Chapter - 3

Complexity of Algorithms
for Solution of Simultaneous
Linear Equations and Matrix
Multiplication
CHAPTER 3

Complexity of Algorithms for Solution of simultaneous linear equations and matrix multiplications

3.1 Introduction

Solving systems of linear equations is at the heart of scientific computing. Many problems in science and engineering give rise to linear equation systems, such as, forecasting, estimation, approximating non-linear problems in numerical analysis and integer factorization. The numerical solution methods for linear systems of equations, $Ax = B$, are broadly classified into two categories: direct methods, such as Gaussian elimination, LU factorization etc.; and iterative methods. Direct methods obtain the exact solution in finitely many operations and are often preferred to iterative methods in real applications because of their robustness and predictable behavior. However, as the size of the systems to be solved increases, they often become almost impractical due to the phenomenon known as fill-in. The fill-in of a sparse matrix is a result of those entries which change from an initial value of zero to a nonzero value during the factorization phase, e.g. when a row of a sparse matrix is subtracted from another row, some of the zero entries in the latter row may become nonzero. Such modifications to the matrix mean that the data structure employed to store the sparse matrix must be updated during the execution of the algorithm. Iterative methods, on the other hand, do not modify matrix $A$; rather, they involve the matrix only in the context of matrix-vector product (MVP) operations. The term “iterative methods” refers to a wide range of techniques that use successive approximations to obtain more accurate solutions to a linear system at each step Barrett et al. (1994). Beginning with a given approximate solution, these methods modify the components of the approximation, until convergence is achieved. They do not guarantee a solution for all systems of equations. However, when they do yield a solution, they are usually less expensive than direct methods. They can be further classified into stationary methods like Jacobi and Gauss-Seidel (GS), and non-stationary methods Saad and Vorst. (2000) present a survey of the iterative methods; Stewart (1994) describes iterative methods in the context of solving Markov chains.
Ever since the dawn of the computer age, researchers have been trying to find an optimal way of multiplying matrices, a fundamental operation that is a bottleneck for many important algorithms. Faster matrix multiplication would give more efficient algorithms for many standard linear algebra problems, such as inverting matrices, solving systems of linear equations, and finding determinants. Even some basic graph algorithms run only as fast as matrix multiplication. The standard method for multiplying \( n \times n \) matrices requires \( O(n^3) \) multiplications.

Winograd (1968) made the discovery that, by using a different method of calculating the inner product, one could find the product of two \( 2^n \times 2^n \) matrices, which, while using a similar number of overall operations, shifted the emphasis more on addition than on multiplication. This was important as addition was computationally less demanding than multiplication. Strassen (1969) provided an explicit algorithm which could multiply two \( 2^n \times 2^n \) matrices in less than \( 6.7^n \) operations, where using Winograd or the trivial algorithm, we would have had approximately \( 8^n \) operations. Using this, it is shown that \( \omega \leq \log_2 \frac{7}{(7)} < 2.81 \).

The algorithms to further reduce by means of the technique of trilinear aggregation by Pan (1978) also in Pan (1979, 1980). This technique uses the fact that computing the trace of the product of three \( n \times n \) matrices is equivalent to the problem of multiplying two \( n \times n \) matrices (in terms of the total number of multiplications). By defining a function on the indices of the entries in the matrices \( A, B \) and \( C \) to be multiplied, we may define an aggregate to do all the required multiplications, plus some extra terms. We unite terms to remove these extra terms in as few calculations as possible. Using this technique, Pan shows that we can multiply two \( 70 \times 70 \) matrix multiplications in 143640 operations. This gives \( \omega \leq \log_{70} \frac{143640}{143640} < 2.79512 \), and further, we can perform a \( 46 \times 46 \) matrix multiplication in 41952 operations, giving \( \omega \leq \log_{46} \frac{41952}{41952} < O(2.78017) \).

Bini (1980) showed that the number of operations required to perform a matrix multiplication could be reduced by considering approximate algorithms. If we change our underlying field \( k \) to be the field of polynomials of, \( \lambda \) a variable which, if \( k \) is \( \mathbb{R} \) can be assumed to be just a small number (allowing negative
powers of) with coefficients in k, we may obtain, using fewer operations, an approximation of the required matrix multiplication (in the sense that each entry will be “out” by a power of $\lambda$). Using this method (which is similar to trilinear aggregation), they obtain $O(2.7799)$.

Schonhage (1981) showed that an algorithm which could approximately compute multiple independent matrix multiplications could be used to further reduce $\omega$. This is the result of his asymptotic sum inequality- using it, he shows that $\omega < 2.5479$. Using similar techniques, Coppersmith and Winograd (1982) showed that one can take an algorithm (of a certain type) that can perform multiple disjoint matrix multiplications and square it. The resulting algorithm will be capable of multiplying larger matrices than expected. This method gives $\omega < 2.4955480$.

Strassen (1986) showed that one could start with an algorithm that was not a matrix product: we have a series of blocks, where the blocks can themselves be seen as elements of a matrix multiplication, and the blocks themselves are matrix multiplications. Raising this original algorithm to a large power, we may set some blocks to zero to obtain a large number of independent matrix products: we then use Schonhage (1981) to find a value for $\omega$. This method yields $\omega < 2.4785$.

