ISSN : 0976-8491 (Online) | ISSN : 2229-4333 (Print)

Comparative Study of Strassen's Matrix Multiplication Algorithm

[1]Juby Mathew, [2]Dr. R Vijaya Kumar
[1]Dept. of CSE, AmalJyothi College of Engg. Kanjirapally, India
[2]School of Computer Science, Mahatma Gandhi University, Kottayam, Kerala, India

Abstract
The main focus of this paper is to compare the execution time complexity and space complexity between Strassen's algorithm and the conventional algorithm for matrix multiplication. The aim is to design a program, which generates two matrices with various dimensions, and multiplies the two matrices using both the Strassen's algorithm and the conventional algorithm. The execution time of each algorithm is recorded to evaluate the performance of each algorithm. The programming language used this project is Java. Some of the main achievements in this project are, successfully divide matrices into blocks, the Strassen's algorithm was applied to each blocks recursively, and the level of recursion was controlled. The overall finding is that the Strassen's algorithm is more efficient than conventional algorithm on large size of matrices. However, in scientific computing, memory has to be considered. The results show that Strassen's algorithm needs more memory allocations than the conventional algorithm, due to the fact in design that more arrays need to be created.

Keywords
Matrix Multiplication, Parallel Algorithm, Strassen's Algorithm

I. Introduction

The multiplication of two matrices is one of the most important operations in linear algebra. The operation plays an important role in scientific computing. The evidence can be found in computer graphics, coordinate transformations such as scaling, rotation, and translation for robotics, the speed for solving those problems are only depends on the execution times of the algorithm. Since different algorithms give various differences in performance, finding a good algorithm seems to be valuable. The investigation of speed up matrix multiplication has become the main focus in scientific computation. The main objective was to design a program, which generates two matrices with various dimensions, and then multiplies the two matrices using the Strassen's algorithm and the conventional algorithm. The performance of both algorithms was compared in terms of their time complexity and space complexity.

A. Various use of Matrix Multiplication

Matrix multiplication is multiplying two matrices together. In linear system of equations, a standard form is given as:

$x - 2y - z = 2$
$2x - y = 4$
$-x + y - 2z = -4$

A matrix derived from a linear system of equations, each in standard form, is called the augmented matrix of the system. The augmented matrix for a linear system is a matrix with the system's variables and equal signs eliminated [4], which is given:

$$\begin{bmatrix} 1 & -2 & -1 & 2 \\ 2 & -1 & 0 & 4 \\ -1 & 1 & -2 & -4 \end{bmatrix}$$

In linear algebra, matrices allow arbitrary linear transformations to be represented in a consistent format, suitable for scientific computation [5]. Computer graphics carries the same aspect in coordinate transformations, for many years computer scientists and mathematicians have been working together to find more efficient algorithms for the benefit in both areas.

B. Conventional Matrix Multiplication

The conventional matrix multiplication is the most important way to multiply matrices, and it is the fundamental basic of other algorithms. The definition of matrix multiplication is only valid if the width of first matrix equals the height of the second matrix. For a matrix A with dimensions m by n, and B with dimensions n by p, the result of A multiply B is an m by p matrix, where the elements of AB are given by

$$(AB)_{i,j} = \sum_{r=1}^{n} A_{i,r} B_{r,j}$$

Matrix product is not commutative. For example:

$$\underset{3\times4 \text{ matrix}}{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 1 & 2 & 3 & 4 \end{bmatrix}} \underset{4\times5 \text{ matrix}}{\begin{bmatrix} \cdot & \cdot & \cdot & a & \cdot \\ \cdot & \cdot & \cdot & b & \cdot \\ \cdot & \cdot & \cdot & c & \cdot \\ \cdot & \cdot & \cdot & d & \cdot \end{bmatrix}} = \underset{3\times5 \text{ matrix}}{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & x_{3,4} & \cdot \end{bmatrix}}$$

The element $X_{3,4}$ of the above matrix product is computed as follows:

$X_{3,4} = (1,2,3,4) \bullet (a,b,c,d) = 1 * a + 2 * b + 3 * c + 4 * d.$

The first subscript in matrix notation denotes the row and the second the column; this order is used both in indexing and in giving the dimensions. The element at the intersection of row i and column j of the product matrix is the dot product (or scalar product) of row i of the first matrix and column j of the second matrix. This explains why the width and the height of the matrices being multiplied must match: otherwise the dot product is not defined.

