DAA Experiment-3

(Batch-A/A1)

Name	Ansari Mohammed Shanouf Valijan
UID Number	2021300004
Class	SY B.Tech Computer Engineering(Div-A)
Experiment Number	3
Date of Performance	02-03-23
Date of Submission	08-03-23

Aim:

To implement Strassen's Matrix Multiplication algorithm.

Theory:

Strassen's Matrix Multiplication algorithm is an approach towards multiplying two matrices. It follows the divide and concur approach. In this algorithm, primary step is to add rows and columns with element-0 in the input matrices so that they have their order in the form 2^n where n is a non-negative integer. Further, each input matrix is divided into four smaller matrices and these matrices are used to obtain the final result. This happens recursively till a matrix can no longer be divided. Given below is a general demonstration of the Strassen's multiplication-

Consider X, Y as two input matrices of order 2^n and Z as the output matrix of the same order, that is, Z=XY.

Let X=$\begin{bmatrix} A & B \\ C & D \end{bmatrix}$ and Y=$\begin{bmatrix} E & F \\ G & H \end{bmatrix}$.

Let Z=$\begin{bmatrix} I & J \\ K & L \end{bmatrix}$.

In all the above matrices, A to H are the initial partitions made to work on and I to L are the result partitions that are combined to obtain the final result of multiplication. Given below is the process that is followed to obtain the result partition-

Note that all variables displayed are matrices of the order 2^{n-1}.
M_1=(A+C)x(E+F).
M_2=(B+D)x(G+H).
M_3=(A-D)x(E+H).
M_4=(A)x(F-H).
M_5=(C+D)x(E).
M_6=(A+B)x(H).
M_7=(D)x(G-E).

From this we obtain,
I=$M_2 + M_3 - M_6 - M_7$
J=$M_4 + M_6$
K=$M_5 + M_7$
L=$M_1 - M_3 - M_4 - M_5$

In the process where we are calculating all the M matrices, we recursively use the Strassen's approach. Obtained I, J, K and L partitions are combined to get the result matrix Z as required.

Algorithm:

[A] For Strassen's Matrix Multiplication-

I. Start.
II. Calculate the order of input matrices in the form of 2^n that will be able to accommodate the provided elements.
III. Partition the input matrices of order 2^n into four matrices of order 2^{n-1}.
IV. Calculate the required seven matrices M_1 to M_7 by recursively performing Strassen's matrix multiplication.
V. Obtain the result partitions through addition and subtraction of M matrices.
VI. Combine the result partitions to get the final product matrix.
VII. End.

