

Ashrut Alok Arora

210968206

DSE-A - Batch A2

# **PARALLEL PROGRAMMING LAB**

## Week-1

### Question 1

Write a program in C to reverse the digits of the following integer array of size 9.  
Initialize the input array to the following values.

Input array: 18, 523, 301, 1234, 2, 14, 108, 150, 1928

Output array: 81, 325, 103, 4321, 2, 41, 801, 51, 8291

```
#include <stdio.h>

void reverseDigits(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        int num = arr[i];
        int reversed = 0;

        while (num > 0)
        {
            reversed = reversed * 10 + num % 10;
            num /= 10;
        }
    }
}
```

```
        arr[i] = reversed;
    }
}

int main()
{
    int size;

    // Getting the size of the array from the user
    printf("Enter the size of the array: ");
    scanf("%d", &size);

    int inputArray[size];

    // Getting array elements from the user
    printf("Enter %d integers for the array:\n", size);
    for (int i = 0; i < size; i++)
    {
        scanf("%d", &inputArray[i]);
    }

    // Displaying the input array
    printf("Input array: ");
    for (int i = 0; i < size; i++)
    {
        printf("%d ", inputArray[i]);
    }
    printf("\n");

    // Reversing the digits
```

```
reverseDigits(inputArray, size);

// Displaying the output array
printf("Output array: ");
for (int i = 0; i < size; i++)
{
    printf("%d ", inputArray[i]);
}
printf("\n");

return 0;
}
```

Outputs:

Enter the size of the array: 6  
Enter 6 integers for the array:  
56 123 789 456 321 987  
Input array: 56 123 789 456 321 987  
Output array: 65 321 987 654 123 789

Enter the size of the array: 6  
Enter 6 integers for the array:  
876 543 210 987 654 321  
Input array: 876 543 210 987 654 321  
Output array: 678 345 012 789 456 123

### Question 2

Write a program in C to simulate the all the operations of a calculator.

Given inputs A and B, find the output for A+B, A-B, A\*B and A/B.

```
#include <stdio.h>

int main() {
    // Declare variables for input
    double A, B;

    // Get input from the user
    printf("Enter the value of A: ");
    scanf("%lf", &A);

    printf("Enter the value of B: ");
    scanf("%lf", &B);

    // Perform calculations
    double sum = A + B;
    double difference = A - B;
    double product = A * B;

    // Check if B is not zero to avoid division by zero
    double quotient = (B != 0) ? A / B : 0;

    // Display the results
    printf("A + B = %.2f\n", sum);
    printf("A - B = %.2f\n", difference);
    printf("A * B = %.2f\n", product);
```

```
// Display division result only if B is not zero
if (B != 0) {
    printf("A / B = %.2f\n", quotient);
} else {
    printf("A / B = Cannot divide by zero.\n");
}

return 0;
}
```

Outputs:

Enter the value of A: 10.5  
Enter the value of B: 3.5  
A + B = 14.00  
A - B = 7.00  
A \* B = 36.75  
A / B = 3.00

Enter the value of A: 15.8  
Enter the value of B: 0  
A + B = 15.80  
A - B = 15.80  
A \* B = 0.00  
A / B = Cannot divide by zero.

Question 3

Write a program in C to toggle the character of a given string. Example:

Input = "HeLLo"

Output = "hElLO"

```
#include <stdio.h>

void toggleCase(char str[]) {
    for (int i = 0; str[i] != '\0'; i++) {
        if (str[i] >= 'A' && str[i] <= 'Z') {
            // If uppercase, convert to lowercase
            str[i] = str[i] + ('a' - 'A');
        } else if (str[i] >= 'a' && str[i] <= 'z') {
            // If lowercase, convert to uppercase
            str[i] = str[i] - ('a' - 'A');
        }
        // Ignore non-alphabetic characters
    }
}

int main() {
    // Declare a string
    char inputString[100];

    // Get input from the user
    printf("Enter a string: ");
    scanf("%s", inputString);

    // Toggle the case of each character
    toggleCase(inputString);
```

```
// Display the result
printf("Toggled Case: %s\n", inputString);

return 0;
}
```

#### Outputs:

Enter a string: HelloWorld123  
Toggled Case: hELLOwORLD123

Enter a string: Data Science is Fun!  
Toggled Case: dATA sCIENCE IS fUN!

