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:

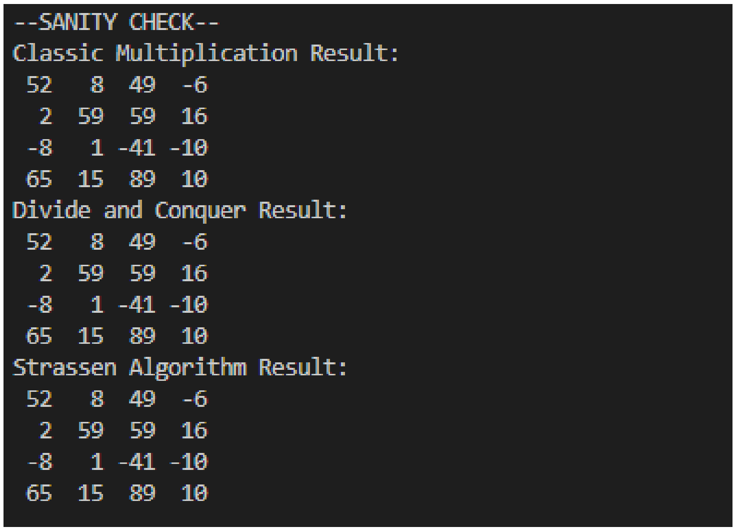
$$\begin{bmatrix} 2 & 0 & -1 & 6 \\ 3 & 7 & 8 & 0 \\ -5 & 1 & 6 & -2 \\ 8 & 0 & 1 & 7 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 6 & 3 \\ -2 & 8 & 7 & 1 \\ 2 & 0 & -1 & 0 \\ 9 & 1 & 6 & -2 \end{bmatrix}$$

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 128 (128 x 128 matrix). The computer couldn’t handle a bigger number of N for this implementation of algorithms.
- Every size of N is repeated 100 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 0-9 integer.
- Then runtimes are streamed into data.csv file.

