Introduction

Matrix multiplication is an example where using a divide-and-conquer strategy may be beneficial to solve the problem faster. The typical approach to solving matrix multiplication would take time, however by utilizing divide-and-conquer, we can trim that down a bit. Divide-and-conquer works by taking the problem, dividing it into smaller digestible sub-problems, and then solving them instead of the problem as a whole. As you will see in the complexity comparisons, there is one divide-and-conquer method that theoretically proves to be the best option. My aim for this project was to test out three methods and to see if they hold true to the assumed theoretical timings.

Theoretical Complexity Comparisons

Classical Matrix Multiplication

If A = () and B = ( are square n x n matrices, then in the product C = AB, you would define for i,j = 1,2,…,n by

Because this method uses 3 for-loops of n iterations, the procedure takes time.

Divide-and-Conquer

Instead of working line by line, you could divide the matrix into 8 submatrices that you solve recursively. Suppose we partition matrix A, B, and C into four n/2 x n/2 matrices where

, , , then we can rewrite C = AB as

= .

From here, you can see that

Through these equations, you can define a recursive, divide-and-conquer algorithm as follows:

Divide-And-Conquer(A,B)

1. Let n = A.rows
2. Let C be a new n x n matrix
3. **If** n==1
4. =
5. **Else**
6. partition A, B, and C into four n/2 x n/2 submatrices
7. = Divide-And-Conquer (, + Divide-And-Conquer (,
8. = Divide-And-Conquer (, + Divide-And-Conquer (,
9. = Divide-And-Conquer (, + Divide-And-Conquer (,
10. = Divide-And-Conquer (, + Divide-And-Conquer (,
11. **return C**

For the base case (line4), it is just a simple multiplication that takes constant time.

T (1) =

Otherwise (line6-10), partitioning takes time, the 8 recursive calls take 8T(n/2), and the four matrix additions (each being take time. Altogether the divide-and-conquer approach is:

Solving for the recurrence relation we see that:

T(n) =

= 8(8T+)+

=

=

=

=

= for some constant k

=

Assume = k

=

=

=

=

(c=1, =1)

So, for the divide-and-conquer algorithm we end up with a solution of which is no better than the classical matrix multiplication.

Strassen’s Method

Strassen improved upon the divide and conquer strategy by lowering the amount of recursive calls by one so that instead of 8 n/2 calls, you would have 7. The cost of this elimination requires more matrix additions, yet the overall cost of the algorithm is lowered.

1. First, divide A and B into 8 n/2 matrices like the previous algorithm taking O(1) time.
2. Second, create 10 matrices in time using the formulas:
3. Recursively call multiply n/2 x n/2 matrices seven times using the formulas:
4. Solve for the parts of C in constant time and return them to C:

All of the above steps give a recurrence relation of:

Solving for the recurrence relation we get:

=

=

=

Assume for some integer k

=

=

=

=

= (b + c)

= O () = O

So, the method developed by Strassen should, theoretically, compute matrix multiplication faster than the other two methods.

Test Strategies

To begin testing, I first needed to implement the three algorithms. I chose Java as my language since I have been coding in it more recently and since I was curious to see if it would follow the theoretical assumptions. Although I knew how the algorithms would work, understanding how to code them was still a challenge. It took me many iterations and a lot of reading my book to get them all correct. Once they began working, I was curious to see how they stacked up.

I decided to run the classical multiplication first with size of matrices n = 2^5 and the values inside the matrices set to 3. So, altogether that’s two 32 x 32 matrices with values set at 3. Not surprisingly, the program immediately finished and the time read 0 seconds. This meant that I would probably need to be a little more accurate with the time so that I could differentiate between values less than 1 second. I made a few changes and ran it again. The time then read out 0.0014 seconds.

From there, I altered the program to run the divide-and-conquer method which would hopefully reveal a similar timing as they are both on the order of n^3. To my surprise, the divide-and-conquer method was actually much slower. The time read out 0.012 seconds. I checked over the code to make sure it was working and decided that it was probably due to the time taken to allocate memory for the submatrices. I then wondered if the Strassen method would show similar results.

After running the Strassen method under the same input, the time read out 0.0198 seconds. The result was very close to the divide-and-conquer result, yet was also higher than the classical result. I was unsure of whether or not I made a mistake and so decided that it needed more testing at different sizes of n.