The fig. below illustrates the product of two matrices A and B, showing how each intersection in the product matrix corresponds to a row of A and a column of B. The size of the output matrix is always the largest possible, i.e. for each row of A and for each column of B there are always corresponding intersections in the product matrix. The product matrix AB consists of all combinations of dot products of rows of A and columns of B [6].

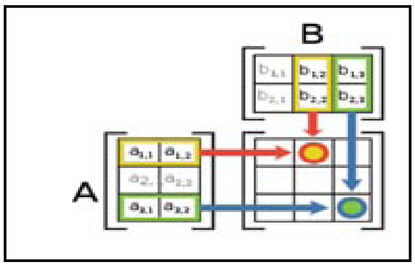

Fig. 1:

The values at the intersections marked with circles are:
$X_{12} = (a_{11}, a_{12}) \bullet (b_{12}, b_{22}) = a_{11}b_{12} + a_{12}b_{22}$
$X_{33} = (a_{31}, a_{32}) \bullet (b_{13}, b_{23}) = a_{31}b_{13} + a_{32}b_{23}$
For n × n conventional matrix multiplication takes O(n3) operations.

C. Block Matrix Multiplication

A block matrix is a partition of a matrix into smaller matrices, and each block itself is treated as a new element of the original matrix. For example, the matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

can be split into four 2 by 2 block matrices

$$A_{11} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad A_{12} = \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix} \quad A_{21} = \begin{bmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \quad A_{22} = \begin{bmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{bmatrix}$$

Then matrix

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

The product of block matrices can be computed using the principle of conventional matrix multiplication. The computation follows the same rule, which means the width of first partitioned matrix equals the height of the second partitioned matrix. For example, for an m by n matrix A with partitioned dimensions a by b, and a n by p matrix B with partitioned dimensions b by c, the product of two partitioned matrices C can be calculated by:

$$C_{i,j} = \sum_{r=1}^{n} A_{i,r} B_{r,j}$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \bullet \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

D. Strassen's Algorithm

"The exponent for matrix multiplication has proved so elusive that it has been given a name: ω. formally, ω is the smallest number such that O (nω + ε) multiplications suffice for all ε > 0.Because all n2 entries must be part of any computation, ω is clear at least 2 The road to the current best upper bound on ω was built on clever ideas and increasingly complex combinatorial techniques. Many researchers long believed that the standard, O(n3) algorithm was the best possible; indeed, one research paper even presented a proof that the standard algorithm is optimal under seemingly reasonable assumptions. Then, in 1969, Volker Strassen's stunned the research world with his O(n2.81) algorithm for multiplying matrices. Still sometimes used in practice, Strassen's algorithm is reminiscent of a shortcut, first observed by Gauss, for multiplying complex numbers. Though finding the product (a + bi)(c + di) = ac – bd + (bc + ad) i seems to require four multiplications, Gauss observed that it can actually be done with three—ac, bd, and (a + b)(c + d)—because bc + ad =(a + b)(c + d) – ac – bd.
In a similar way, Strassen's algorithm reduces the number of multiplications required for computing the product of two 2 × 2 matrices A and B from eight to seven, by expressing the entries of AB as linear combinations of products of linear combinations of the entries of A and B. This trick, applied to four blocks of 2n×2n matrices, reduces the problem to seven multiplications of 2n–1×2n–1 matrices. Recursive application gives an algorithm that runs in O (nlog27) = O (n2.81) time [7].
Strassen's algorithm carries the same aspect of Block matrix multiplication, and using the divide and conquers technique to perform faster. However, this is only theoretical, in practice the divide and conquer approach has two potential drawbacks. First, if the division proceeds to the level of single matrix elements, the recursion overhead (measured, for instance, by the recursion depth and additional temporary storage) becomes significant, this means a large additional temporary storage is required. Therefore it reduces the potential advantage in performance. This overhead is generally limited by stopping the recursion early and performing a conventional matrix multiplication on sub matrices at the crossover point. Second, the division step must efficiently handle odd-sized matrices. This can be solved by one of several schemes: by embedding the matrix inside a larger one (called static padding), by decomposing into sub matrices that overlap by a single row or column (called dynamic overlap), or by performing special case computation for the boundary cases (called dynamic peeling) [9].
Strassen's algorithm works more efficient with relatively large matrices, such as matrices with dimensions of 2n by 2n where n ≥ 2. Partition the two input matrices A and B into four equal size blocks as mentioned in block matrix multiplication, and then the algorithm carries out as following