#### Question 4

Write a C program to read a word of length N and produce the pattern as shown in the example.

Example:

Input: PCBD

Output: PCCBBBDDDD

```
#include <stdio.h>

void printPattern(char str[]) {
    int i, j;
    for (i = 0; str[i] != '\0'; i++) {
        for (j = 0; j <= i; j++) {
            printf("%c", str[i]);
        }
    }
}
```

```

    }
}

int main() {
    // Declare a string
    char inputString[100];

    // Get input from the user
    printf("Enter a word: ");
    scanf("%s", inputString);

    // Display the pattern
    printf("Pattern: ");
    printPattern(inputString);
    printf("\n");

    return 0;
}

```

#### Outputs:

Enter a word: PROGRAM

Pattern: PRRRRRROOOOOOOGGGGGGRRRRRRRAAAAAAAAAAMMMMMMMMMMM

Enter a word: CODE

Pattern: CCOODDEE

#### Question 5



Write a C program to read two strings S1 and S2 of same length and produce the resultant string as shown below.

Example:

S1: string S2: length

Resultant String: slternigntgh

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

void generateResultantString(char s1[], char s2[], char result[], int length) {
    int i;
    for (i = 0; i < length; i++) {
        result[i * 2] = s1[i];
        result[i * 2 + 1] = s2[i];
    }
    result[i * 2] = '\0'; // Add null terminator to end the string
}

int main() {
    // Declare strings
    char s1[100], s2[100], resultantString[200];

    // Get input from the user
    printf("Enter the first string (S1): ");
    scanf("%s", s1);

    printf("Enter the second string (S2): ");
    scanf("%s", s2);

    // Check if the strings are of the same length
```

```
if (strlen(s1) != strlen(s2)) {  
    printf("Error: Strings must be of the same length.\n");  
    return 1; // Return an error code  
}  
  
// Generate the resultant string  
generateResultantString(s1, s2, resultantString, strlen(s1));  
  
// Display the resultant string  
printf("Resultant String: %s\n", resultantString);  
  
return 0;  
}
```

Outputs:

Enter the first string (S1): Hello  
Enter the second string (S2): World  
Resultant String: HWeolrldo

Enter the first string (S1): Data  
Enter the second string (S2): Science  
Resultant String: Dseaciatnece

Question 6

Write a C program to perform Matrix times vector product operation

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

void matrixVectorProduct(int size, int matrix[size][size], int vector[size], int result[size]) {
    #pragma omp parallel for
    for (int i = 0; i < size; i++) {
        result[i] = 0;
        for (int j = 0; j < size; j++) {
            result[i] += matrix[i][j] * vector[j];
        }
    }
}

int main() {
    // Get input for the array size from the user
    int size;
    printf("Enter the size of the matrix and vector: ");
    scanf("%d", &size);

    // Declare arrays for the matrix, vector, and result
    int matrix[size][size];
    int vector[size];
    int result[size];

    // Get input for the matrix from the user
    printf("Enter the matrix:\n");
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            printf("Enter element at position (%d, %d): ", i + 1, j + 1);
            scanf("%d", &matrix[i][j]);
        }
    }
}
```

```

    }
}

// Get input for the vector from the user
printf("Enter the vector of size %d:\n", size);
for (int i = 0; i < size; i++) {
    printf("Enter element at position %d: ", i + 1);
    scanf("%d", &vector[i]);
}

// Perform matrix-vector product
matrixVectorProduct(size, matrix, vector, result);

// Display the result
printf("Matrix times Vector Result: ");
for (int i = 0; i < size; i++) {
    printf("%d ", result[i]);
}
printf("\n");

return 0;
}

```

### Outputs:

Enter the size of the matrix and vector: 3

Enter the matrix:

Enter element at position (1, 1): 1

Enter element at position (1, 2): 2

Enter element at position (1, 3): 3

Enter element at position (2, 1): 4

Enter element at position (2, 2): 5

Enter element at position (2, 3): 6

Enter element at position (3, 1): 7

Enter element at position (3, 2): 8

Enter element at position (3, 3): 9

Enter the vector of size 3:

