Matrix Multiplication Using Divide And Conquer

Aim: The aim of the experiment is to implement the Strassen algorithm for efficient matrix multiplication and demonstrate its performance compared to the conventional matrix multiplication approach.

Theory:
The Strassen algorithm is a divide-and-conquer method for matrix multiplication that reduces the number of arithmetic operations required compared to the standard matrix multiplication algorithm (O(n^3) for conventional matrix multiplication). Strassen's algorithm divides the input matrices into smaller submatrices, recursively multiplies these submatrices, and combines the results to obtain the final product. This approach reduces the number of multiplications from 8 to 7 for each submatrix, resulting in an overall time complexity of O(n^log2(7)), which is approximately O(n^2.81).

Brute Force Method

Code:

```
#include <stdio.h>
 #define MAX_SIZE 10
void Multiply(int a[MAX_SIZE][MAX_SIZE], int b[MAX_SIZE][MAX_SIZE], int result[MAX_SIZE][MAX_SIZE], int r, int c) {
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            result[i][j] = 0;
            for (int k = 0; k < c; k++) { // Assuming square matrices for simplicity
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}
 void PrintMatrix(int matrix[MAX_SIZE][MAX_SIZE], int r, int c) {
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
 int main() {
    int r, c;
                int a[MAX_SIZE][MAX_SIZE], b[MAX_SIZE][MAX_SIZE], result[MAX_SIZE][MAX_SIZE];
     printf("Enter the number of rows and columns of matrix: ");
    scanf("%d%d", &r, &c);
    printf("Enter the elements of matrix A:\n");
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            scanf("%d", &a[i][j]);
        }
    }
     printf("Enter the elements of matrix B:\n");
```

```
    for (int i = 0; i < r; i++) {
        for (int j = 0; j < c; j++) {
            scanf("%d", &b[i][j]);
        }
    }

    // No need to input matrix C, as it will be the result of A * B
    Multiply(a, b, result, r, c);

    printf("Resultant matrix C after multiplication:\n");
    PrintMatrix(result, r, c);

    return 0;
}
```

Output:

```
zoro-d-code@wado-ichimonji:~/Roger/temp$ cd "/home/zoro-d-code/Roger/temp/" && gcc mat_mult.c -o mat_mult && "/home/zoro-d-code/
Roger/temp/"mat_mult
Enter the number of rows and columns of matrix: 2 2
Enter the elements of matrix A:
1 2 3 4
Enter the elements of matrix B:
5 6 7 8
Resultant matrix C after multiplication:
19 22
43 50
```

Algorithm Steps:

1. Input Matrices:
- Accept the number of rows and columns (r, c) for the matrices A and B.
- Prompt the user to input the elements of matrices A and B.

2. Matrix Multiplication (Multiply):
- Initialize an empty result matrix to store the product of matrices A and B.
- Iterate through each element of the result matrix:
  - For each element at position (i, j), initialize it to 0.
  - Perform dot product of the ith row of matrix A and the jth column of matrix B to calculate the value of the element at position (i, j) in the result matrix.
  - Update the element at position (i, j) in the result matrix with the calculated value.

3. Print Resultant Matrix (PrintMatrix):
- Iterate through the result matrix and print each element.
- Print a newline character after printing each row to display the matrix properly.

4. Main Function:
- Declare variables for matrices A, B, and the resultant matrix.
- Accept input for the number of rows and columns of the matrices A and B.
- Input the elements of matrices A and B.
- Call the Multiply function with matrices A, B, and the resultant matrix, along with the number of rows and columns.
- Print the resultant matrix after multiplication.

5. Output:
- Display the resultant matrix C, which is the product of matrices A and B.

Time Complexity Analysis:

1. **Input Matrices:**
- Accepting input for the matrices A and B requires traversing each element once, which takes O(r * c) time, where 'r' is the number of rows and 'c' is the number of columns.

