CS3310 - Design and Analysis of Algorithms

Cal Poly Pomona

Project 1

Fall 2022

Description:

Matrix Multiplication Algorithms

Name: Fidelis Prasetyo

Email: (fprasetyo@cpp.edu)

BroncoID: 015765555

Github: https://github.com/fidelisprasetyo

Methodology

The test was divided into 2 parts. The first part is the sanity check to make sure that the matrix multiplication algorithms are implemented correctly. This part is done by passing same 2 4x4 matrices into each algorithm, if every algorithm yields the same correct result, then we can conclude that the algorithms are at least implemented correctly as matrix multiplication algorithms regardless of their performance. The matrix multiplication performed in this part is as follows:

$$\begin{bmatrix} 2 & 0 & -1 & 6 \\ 3 & 7 & 8 & 0 \\ -5 & 1 & 6 & -2 \\ 8 & 0 & 1 & 7 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 6 & 3 \\ -2 & 8 & 7 & 1 \\ 2 & 0 & -1 & 0 \\ 9 & 1 & 6 & -2 \end{bmatrix}$$

The second test is the runtimes observation to analyze the performance of the algorithms. The test strategies are as follows:

- The test uses different sizes of NxN matrices, with N is a power of 2. In this test, the smallest N used is 2 (2x2 matrix) and the largest is 512 (512 x 512 matrix). The computer couldn't handle a bigger number of N for this implementation of algorithms.
- Every size of N is repeated 5 times with different matrices of the same size of N to ensure the accuracy of the test.
- Every iteration of the test uses the same matrix for three algorithms. The matrix is filled with random 0-99 integer.
- Then runtimes are streamed into a data.csv file.

