be the same cache line. This means we are more likely to have more hits than misses, resulting in more efficiency. Hence, we want to maximize the

chances of accessing different columns in the same row, which happens when j is the index of the innermost loop (as both Result Matrix and Matrix B use j).

Another possibility could have been writing the matrices in terms of (columns * dimension + row) instead of (row * dimension + column). This would also drastically reduce the number of different cache lines accessed.

6 Padding

To make Strassen work when n is odd and is not a power of two, I pad the initial two matrices to the next highest power of 2 with zeroes.

To be fair, this can be inefficient when n is large (e.g. an n of 518 is padded to 1024). An alternative was padding a single column and row of 0's within the Strassen function immediately when function strassenMultiply begins. However, this makes it harder to visualize where the padding is occuring as Strassen is recursive. Hence, I just pad the matrix once at the start and then remove the padding at the end while printing out the diagonal.

7 Experiments

Initially, I ran Conventional and Strassen Multiplication with a crossover point of 1 (base case = 1). While this wasn't strictly required, I just wanted to get a sense of what I was working with. See Figure 1. Note: Tables to go with ALL graphs are in Appendix. Clearly, Strassen is quite inefficient if we don't switch over to conventional in the middle.

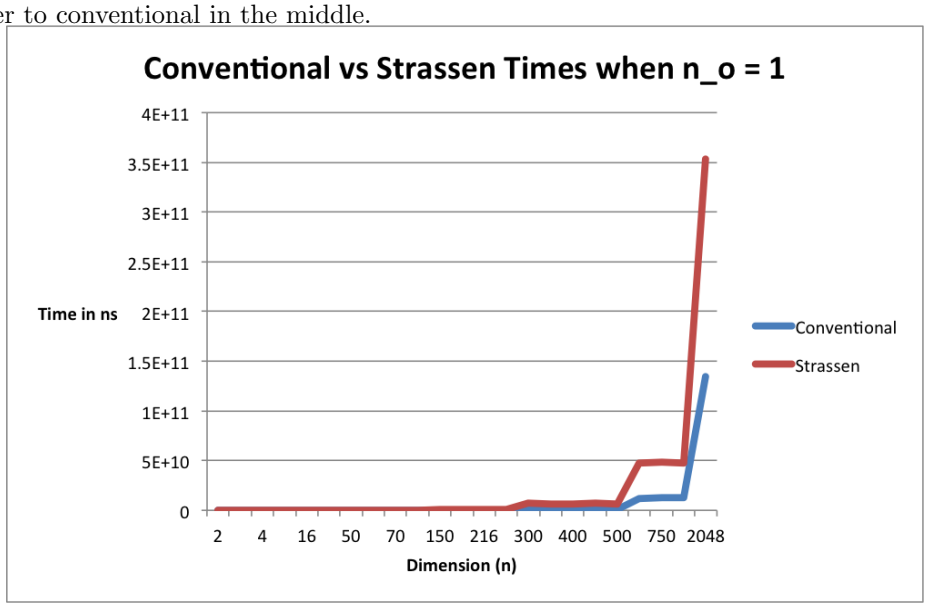

Figure 1

After that, I tried to find the optimal cross over point.

Initially, I fixed the cross points and then made tables of times at different dimensions. I made a few tables like this, changing the cross over point in each table. However, then I realized that this was not giving me a lot of data about the cross over points, so I fixed the dimensions and measured times for a range of cross over points.

Initially, I selected crossover points to find a general trend in the data. Hence, I chose crossover points that are slightly far apart. I find that the lowest values occur within the range $16 \leq n_o \leq 216$ (See Figure 2). The legend shows different dimensions.This crossover point range produces low times for all dimensions. The times start increasing when $n_o \leq 16$ and when $n_o \geq 216$. Figure 3 has a red circle that labels the range we are looking at.