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \quad C = \begin{bmatrix} C_{11} & C_{12} \\ C_{12} & C_{22} \end{bmatrix}$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \bullet \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Strassen's algorithm is formulated by 7 intermediate matrices P_i.
At the end of computation 4 block matrices are merged to construct the result matrix.
$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$ $C_{11} = P_1 + P_4 - P_5 + P_7$
$P_2 = (A_{21} + A_{22})B_{11}$ $C_{12} = P_3 + P_5$
$P_3 = A_{11} (B_{12} - B_{22})$ $C_{21} = P_2 + P_4$
$P_4 = A_{22} (B_{21} - B_{11})$ $C_{22} = P_1 - P_2 + P_3 + P_6$
$P_5 = (A_{11} + A_{12}) B_{22}$
$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$
$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$
In practice, recursion is performed down to a certain level. According to Steve Skiena [10], that "Strassen's algorithm is unlikely to beat the straightforward algorithm for n ≤ 100, and it is less numerically stable to boot. Other studies have been more encouraging, claiming that the crossover point is as low as n ≤32. Earlier experiments put the crossover point where Strassen's algorithm beats the cubic algorithm at about n = 128".
The important part of implementation of Strassen's algorithm is handling arbitrary matrices. According to Strassen's theory it can be observed that only matrices with even dimensions works with Strassen's algorithm, thus it cannot deal with non even size matrices due to the structure of partitions.

E. Programming Language and Platform

The object-orientated programming language was used to develop the program. It was chosen because of the “the concept of a data class makes it possible to define subclasses of data objects that share some or all of the main class characteristics,called inheritance, this property of OOP forces a more thorough data analysis, reduces development time, and ensures more accurate coding. Since a class defines only the data it needs to be concerned with, when an instance of that class (an object) is run, the code will not be able to accidentally access other program data. This characteristic of data hiding provides greater system security and avoids unintended data corruption

F. Construction of Matrix

The matrix class defines the construction of matrices. The constructor creates a two-dimensional array object of matrix, and it should enable instance creation of matrix with arbitrary size. In order to simplify the problem but keep the concept the program should be able to allow the user to input the size of n × n matrix through command line arguments. For purpose of evaluating large matrices, a random generator should generate the value of elements in two input matrices.

G. Methods Specification

Matrix multiplication is the main class. The main objective of this class is to perform Strassen’s algorithm and conventional algorithm.

A random generator should be created at the beginning to generate an n × n matrix with various dimensions. This method requires two input parameters, which are a two-dimension array object to hold the matrix that generated by this method, and the other is the value of input size of the matrix from command line arguments to control the iteration of the increment. The matrix should be printed out after generates The addition and subtraction methods take two input matrix objects, and the value of input size of the matrix from command line arguments to control the iteration of the increment, and then perform the addition or subtraction of those two input matrices. The result of new matrix would store into a new two dimensional array matrix object.