Test Results

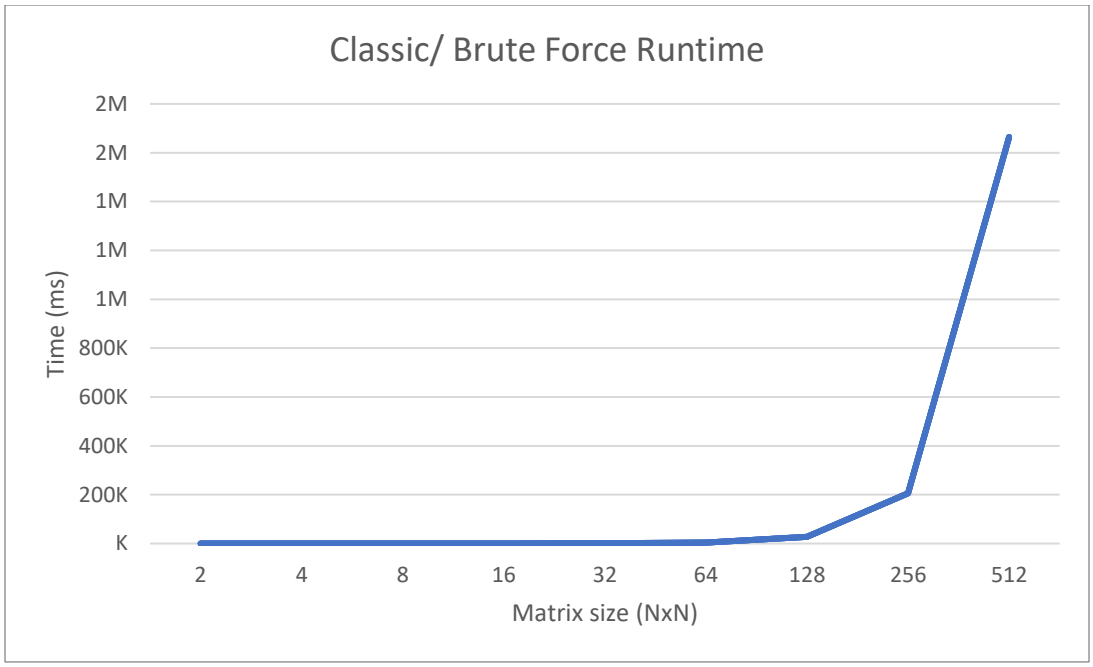
- Sanity Check

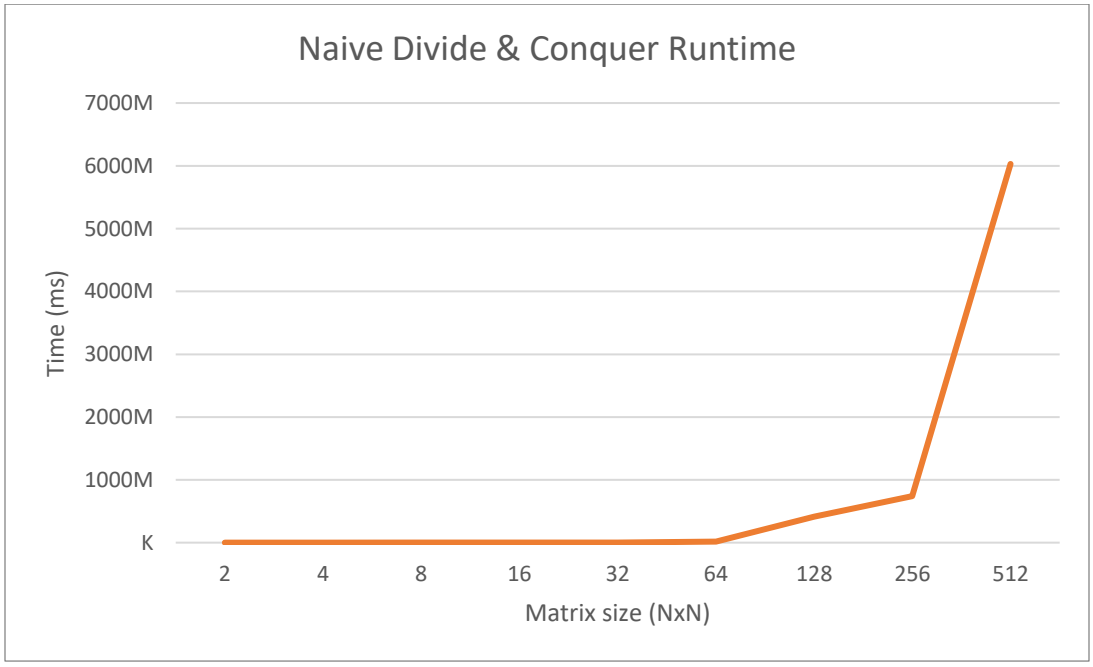
```
--SANITY CHECK--
Classic Multiplication Result:
 52   8  49  -6
  2  59  59  16
 -8   1 -41 -10
 65  15  89  10
Divide and Conquer Result:
 52   8  49  -6
  2  59  59  16
 -8   1 -41 -10
 65  15  89  10
Strassen Algorithm Result:
 52   8  49  -6
  2  59  59  16
 -8   1 -41 -10
 65  15  89  10
```