Enter element at position 1: 2

Enter element at position 2: 1

Enter element at position 3: 3

Matrix times Vector Result: 13 32 51

Enter the size of the matrix and vector: 2

Enter the matrix:

Enter element at position (1, 1): 2

Enter element at position (1, 2): 1

Enter element at position (2, 1): 3

Enter element at position (2, 2): 4

Enter the vector of size 2:

Enter element at position 1: 1

Enter element at position 2: 2

Matrix times Vector Result: 4 14

### Question 7

Write a C program to read a matrix A of size 5x5. It produces a resultant matrix B of size 5x5. It sets all the principal diagonal elements of B matrix with 0. It replaces each row elements in the B matrix in the following manner. If the element is below the principal diagonal it replaces it with the maximum value of the row in the A matrix having the

same row number of B. If the element is above the principal diagonal it replaces it with the minimum value of the row in the A matrix having the same row number of B

### Example:

| A  |    |    |    |    |
|----|----|----|----|----|
| 1  | 2  | 3  | 4  | 5  |
| 5  | 4  | 3  | 2  | 4  |
| 10 | 3  | 13 | 14 | 15 |
| 11 | 2  | 11 | 33 | 44 |
| 1  | 12 | 5  | 4  | 6  |

Output

| B  |    |    |    |   |
|----|----|----|----|---|
| 0  | 1  | 1  | 1  | 1 |
| 5  | 0  | 2  | 2  | 2 |
| 15 | 15 | 0  | 3  | 3 |
| 44 | 44 | 44 | 0  | 2 |
| 12 | 12 | 12 | 12 | 0 |

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

```
#define SIZE 5
```

```
void printMatrix(int matrix[SIZE][SIZE])
```

```
{
```

```
    for (int i = 0; i < SIZE; i++)
```

```
    {
```

```
        for (int j = 0; j < SIZE; j++)
```

```
        {
```

```
            printf("%d ", matrix[i][j]);
```

```
        }
```

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

```

    }
}

void processMatrix(int A[SIZE][SIZE], int B[SIZE][SIZE])
{
    // Initialize B with zeros on the principal diagonal
    for (int i = 0; i < SIZE; i++)
    {
        B[i][i] = 0;
    }

    // Process each element in B based on the given rules
    for (int i = 0; i < SIZE; i++)
    {

        for (int j = 0; j < SIZE; j++)
        {

            if (j < i)
            {
                // Below principal diagonal
                int maxRowValue = A[i][0];

                for (int k = 1; k < SIZE; k++)
                {

                    if (A[i][k] > maxRowValue)
                    {
                        maxRowValue = A[i][k];
                    }
                }
            }
        }
    }
}

```

```

    }
    B[i][j] = maxRowValue;
}
else if (j > i)
{
    // Above principal diagonal
    int minRowValue = A[i][0];

    for (int k = 1; k < SIZE; k++)
    {

        if (A[i][k] < minRowValue)
        {
            minRowValue = A[i][k];
        }
    }
    B[i][j] = minRowValue;
}
}
}
}

```

```

int main()
{
    int matrixA[SIZE][SIZE] = {
        {1, 2, 3, 4, 5},
        {5, 4, 3, 2, 4},
        {10, 3, 13, 14, 15},
        {11, 2, 11, 33, 44},
        {1, 12, 5, 4, 6}};
}

```



```
int matrixB[SIZE][SIZE];

processMatrix(matrixA, matrixB);

printf("\nMatrix B:\n");
printMatrix(matrixB);

return 0;
}
```

### Outputs:

Matrix A:

```
1 2 3 4 5
5 4 3 2 4
10 3 13 14 15
11 2 11 33 44
1 12 5 4 6
```

Matrix B:

```
0 0 0 0 0
5 0 0 0 0
10 3 0 0 0
11 2 11 0 0
1 12 5 4 0
```

Matrix A:

2 5 8 4 1

6 1 7 3 9

4 2 8 7 5

3 6 2 8 4

1 9 3 5 7

Matrix B:

0 0 0 0 0

6 0 0 0 0

4 2 0 0 0

3 2 3 0 0

1 6 2 4 0

### Question 8

Write a C program that reads a matrix of size  $M \times N$  and produce an output matrix B of same size such that it replaces all the non-border elements of A with its equivalent 1's complement and remaining elements same as matrix A. Also produce a matrix D as shown below.