The conventional matrix multiplication would have 3 input parameters; two n × n matrices with various dimensions should be passed into this method for conventional matrix multiplication, and the value of input size of the matrix from command line arguments should also be passed into this method in order to control the iteration of the increment. At the end the result of conventional matrix multiplication is stored in a new matrix object.

According to Strassen’s algorithm input matrices should be split into sub-matrices, and perform Strassen’s algorithm recursively to those sub-matrices

The matrix A, B

$[a_{11}\ a_{12}\ a_{13}\ a_{14}]\ \ [b_{11}\ b_{12}\ b_{13}\ b_{14}]$

$$A = \begin{vmatrix} a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{vmatrix} \qquad B = \begin{vmatrix} b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{vmatrix}$$

can be split into 4 2 by 2 block matrices

$$A_{11}=\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad A_{12}=\begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix} \quad A_{21}=\begin{bmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix} \quad A_{22}=\begin{bmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{bmatrix}$$

$$B_{11}=\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \quad B_{12}=\begin{bmatrix} b_{13} & b_{14} \\ b_{23} & b_{24} \end{bmatrix} \quad B_{21}=\begin{bmatrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{bmatrix} \quad B_{22}=\begin{bmatrix} b_{33} & b_{34} \\ b_{43} & b_{44} \end{bmatrix}$$

Then matrix product of A•B is matrix C

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \bullet \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

In this case the 16 × 16 matrices A and B are divided into four 4 × 4 matrices respectively, and the result matrix C is merged from its sub-matrices. The divide method should take four matrix objects to store each partition sub-matrices, and one matrix object for splitting. There should not be a return call as each sub-matrix is stored for later use in Strassen’s algorithm method. The merge method should be similar to divide method, instead of having a matrix object for splitting, it creates a matrix object at run time for hold the value after merge is finish. It should have a return call of merged matrix for later use Strassen’s algorithm method with one level is petty straightforward. The method should take two generated input matrices as input parameters, and using the Strassen’s algorithm structure to calculate the intermediate matrices, then the result matrix formulates from the calculation between 7 intermediate matrices.

The recursive Strassen’s algorithm could be implemented by splitting the input matrix A and B with 4 equal dimension block matrices. Intermediate matrices are calculated by using Strassen’s algorithm structure and recursively applied to the

Sub-matrices, then the result matrix formulates from the calculation between 7 intermediate matrices. The merge method call should be used to construct the result matrix. The following diagrams illustrate the recursion

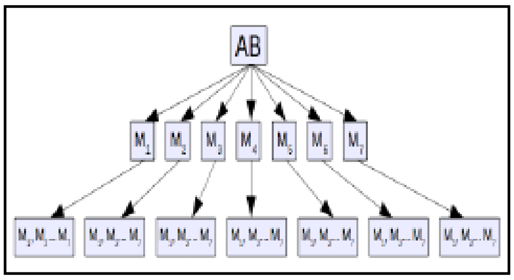

Fig. 2:

It is necessary to have a method to check the input size of the matrix is even number. This method would allow user to use different techniques (static padding, dynamic padding, dynamic peeling, and dynamic overlap) for odd size matrices

H. Construction of Matrix

The construction of the matrix is processed by a constructor that constructs a two-dimensional array object. The size of the matrix is fixed in order to simplify the complexity of the code and to run early testing. The matrix class has a universal commensurability, which means it can be created anywhere in the main class where creation of matrix is needed.

```
Class matrix
{
Public int[][] m = new int[100][100];
}//class Create matrix
```

The random generator was created at early stage of the implementation step. This generator generates a n × n matrix. The following code indicates the implementation of this method.

```
public matrix GetMatrix(matrix X,int n)
{
int i,j;
X = new matrix();
Random random = new Random();
for(i=0;i<n;i++)
for(j=0;j<n;j++)
X.m[i][j] = (int)(Math.random()*10);
for(i=0;i<n;i++)
{
 for(j=0;j<n;j++)
 {
  System.out.print("[" + X.m[i][j] + "]" + "");
 }
System.out.println();
}//for loop printing GetMatrix
return X;
}//GetMatrix
```

In the code, matrix X is the first parameter, which is a matrix object for holding the values of the matrix generated. The second parameter is int n, which represents the size of the matrix to be created.