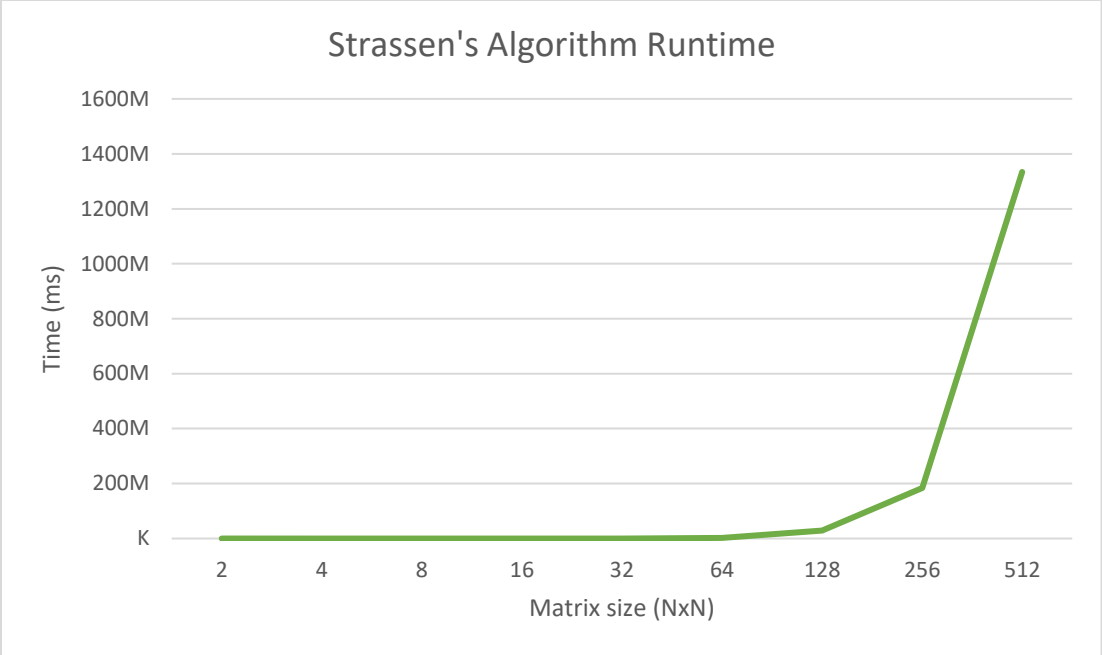
- Performance

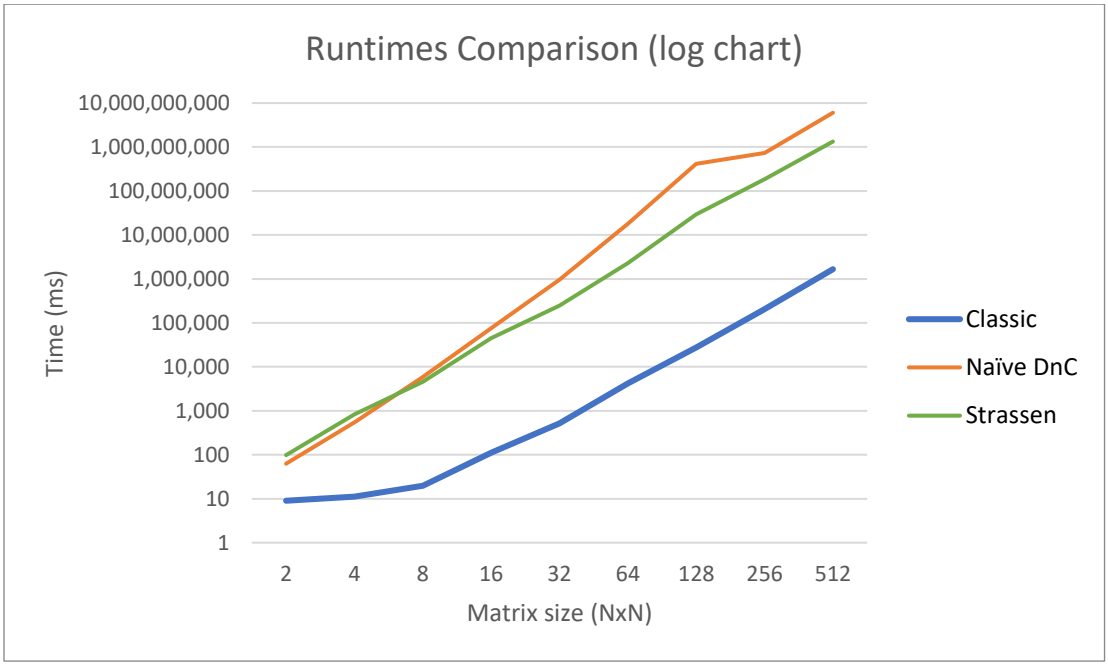
Size	Classic (ms)	Divide and Conquer (ms)	Strassen's Algorithm (ms)
2	9	62.6	98
4	11.2	546.6	825.4
8	19.8	5873.8	4605
16	111.6	74821	44729.2
32	517.4	956359.2	246902
64	4204	18021247.4	2293315.6
128	27604	416423271.2	29209295.8
256	205773.2	738247803.6	184330600.6
512*	1664226	6029920980	1334087725

**Had to be stopped prematurely as one iteration alone took a very long of time.*


Classic/ Brute Force Runtime
2M
2M
1M
1M
1M
800K
600K
400K
200K
K
Time (ms)
2
4
8
16
32
64
128
256
512
Matrix size (NxN)