A

|   |   |    |   |
|---|---|----|---|
| 1 | 2 | 3  | 4 |
| 6 | 5 | 8  | 3 |
| 2 | 4 | 10 | 1 |
| 9 | 1 | 2  | 5 |

B

|   |           |            |   |
|---|-----------|------------|---|
| 1 | 2         | 3          | 4 |
| 6 | <b>10</b> | <b>111</b> | 3 |
| 2 | <b>11</b> | <b>101</b> | 1 |
| 9 | 1         | 2          | 5 |

D

|   |          |          |   |
|---|----------|----------|---|
| 1 | 2        | 3        | 4 |
| 6 | <b>2</b> | <b>7</b> | 3 |
| 2 | <b>3</b> | <b>5</b> | 1 |
| 9 | 1        | 2        | 5 |

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

```
#define M 4
```

```
#define N 4
```

```
void printBinary(int num) {
```

```
    int mask = 1 << 3; // considering 4 bits as the maximum value in your matrix is 10  
    (1010 in binary)
```

```
    while(mask > 0) {
```

```
        printf("%d", (num & mask) ? 1 : 0);
```

```
        mask >>= 1;
```

```
}
```

```
}
```

```
int binaryToDecimal(int binary) {
```

```
    int decimal = 0, base = 1;
```

```
    while (binary > 0) {
```

```
        decimal += (binary % 10) * base;
```

```
        binary /= 10;
```

```
        base *= 2;
```

```
    }
```

```
    return decimal;
```

```
}
```

```
int main() {
```

```
    int A[M][N] = {{1, 2, 3, 4}, {6, 5, 8, 3}, {2, 4, 10, 1}, {9, 1, 2, 5}};
```

```
    printf("Matrix B (Binary):\n");
```

```
    for(int i=0; i<M; i++) {
```

```
        for(int j=0; j<N; j++) {
```

```
            if(i==0 || i==M-1 || j==0 || j==N-1) {
```

```
                printf("%d ", A[i][j]);
```

```
            } else {
```

```
                printBinary(~A[i][j] & 0x0F); // Mask the higher bits
```

```
                printf(" ");
```

```
            }
```

```
        }
```

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

```
    }
```

```
    printf("\nMatrix D (Decimal):\n");
```

```
    for(int i=0; i<M; i++) {
```

```

for(int j=0; j<N; j++) {
    if(i==0 || i==M-1 || j==0 || j==N-1) {
        printf("%d ", A[i][j]);
    } else {
        int binaryValue = ~A[i][j] & 0x0F; // Mask the higher bits
        int decimalValue = binaryToDecimal(binaryValue);
        printf("%d ", decimalValue);
    }
}
printf("\n");
}

return 0;
}

```

### Outputs:

Matrix A:

1 2 3 4

6 5 8 3

2 4 10 1

9 1 2 5

Matrix B (Binary):

1 2 3 4

6 0101 1000 3

2 0100 0001 1

9 1 2 5

Matrix D (Decimal):

1 2 3 4

6 5 8 3

2 4 10 1

9 1 2 5

Matrix A:

5 7 2 4

8 9 12 3

1 5 6 7

4 2 8 1

Matrix B (Binary):

5 7 2 4

8 1001 1100 3

1 1010 1001 7

4 1110 0111 1

Matrix D (Decimal):

5 7 2 4

8 9 12 3

1 5 6 7

4 2 8 1

### Question 9

Write a C program that reads a character type matrix and integer type matrix B of size  $M \times N$ . It produces and output string STR such that, every character of A is repeated r

times (where r is the integer value in matrix B which is having the same index as that of the character taken in A).

| Example: | A     | B       |
|----------|-------|---------|
| p        | C a P | 1 2 4 3 |
| e        | X a M | 2 4 3 2 |

