CS 124 Programming Assignment 2: Spring 2023

Your name(s) (up to two): Tomas Arevalo & Joanna Bai

Collaborators: None.

No. of late days used on previous psets: (6, 6)

No. of late days used after including this pset: (6, 6)

Overview This report details our process of finding an efficient cross-over point from Strassen's algorithm to conventional matrix-matrix multiplication. We outline our process in the following sections: Analytical Cross-Over Point, Experimental Cross-Over Point, Counting Triangles in a Graph, & Discussion.

1 Analytical Cross-Over Point

For conventional matrix-matrix multiplication,

$$T_c(n) = 2n^3 - n^2$$

$T_c(n)$ represents the number of operations conventional matrix-multiplication uses. For each entry we have n multiplications and we have $n-1$ additions. Since we have n^2 entries, then we $n^2(n+n-1)$ operations. This comes out to be $2n^3 - n^2$ operations which is what we have above.

For Strassen's matrix-matrix multiplication,

$$T_s(n) = 7 \cdot T_s(\lceil n/2 \rceil) + 18 \cdot (\lceil n/2 \rceil)^2$$

$T_s(n)$ represents the number of operations Strassen's algorithm takes, where there are 7 recursive calls on $n/2$ sized matrices (multiplication) and there are 18 operations to account for on $n/2$ sized matrices (addition/subtraction is n^2)

The cross-over point can be calculated as the value of n where there is no time difference between the two approaches, or where the two equations are equal to one another. To find the cross-over value, we set the two equations equal to each other and solve for n.

$$7 \cdot T_c(\lceil n/2 \rceil) + 18 \cdot (\lceil n/2 \rceil)^2 = T_c(n)$$

$$7 \cdot T_c(\lceil n/2 \rceil) + 18 \cdot (\lceil n/2 \rceil)^2 = 2n^3 - n^2$$

Note, we replaced T_s with T_c because the cross-over point will represent when we stop using Strassen's algorithm and switch over to the conventional algorithm. Thus, the recursions will switch over to the conventional multiplication for the value of n we are solving for.

We will consider two cases to calculate the analytical cross-over point:

1.1 Case 1: n is even

$$7 \cdot T_c(\frac{n}{2}) + 18 \cdot (\frac{n}{2})^2 = 2n^3 - n^2$$

Substituting in $T_c(n/2)$

$$7 \cdot (2(\frac{n}{2})^3 - (\frac{n}{2})^2) + 18 \cdot (\frac{n}{2})^2 = 2n^3 - n^2$$

$$\frac{7}{4}(n^3 - n^2) + \frac{18}{4}n^2 = 2n^3 - n^2$$

$$\frac{1}{4}n^3 - \frac{15}{4}n^2 = 0$$

$$n^2(n - 15) = 0$$

$$n = 15$$

In the case where n is even, our cross-over point is $n = 15$, meaning we should switch to the conventional matrix-matrix multiplication algorithm when n is less than or equal to 15.

1.2 Case 2: n is odd

$$7 \cdot T_c(\frac{n+1}{2}) + 18 \cdot (\frac{n+1}{2})^2 = 2n^3 - n^2$$

Substituting in $T_c(n + 1/2)$

$$7 \cdot (2(\frac{n+1}{2})^3 - (\frac{n+1}{2})^2) + 18 \cdot (\frac{n+1}{2})^2 = 2n^3 - n^2$$

$$\frac{7n^3}{4} + 8n^2 + \frac{43n}{4} + \frac{9}{2} = 2n^3 - n^2$$

$$-\frac{n^3}{4} + 9n^2 + \frac{43n}{4} + \frac{9}{2} = 0$$

$$n^3 - 36n^2 - 43n - 18 = 0$$

$$n \approx 37.1699$$

In the case where n is odd, our cross-over point is $n \approx 37.17$. Since we can't have non-integer sized matrices, we interpret this as 37, meaning we should switch over to the conventional matrix-matrix multiplication algorithm when n is less than or equal to 37.

2 Experimental Cross-Over Point

2.1 Conventional Matrix-Matrix Multiplication

Our implementation of conventional matrix multiplication follows the steps which one would do by hand exactly. We have 3 for loops, looping through the row of one matrix and the column of the other, multiplying the elements, and summing the products, and then adding that element to the resulting matrix, and then repeating this for all rows and columns.

2.2 Strassen's Algorithm

In order to test different values n_0, where n_0 is the cross-over point from Strassen's algorithm to conventional matrix multiplication, we created a function to create random matrices of size d (as given in the command line arguments). The function randomly populates the matrix with integers. We ran a majority of our experiments with negative and non-negative single-digit integers.

Below is a table of the runtimes (in seconds) of a random matrix of size $n = 1024$ with random single-digit integers. We ran these with and without compiler optimizations.

n_0	Without Compiler Optimizations	With Compiler Optimizations
1	3999.436768	1216.026123
2	883.812439	257.617004
4	232.130829	62.418987
8	75.613136	16.176100
16	32.023777	4.690466
32	18.771671	1.733937
64	14.629937	**1.031445**
128	**13.888178**	1.097120
256	16.044458	1.564642
512	21.650469	2.814819
1024	28.646996	3.069768

Our cross-over point with compiler optimizations is around $n_0 = 64$, and without optimizations is around $n_0 = 128$.