Coppersmith and Winograd (1987) used this method to great effect to provide the current record for $\omega$. They start with an algorithm, raise it to the Nth power and show that setting certain variables as being zero will lead to the algorithm calculating a large number of independent matrix products of a certain size: using Schonhage, we get that $\omega < 2.376$.

Cohn and Umans (2005) placed $\omega$, the group-theoretic context provides new conjectures, which, if proved, will show that $\omega = 2$. A related problem is determining the rank of Matrix Multiplication. The rank is the total number of non-scalar multiplications required to evaluate a Matrix product (including scalar multiplications this becomes the Multiplicative Complexity).

The multiplication of two matrices is one of the most important operations in linear algebra. Matrix multiplication is critical in the areas of regression analysis and
statistical probability. The operation plays an important role in scientific computing. There are several methods available for using a computer program to calculate the product of two matrices. Since different algorithms give various differences in performance, finding a good algorithm seems to be valuable.

The main focus of this chapter is to compare the time complexity and space complexity between algorithms used for solution of linear equations and 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 complexity of algorithms used for matrix multiplication for both algorithms is calculated. The execution time of each algorithm is recorded to evaluate the performance of each algorithm. The Strassen’s algorithm 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.

### 3.2. Solution of Linear Equations

Solution of system of linear equations is assignment of value to variables that satisfy the equations. To solve the system of linear equations, we considered the direct method: Gaussian elimination. It is a numerical method for solving the system of linear equations $AX=B$, where $A$ is a known matrix of size $nxn$, $X$ is the required solution vector, and $B$ is a known vector of size $n$. Consider the $n$ linear equation in $n$ unknowns as

$$
\begin{align*}
    a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= a_{1,n+1} \\
    a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= a_{2,n+1} \\
    a_{31}x_1 + a_{32}x_2 + \ldots + a_{3n}x_n &= a_{3,n+1} \\
    \vdots & \quad \vdots \\
    a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= a_{n,n+1}
\end{align*}
$$

(3.1)

Where $a_{i,j}$ and $a_{i,j+1}$ are known constants and $x_i$'s are unknowns.
The sequential algorithm of gauss elimination is given below:

3.2.1. Sequential algorithm

1. Input: Given Matrix $a \ [1: n, 1: n+1]$
2. Output: $x \ [1: n]$
3. // Test for $a_{ij} <> 0$
   For i=1 to n
   For j = 1 to n
   If $a_{ij}==0$ then write “No solution”
   End for j
   End for i
4. // Triangularization process
5. For k = 1 to n-1
   //Find pivot row by searching the row with greatest absolute value among the elements of Column $k$
6. Swap the pivot row with row $k$
7. For i = k+1 to n
8. Set $m_{i,k} = \frac{a_{i,k}}{a_{k,k}}$
9. For j = k to n+1
10. Set $a_{i,j} = a_{i,j} - m_{i,k} \cdot a_{k,j}$
11. End for j
12. End for i
13. End for k
14. // Back substitution process
15. Set $x_n = \frac{a_{n,n+1}}{a_{n,n}}$
16. For i = n-1 to 1 step -1 do
17. Set $sum = 0$
18. For j = i+1 to n do
19. Set $sum = sum + a_{i,j} \cdot x_j$
20. End for j
21. \[ x_i = \frac{a_{i,n+1} - \text{sum}}{a_{i,i}} \]

22. End for i

23. for i=1 to n

24. print x[i]

25. End for i

26. End

### 3.2.2. Counting Primitive Operations

In a system of n equations with just one right hand side, we compute the number of operations as follows. The augmented matrix is nxn+1 in size.

To reduce the elements below the diagonal in column 1, we first compute (n-1) multiplying factor [takes (n-1) divides]. We multiply each of these by all the elements in row 1 except the first element [takes n multiplies] and subtract these products from the n elements in each of n-1 rows below row 1, ignoring the first element because these are known to become zero takes n-(n-1) multiplies and same number subtracts].

In summary:

- Divide = (n-1)
- Multiply = n-(n-1)
- Subtract = n-(n-1)

In the other column, we do similarly except each succeeding column has one fewer elements. So, we have for column i:

- Divides: (n-i)
- Multiplies: \((n - i + 1)(n - i)\)
- Subtracts: \((n-i+1) \ (n-i)\)

We add these quantities together for the reduction in column 1 through n-1 to get:

\[
\text{Divides} = \sum_{i=1}^{n-1} (n - i) = \frac{n^2}{2} - \frac{n}{2}
\]
Multplies=\sum_{i=1}^{n-1} (n-i+1)(n-i) = \sum_{i=1}^{n-1} i(i+1) = \frac{n^3}{3} - \frac{n}{3}

Subtracts are the same as multplies = \frac{n^3}{3} - \frac{n}{3}. If we add these together, we get, for the triangularisation part \(O(n^3)\) total operations. For back substitutions we requires:

Multplies = \sum_{i=1}^{n-1} i = \frac{n^2}{2} - \frac{n}{2}

Subtract=\sum_{i=1}^{n-1} i = \frac{n^2}{2} - \frac{n}{2}

Divide= n.

So back substitution requires \(n^2\) operations.