I. Addition and Subtraction

The implementation process on addition and subtraction are straightforward. The values in each matrix are either added up or subtracted. A new matrix object is therefore created at the run time to store the values after computation.

```
public matrix MatrixPlus(matrix f,matrix g,int n)
{
int i,j;
matrix h = new matrix();
for(i=0;i<n;i++)
for(j=0;j<n;j++)
h.m[i][j]=f.m[i][j]+g.m[i][j];
return h;
}//MatrixPlus
public matrix MatrixMinus(matrix f,matrix g,int n)
{
int i,j;
matrix h = new matrix();
for(i=0;i<n;i++)
for(j=0;j<n;j++)
h.m[i][j]=f.m[i][j]-g.m[i][j];
return h;
}//MatrixMinus
```

The above code explains that the method for addition and subtraction of each value in the two-dimensional array matrix object. Two for loop has been used to control the iteration in both rows and columns. After execution of the result is stored in the new created matrix object and returned

J. Implementation of Conventional Matrix Multiplication

Conventional matrix multiplication works similar as matrix addition and subtraction. However, the conventional matrix multiplication requires three for loop iterations. Moreover, the most inner one is the row by column matrix multiplication. The definition of n

$$(AB)_{i,j} = \sum_{r=1}^{n} A_{i,r} B_{r,j}$$

The following code illustrates the implementation based on such definition.

```
for (i=0;i<n;i++)
for (j=0;j<n;j++)
{
for (k=0,t=0;k<n;k++)
t+=A.m[i][k]*B.m[k][j];
C.m[i][j]=t;
}
```

K. Matrix Partition and Merge

The divide and conquer algorithm requires partition on the matrix during recursion. At the end of recursion, all results have to merge together to form the finial outcome. As a consequence, the divide method needs 4 input parameters to store the sub matrices after a partition into block matrices. There is also an input parameter for the input matrix, which needs to be partitioned. In addition, there is no requirement for a return function in the partition phase. The divide method is implemented by following code public void Divide(matrix d,matrix d11,matrix d12,matrix d21,matrix d22)

```
{
int i,j;
int n = a11.length;
for(i=0;i<n;i++)
for(j=0;j<n;j++)
{
d11.m[i][j]=d.m[i][j];
d12.m[i][j]=d.m[i][j+n];
d21.m[i][j]=d.m[i+n][j];
d22.m[i][j]=d.m[i+n][j+n];
}//for
}//Divide
```

The merge method is implemented in a similar way as the following:

```
public matrix Merge(matrix a11,matrix a12,matrix a21,matrix a22)
{
int i,j;
matrix a = new matrix();
int n = a.length;
for(i=0;i<n;i++)
for(j=0;j<n;j++)
{
a.m[i][j]=a11.m[i][j];
a.m[i][j+n]=a12.m[i][j];
a.m[i+n][j]=a21.m[i][j];
a.m[i+n][j+n]=a22.m[i][j];
}
return a;
}//Merge
```

Rather than splitting the matrix, merge method allows to merge the partitioned block matrices into one matrix to reach the finial result. Hence, the result matrix is able to return at the end.

