Algorithms: Design & Analysis
Project Report

Owais Bin Asad (oa05007) Aaron Lucas Soares (as04345)
Bahzad Ahmed Badvi (bb05083)

5th December 2020

1 Introduction

This project deals with the topic of Matrix multiplication, the relevance of this topic stems from the fact that matrix multiplication is a hefty operation in terms of time and is used quite often. This project aims to explore two different algorithms for matrix multiplication and analyze them, both empirically and theoretically.

2 Repository

The algorithm implementations, empirical analysis code, and report .tex file can be found here.

3 Algorithms

- Naive Method
- Strassen's Method

4 Implementation

4.1 Naive Method

```python
import numpy as np

A = np.random.randint(20, size=(4096,4096))
B = np.random.randint(20, size=(4096,4096))
n = 4096

ans = [[0 for y in range(n)] for x in range(n)]

for i in range(n):
    for j in range(n):
        for k in range(n):
            ans[i][j] += A[i][k] * B[k][j]

print(ans)
```

Listing 1: Naive Method

4.2 Strassen's Method

```
import numpy as np

A = np.random.randint(20, size=(4096,4096))
B = np.random.randint(20, size=(4096,4096))

def split(matrix):
    row, col = matrix.shape
    row2, col2 = row//2, col//2
    return (matrix[:row2, :col2], matrix[:row2, col2:], matrix[row2
    :, :col2], matrix[row2:, col2:])


def strassen(x, y, t):

    if len(x) == 1:
        return x * y

    a, b, c, d = split(x)
    e, f, g, h = split(y)

    p1 = strassen(a, f - h, t)
    p2 = strassen(a + b, h, t)
    p3 = strassen(c + d, e, t)
    p4 = strassen(d, g - e, t)
    p5 = strassen(a + d, e + h, t)
    p6 = strassen(b - d, g + h, t)
    p7 = strassen(a - c, e + f, t)

    c11 = p5 + p4 - p2 + p6
    c12 = p1 + p2
    c21 = p3 + p4
    c22 = p1 + p5 - p3 - p7

    c = np.vstack((np.hstack((c11, c12)), np.hstack((c21, c22))))

    return c

print(strassen(A,B,0))
```

Listing 2: Strassen's Method

5 Theoretical Comparison

5.1 Naive Method

The naive implementation employs the simple method of multiplying a row of one matrix with the corresponding column of the other matrix to obtain an

element of the resultant matrix.
Suppose we had two $n \times n$ matrices A and B which were being multiplied and a $n \times n$ matrix R which is our resultant matrix. To obtain the $R_{1,1}$ element, we would multiply the first row of A with the first column of B. Similarly, to obtain $R_{2,3}$, we would need to multiply the second row of A with the third row of B.
To carry out this simple multiplication, the naive method uses three loops each running for n iterations. Therefore, this implementation has a time complexity of $O(n^3)$.

5.2 Strassen's Method

5.2.1 A Divide and Conquer Approach

Suppose we have 2 matrices with dimensions n by n. The algorithm will recursively divide each of these matrices into 4 smaller sub-matrices, each having dimensions n/2 by n/2. It then calculates 8 multiplications which take O(n) time and 4 additions which will take $O(n^2)$ time.

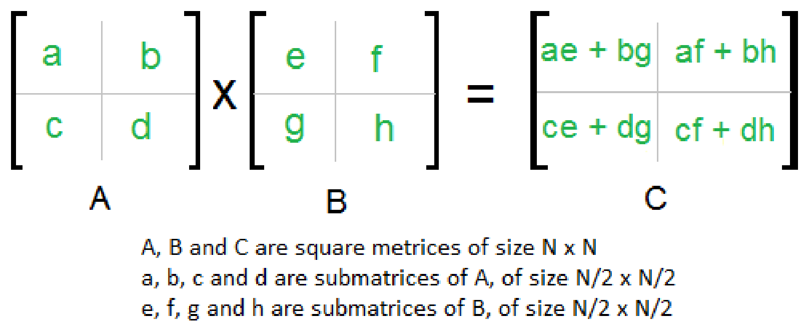

a
b
c
d
A
X
e
f
g
h
B
=
ae + bg
af + bh
ce + dg
cf + dh
C
A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