If we add the operations needed for the entire solution of a system on \(n\) equations we get:

\[
\frac{2n^3}{3} + \frac{3n^2}{2} - \frac{7n}{6}
\] (3.2)

So the complexity of algorithm for Gaussian Elimination method is \(O(n^3)\).

```
#include "stdafx.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>
int _tmain(int argc, _TCHAR* argv[])
{
    double matrix[100][100],a,b, temp[100],
    time_begin,time_elapsed,time_stop;
    int  i, j, k, n;
    printf("Enter the no of variables: ");
    scanf_s("%d", &n);
    printf("Enter the agumented matrix\n");
    for(i = 0; i < n ; i++)
```
{ 
    for(j = 0; j < (n+1); j++)  
    {  
        matrix[i][j]=rand();  
        // scanf_s("%lf", &matrix[i][j]);  
    }  

time_begin = omp_get_wtime ( );  
#pragma omp parallel
for(i = 0; i < n; i++)  
{  
    for(j = 0; j < n; j++)  
    {  
        if(j>i)  
        {  
            a = matrix[j][i];  
            b = matrix[i][i];  
            for(k = 0; k < n+1; k++)  
            {  
                matrix[j][k] = matrix[j][k] - (a/b) * matrix[i][k];  
            }  
        }  
    }  
}  
printf("The Upper triangular matrix is: \n");  
for( i = 0; i < n; i++)  
{  
    for(j = 0; j < n+1; j++)  
    {  
        printf("%.2f", matrix[i][j]);  
        printf("\t");  
    }  
}  
printf("\n");  
}  
printf("\nThe required result is: ");  
for(i = n-1; i>=0; i--)  
{  
    b = matrix[i][n];  
}
The above program code inputs number of variable for argumented matrix, generate random matrix and then find the value of variables using sequential and parallel execution time.

### 3.2.3 Result

The time taken by sequential and parallel algorithms are measured on many number of equations. For parallel execution #pragma omp parallel for directive is used for parallel execution using thread. The results are shown as in following table:

#### Table 3.1: Comparative time taken in sequential and parallel execution

<table>
<thead>
<tr>
<th>S. No</th>
<th>No of Equations</th>
<th>Time Taken in by Algorithms in ms.</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sequential</td>
<td>Parallel</td>
</tr>
<tr>
<td>1.</td>
<td>3</td>
<td>0.001469</td>
<td>0.001138</td>
</tr>
<tr>
<td>2.</td>
<td>10</td>
<td>0.020924</td>
<td>0.019781</td>
</tr>
<tr>
<td>3.</td>
<td>15</td>
<td>0.052480</td>
<td>0.040239</td>
</tr>
<tr>
<td>4.</td>
<td>20</td>
<td>0.100181</td>
<td>0.09234</td>
</tr>
<tr>
<td>5.</td>
<td>25</td>
<td>0.172766</td>
<td>0.157355</td>
</tr>
<tr>
<td>6.</td>
<td>30</td>
<td>0.221344</td>
<td>0.219455</td>
</tr>
<tr>
<td>7.</td>
<td>35</td>
<td>0.359364</td>
<td>0.315594</td>
</tr>
<tr>
<td>8.</td>
<td>40</td>
<td>0.448625</td>
<td>0.382662</td>
</tr>
<tr>
<td>9.</td>
<td>45</td>
<td>0.463426</td>
<td>0.451316</td>
</tr>
<tr>
<td>10.</td>
<td>50</td>
<td>0.702279</td>
<td>0.512336</td>
</tr>
</tbody>
</table>
3.3. **Iterative Methods**

Gaussian elimination method are called direct method. An entirely different way to solve many system through iteration. In this way we start with initial estimate of the solution vector and proceed to refine this estimate. This is especially true when the coefficient matrix is true.

3.3.1. **Jacobi Method**

Jacobi method belongs to the category of so-called stationary iterative methods. These methods can be expressed in the simple \( A = D - (L + U) \) form \( x^{(k)} = Fx^{(k-1)} + c \), where \( x^{(k)} \) is the approximation to the solution vector at the k-th iteration and neither \( F \) nor \( c \) depend on \( k \). To solve a system \( AX = B \), where \( A \in \mathbb{R}^{n \times n} \) and \( x, b \in \mathbb{R}^n \), the Jacobi method performs the following computations in its k-th iteration:

\[
x_i^{(k)} = a_{ii}^{-1}(b_i - \sum_{j \neq i} a_{ij}x_j^{(k-1)}),
\]

for all \( i, 0 \leq i < n \). In the equation, \( a_{ij} \) denotes the element in row \( i \) and column \( j \) of matrix \( A \) and, \( x_i^{(k)} \) and \( x_i^{(k-1)} \) indicate the \( i \)-th element of the iteration vector for the iterations numbered \( k \) and \( k - 1 \), respectively. The Jacobi equation given above can also be written in matrix notation as:

\[
\begin{bmatrix}
x_1^{(k)} \\
x_2^{(k)} \\
\vdots \\
x_n^{(k)}
\end{bmatrix} = D^{-1}(L + U)\begin{bmatrix}
x_1^{(k-1)} \\
x_2^{(k-1)} \\
\vdots \\
x_n^{(k-1)}
\end{bmatrix} + D^{-1}b
\]

Where \( A = D - (L + U) \) is a partitioning of \( A \) into its diagonal, lower-triangular and upper-triangular parts, respectively.