Program:

```
//header files
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<stdbool.h>
#include<time.h>
```

```
//functions for dynamic creation, destruction of matrices
int** createMatrix(int order){
    int** matrix;
    matrix=(int**)calloc(order,sizeof(int*));
    for(int i=0; i<order; i++){
        matrix[i]=(int*)calloc(order,sizeof(int));
    }
    return matrix;
}

void destroyMatrix(int** matrix, int order){
    for(int i=0; i<order; i++){
        free(matrix[i]);
    }
    free(matrix);
}

//function for addition of matrices
int** addSubMat(int** matA, int** matB, int order, bool isAdd){
    int** resultMat=createMatrix(order);
    for(int i=0; i<order; i++){
        for(int j=0; j<order; j++){
            if(isAdd)
                resultMat[i][j]=matA[i][j]+matB[i][j];
            else
                resultMat[i][j]=matA[i][j]-matB[i][j];
        }
    }
    return resultMat;
}

//functions for matrix input and output
void takeMatInput(int** matrix, int rowM, int colM){
    for(int i=0; i<rowM; i++){
        for(int j=0; j<colM; j++){
            //scanf("%d",&matrix[i][j]);
            matrix[i][j]=1;
        }
    }
}

void displayMat(int** matrix, int rowM, int colM){
    for(int i=0; i<rowM; i++){
        for(int j=0; j<colM; j++){
            printf("%d ",matrix[i][j]);
        }
        printf("\n");
    }
}

//functions for partioning and joining of matrices
void partitionMat(int*** matrices, int** matA, int** matB, int order){
    int tempi=0, tempj=0;
    for(int i=0; i<order/2; i++){
```

```
        for(int j=0; j<order/2; j++){
            matrices[0][i][j]=matA[i][j];
            matrices[4][i][j]=matB[i][j];
        }
    }
    for(int i=0; i<order/2; i++){
        for(int j=order/2; j<order; j++){
            matrices[1][i][tempj]=matA[i][j];
            matrices[5][i][tempj]=matB[i][j];
            tempj++;
        }
        tempj=0;
    }
    for(int i=order/2; i<order; i++){
        for(int j=0; j<order/2; j++){
            matrices[2][tempi][j]=matA[i][j];
            matrices[6][tempi][j]=matB[i][j];
        }
        tempi++;
    }
    tempi=tempj=0;
    for(int i=order/2; i<order; i++){
        for(int j=order/2; j<order; j++){
            matrices[3][tempi][tempj]=matA[i][j];
            matrices[7][tempi][tempj]=matB[i][j];
            tempj++;
        }
        tempi++;
        tempj=0;
    }
}

void joinMats(int*** matrices, int** resultMat, int order){
    int tempi=0, tempj=0;
    for(int i=0; i<order/2; i++){
        for(int j=0; j<order/2; j++){
            resultMat[i][j]=matrices[8][i][j];
        }
    }
    for(int i=0; i<order/2; i++){
        for(int j=order/2; j<order; j++){
            resultMat[i][j]=matrices[9][i][tempj];
            tempj++;
        }
        tempj=0;
    }
    for(int i=order/2; i<order; i++){
        for(int j=0; j<order/2; j++){
            resultMat[i][j]=matrices[10][tempi][j];
        }
        tempi++;
    }
    tempi=tempj=0;
    for(int i=order/2; i<order; i++){
```

```
        for(int j=order/2; j<order; j++){
            resultMat[i][j]=matrices[11][tempi][tempj];
            tempj++;
        }
        tempi++;
        tempj=0;
    }
}

//using strassens method of matrix multiplication
int** strassensMul(int** matA, int** matB, int order){
    int** resultMat=createMatrix(order);
    int*** matrices;
    //base condition
    if(order==1){
        resultMat[0][0]=matA[0][0]*matB[0][0];
        return resultMat;
    }
    //strassen's logic
    matrices=(int***)malloc(19*sizeof(int**));
    for(int i=0; i<19; i++)
        matrices[i]=createMatrix(order/2);
    //partitioning the original matrix into 4 smaller matrices
    partitionMat(matrices,matA,matB,order);
    //using the obtained partitions to get the result matrix
    matrices[12]=strassensMul(addSubMat(matrices[0],matrices[2],order/2,true),addSubMat(ma
trices[4],matrices[5],order/2,true),order/2);
    matrices[13]=strassensMul(addSubMat(matrices[1],matrices[3],order/2,true),addSubMat(ma
trices[6],matrices[7],order/2,true),order/2);
    matrices[14]=strassensMul(addSubMat(matrices[0],matrices[3],order/2,false),addSubMat(m
atrices[4],matrices[7],order/2,true),order/2);
    matrices[15]=strassensMul(matrices[0],addSubMat(matrices[5],matrices[7],order/2,false)
,order/2);
    matrices[16]=strassensMul(addSubMat(matrices[2],matrices[3],order/2,true),matrices[4],
order/2);
    matrices[17]=strassensMul(addSubMat(matrices[0],matrices[1],order/2,true),matrices[7],
order/2);
    matrices[18]=strassensMul(matrices[3],addSubMat(matrices[6],matrices[4],order/2,false)
,order/2);
    //obtaining the result partitions
    matrices[8]=addSubMat(addSubMat(matrices[13],matrices[14],order/2,true),addSubMat(matr
ices[17],matrices[18],order/2,true),order/2,false);
    matrices[9]=addSubMat(matrices[15],matrices[17],order/2,true);
    matrices[10]=addSubMat(matrices[16],matrices[18],order/2,true);
    matrices[11]=addSubMat(addSubMat(matrices[12],matrices[14],order/2,false),addSubMat(ma
trices[15],matrices[16],order/2,true),order/2,false);
    //combining the partitions to get the final result
    joinMats(matrices,resultMat,order);
    //deallocating the used matrices
    for(int i=0; i<19; i++)
        destroyMatrix(matrices[i],order/2);
    free(matrices);
    return resultMat;
}
```

```
//defining naive approach of matrix multiplication for comparison
int** naiveMul(int** matA, int** matB, int rA, int cA, int cB, int order){
    int** resultMat=createMatrix(order);
    int sum;
    for(int i=0; i<rA; i++){
        for(int j=0; j<cB; j++){
            sum=0;
            for(int k=0; k<cA; k++){
                sum+=matA[i][k]*matB[k][j];
            }
            resultMat[i][j]=sum;
        }
    }
    return resultMat;
}

//main function
void main(){
    clock_t time;
    //taking user input of order of matrices and converting it to suitable square order
    int rA, cA, rB, cB;
    printf("\nEnter the number of rows and columns in matrix-A -----> ");
    scanf("%d %d",&rA,&cA);
    printf("Enter the number of rows and columns in matrix-B -----> ");
    scanf("%d %d",&rB,&cB);
    int max1=rA>cA?rA:cA;
    int max2=rB>cB?rB:cB;
    int max=max1>max2?max1:max2;
    int order=(int)pow(2,ceil((log(max)/log(2))));
    int** matA=createMatrix(order);
    int** matB=createMatrix(order);
    int** resultMat=createMatrix(order);
    //printf("\nEnter Matrix-A-\n\n");
    takeMatInput(matA,rA,cA);
    //printf("\nEnter Matrix-B-\n\n");
    takeMatInput(matB,rB,cB);
    printf("\nAll cells in the matrices were initialized with 1.....\n");
    time=clock();
    resultMat=strassensMul(matA,matB,order);
    time=clock()-time;
    //printf("\nMatrix AxB as obtained-\n\n");
    //displayMat(resultMat,rA,cB);
    printf("\nTime taken to perform the strassen's multiplication: %lf
seconds",(double)time/CLOCKS_PER_SEC);
    //temp
    time=clock();
    resultMat=naiveMul(matA,matB,rA,cA,cB,order);
    time=clock()-time;
    //displayMat(resultMat,rA,cB);
    printf("\nTime taken to perform the naive matrix multiplication: %lf
seconds\n\n",(double)time/CLOCKS_PER_SEC);
    //temp
    destroyMatrix(matA,order);
    destroyMatrix(matB,order);
```

```
    destroyMatrix(resultMat,order);
    printf("\n\n");
}
```