Test Results

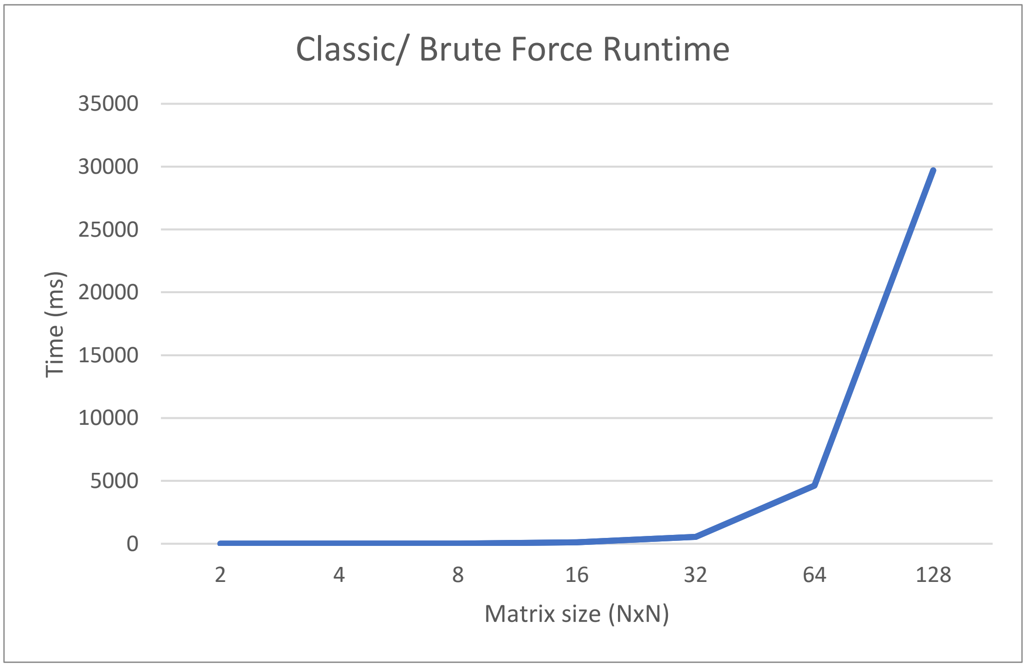
- Sanity Check

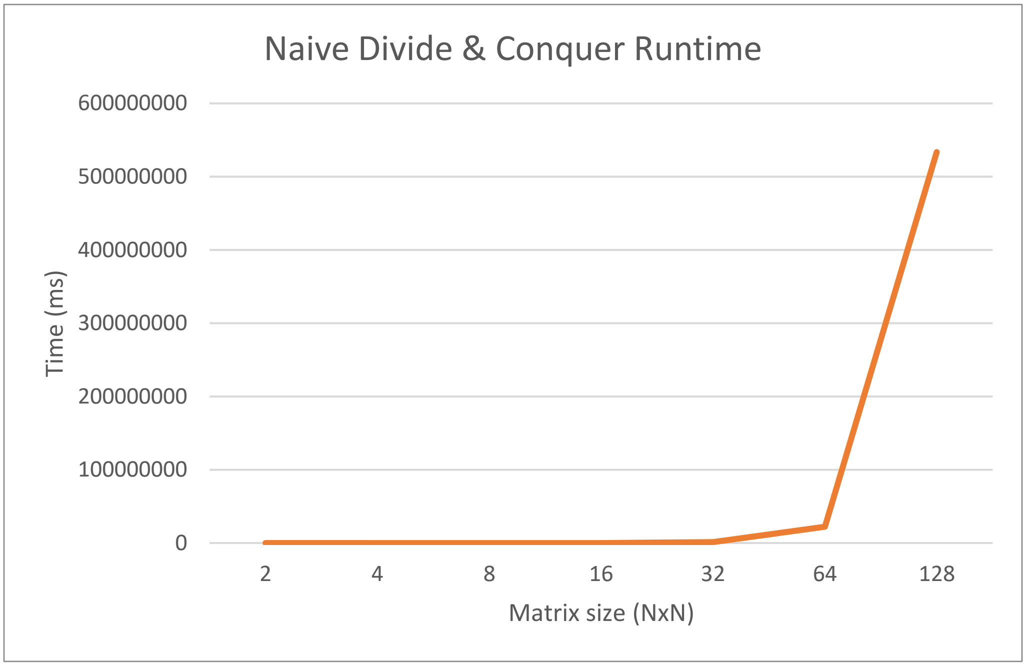


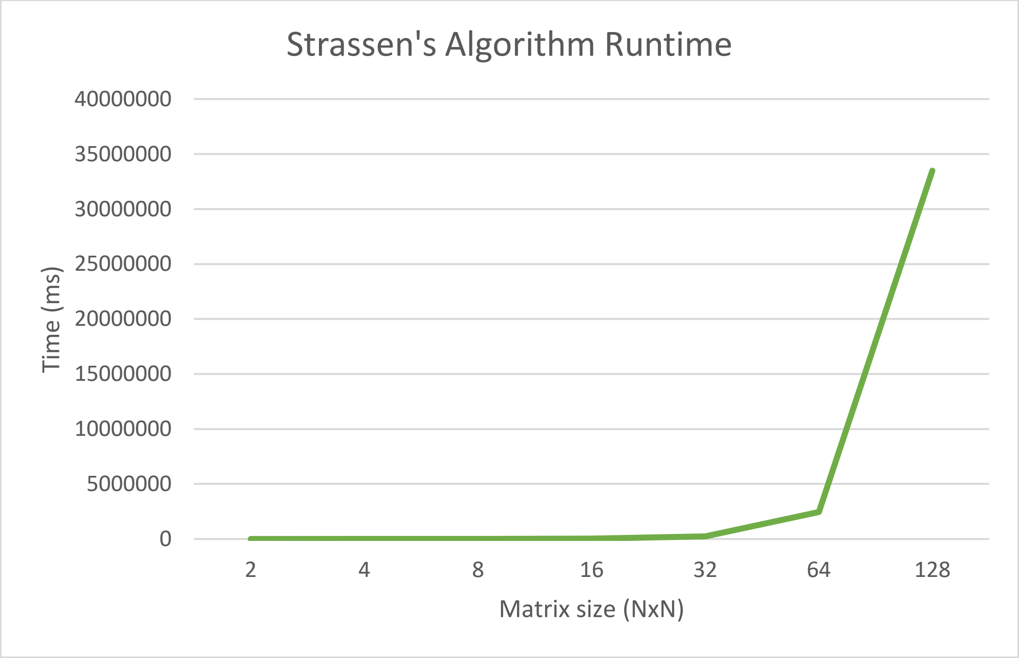
```
--SANITY CHECK--
Classic Multiplication Result:
 52   8  49  -6
  2  59  59  16
 -8   1 -41 -10
 65  15  89  10
Divide and Conquer Result:
 52   8  49  -6
  2  59  59  16
 -8   1 -41 -10
 65  15  89  10
Strassen Algorithm Result:
 52   8  49  -6
  2  59  59  16
 -8   1 -41 -10
 65  15  89  10
```