Figure 1: Divide and Conquer Approach

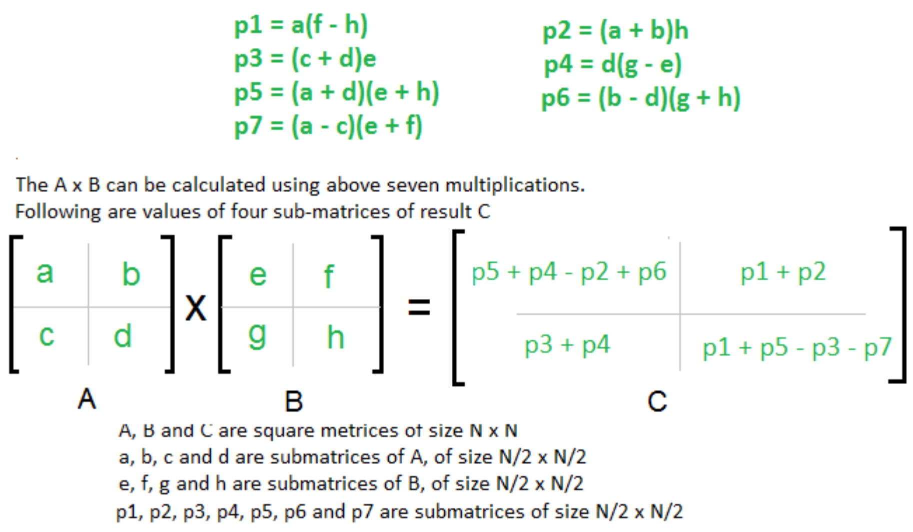
Source: www.geeksforgeeks.com

$$T(n) = 8T(\frac{n}{2}) + O(n^2)$$

Using the Master's Theorem, we can calculate the total time complexity which is $O(n^3)$. However, this is the same as the naive implementation.

5.2.2 What does Strassen's Algorithm do better?

Strassen's algorithm applies a similar approach. It divides the matrices into smaller sub-matrices, but instead of 8 recursive calls, the algorithm makes 7 calls.



p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Figure 2: Strassen's Method

Source: www.geeksforgeeks.com

$$T(n) = 7T(\frac{n}{2}) + O(n^2)$$

We can use the Master's Theorem to compute the time complexity of Strassen's algorithm. We have 7 recursive calls and all the additions and subtractions take $O(n^2)$ time. Therefore the total time complexity turns out to be $O(n^{log7})$.

6 Empirical Comparison

6.1 System Specifications

Processor: Intel Core i5 8250U
RAM: 12 GB
Power Saving: Max Performance Mode
Language: Python 3.8.1

6.2 Data

		Dimensions $(n \times n)$			
		1024	**512**	**256**	**128**
Avg. Run-time (s)	**Naive Method**	2207	286	32	5
	Strassen's Method	3211	496	64	8.7

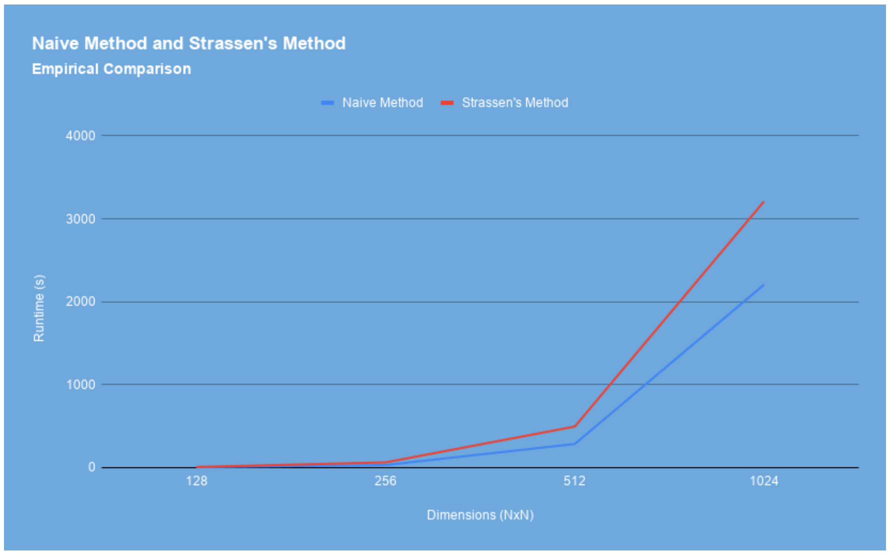


Figure 3: Empirical Comparison between Naive Method & Strassen's Method

6.3 Why Strassen's Method took longer?

As can be noted from the aforementioned table Strassen's empirical analysis shows that it takes longer than the naive method regardless of the size of the matrix n. This is the result of communication bottlenecks that happen in real-time. These bottle necks arise due to the fact that a single call to the function implies that recursively the function will be called again and these recursive calls are waiting for each subsequent calls to reach the base case causing a lag.

7 Feedback

- C++ should've been chosen as a language since it is faster than Python

8 Conclusion

Matrix multiplication is a very important aspect of Computer Science since it allows faster and more efficient mathematical modelling of real-world phenomena. There are other algorithms that do exist for this purpose but are beyond the goal of this project.