**Output string STR: pCCaaaaPPPeXXXXaaaMM**

```
#include <stdio.h>

int main()
{
    int M, N;

    // Input matrix A
    printf("Enter the number of rows (M) and columns (N) for matrix A: ");
    scanf("%d %d", &M, &N);

    char matrixA[M][N];

    printf("Enter the characters for matrix A:\n");
    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            scanf(" %c", &matrixA[i][j]);
        }
    }

    // Input matrix B
```

```
int matrixB[M][N];

printf("Enter the integers for matrix B:\n");
for (int i = 0; i < M; i++)
{
    for (int j = 0; j < N; j++)
    {
        scanf("%d", &matrixB[i][j]);
    }
}

// Output string STR
printf("Output string STR: ");
for (int i = 0; i < M; i++){
    for (int j = 0; j < N; j++)
    {
        for (int k = 0; k < matrixB[i][j]; k++)
        {
            printf("%c", matrixA[i][j]);
        }
    }
}
printf("\n");
return 0;
}
```

Outputs:

Enter the number of rows (M) and columns (N) for matrix A: 3 4  
Enter the characters for matrix A:



a b c d

e f g h

i j k l

Enter the integers for matrix B:

2 3 1 0

4 1 2 3

0 2 1 4

Output string STR: aabbcccffghhijklkkkk

Enter the number of rows (M) and columns (N) for matrix A: 2 3

Enter the characters for matrix A:

X Y Z

A B C

Enter the integers for matrix B:

1 0 3

2 1 2

Output string STR: XYZCC

## Week-2(Parallelize the Programs of Week-1)

### Question 1

Write a program in C to reverse the digits of the following integer array of size 9.  
Initialize the input array to the following values.

Input array: 18, 523, 301, 1234, 2, 14, 108, 150, 1928

Output array: 81, 325, 103, 4321, 2, 41, 801, 51, 8291

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

void reverseDigits(int *arr, int size) {
    #pragma omp parallel for
    for (int i = 0; i < size; i++) {
        int num = arr[i];
        int reversed = 0;

        while (num > 0) {
            reversed = reversed * 10 + num % 10;
            num /= 10;
        }

        arr[i] = reversed;
    }
}

int main() {
    int size;
    printf("Enter the size of the array: ");
    scanf("%d", &size);
```

```

int* inputArray = (int*)malloc(size * sizeof(int));
if (inputArray == NULL) {
    fprintf(stderr, "Memory allocation failed.\n");
    return 1;
}

printf("Enter %d integers for the array:\n", size);
for (int i = 0; i < size; i++) {
    scanf("%d", &inputArray[i]);
}

// Set a constant value for the number of threads
int numThreads = 4; // You can adjust this value based on your system

// Measure time before parallel execution
clock_t startClock = clock();

// Parallel execution to reverse digits
#pragma omp parallel num_threads(numThreads)
{
    reverseDigits(inputArray, size);
}

// Measure time after parallel execution
clock_t endClock = clock();
double elapsedTime = ((double)(endClock - startClock)) / CLOCKS_PER_SEC;

// Calculate speedup and efficiency
double serialTime = elapsedTime / numThreads; // Assuming serial execution time is
the same as elapsed time

```

```

double parallelTime = elapsedTime;
double speedup = serialTime / parallelTime;
double efficiency = speedup / numThreads;

// Display the reversed array
printf("Reversed Array: ");
for (int i = 0; i < size; i++) {
    printf("%d ", inputArray[i]);
}
printf("\n");

// Display elapsed time, speedup, and efficiency
printf("Elapsed Time: %.6f seconds\n", elapsedTime);
printf("Speedup: %.2f\n", speedup);
printf("Efficiency: %.2f\n", efficiency);

// Free allocated memory
free(inputArray);

return 0;
}

```

### Outputs:

```

Enter the size of the array: 6
Enter 6 integers for the array:
56 123 789 456 321 987
Input array: 56 123 789 456 321 987

```

Output array: 65 321 987 654 123 789

Elapsed Time: 0.000003 seconds

Speedup: 0.25

Efficiency: 0.06

Enter the size of the array: 6

Enter 6 integers for the array:

876 543 210 987 654 321

Input array: 876 543 210 987 654 321

Output array: 678 345 012 789 456 123

Elapsed Time: 0.000001 seconds

Speedup: 0.25

Efficiency: 0.06

## Question 2

Write a program in C to simulate the all the operations of a calculator.

Given inputs A and B, find the output for A+B, A-B, A\*B and A/B.

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

```
#include <omp.h>
```

```
#include <time.h>
```

```
int main() {
```

```
    int A, B;
```

```
// Input
printf("Enter value for A: ");
scanf("%d", &A);

printf("Enter value for B: ");
scanf("%d", &B);