Naive Divide & Conquer Runtime
7000M
6000M
5000M
4000M
3000M
2000M
1000M
K
Time (ms)
2
4
8
16
32
64
128
256
512
Matrix size (NxN)


Strassen's Algorithm Runtime
1600M
1400M
1200M
1000M
800M
600M
400M
200M
K
Time (ms)
2
4
8
16
32
64
128
256
512
Matrix size (NxN)


Runtimes Comparison (log chart)
10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
Time (ms)
2
4
8
16
32
64
128
256
512
Matrix size (NxN)
Classic
Naïve DnC
Strassen

Data Analysis

From the result of the first test, it can be concluded that the three algorithms are working as intended, since all of them give a same correct result.

The second test gives results as described on the graphs above. As we can see here, with these data sets, the best algorithm in terms of performance is the classic/ brute force algorithm, and the difference in performance compared to the other algorithms is very significant. Theoretically, the Strassen's algorithm should be the best in terms of performance with time complexity $O(n^{\log 7})$, followed by the naïve divide and conquer algorithm $O(n^3)$, and lastly the naïve approach $\Omega(n^3)$. However, this is not the case with this implementation. Why? The reason is because of the overhead in the implemented divide and conquer and Strassen's algorithm. The brute force matrix multiplication algorithm simply does n^3 operations (additions and integer multiplications), meanwhile, in divide and conquer and Strassen, the matrices are divided into quadrants recursively, which means there are additional workloads (copying and creating new objects) that affect the overall performance of the algorithm. Turns out in practical application, the constant workload, which is usually ignored in asymptotic analysis, does matter when the data set used is not big enough. Not to mention, due to how the computer handles recursive functions, the stack memory will be cluttered with the previous recursive calls, which hurts the performance even more.

As we can see from the graph above, naïve divide and conquer runs better than Strassen's algorithm when N is less than 8, even though the difference is not that significant. When N is larger than 8, the Strassen starts to outperform the naïve divide and conquer algorithm. Theoretically, when N is not big enough, the difference between $O(n^3)$ and $O(n^{\log 7})$ should not be that significant. However, as the n grows, the Strassen's algorithm will eventually run much faster than naïve divide and conquer.

The algorithms could be improved by minimizing the number of copying and creating new objects. This could be done by modifying the algorithm to do the computations in-place instead of passing new smaller-size objects recursively. We could also implement the divide and conquer and Strassen algorithms iteratively instead of using recursion to minimize memory clutter. Since brute force is much faster in multiplying small matrices, we could use brute force to solve the base case of the divide and conquer algorithms when the quadrant reaches a certain number of N, specifically when the existence of the overhead simply negates the advantage of divide and conquer over the brute force. Unfortunately, in this implementation of algorithms, the computer couldn't reach that N where both divide and conquer and Strassen's algorithm starts to outperform the brute force algorithm.

In conclusion, divide and conquer and Strassen's algorithm can only be advantageous over the classic way when multiplying large matrices, specifically when the overhead caused by the algorithm is no longer significant to the overall performance.

Strength and Constrains

Strength:

- The code is easy to read.
- Easy to adjust the parameters of the test.
- For the divide and conquer and Strassen's algorithms, since they were implemented recursively, it's easy to follow how the algorithm works.
- In terms of functionality, all 3 matrix multiplications algorithms are working properly.

Constrains:

- Input matrices must be a square matrix, and the size must be a power of 2.
- Plenty of overhead that could be optimized. First, use indexing (in-place) to minimize the process of copying and making new objects. Second, use a bigger number of n for the base case and solve the rest of the quadrant by using brute force.
- Recursive implementation is memory intensive, which naturally hurts the performance of the algorithm.
- Not enough data variations especially for bigger number of N (more than 512) to observe when divide and conquer algorithms start to outperform the brute force algorithm, and the number of repeated tests is not large enough as computer can't handle the workload.

Source Code & Supporting Files

The source code, this pdf file, and data.csv, which contains the raw data of the second test, can be obtained from this GitHub repository:

https://github.com/fidelisprasetyo/MatrixMultiplication