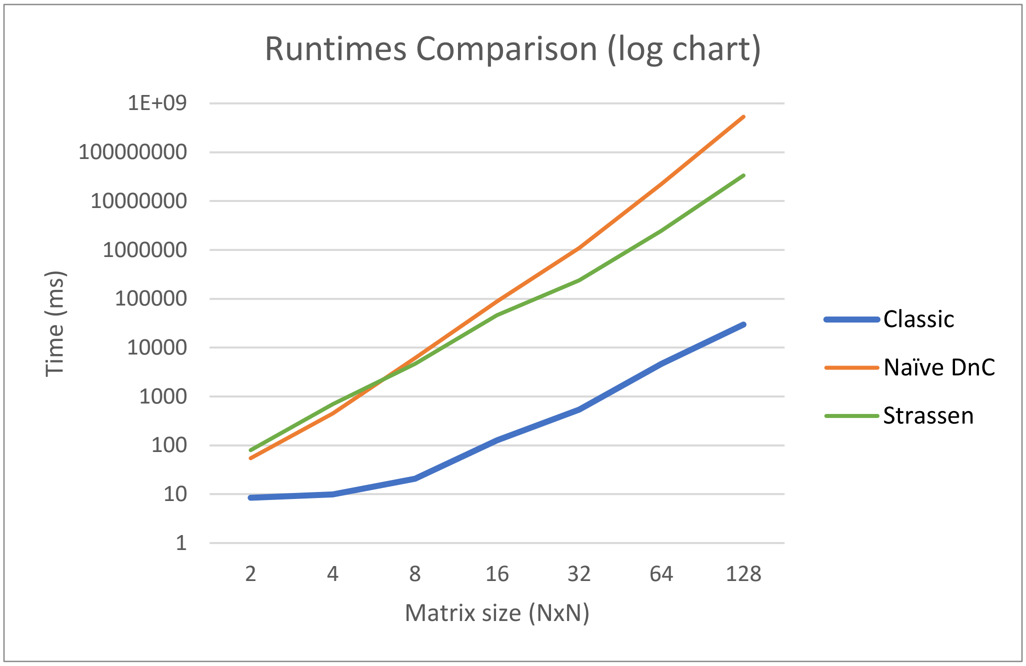
- Performance

SIZE	CLASSIC	DIVIDE AND CONQUER	STRASSEN
2	8.45	54.45	79.61
4	9.83	451.85	699.95
8	20.58	6142	4621.95
16	126.1	88326.84	45899.1
32	537.8	1088489.33	239196.79
64	4632.33	22363676.26	2462646.2
128	29707.88	533537394.4	33488993.88




Naive Divide & Conquer Runtime
600000000
500000000
400000000
300000000
200000000
100000000
0
Time (ms)
2
4
8
16
32
64
128
Matrix size (NxN)


Strassen's Algorithm Runtime
40000000
35000000
30000000
25000000
20000000
15000000
10000000
5000000
0
Time (ms)
2
4
8
16
32
64
128
Matrix size (NxN)



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.

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, which where most of the overload come from. We could also modify the base case of the algorithms. When the n is not large enough, the naïve/

classic approach is preferred, so when we reach that particular number of n, we solve the base case of the recursive loop by using the classic matrix multiplication.

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 doesn't really affect the overall performance.

Strength and Constrains

Strength:

- Code is easy to read.
- Easy to adjust the parameters of the test.
- Plenty of repeated test for every variation of N, thus the data should be accurate.
- For divide and conquer and Strassen's algorithms, since we only pass 2 parameters, 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 by using indexing (in-place) instead of keep making new copies for every recursive case.
- Not enough data variations especially for bigger number of N (more than 128) 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