// Serial execution
printf("Serial execution:\n");
clock_t serial_start_time = clock();

// Addition
printf("%d + %d = %d\n", A, B, A + B);

// Subtraction
printf("%d - %d = %d\n", A, B, A - B);

// Multiplication
printf("%d * %d = %d\n", A, B, A * B);

// Division
if (B != 0) {
    printf("%d / %d = %d\n", A, B, A / B);
} else {
    printf("Cannot divide by zero.\n");
}

clock_t serial_end_time = clock();

double serial_elapsed_time = ((double)(serial_end_time - serial_start_time)) /
CLOCKS_PER_SEC;
```

```
printf("Serial execution time: %f seconds\n\n", serial_elapsed_time);
```

```
// Parallel execution
```

```
printf("Parallel execution:\n");
```

```
for (int num_threads = 2; num_threads <= 4; num_threads++) {
```

```
    // Start the clock
```

```
    clock_t parallel_start_time = clock();
```

```
    #pragma omp parallel num_threads(num_threads)
```

```
    #pragma omp sections
```

```
{
```

```
    // Addition
```

```
    #pragma omp section
```

```
{
```

```
    printf("%d + %d = %d\n", A, B, A + B);
```

```
}
```

```
    // Subtraction
```

```
    #pragma omp section
```

```
{
```

```
    printf("%d - %d = %d\n", A, B, A - B);
```

```
}
```

```
    // Multiplication
```

```
    #pragma omp section
```

```
{
```

```
    printf("%d * %d = %d\n", A, B, A * B);
```

```
}
```

```

// Division
#pragma omp section
{
    if (B != 0) {
        printf("%d / %d = %d\n", A, B, A / B);
    } else {
        printf("Cannot divide by zero.\n");
    }
}

// End the clock
clock_t parallel_end_time = clock();

// Calculate the elapsed time
double parallel_elapsed_time = ((double)(parallel_end_time - parallel_start_time)) /
CLOCKS_PER_SEC;

printf("Parallel execution time with %d threads: %f seconds\n", num_threads,
parallel_elapsed_time);

// Calculate speedup and efficiency
double speedup = serial_elapsed_time / parallel_elapsed_time;
double efficiency = speedup / num_threads;

printf("Speedup: %f\n", speedup);
printf("Efficiency: %f\n\n", efficiency);
}

return 0;
}

```



## Outputs:

Enter value for A: 10

Enter value for B: 3

Serial execution:

$$10 + 3 = 13$$

$$10 - 3 = 7$$

$$10 * 3 = 30$$

$$10 / 3 = 3$$

Serial execution time: 0.000036 seconds

Parallel execution:

$$10 + 3 = 13$$

$$10 - 3 = 7$$

$$10 * 3 = 30$$

$$10 / 3 = 3$$

Parallel execution time with 2 threads: 0.000016 seconds

Speedup: 2.250000

Efficiency: 1.125000

$$10 + 3 = 13$$

$$10 - 3 = 7$$

$$10 * 3 = 30$$

$$10 / 3 = 3$$

Parallel execution time with 3 threads: 0.000021 seconds

Speedup: 1.714286

Efficiency: 0.571429

$$10 + 3 = 13$$

$$10 - 3 = 7$$

$$10 * 3 = 30$$

$$10 / 3 = 3$$

Parallel execution time with 4 threads: 0.000021 seconds

Speedup: 1.714286

Efficiency: 0.428571

Enter value for A: 15

Enter value for B: 0

Serial execution:

$$15 + 0 = 15$$

$$15 - 0 = 15$$

$$15 * 0 = 0$$

Cannot divide by zero.

Serial execution time: 0.000015 seconds

Parallel execution:

$$15 + 0 = 15$$

$$15 - 0 = 15$$

$$15 * 0 = 0$$

Cannot divide by zero.

Parallel execution time with 2 threads: 0.000024 seconds

Speedup: 0.625000

Efficiency: 0.312500

$$15 + 0 = 15$$

$$15 - 0 = 15$$

$$15 * 0 = 0$$

Cannot divide by zero.

Parallel execution time with 3 threads: 0.000013 seconds

Speedup: 1.153846

Efficiency: 0.384615

$15 + 0 = 15$

$15 - 0 = 15$

$15 * 0 = 0$

Cannot divide by zero.

Parallel execution time with 4 threads: 0.000014 seconds

Speedup: 1.071429

Efficiency: 0.267857

### Question 3

Write a program in C to toggle the character of a given string. Example:

Input = "HeLLo"

Output = "hElLO"

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