L. Recursive Strassen's Algorithm

The implementation of the recursive Strassen's algorithm constitutes the most important part of the whole program. It is processed through the subsequent steps. The recursive Strassen's algorithm begins with dividing the input matrix A and B by 4 equal dimension block matrices. Intermediate matrices are calculated by using Strassen's algorithm structure and are recursively applied to the sub-matrices. It then followed by a result matrix being obtained from the calculation between 7 intermediate matrices. The merge method is called to merge the
partitioned block matrices into the result matrix and is stored in matrix C. At the end of the method, the result matrix C gets returned. The implementation code is shown as following:

```
public matrix MatrixMultiply(matrix a,matrix b,)
{
. . .
Divide(a,a11,a12,a21,a22);
Divide(b,b11,b12,b21,b22);
m1 = MatrixMultiply(Add(a11,a22),Add(b11,b22));
m2 = MatrixMultiply(Add(a21,a22),b11);
m3 = MatrixMultiply(a11,Subtract(b12,b22));
m4 = MatrixMultiply(a22,Subtract(b21,b11));
m5 = MatrixMultiply(Add(a11,a12),b22);
m6 = MatrixMultiply(Subtract(a21,a11),Add(b11,b12));
m7 = MatrixMultiply(Subtract(a12,a22),Add(b21,b22));
c11=Add(Subtract(Add(m1,m4),m5),m7);
c12=Add(m3,m5);
c21=Add(m2,m4);
c22=Add(Subtract(Add(m1,m3),m2),m6);
c=Merge(c,c11,c12,c21,c22); //Join matrix C back
return c;
}
```

M. Solving Arbitrary Matrix

It is crucial to recognize if the size of input matrix shows a power of 2. According to the structure of Strassen's algorithm,the recursion can only be applied to the matrices with size n2×n2. The method for checking if the input matrix is an n2 × n2 matrix is given by:

```
public boolean isPowerOfTwo(int n)
{
while (((n % 2) == 0) && n > 1)
// While x is even and > 1
n /= 2;
return (n == 1);
}//isPowerOfTwo
```

This method takes the size of input matrix n and keeps dividing n by 2. The common rule is that if the remainder results in 0, it is safe to say that n = k2. It is constrained that n has to be greater than 1.

N. Recording Execution Time

The objective of the program was to compare the execution time complexity for both Strassen's algorithm and conventional matrix multiplication. Therefore, it is essential to implement a function for obtaining the execution time of both algorithms. It is helpful that Java has a built in method to obtain the current run time of the execution. Implementation is given by following code:

```
final long startStrassen = System.currentTimeMillis ();
C=instance.MatrixMultiply(A,B);
final long finishStrassen = System.currentTimeMillis ();
strassenTime = finishStrassen - startStrassen;
final long startConventional= System.currentTimeMillis ();
C=instance.ConventionalMatrixMultiply (A, B);
final long finishConventional= System.currentTimeMillis ();
ConventionalTime = finishConventional – startConventional
```

O. Crossover Point and Creation of Sub-matrices

In practice, the execution time complexity is difficult to match the theoretical complexity in the implementation process. Complexity has two common measures. First one is the time complexity, which refers to the number of counting operations.
The second one is the space complexity, which refers to the requirement of the number of storage space. The time complexity is difficult to handle in terms of the structure of the algorithm. For this reason, an improvement in the run time complexity of the algorithm in implementation means an improvement in the space complexity for memory efficiency

P. Result Evaluation

In result evaluation, some tested data needs to be obtained and investigated in terms of execution time (time complexity and space complexity). Table below is an execution time comparison between conventional matrix multiplication and Strassen's algorithm. The input matrices for testing have dimension of n2 × n2. In order to the find the crossover point, some matrices were tested more than once with different recursion levels. Due to the reason that the input matrix is generated randomly, the result data is different every time during execution. This is true even for the matrix with same dimension. As a result, the average result is recorded to be the general case. The execution time is in ms.
It is necessary to increase the memory heap size for more memory allocation, in which testing data is some large size matrix. This was done by using command line options.

```
java –Xms <initial heap size>
java –Xmx <maximum heap size>
```

Table 1:

Matrix Dimension	Recursion Level	Conventional Matrix Multiplication	Strassen's Algorithm
16	2	1.0ms	6.0ms
32	2	1.0ms	61.0ms
64	2	2.0ms	196.0ms
128	2	9.0ms	983.0ms
512	2	1416.0 ms	28018.0ms
512	8	1593.0ms	2490.0ms
512	16	1569.0ms	1241.0ms
512	64	2053.0ms	679.0ms
512	128	1550.0ms	741.0ms
512	256	1540ms	1084.0ms
1024	32	15182.0ms	6222.0ms
1024	64	15558.0ms	4506.0ms
1024	128	15251.0ms	5860.0ms
1024	512	17901ms	11984.0ms

Results in Table 1, indicate when the dimension of matrices are at 24, 25, 26, 27 and 28, conventional matrix multiplication beats Strassen's algorithm, where enables the recursion to proceed all the way down to one level. The differences in execution time are significant when the dimension of matrix is at 28. It is simple to observe the crossover points for both 512 ×512 and 1024 × 1024 matrices are at recursion level 64.