3.3.1.1 **Algorithm for Jacobi Iteration**

We assume that the system \( Ax = b \) has been rearranged so that the matrix \( A \) is diagonally dominant. That is, for each row of \( A \):

\[
|a_{ii}| > \sum_{\substack{j=1 \quad j \neq i \quad j \neq i}}^{n} |a_{ij}|, \quad i = 1, 2, \ldots, n
\]
This is a sufficient condition for convergence. We begin with initial approximation to the solution vector, which we store in the vector $x_1$.

1. For $i=1$ to $n$
   
   Set $b[i] = \frac{b[i]}{a[i]}$

   Set $x_2[i] = x_1[i]$

   For $j=i$ to $n$
   
   If $i <> j$
   
   Set $a[i, j] = \frac{a[i, j]}{a[i]}$

   End if

   End for $j$

   End for $i$

2. Repeat
   
   For $i=1$ to $n$
   
   Set $x_1[i] = x_2[i]$

   Set $x_2[i] = b[i]$

   End for $i$

   For $i=1$ to $n$
   
   For $j=1$ to $n$
   
   If $(i <> j)$ then
   
   Set $x_2[i] = x_2[i] - a[i, j] \cdot x_1[i]$

   End if

   End for $j$

   End for $i$

   Until ($x_2[i]$ and $x_1[i]$ convergence to each other)

3. End

### 3.3.3.2 Counting Primitive operations using RAM model

1. The loop executes $n$ times as:

   i. Set $b[i] = \frac{b[i]}{a[i]}$ contributes two unit of count.
ii. Set \( x_2[i] = x_2[i] \) contributes one unit to count.
   \[
   = n(2+1)
   \]
   \[
   = 3n.
   \]

iii. The body of loop executes \( n \) times as:
   (a) Comparison \( i \) with \( j \) and compute
   \[
   \frac{d[i,j]}{d[i]} \]
   count two unit
   operation.
   \[
   = n(n(2))
   \]
   \[
   = 2n^2.
   \]

2. The body of loop will execute \( n \) times as:
   i. Set \( x_1[i] = x_2[i] \) contributes one unit to count.
   ii. Set \( x_2[i] = b[i] \) contributes one unit to count.
   \[
   = 2n.
   \]

3. The body of loop will execute \( n \) times as:
   The body of loop executes \( n \) times as:
   (a) Comparison \( i \) with \( j \) and compute
   \[
   x_1[i] = x_1[i] - a[i,j] \cdot x_1[i] \]
   count four
   unit operation.
   \[
   = n(n(4))
   \]
   \[
   = 4n^2
   \]
   \[
   t(n) = 5n + 3n^2 + 2n + 4n^2
   \]
   \[
   = 7n^2 + 7n
   \]
   So the complexity of Jacobi algorithm is \( O(n^2) \)

Program coding for Jacobi Method

```cpp
// jacobi.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>
```

```c
#include<conio.h>
#define ESP 0.0001
// Example defined for linear equation to find the value of x1,x2,and x3
#define X1(x2,x3) (1.8333 +0.3333*(x2) -0.1667*(x3))
#define X2(x1,x3) (0.7143 +0.2857*(x1) -0.2857*(x3))
#define X3(x1,x2) (0.2000 +0.2000*(x1) + 0.4000*(x2))
int _tmain(int argc, _TCHAR* argv[])
{
    double time_begin,time_elapsed,time_stop;
    double x1=0,x2=0,x3=0,y1,y2,y3;
    int i=0;
    printf("_______________________________________ ___
    \n   x1		   x2		   x3
    \n_______________________________________ ___
    \n%f	%f	%f",x1,x2,x3);
    time_begin = omp_get_wtime ( );
    #pragma omp parallel for
    do
    {
        y1=X1(x2,x3);
        y2=X2(x1,x3);
        y3=X3(x1,x2);
        if(fabs(y1-x1)<ESP && fabs(y2-x2)<ESP && fabs(y3 -x3)<ESP )
        {
            printf("\n%f%f%f",x1,x2,x3);
            printf("x1 = %.3lf",y1);
            printf("x2 = %.3lf",y2);
            printf("x3 = %.3lf",y3);
            i = 1;
        }
    }
    else
    {
        x1 = y1;
        x2 = y2;
        x3 = y3;
        printf("\n%f%f%f",x1,x2,x3);
    }
}
```
3.3.2. Gauss-Seidel Method

The Gauss-Seidel method typically converges faster than the Jacobi method by using the most recently available approximations of the elements of the iteration vector. The other advantage of the Gauss-Seidel algorithm is that it can be implemented using only one iteration vector, which is important for large linear equation systems where storage of a single iteration vector alone may require 10GB or more. However, a consequence of using the most recently available solution approximation is that the method is inherently sequential – it does not possess natural parallelism. The Gauss-Seidel method has been used for parallel solutions of Markov chains, given in Kwiatkowska et al. (2004).