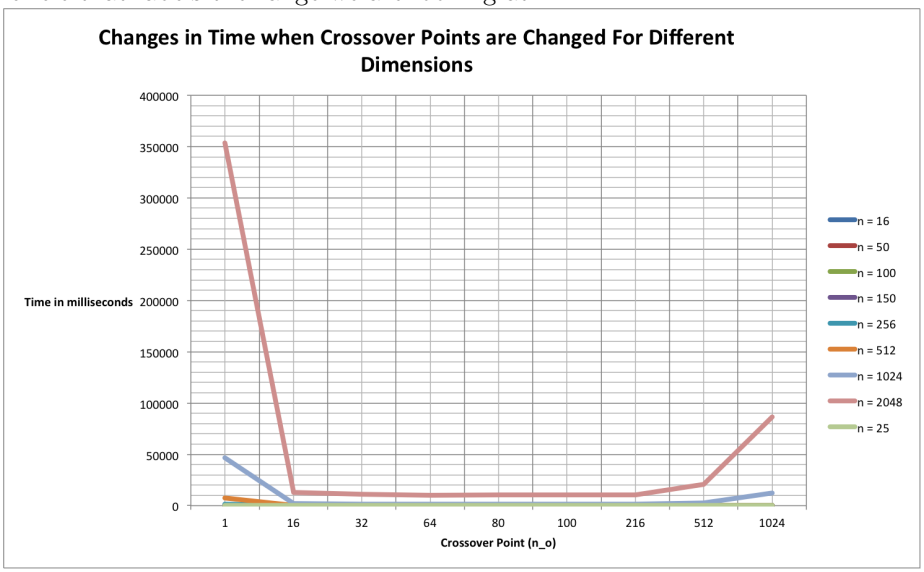

Figure 2

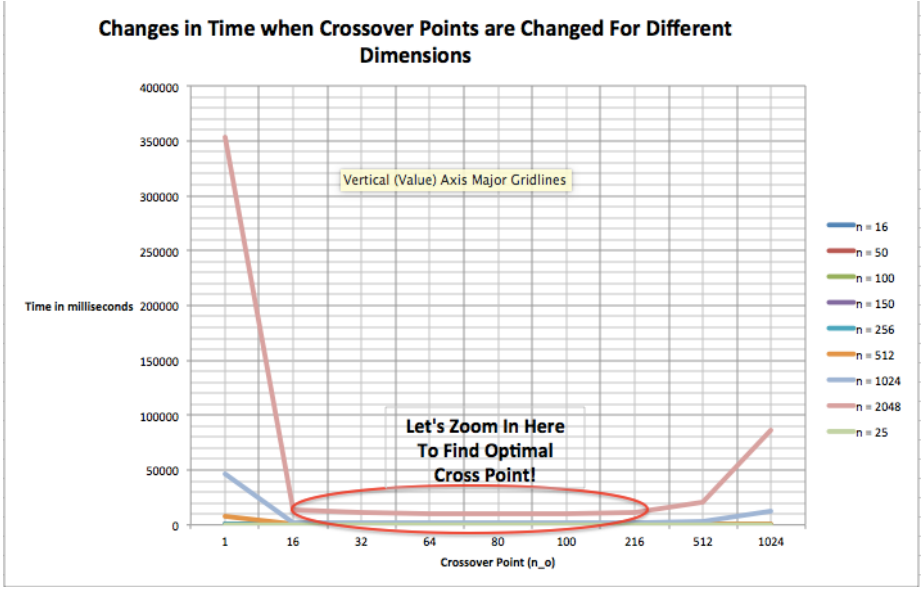

Figure 3

Hence, in my fourth graph, I zoom into this range and measure the times for a lot of crossover points within this range. This graph shows that while the variation isn't huge for when the dimension size is low, the greatest dip in times occurs when $n_o = 75$. Hence, my optimal cross over point is 75. This is quite close to my mathematical findings too ($n_o = 69$).