2. **Matrix Multiplication (Multiply):**
- The multiplication of two matrices A and B involves three nested loops:
- The outer loop iterates 'r' times (number of rows of A).
- The middle loop iterates 'c' times (number of columns of B).
- The inner loop iterates 'c' times (assuming square matrices for simplicity).
- Inside the innermost loop, there's a constant-time operation (addition and multiplication) performed 'c' times.
- Therefore, the time complexity of the multiplication step is O(r * c * c).

3. **Print Resultant Matrix (PrintMatrix):**
- Printing the elements of the resultant matrix takes O(r * c) time, where 'r' is the number of rows and 'c' is the number of columns.

4. **Main Function:**
- In the main function, the time complexity is dominated by the matrix multiplication step.
- The time complexity of accepting input for matrices and printing the resultant matrix is comparatively negligible compared to the matrix multiplication operation.
- Hence, the overall time complexity of the algorithm is O(r * c * c).

Space Complexity Analysis:

1. **Input Matrices:**
- Space required to store the input matrices A and B: O(r * c) + O(r * c) = O(r * c), where 'r' is the number of rows and 'c' is the number of columns.

2. **Resultant Matrix:**
- Space required to store the resultant matrix C: O(r * c), where 'r' is the number of rows and 'c' is the number of columns.

3. **Other Variables:**
- Additional space required for loop indices, temporary variables, and function parameters is constant and does not depend on the size of the input matrices. Therefore, it can be considered negligible.

4. **Total Space Complexity:**
- The dominant contributors to space complexity are the input matrices A and B, as well as the resultant matrix C.
- Hence, the overall space complexity of the algorithm is O(r * c), where 'r' is the number of rows and 'c' is the number of columns in the input matrices.

Strassen’s Matrix Multiplication Method

Code:

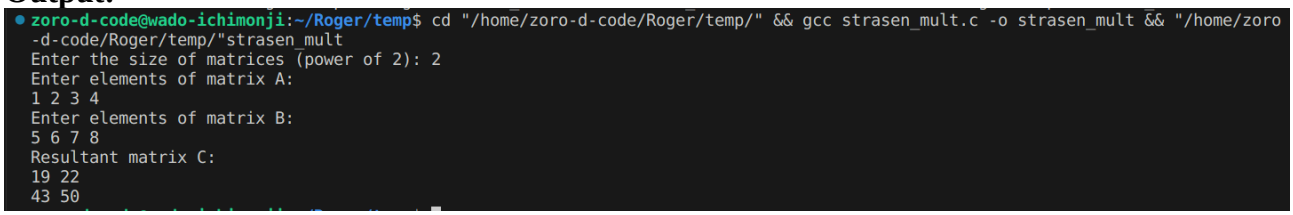
```
#include <stdio.h>
```

```
#include <stdlib.h>

// Function to add two matrices
void matrixAddition(int n, int A[][n], int B[][n], int C[][n]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] = A[i][j] + B[i][j];
}

// Function to subtract two matrices
void matrixSubtraction(int n, int A[][n], int B[][n], int C[][n]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] = A[i][j] - B[i][j];
}

// Function to multiply two matrices using Strassen algorithm
void strassen(int n, int A[][n], int B[][n], int C[][n]) {
    if (n == 1) {
        C[0][0] = A[0][0] * B[0][0];
        return;
    }

    int newSize = n / 2;
      int A11[newSize][newSize], A12[newSize][newSize], A21[newSize][newSize], A22[newSize]
[newSize];
      int B11[newSize][newSize], B12[newSize][newSize], B21[newSize][newSize], B22[newSize]
[newSize];
      int C11[newSize][newSize], C12[newSize][newSize], C21[newSize][newSize], C22[newSize]
[newSize];
       int M1[newSize][newSize], M2[newSize][newSize], M3[newSize][newSize], M4[newSize]
[newSize];
    int M5[newSize][newSize], M6[newSize][newSize], M7[newSize][newSize];
    int temp1[newSize][newSize], temp2[newSize][newSize];

    // Divide matrices A and B into submatrices
    for (int i = 0; i < newSize; i++) {
        for (int j = 0; j < newSize; j++) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + newSize];
            A21[i][j] = A[i + newSize][j];
            A22[i][j] = A[i + newSize][j + newSize];
            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + newSize];
            B21[i][j] = B[i + newSize][j];
            B22[i][j] = B[i + newSize][j + newSize];
        }
    }

    // Calculating M1 to M7
    matrixAddition(newSize, A11, A22, temp1);
    matrixAddition(newSize, B11, B22, temp2);
```

```
    strassen(newSize, temp1, temp2, M1);

    matrixAddition(newSize, A21, A22, temp1);
    strassen(newSize, temp1, B11, M2);

    matrixSubtraction(newSize, B12, B22, temp1);
    strassen(newSize, A11, temp1, M3);

    matrixSubtraction(newSize, B21, B11, temp1);
    strassen(newSize, A22, temp1, M4);

    matrixAddition(newSize, A11, A12, temp1);
    strassen(newSize, temp1, B22, M5);

    matrixSubtraction(newSize, A21, A11, temp1);
    matrixAddition(newSize, B11, B12, temp2);
    strassen(newSize, temp1, temp2, M6);

    matrixSubtraction(newSize, A12, A22, temp1);
    matrixAddition(newSize, B21, B22, temp2);
    strassen(newSize, temp1, temp2, M7);

    // Calculating C matrices
    matrixAddition(newSize, M1, M4, temp1);
    matrixSubtraction(newSize, temp1, M5, temp2);
    matrixAddition(newSize, temp2, M7, C11);

    matrixAddition(newSize, M3, M5, C12);

    matrixAddition(newSize, M2, M4, C21);

    matrixAddition(newSize, M1, M3, temp1);
    matrixSubtraction(newSize, temp1, M2, temp2);
    matrixAddition(newSize, temp2, M6, C22);

    // Combining C matrices into one
    for (int i = 0; i < newSize; i++) {
        for (int j = 0; j < newSize; j++) {
            C[i][j] = C11[i][j];
            C[i][j + newSize] = C12[i][j];
            C[i + newSize][j] = C21[i][j];
            C[i + newSize][j + newSize] = C22[i][j];
        }
    }
}

// Function to print a matrix
void printMatrix(int n, int matrix[][n]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", matrix[i][j]);
        }
```

```
            printf("\n");
        }
    }

    int main() {
        int n;
        printf("Enter the size of matrices (power of 2): ");
        scanf("%d", &n);

        int A[n][n], B[n][n], C[n][n];

        printf("Enter elements of matrix A:\n");
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                scanf("%d", &A[i][j]);

        printf("Enter elements of matrix B:\n");
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                scanf("%d", &B[i][j]);

        strassen(n, A, B, C);

        printf("Resultant matrix C:\n");
        printMatrix(n, C);

        return 0;
    }
```

Output:



Algorithm Steps:

1. **Input Matrices:**
- Accept the size of the square matrices (power of 2) 'n'.
- Prompt the user to input the elements of matrices A and B of size 'n x n'.

2. **Strassen Algorithm (strassen function):**
- Check if the size of the matrices is reduced to 1 x 1.
- If true, perform simple multiplication and store the result in matrix C.
- Otherwise, proceed with Strassen algorithm steps.
- Divide matrices A and B into four submatrices each: A11, A12, A21, A22, and B11, B12, B21, B22, respectively.
- Compute seven intermediate matrices (M1 to M7) using matrix additions, subtractions, and recursive calls to Strassen's algorithm.

- Calculate the four quadrants of the resultant matrix C (C11, C12, C21, and C22) by combining the intermediate matrices.

3. **Print Resultant Matrix (printMatrix function):**
- Print the resultant matrix C, which represents the product of matrices A and B.