3.3.2.1 Algorithm for Gauss-Seidel Iteration

We assume that the system AX=B has been rearranged so that the coefficient matrix A, is diagonally dominant. We begin with initial approximation to the solution vector which we store in the vector X

1. For i=1 to n
   Set $b[i] = \frac{b[i]}{a[i,i]}$
   For j=1 to n
     If ($i \neq j$)
     Set $a[i, j] = \frac{a[i, j]}{a[i, i]}$
     End if
   End for j
End for i
2. While (not convergence) 
for i=1 to n
Set $x[i] = b[i]$
For j=1 to n
If $(i \neq j)$ 
Set $x[i] = x[i] - a[i, j] \cdot x[j]$
End for j
End for i
End While

3. End

3.3.2.2 Counting primitive operations
1. The loop executes n times as:
   i. Set $b[i] = \frac{b[i]}{a[i, i]}$ contributes two unit to count.
      
      \[= 2\]
   ii. J loops executes n times as:
        (a) To compare i with j and calculate $a[i, j] = \frac{a[i, j]}{a[i, i]}$ contributes three unit to count.
        
        \[= 3n\]
        \[= n(2 + 3n)\]
        \[= 2n + 3n^2\]

2. The i loop executes n times:
   i. Set $x[i] = b[i]$ contributes one unit to count.
      
      \[= 1\]
   ii. The j loop executes n times:
        (a) To compare I with j and $x[i] = x[i] - a[i, j] \cdot x[j]$ contributes four unit to count.
        
        \[= 4n\]
        \[= n(1 + 4n)\]
\[ n + 4n^2 \]

Total (1+2) \[ = 2n + 3n^2 + n + 4n^2 \]

\[ = 7n^2 + 3n \]

So the complexity of Gauss-sideal algorithm is \( O(n^2) \)

Program coding for Gauss-sideal

```cpp
// jacobi.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdio.h>
#include <conio.h>

int main()
{
    int i,j,k,n;
    float a[10][10],c[10],x[10];
    double time_begin,time_elapsed,time_stop;
    time_begin = omp_get_wtime ();
    #pragma omp parallel for
    printf("Enter the number of elements: ");
    scanf("%d",&n);
    printf("Enter constants: ");
    for(i=0;i<n;i++)
        scanf("%f",&c[i]);
    printf("Enter the matrix: ");
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%f",&a[i][j]);
    for(k=0;k<n-1;i++)
        for(i=k+1;i<n;i++)
            for(j=k+1;j<n;j++)
                a[i][j]=a[i][j]-(a[i][k]/a[k][k])*a[k][j];
    c[i]=c[i]-(a[i][k]/a[k][k])*c[k];
}

x[n-1]=c[n-1]/a[n-1][n-1];
```
3.3.3. Result

The complexity of both algorithms Jacobi and Gauss-sidel are $O(n^2)$, while the complexity of Gauss elimination method is $O(n^3)$. The time complexity of iterative methods are better than Gauss elimination method. Using the OpenMP method for parallel processing with help of `#pragma` directive the program run faster than sequential access algorithms is given in following table:

Using following two examples we find the execution time for sequential and parallel execution.

Example 1:

\[
\begin{align*}
x_1(x_2, x_3) &= (1.8333 + 0.3333 \cdot (x_2) - 0.1667(x_3)) \\
x_2(x_1, x_3) &= (0.7143 + 0.2857 \cdot (x_1) - 0.2857 \cdot (x_3)) \\
x_3(x_1, x_2) &= (0.2000 + 0.2000 \cdot (x_1) - 0.4000 \cdot (x_2))
\end{align*}
\]

Example 2:

\[
\begin{align*}
x_1(x_2, x_3) &= ((17 - 20 \cdot (x_2) + 2 \cdot (x_3)) / 20) \\
x_2(x_1, x_3) &= ((-18 - 3 \cdot (x_1) + (x_3)) / 20) \\
x_3(x_1, x_2) &= ((25 - 2 \cdot (x_1) + 3 \cdot (x_2)) / 20)
\end{align*}
\]
Table – 3.2: Comparison between Jacobi and Gauss-seidel Methods

<table>
<thead>
<tr>
<th>S. No</th>
<th>Example</th>
<th>Time in ms for Jacobi Method</th>
<th>Gauss-seidel Method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sequential</td>
<td>Parallel</td>
</tr>
<tr>
<td>1.</td>
<td>Example 1</td>
<td>0.001594</td>
<td>0.001019</td>
</tr>
<tr>
<td>2.</td>
<td>Example 2</td>
<td>0.003311</td>
<td>0.001006</td>
</tr>
</tbody>
</table>

3.4 Matrix Multiplication

3.4.1 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 $(AB)_{i,j} = \sum A_{i,k}B_{k,j}$.

Matrix product is not commutative. 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. Matrix multiplication is the process of taking two or more $n \times n$ matrices and calculating a product by summing the product of each row in one matrix with the column of the second matrix. The simple algorithm to calculate this product can be summarized in the following algorithm:

3.4.1.1 Algorithm for Matrix Multiply $(A, B)$

1. if columns[A] <> rows[B]
2. then ERROR "Incompatible Dimensions"
3. else for $i = 1$ to rows[A]
4. do for $j = 1$ to columns[B]
5. do $C[i,j] = 0$
6. do for $k = 1$ to columns[B]
8. return $C$
3.4.1.2 Counting Primitive Operations using RAM model.
2. To initialize i=1 requires 1 operation and it repeats n times.
3. To initialize j=1 requires 1 operations and it repeats n times.
4. C[i,j]=0 requires 1 operation.
5. To initialize k=1 requires 1 operation and it repeats n times.
6. Step 7 requires 3 operations.
7. Step 8 requires 1 operations.
So total number of operations= 1 + n(1 + n(3n + 1) + 1
= 3n³ + n² + n + 2.