Q. Interesting Thoughts and Changes

The complex testing between conventional matrix multiplication and Strassen's algorithm was successful. However, there are still some changes can be made for further investigation and evaluation. Firstly, from the testing, it can be observed that the Strassen's algorithm was not efficient when applied to small size matrix. In that, preceding the recursion all the way down to one level would result in significant recursion overhead.
Nevertheless, it would still be interesting to know at which point Strassen's algorithm can beat conventional matrix multiplication. However, this was not carried out in the testing due to the huge requirement of memory storage and long and ongoing execution time. The second change could be to calculate the space complexity for Strassen's algorithm as the depth of the recursion goes down.

References

[1] Stetson edu., [Online] Available: http://www2.stetson.edu/~efriedma/periodictable/html/C.html>
[2] Sara Robinson,"Toward an Optimal Algorithm for Matrix Multiplication", From SIAM News, Vol. 38, No. 9, November 2005. [Online] Available: http://www.siam.org/pdf/news/174.pdf>
[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein,"Introduction to Algorithms", Second Edition. Chapter 28 pp. 769.
[4] Marsha Austin, (2007),"Matrix solution of Linear Systems", [Online] Available: http://www.occc.edu/maustin/matrix_solutions/matrix solution of linear systems.htm>
[5] Wikipedia, (2007),"Transformation matrix", [Online] Available: http://en.wikipedia.org/wiki/Transformation_matrix>
[6] Wikipedia,"Ordinary matrix product", [Online] Available: http://en.wikipedia.org/wiki/Matrix_multiplication - cite_note-0>
[7] Sara Robinson,"Toward an Optimal Algorithm for Matrix Multiplication", From SIAM News, Vol. 38, No. 9, November 2005. [Online] Available: http://www.siam.org/pdf/news/174.pdf>
[8] Lehigh University fall, (2008),"Fast Matrix Multiplication and Inversion", [Online] Available: http://www.lehigh.edu/~gi02/m242/08linstras.pdf>
[9] Mithuna Thottethodi, Siddhartha Chatterjee, Alvin R. Lebeck,"Tuning Strassen's Matrix Multiplication for Memory Efficiency", [Online] Available: http://www.cs.duke.edu/~alvy/papers/sc98/index.htm>
[10] Skiena, Steven S.,"The Algorithm Design Manual second edition", chapter 13, pp 402. Berlin, New York: Springer-Verlag

Strassen Matrix Multiplication Strassen's algorithm will need 7 multiplications instead of 8.Strassen's algorithm is faster for matrices with dimension from 32 to 128,and the principle is to apply this algorithm recursively to the block of matrices	
Time Complexity The standard matrix multiplication takes approximately 2N3 where N=2n arithmetic operations is O(N3).When using the strassen algorithm the operation is approximately O(N2.807)	Space Complexity Compare to the standard matrix multiplication,Strassen's algorithm will require more spaces as it has intermediate matrices

Juby Mathew pursued his MCA from Periyar University Salem and an MPhil in Computer Science from Madurai Kamaraj University. Currently he is pursuing his PhD in parallel algorithms from school of Computer science, Mahatma Gandhi University Kottayam. so far he has published three international journals and presented papers in more than ten National and International conferences. He has more than eight years of teaching and corporate experience and continuing his profession in teaching. Now he is working as an Assistant Professor in Computer Science Department of AmalJyothi College of Engineering Kanjirapally, Kerala, India.