4. **Main Function:**
- Declare variables for matrices A, B, and the resultant matrix C.
- Accept input for the size of matrices and their elements.
- Call the strassen function with matrices A, B, and the resultant matrix C.
- Print the resultant matrix C after multiplication.

5. **Output:**
- Display the resultant matrix C, which represents the product of matrices A and B using the Strassen algorithm.

Time Complexity:

1. **Matrix Division:**
- The algorithm begins by dividing the input matrices A and B into four equal-sized submatrices, each of size n/2 x n/2.
- This step involves copying the elements of the input matrices into smaller submatrices, which takes constant time and can be ignored in the overall time complexity analysis.

2. **Intermediate Matrix Computations:**
- The algorithm computes seven intermediate matrices (M1 to M7) using matrix additions, subtractions, and recursive calls.
- Each intermediate matrix is calculated using arithmetic operations on submatrices, which involve traversing through n/2 x n/2 matrices.
- Since each intermediate matrix involves a constant number of arithmetic operations (additions, subtractions, and multiplications), their computation time can be considered proportional to O(n^2).

3. **Recursive Calls:**
- After dividing the matrices into smaller submatrices, the algorithm recursively calls itself on each of the submatrices.
- The recurrence relation for the time complexity of the recursive calls is T(n) = 7T(n/2), where each call processes matrices of half the size.
- The recursive calls proceed until the base case is reached when the matrices become 1 x 1, at which point a simple multiplication operation is performed.

4. **Combining Intermediate Matrices:**
- Once the intermediate matrices are computed, the algorithm combines them to calculate the four quadrants of the resultant matrix C.
- Combining the matrices involves additional matrix additions and subtractions, which take O(n^2) time.

5. **Base Case Handling:**
- At each level of recursion, the algorithm checks if the size of the matrices has reduced to 1 x 1.
- If true, a simple multiplication operation is performed to calculate the base case, which takes constant time.

6. **Overall Time Complexity:**

- The time complexity of the Strassen algorithm is determined by the recurrence relation T(n) = 7T(n/2) + O(n^2), where T(n) represents the time taken to multiply two square matrices of size n x n.
- Using the Master theorem for divide-and-conquer recurrences, the time complexity is approximately $O(n^{\log_2(7)}) \approx O(n^{2.81})$.

Space Complexity:

1. **Input Matrices:**
   - Space required to store the input matrices A and B of size n x n: O(n^2) + O(n^2) = O(n^2), where 'n' is the size of the input matrices.

2. **Intermediate Matrices:**
   - The algorithm computes seven intermediate matrices (M1 to M7) during its execution, each of size n/2 x n/2.
   - Space required to store each intermediate matrix: O((n/2)^2) = O(n^2/4) = O(n^2).
   - Since there are seven intermediate matrices, the total space complexity for intermediate matrices is 7 * O(n^2) = O(n^2).

3. **Submatrices:**
   - During the recursive calls, the input matrices are divided into smaller submatrices at each level.
   - These submatrices are stored separately, requiring additional space.
   - The space required for storing submatrices is proportional to the space required for the input matrices.
   - As the algorithm progresses, the number of submatrices created and stored also increases.

4. **Stack Space:**
   - Recursive calls in the algorithm lead to the activation of multiple function calls on the stack.
   - Each function call consumes additional space on the stack for storing local variables, function arguments, and return addresses.
   - The depth of the recursion tree is $\log_2(n)$, where 'n' is the size of the input matrices, leading to $\log_2(n)$ levels of function calls on the stack.
   - Therefore, the space complexity due to stack space is O(log(n)).

5. **Overall Space Complexity:**
   - The dominant contributors to space complexity are the input matrices, intermediate matrices, and stack space.
   - The overall space complexity of the Strassen algorithm is the sum of these components: O(n^2) + O(n^2) + O(log(n)) = O(n^2).