This simple algorithm can be measured using Big O analysis:

\[ t(n) = O(n^3) \]
Hence the complexity of this algorithm is \( O(n^3) \).

Program Coding for Matrix Multiplication:

```c
#include <iostream.h>
#include <time.h>
const int SIZE=100;
void Matrix_Mult ( int A[SIZE][SIZE], int B[SIZE][SIZE], int C[SIZE][SIZE]);
void Prn ( int C[SIZE][SIZE]);
int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE], count=0;
void main()
{
  long unsigned int start, end, i;
  for ( int a=1; a<=SIZE; a++ )
      for ( int b=1; b<=SIZE; b++ )
        C[a][b]=0;
  start=clock();       // Starts the timing clock.
  for (i=1; i<=1000; i++)
    {
      for ( int a = 0; a < SIZE; a++ )
        for ( int b = 0; b < SIZE; b++ )
          {
            A[a][b] = count;
            count++;
          }
    }
```
count = SIZE * SIZE;
for (a = 0; a < SIZE; a++)
    for (int b = 0; b < SIZE; b++)
    {
        B[a][b] = count;
        count--;
    }
Matrix_Mult(A,B,C); // Calls the multiplication function.
}
end=clock(); //Ends the timing clock.
Prn(C); // Calls the print function.
cout<>"Matrix size = "<<SIZE<<"x"<<SIZE;
cout<>"\nThis algorithm took "<<end-start<<" milliseconds.\n"
}

void Matrix_Mult(int a1[SIZE][SIZE], int a2[SIZE][SIZE], int a3[SIZE][SIZE])
{
    int i, j, k;
    for(i = 0; i < SIZE; i++)
        for(j = 0; j < SIZE; j++)
            for(k=0; k < SIZE; k++)
            {
                a3[i][j] += a1[i][k] * a2[k][j];
            }
}

void Prn(int C[SIZE][SIZE])
{
    // prints out the contents of a NxN array.
    int iii, jjj;
    for(iii = 0; iii < SIZE; iii++)
    {
        for (jjj = 0; jjj < SIZE; jjj++)
        {
            cout<>C[iii][jjj]/1000<<"\n";
            cout<>"\n";
        }
    }
}

3.4.2. STRASSEN’S METHOD

Strassen (1969) proposed a method of divide and conquer to try to break the $O(n^3)$ barrier. Where the idea came from is unclear, however the goal was to reduce the number of multiplications needed to complete the algorithm. By splitting a matrix down into smaller partitions of 2 x 2 dimensions, Strassen discovered that the eight (8)
Complexity of Algorithms for Solution of Simultaneous......

multiplications needed in the original simple iterative algorithm can be reduced to seven (7). Strassen’s Recursive Algorithm is outlined in the following algorithm:

### 3.4.2.1 Algorithm for Strassen’s Method

```
PROD (A, B: [n x n] matrix)
1. if n = 1 then
2. C = A * B
3. else
4. parity = n MOD 2
5. if parity = 1 then
6. n = n + 1 // add a row and a column of 0’s to A & B
7. end if
9. [A21 A22] [B21 B22]
10. M1 = PROD(A12 - A22, B21 + B22)
11. M2 = PROD(A11 + A22, B11 + B22)
12. M3 = PROD(A11 - A22, B11 + B22)
13. M4 = PROD(A11 + A12, B22)
14. M5 = PROD(A11, B12 - B22)
15. M6 = PROD(A22, B21 - B11)
16. M7 = PROD(A21 + A22, B11)
17. C11 = M1 + M2 - M4 + M6
18. C12 = M4 + M5
19. C21 = M6 + M7
20. C22 = M2 - M3 + M5 - M7
21. if parity = 1 then
22. n = n - 1 // remove the last row & column from C
23. end if
24. return C
```

Strassen was able to reduce the number of multiplications from eight (8) to seven (7) as outlined below:
Assuming a matrix of 2 x 2 dimension, we can calculate the additions and multiplications of the simple iterative algorithm:

Multiplications: \[ M(n) = 8M\left(\frac{n}{2}\right) . M(1) = 8 \]

Additions: \[ A(n) = 8A\left(\frac{n}{2}\right) + n^2 \]
\[ t(n) = 8\log n = n\log 8 = n^3 \]
\[ O(n^3) \]

Strassen was able to reduce the number of multiplications required in the basic iterative algorithm yielding the following computations:

Multiplications: \[ M(n) = 7M\left(\frac{n}{2}\right) . M(1) = 7 \]

Additions: \[ A(n) = 7A\left(\frac{n}{2}\right) + 18\left(\frac{n^2}{4}\right) \]
\[ t(n) = 7t\left(\frac{n}{2}\right) + O(n) = O(n^{2.81}) \]
\[ .Hence\ complexity\ of\ this\ algorithm\ is\ O(n^{2.81}). \]