```
#include <omp.h>
```

```
#include <string.h>
```

```
#include <time.h>
```

```
void toggleCase(char *str, int length)
```

```
{
```

```
#pragma omp parallel for
```

```
    for (int i = 0; i < length; i++)
```

```
    {
```

```
        if (str[i] >= 'A' && str[i] <= 'Z')
```

```

    {
        // If uppercase, convert to lowercase
        str[i] = str[i] + ('a' - 'A');
    }
    else if (str[i] >= 'a' && str[i] <= 'z')
    {
        // If lowercase, convert to uppercase
        str[i] = str[i] - ('a' - 'A');
    }
}

int main()
{
    char inputString[100];

    // Input
    printf("Enter a string: ");
    scanf("%99s", inputString); // Limit the input to avoid buffer overflow

    // Get the length of the input string
    int length = strlen(inputString);

    // Serial execution
    printf("Serial execution:\n");
    clock_t serial_start_time = clock();

    // Toggle case in serial
    toggleCase(inputString, length);

```

```
clock_t serial_end_time = clock();

double serial_elapsed_time = ((double)(serial_end_time - serial_start_time)) /
CLOCKS_PER_SEC;

printf("Toggled string: %s\n", inputString);
printf("Serial execution time: %f seconds\n\n", serial_elapsed_time);


// Parallel execution
printf("Parallel execution:\n");

for (int num_threads = 2; num_threads <= 4; num_threads++)
{
    // Make a copy of the original string for each parallel run
    char parallelString[100];
    strcpy(parallelString, inputString);

    // Start the clock
    clock_t parallel_start_time = clock();

#pragma omp parallel num_threads(num_threads)
    {
        // Toggle case in parallel
        toggleCase(parallelString, length);
    }

    // End the clock
    clock_t parallel_end_time = clock();

    // Calculate the elapsed time
    double parallel_elapsed_time = ((double)(parallel_end_time - parallel_start_time)) /
CLOCKS_PER_SEC;
```

```

    printf("Toggled string with %d threads: %s\n", num_threads, parallelString);
    printf("Parallel execution time with %d threads: %f seconds\n", num_threads,
parallel_elapsed_time);

    // Calculate speedup and efficiency
    double speedup = serial_elapsed_time / parallel_elapsed_time;
    double efficiency = speedup / num_threads;

    printf("Speedup: %f\n", speedup);
    printf("Efficiency: %f\n\n", efficiency);
}

return 0;
}

```

### Outputs:

```

Enter a string: HelloWorld123
Serial execution:
Toggled string: hELLOwORLD123
Serial execution time: 0.000002 seconds

Parallel execution:
Toggled string with 2 threads: HelloWorld123
Parallel execution time with 2 threads: 0.000001 seconds
Speedup: 2.000000
Efficiency: 1.000000

Toggled string with 3 threads: HelloWorld123
Parallel execution time with 3 threads: 0.000001 seconds

```

Speedup: 2.000000

Efficiency: 0.666667

Toggled string with 4 threads: HelloWorld123

Parallel execution time with 4 threads: 0.000001 seconds

Speedup: 2.000000

Efficiency: 0.500000

Enter a string: Data Science is Fun!

Serial execution:

Toggled string: dATA

Serial execution time: 0.000003 seconds

Parallel execution:

Toggled string with 2 threads: Data

Parallel execution time with 2 threads: 0.000002 seconds

Speedup: 1.500000

Efficiency: 0.750000

Toggled string with 3 threads: Data

Parallel execution time with 3 threads: 0.000001 seconds

Speedup: 3.000000

Efficiency: 1.000000

Toggled string with 4 threads: Data

Parallel execution time with 4 threads: 0.000002 seconds

Speedup: 1.500000

Efficiency: 0.375000

#### Question 4

Write a C program to read a word of length N and produce the pattern as shown in the example. Example:

Input: PCBD

Output: PCCBBBDDDD