Implementation:

Following examples were utilized to verify the correctness of the code and also to compare Strassen's method and Naïve method of matrix multiplication-

```
Enter the number of rows and columns in matrix-A -----> 4 5
Enter the number of rows and columns in matrix-B -----> 5 3

Enter Matrix-A-

1 5 4 2 1
7 4 1 2 5
1 5 4 8 4
9 5 4 1 2

Enter Matrix-B-

7 4 8
1 4 2
1 5 4
2 6 5
7 4 1

Matrix AxB as obtained-

27 60 45
93 81 83
60 108 78
88 90 105

Time taken to perform the strassen's multiplication: 0.001000 seconds
Time taken to perform the naive matrix multiplication: 0.000000 seconds
```

```
Enter the number of rows and columns in matrix-A -----> 2 2
Enter the number of rows and columns in matrix-B -----> 2 2

Enter Matrix-A-

1 3
7 5

Enter Matrix-B-

6 8
4 2

Matrix AxB as obtained-

18 14
62 66

Time taken to perform the strassen's multiplication: 0.000000 seconds
Time taken to perform the naive matrix multiplication: 0.000000 seconds
```

```
Enter the number of rows and columns in matrix-A -----> 4 4
Enter the number of rows and columns in matrix-B -----> 4 4

Enter Matrix-A-

4 1 5 2
7 4 9 5
1 5 4 2
1 4 5 2

Enter Matrix-B-

9 5 4 2
8 5 4 1
1 2 5 4
1 6 5 4

Matrix AxB as obtained-

51 47 55 37
109 103 114 74
55 50 54 31
48 47 55 34

Time taken to perform the strassen's multiplication: 0.000000 seconds
Time taken to perform the naive matrix multiplication: 0.000000 seconds
```

```
Enter the number of rows and columns in matrix-A -----> 12 12
Enter the number of rows and columns in matrix-B -----> 12 12

All cells in the matrices were initialized with 1.....

Matrix AxB as obtained-

12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12
12 12 12 12 12 12 12 12 12 12 12 12

Time taken to perform the strassen's multiplication: 0.007000 seconds
Time taken to perform the naive matrix multiplication: 0.000000 seconds
```

```
Enter the number of rows and columns in matrix-A -----> 15 19
Enter the number of rows and columns in matrix-B -----> 19 12

All cells in the matrices were initialized with 1.....

Matrix AxB as obtained-

19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19
19 19 19 19 19 19 19 19 19 19 19 19

Time taken to perform the strassen's multiplication: 0.029000 seconds
Time taken to perform the naive matrix multiplication: 0.000000 seconds
```

```
Enter the number of rows and columns in matrix-A -----> 50 50
Enter the number of rows and columns in matrix-B -----> 50 50

All cells in the matrices were initialized with 1.....

Time taken to perform the strassen's multiplication: 0.186000 seconds
Time taken to perform the naive matrix multiplication: 0.000000 seconds
```

```
Enter the number of rows and columns in matrix-A -----> 200 200
Enter the number of rows and columns in matrix-B -----> 200 175

All cells in the matrices were initialized with 1.....

Time taken to perform the strassen's multiplication: 8.681000 seconds
Time taken to perform the naive matrix multiplication: 0.017000 seconds
```

Inference:

Upon observation, I noticed that Naïve method of matrix multiplication usually takes negligible amount of time when dealing with matrices with order below 100. Even for order above 100, it usually provides the result in milliseconds. On the other hand, Strassen’s method of matrix multiplication comparatively takes much longer for smaller matrices as well as larger matrices to provide the result.

Theoretically speaking, Naïve method has a time complexity of O(n^3) while Strassen's method has a time complexity of O($n^{\log_2 7}$) which approximates to O($n^{2.808}$).

Hence, one might think that Strassen's method would be faster in general. However, there are multiple things that should be taken into consideration before arriving at a conclusion. Even though Strassen's method has a better time complexity, it has a much larger constant factor attached to it. Also, it makes use of more memory than the Naïve method. Hence the speed of this algorithm would also depend on the hardware it is tested on. In my case, Naïve method was found to be generally faster, however it would defer from hardware to hardware.

Conclusion:

By performing this experiment, I was able to understand Strassen's Matrix Multiplication algorithm. I was also able to implement the same and compare it with Naïve method of matrix multiplication.