Program Coding:

```cpp
#include <iostream.h>
#include <time.h>
const int SIZE = 40;
const int THRESHOLD = 2;
void strassen (int matrixA[][SIZE], int matrixB[][SIZE],
int productC[][SIZE],
    int matrixSize, long double& multCount, long double& addSubtCount );
void matrix_multiply (int array_size, int arrA[][SIZE], int arrB[][SIZE],
int productArray[][SIZE], long double& multi_count , long double&
    add_count);
int add (int num1, int num2, long double& count_add );
int multiply (int num1, int num2, long double& count_multi);
void addMatrices (int matrixA[][SIZE], int matrixB[][SIZE],
int resultMatrix[][SIZE], int matrixSize, long double& addSubtCount);```
void subtMatrices(int matrixA[][SIZE], int matrixB[][SIZE],
                 int resultMatrix[][SIZE], int matrixSize, long double& addSubtCount);

void main()
{
  int a1[SIZE][SIZE], a2[SIZE][SIZE], final[SIZE][SIZE];
  int a, b;
  long unsigned int start, end, iterations;
  long double addSubtCounter = 0, multCounter = 0;
  int count = 0;
  for (a = 0; a < SIZE; a++)
  {
    for (b = 0; b < SIZE; b++)
    {
      a1[a][b] = count;
      count++;
    }
  }
  count = SIZE * SIZE;
  for (a = 0; a < SIZE; a++)
  {
    for (b = 0; b < SIZE; b++)
    {
      a2[a][b] = count;
      count--;
    }
  }
  start = clock();
  for (iterations = 1; iterations <= 1000; iterations++)
    strassen(a1, a2, final, SIZE, multCounter, addSubtCounter);
  end = clock();
  cout << "Number of times multiplications were done: " << multCounter
       << "nNumber of times additions/subtractions were done: "
       << addSubtCounter << endl;
  cout << "Matrix size = " << SIZE << "x" << SIZE << "n";
  cout << "This algorithm took " << end - start << " milliseconds to run.n";
}

void strassen(int matrixA[][SIZE], int matrixB[][SIZE], int
              productC[][SIZE],
              int matrixSize, long double& multCount, long double& addSubtCount)
{
  int nextMatrixSize;
int a11[SIZE][SIZE], a12[SIZE][SIZE], a21[SIZE][SIZE],
a22[SIZE][SIZE];
int b11[SIZE][SIZE], b12[SIZE][SIZE], b21[SIZE][SIZE],
b22[SIZE][SIZE];
int c11[SIZE][SIZE], c12[SIZE][SIZE], c21[SIZE][SIZE],
c22[SIZE][SIZE];
int m1[SIZE][SIZE];
int m2[SIZE][SIZE];
int m3[SIZE][SIZE];
int m4[SIZE][SIZE];
int m5[SIZE][SIZE];
int m6[SIZE][SIZE];
int m7[SIZE][SIZE];
int addResult1[SIZE][SIZE], addResult2[SIZE][SIZE],
subtResult[SIZE][SIZE];
multCount++;
if (matrixSize <= THRESHOLD)
{
    matrix_multiply (matrixSize, matrixA, matrixB, productC,
multCount,
    addSubtCount);
}
else
{
    nextMatrixSize = matrixSize/2;
    for (int i = 0; i < nextMatrixSize; i++)
        for (int j = 0; j < nextMatrixSize; j++)
        {
            a11[i][j] = matrixA[i][j];
a12[i][j] = matrixA[i][j + nextMatrixSize];
a21[i][j] = matrixA[i + nextMatrixSize][j];
a22[i][j] = matrixA[i + nextMatrixSize][j +
nextMatrixSize];
b11[i][j] = matrixB[i][j];
b12[i][j] = matrixB[i][j + nextMatrixSize];
b21[i][j] = matrixB[i + nextMatrixSize][j];
b22[i][j] = matrixB[i + nextMatrixSize][j +
nextMatrixSize];
        }
    addMatrices (a11, a22, addResult1, nextMatrixSize, addSubtCount);
    addMatrices (b11, b22, addResult2, nextMatrixSize, addSubtCount);
    strassen (addResult1, addResult2, m1, nextMatrixSize, multCount,
addSubtCount);
addMatrices (a21, a22, addResult1, nextMatrixSize, addSubtCount);
strassen (addResult1, b11, m2, nextMatrixSize, multCount, addSubtCount);
addSubtCount);
subtMatrices (b12, b22, subtResult, nextMatrixSize, addSubtCount);
strassen (a11, subtResult, m3, nextMatrixSize, multCount, addSubtCount);
subtMatrices (b21, b11, subtResult, nextMatrixSize, addSubtCount);
strassen (a22, subtResult, m4, nextMatrixSize, multCount, addSubtCount);
addMatrices (a11, a12, addResult1, nextMatrixSize, addSubtCount);
strassen (addResult1, b22, m5, nextMatrixSize, multCount, addSubtCount);
subtMatrices (a21, a11, subtResult, nextMatrixSize, addSubtCount);
addMatrices (b11, b12, addResult1, nextMatrixSize, addSubtCount);
strassen (subtResult, addResult1, m6, nextMatrixSize, multCount, addSubtCount);
addMatrices (a12, a22, subtResult, nextMatrixSize, addSubtCount);
addMatrices (b21, b22, addResult1, nextMatrixSize, addSubtCount);
strassen (subtResult, addResult1, m7, nextMatrixSize, multCount, addSubtCount);