2.3 Double-Digit Elements

We wanted to see how the n_0 value would change if we used double-digit numbers as the elements of the matrices. When calculating our analytical cross-over point, we treated multiplication and addition operations as equals, but in reality, multiplying values with n-digits takes $O(n^2)$ time and adding values with n-digits takes $O(n)$ time. In the conventional matrix-matrix multiplication algorithm, there are 8 multiplications and 4 additions. In Strassen's algorithm, there are 7 multiplications and 18 additions/subtractions. With single digit elements, these runtimes are equal, so in theory, the value of n_0 would be relatively greater as the cost of Strassen's algorithm is effectively 25 operations to 12 in the conventional. Then it follows that as the number of digits increase, the one additional $O(n^2)$ multiplication in the conventional algorithm outweighs the time tradeoff of the 14 additional $O(n)$ addition/subtraction operations.

Below is a table of the runtimes (in seconds) of a random matrix of size $n = 512$ with random double-digit integers. We ran these exclusively with the compiler optimization.

n_0	With Compiler Optimizations
1	175.266953
2	36.823811
4	8.827707
8	2.294519
16	0.680369
32	0.245419
64	**0.152134**
128	0.156302
256	0.215568
512	0.362572

As you can see, using double digits as the elements in the matrices did not lower our value of n_0 as suspected, but it is still around $n_0 = 64$. This still could be more consequential for greater n-digit numbers, since double digit numbers are calculated with high efficiency in computer systems.

2.4 Compiler Optimizations

We mentioned earlier that we had compiler optimizations. The following is what we did. We attempted testing both with and without compiler optimizations, using the -O3 flag vs the -O2 flag, and immediately when we enabled compiler optimizations, our code ran faster. Thus, we can hypothesize that with compiler optimizations, our crossover point will be higher than if we had tested for our crossover point without it. This is because matrix multiplication involves a significant amount of arithmetic operations, and the -O3 flag can enable additional optimizations that are not included in the -O2 flag. For example, the -O3 flag can enable loop vectorization, which can speed up the computation by using vector instructions to perform multiple arithmetic operations simultaneously. Since the conventional matrix multiplication algorithm can be vectorized by the compiler more simply than Strassen's algorithm, we would expect the conventional algorithm to perform relatively better than Strassen's with compiler optimizations enabled. This would then lead to a higher crossover point with compiler optimizations enabled. While this hypothesis is somewhat just theoretical, because of the differences in runtime we saw, we can expect our hypothesis to be correct.

3 Triangles in a Random Graph

Using our implementation of Strassen's algorithm and the problem specifications, we counted the number of triangles in a graph. First, we randomly created the graph with 1024 vertices and represented its edges in a matrix A. Using various probabilities $p = 0.01, 0.02, 0.03, 0.04, 0.05$, we calculated whether or not to include certain edges. Then, we used Strassen's to find A^3, added the values in the diagonal, and divided by six.

The expected number of triangles is represented as $\binom{1024}{3}p^3$. Our results are outlined in the table below across five different runs, compared against the expected number of triangles based on the formula given. We also specify the average number of triangles counted across our five runs.

p	Expected	Run 1	Run 2	Run 3	Run 4	Run 5	Average
0.01	178.433	176	197	165	170	188	179.2
0.02	1427.464	1416	1467	1506	1456	1465	1462
0.03	4817.692	5239	4819	4877	4882	4895	4942.4
0.04	11419.71	11668	11840	11320	11300	11498	11525.2
0.05	22304.13	21442	22433	21890	21657	22100	21904.4

Our experimental results are similar to the expected results, though we were surprised with the variance we saw between runs. Logically, it makes sense that with greater probability that an edge is included, the number of triangles increases. Additionally, with greater probability an edge is included comes greater variance of the number of triangles counted between runs.

4 Discussion

4.1 Extending Strassen's

When implementing Strassen's Algorithm we followed the recursive definition and the multiplication modifications mentioned in Lecture Notes 9. However, we quickly realized that our algorithm only worked when the inputted dimension was a was a power of 2. Thus, the first step we did was read and fill our martices based off the input-file normally. Next we, implemented a function that checked whether the dimension inputted in the command line argument was a power of 2 or not. If it was, then we call Strassen's normally. However, if it was not a power of 2, which includes even and odd numbers meaning our Strassens should work for all positive integers, then it would resize the matrix using the *.resize()* function that already exists in C++. We resized it by taking the *dimension* x *dimension* matrix, and making it an m x m matrix, where m is the first power of 2 greater than the inputted dimension. Luckily, the *.resize()* function takes your existing matrix and resizes while preserving it current entries, and thus, the new matrix is exactly as it was before except that it is now padded with extra rows and columns of 0's. Finally, we pass in this padded matrix into Strassen's the padded matrices will return the correct result, but also padded with extra rows and columns of 0's. Thus, when printing the diagonal, we still only print the first *dimension* entries so that we don't print the padded 0's.