Notice that for high dimensions, some values of times produced are redundant e.g. if I am looking at a crossover point of 69, the times produced by dimensions 32*32 are just the same as naive multiplication. However, I have added these times in my tables and graphs for consistency's sake.

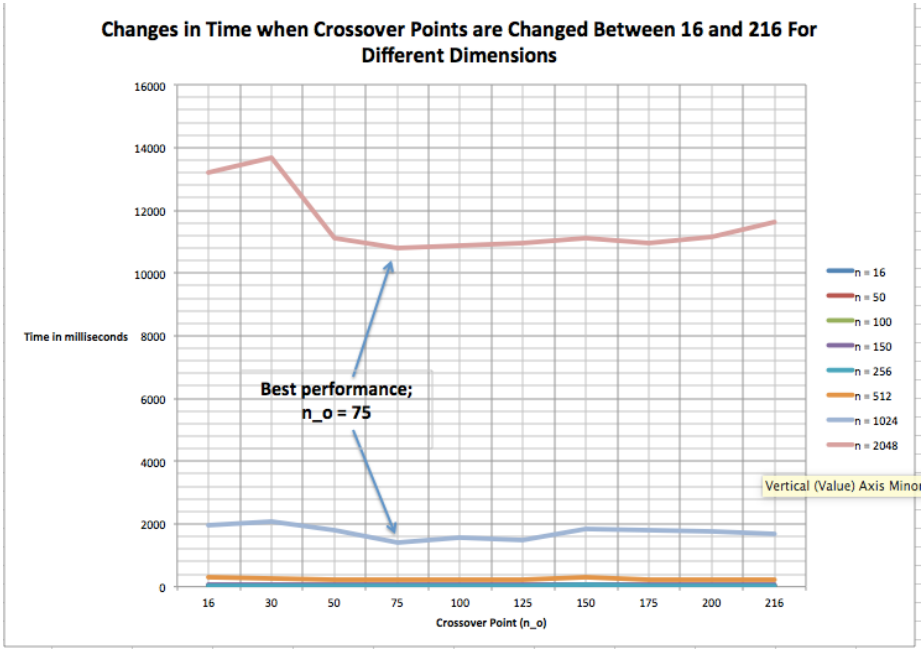

Figure 4

My last figure will compare the performance of Strassen when $n_o = 75$ and conventional multiplication just to give you an idea of how improved this version is.

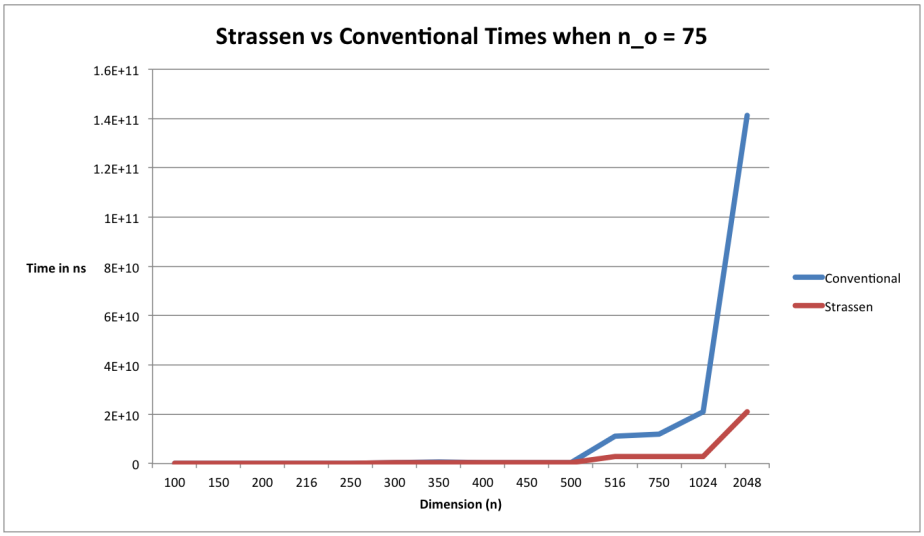

Figure 5

Notes: While my implementation can deal with odd numbers, most of the dimensions in the graph are even. This is because odd dimensioned matrices are just padded to become even dimensioned ones so it doesn't make a lot sense

to use them as their times will be the same as the even dimensioned matrices they are padded up to. However, just to ensure that the grader knows that my implementation works for odd powers, I have added a few odd dimension (e.g. 25X25 in Figure 2 - the last green line in the legend).