addMatrices (m1, m4, addResult1, nextMatrixSize, addSubtCount);
addMatrices (addResult1, m7, addResult2, nextMatrixSize, addSubtCount);
subtMatrices (addResult2, m5, c11, nextMatrixSize, addSubtCount);
addMatrices (m3, m5, c12, nextMatrixSize, addSubtCount);
addMatrices (m2, m4, c21, nextMatrixSize, addSubtCount);
addMatrices (m1, m3, addResult1, nextMatrixSize, addSubtCount);
addMatrices (addResult1, m6, addResult2, nextMatrixSize, addSubtCount);
subtMatrices (addResult2, m2, c22, nextMatrixSize, addSubtCount);
for (int k = 0; k < nextMatrixSize; k++)
    for (int j = 0; j < nextMatrixSize; j++)
    {
        productC[k][j] = c11[k][j];
        productC[k][j + nextMatrixSize] = c12[k][j];
        productC[k + nextMatrixSize][j] = c21[k][j];
        productC[k + nextMatrixSize][j + nextMatrixSize] = c22[k][j];
    }
}
void matrix_multiply (int array_size, int arrA[][SIZE], int arrB[][SIZE],

int productArray[][SIZE], long double& multi_count, long double& add_count)
{
    int i,j,k;
    int multi;
    for (i = 0; i < array_size; i++)
        for (j = 0; j < array_size; j++)
            {
            productArray[i][j] = 0;
            for (k = 0; k < array_size; k++)
                {
                multi = multiply (arrA[i][k],arrB[k][j],
multi_count);
                productArray[i][j] = add (productArray[i][j],
multi, add_count);
                }
            }
}
int add (int num1, int num2, long double& count_add)
{
    int add_result = num1 + num2;
    count_add++;
    return add_result;
}
int multiply (int num1, int num2, long double& count_multi)
{
    int multi_result = num1 * num2;
    count_multi++;
    return multi_result;
}
// addMatrices function - adds two matrices together and places the resulting
// matrix into the resultMatrix.
void addMatrices (int matrixA[][SIZE], int matrixB[][SIZE],
    int resultMatrix[][SIZE], int matrixSize, long double& addSubtCount)
{
    for (int i = 0; i < matrixSize; i++)
        for (int j = 0; j < matrixSize; j++)
            resultMatrix[i][j] = add (matrixA[i][j], matrixB[i][j],
addSubtCount);
}
// subMatrices function - subtracts matrixA from matrixB and places the
// result in the resultMatrix.
void subtMatrices (int matrixA[][SIZE], int matrixB[][SIZE],
    int resultMatrix[][SIZE], int matrixSize, long double& addSubtCount)
{
    for (int i = 0; i < matrixSize; i++)
        for (int j = 0; j < matrixSize; j++)
            resultMatrix[i][j] = add (matrixA[i][j], (0 -
                matrixB[i][j]),
                addSubtCount);
}

Here, some tested data needs to be obtained and investigated in terms of
evaluation time (time complexity and space complexity). Table 3.3 is an execution
time comparison between conventional matrix multiplication and Strassen’s
algorithm. The input matrices for testing have dimension of $n \times m$. 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. Finally, comparing all of our timing results from
both Strassen and the simple iterative logic, we see the following results table:

<table>
<thead>
<tr>
<th>Matrix Dimension</th>
<th>Recursion Level</th>
<th>Conventional Matrix</th>
<th>Strassan’s Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2</td>
<td>1.0 ms</td>
<td>5.0 ms</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>1.0 ms</td>
<td>51.0 ms</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
<td>2.0 ms</td>
<td>191.0 ms</td>
</tr>
<tr>
<td>128</td>
<td>2</td>
<td>9.0 ms</td>
<td>973.0 ms</td>
</tr>
<tr>
<td>512</td>
<td>2</td>
<td>1411 ms</td>
<td>28011 ms</td>
</tr>
<tr>
<td>512</td>
<td>8</td>
<td>1591 ms</td>
<td>2220 ms</td>
</tr>
<tr>
<td>512</td>
<td>16</td>
<td>1560 ms</td>
<td>1240 ms</td>
</tr>
<tr>
<td>512</td>
<td>64</td>
<td>2051 ms</td>
<td>671 ms</td>
</tr>
<tr>
<td>512</td>
<td>128</td>
<td>1551 ms</td>
<td>740 ms</td>
</tr>
<tr>
<td>512</td>
<td>256</td>
<td>1543 ms</td>
<td>1080 ms</td>
</tr>
</tbody>
</table>
Conclusion

Time Complexity: The standard matrix multiplication takes approximately $2N^3$ where $N=2n$ arithmetic operations is $O(N^3)$. When using the Strassen algorithm the operation is approximately $O(N^{2.81})$.

Space Complexity: Compare to the standard matrix multiplication, Strassen’s algorithm will require more spaces as it has intermediate matrices.