I began testing matrix sizes of 2^n from n=1 up to n=12, at around 10 tests at each size to get an average timing. I figured that at the highest input of 2^10 (or n = 1024) that the result would take quite a while to compute and so I only tested the higher values once or twice as I was limited on time. I wrote down the timing of each trial and tried to understand the results as best I could. Unfortunately, my computer did not make it far before it began to error when testing with the divide-and-conquer and Strassen methods. The editor, NetBeans, gave me a Java Heap Error which I promptly looked up. It turns out that the process ran out of memory which was probably due to it having to create new copies of the submatrices in the recursive calls. I then researched an explanation of the Strassen method in my book, hoping to find a better way to do the partitioning. The book mentioned that you could do index calculations rather than copying entries, however it did not help with achieving that. I did my best to make this fix but kept hitting dead end after dead end. I actually have no idea how to do this in Java and only a small idea of how it might be accomplished in C++. I still needed a fix though, to be able to test at higher values.

Through some googling, I found I could change the amount of memory allowed to be used by the program. The initial setting was supposed to be 512MB, so I doubled it. That did not help much so I then raised it to what I considered my max: 6GB. Success! I could now do testing at higher values.

I continued my testing but soon reached values that would take large amounts of time. What initially finished in a few seconds was now taking over a half hour. Then, half an hour quickly became many hours of one single test. There were many attempts that failed either because my computer shut off, or my computer decided to update during testing, or I just gave up after 5 hours when I had other things to do. Even so, I still collected a decent amount of data and was beginning to draw some conclusions.

Before I got around to conclusions, I wondered if there was anything else I could do to improve my Strassen method. Since I had initially based the method off of pseudocode in my book, my version of the Strassen method had a base case of 1. Through reading class notes and google searches, I found that it might improve if I were to use a higher base case, such as n = 2. I made some changes in my code and added a second copy of the Strassen method but with a base case of 2. It turns out that this did improve the timing of the Strassen method by a decent amount. Still, I had one more change that I was curious about.

I wanted to see whether or not there would be a difference in timing if the matrices had their values set to 0 or if they instead were initialized with random values. My assumption was that there would be no difference, other than the normal variance, in the timing between the different value sets. From my results, it would seem that my assumption was correct. There was no difference in the way the algorithm ran, so it would follow that the results would also be the same.

Results

The general results I compiled after running the tests showed a ranking of the methods used. The worst performance came out of the divide-and-conquer method while the best performance was found in the classical multiplication method. The Strassen method with a base case of 1 performed similarly to the divide-and-conquer method, while the Strassen method with a base case of 2 performed much better and seemed to be closing the gap on the classical multiplication method. I would venture to guess that once your input passes size 2^14 or possibly 2^15, the time taken to allocate memory would begin to not matter and the Strassen method would begin to leap ahead as the fastest time.

|  |  |  |  |
| --- | --- | --- | --- |
| **n = 2048** | **entries = 3** | **entries = 0** | **entries = rand(0-9)** |
| Classical | 68s | 69s | 68s |
| Divide-and-Conquer | 2113s | 2112s | 2114s |
| Strassen's (base = 1) | 969s | 961s | 965s |

When comparing the timing with different values set in the matrices, I found little to no effect on the computation time. For example, the time to do multiplication for size n = 2048 with the DNC method only varied by a second. That was a less than 1% of a change.

I found similar results for the other methods. They only different in time by very little and it seemed to be clear that it did not matter. When using random inputs on the matrices, I found the same effect.

When comparing the timing for the different base cases of Strassen’s method, I saw quite an improvement. The implementation that fell all the way down to a matrix multiplication of size 1 x 1 took much longer to solve. The variation that instead stopped at size 2 x 2 was an improvement, even at small trial sizes. You can see from the graph below, that the base case 2’s line was growing slower when compared to the base case 1’s line, meaning that it was much faster.

Strengths and Constraints

My greatest takeaway from this project was the effect that allocation of memory had on the timing. It is a good reminder that efficiency with memory usage can be very important and that it has real-world effects. I think Java struggles in this area since it does not have pointers (to my knowledge). If I were to redo this project, I would try to implement it in a more efficient manner with regards to memory as well as see how it varies between different languages. I’m sure that C++ or Python would have different, and possibly faster, results. Another easy improvement on the project would be to use a stronger pc. The laptop I used for the project was limited in RAM and did not have a fast CPU. Both of those improvements would let me test on higher values to see if my theory is correct that copying the matrices does not matter in the long run.

The following pages include references, the average data set timings, and the code used in the project.