8 Notes

I have added a makefile. Without the makefile, just type java strassen 0 dimension inputFile

9 Citations

1. CS61 Caching 2. Book & Lecture Notes: Strassen Algorithm

Table 1: Conventional vs Strassen Times when n_o = 1 in nanoseconds		
n_o = 1		
n	**Simple Multiplication**	**Strassen Multiplication**
2	2000	17000
3	7000	175000
4	8000	131000
10	201000	7944000
16	11000	4650000
20	109000	43358000
50	856000	23584000
64	539000	26907000
70	5587000	1.59E+08
100	5320000	1.5E+08
150	46131000	9.65E+08
200	50293000	1.01E+09
216	46146000	9.43E+08
250	52168000	9.34E+08
300	461134000	6.83E+09
350	474431000	6.58E+09
400	523902000	6.44E+09
450	507380000	6.93E+09
500	494843000	6.64E+09
516	11826164000	4.75E+10
750	12476153000	4.84E+10
1024	12335738000	4.73E+10

2048	1.34402E+11	3.54E+11

Table 2 :Times in milliseconds to find a good range of cross over points

Dimension(vertical) / Cross Over (horizontal)	1	16	32	64	80
16	13	0	0	0	0
25	25	1	1	1	2
50	23	0	13	14	13
100	206	7	27	14	16
150	1020	49	69	53	44
256	1205	38	32	25	27
512	7602	288	218	208	207
1024	46538	1946	1604	1445	1501
2048	353616	13197	10856	10142	10265

Table 2 CONTINUED :Times in milliseconds to find a good range of cross over points

Dimension(vertical) / Cross Over (horizontal)	100	216	512	1024	Conventional
16	0	0	0	0	0
25	2	1	2	1	1
50	13	11	14	14	0
100	14	7	5	6	10
150	37	54	41	41	77
256	25	43	40	42	43
512	212	207	420	450	385
1024	1508	1552	2890	12288	12622
2048	10555	10794	20949	86631	132164

Table 3 :Times in milliseconds for cross over points between 16 and 216					
Dimension(vertical) / Cross Over (horizontal)	16	30	50	75	100
16	0	0	0	0	0
50	0	15	17	11	18
100	7	8	26	18	13
150	49	53	47	62	49
256	38	40	32	25	32
512	288	270	220	221	223
1024	1946	2060	1796	1407	1549
2048	13197	13681	11125	10790	10869

Table 3 CONTINUED :Times in milliseconds for cross over points between 16 and 216					
Dimension(vertical) / Cross Over (horizontal)	125	150	175	200	216
16	0	0	0	0	0
50	12	14	14	18	12
100	15	10	7	6	7
150	54	48	52	47	74
256	28	47	31	32	42
512	224	289	240	221	233
1024	1470	1825	1784	1754	1693
2048	10947	11109	10949	11160	11634

Table 4: Conventional vs Strassen Times when n_o = 75 in nanoseconds		
n	**Simple Multiplication**	**Strassen Multiplication**
100	5238000	28837000
150	35528000	68201000
200	58821000	2.04E+08
216	35119000	96344000
250	34298000	72083000
300	465972000	4.6E+08
350	572742000	4.89E+08
400	457681000	3.9E+08
450	444504000	4.15E+08
500	439685000	3.94E+08
516	11172838000	2.98E+09
750	11823243000	2.98E+09
1024	21057247000	2.91E+09
2048	1.41164E+11	2.11E+10