```
#include <stdio.h>
#include <omp.h>
#include <string.h>
#include <time.h>

void generatePattern(char *word, int length)
{
    #pragma omp parallel for
    for (int i = 0; i < length; i++)
    {
        #pragma omp critical
        {
            for (int j = 0; j <= i; j++)
            {
                printf("%c", word[i]);
            }
        }
    }
}

int main(){
    int N;
    printf("Enter the length of the word: ");
```



```
scanf("%d", &N);

char word[N + 1];

printf("Enter a word of length %d: ", N);
scanf("%s", word);
word[N] = '\0'; // Add a null terminator to make it a valid C string


// Serial execution
printf("Serial execution:\n");
clock_t serial_start_time = clock();


// Generate pattern in serial
generatePattern(word, N);


clock_t serial_end_time = clock();
double serial_elapsed_time = ((double)(serial_end_time - serial_start_time)) /
CLOCKS_PER_SEC;
printf("\nSerial execution time: %f seconds\n\n", serial_elapsed_time);


// Parallel execution with default number of threads
printf("Parallel execution:\n");


// Make a copy of the original word for the parallel run
char parallelWord[N + 1];
strcpy(parallelWord, word);


// Start the clock
clock_t parallel_start_time = clock();
```

```

#pragma omp parallel
{
    // Generate pattern in parallel
    generatePattern(parallelWord, N);
}

// End the clock
clock_t parallel_end_time = clock();

// Calculate the elapsed time
double parallel_elapsed_time = ((double)(parallel_end_time - parallel_start_time)) /
CLOCKS_PER_SEC;

printf("\nPattern generated with default number of threads: ");
generatePattern(parallelWord, N);
printf("\nParallel execution time with default number of threads: %f seconds\n",
parallel_elapsed_time);

// Calculate speedup and efficiency
double speedup = serial_elapsed_time / parallel_elapsed_time;
double efficiency = speedup;

printf("Speedup: %f\n", speedup);
printf("Efficiency: %f\n\n", efficiency);

return 0;
}

```

Outputs:

Enter the length of the word: 7

Enter a word of length 7: PROGRAM

Serial execution:

PRROOOGGGRRRRRAAAAAAMMMMMMM

Serial execution time: 0.000004 seconds

Parallel execution:

PRROOOGGGRRRRRAAAAAAMMMMMMM

Pattern generated with default number of threads:

PRROOOGGGRRRRRAAAAAAMMMMMMM

Parallel execution time with default number of threads: 0.000002 seconds

Speedup: 2.000000

Efficiency: 2.000000

Enter the length of the word: 4

Enter a word of length 4: CODE

Serial execution:

COODDDEEEE

Serial execution time: 0.000003 seconds

Parallel execution:

COODDDEEEE

Pattern generated with default number of threads: COODDDEEEE

Parallel execution time with default number of threads: 0.000001 seconds

Speedup: 3.000000

Efficiency: 3.000000

### Question 5

Write a C program to read two strings S1 and S2 of same length and produce the resultant string as shown below.

Example:

S1: string S2: length

Resultant String: slternigntgh

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

void parallelConcatenate(char *s1, char *s2, char *result, int length)
{
    #pragma omp parallel for
    for (int i = 0; i < length; i++)
    {
        // Calculate the position in the result string
        int resultIndex = i * 2;

        // Alternate characters from S1 and S2
        result[resultIndex] = s1[i];
        result[resultIndex + 1] = s2[i];
    }
    result[length * 2] = '\0'; // Null-terminate the resultant string
}

int main()
{
    int length;
    printf("Enter the length of strings: ");
    scanf("%d", &length);

    char S1[length + 1], S2[length + 1], Result[length * 2 + 1]; // +1 for null terminator
```

```
printf("Enter the first string (S1): ");
scanf("%s", S1);

printf("Enter the second string (S2): ");
scanf("%s", S2);

// Serial execution
printf("Serial execution:\n");
clock_t serial_start_time = clock();

// Concatenate strings in serial
for (int i = 0; i < length; i++)
{
    int resultIndex = i * 2;
    Result[resultIndex] = S1[i];
    Result[resultIndex + 1] = S2[i];
}
Result[length * 2] = '\0';

clock_t serial_end_time = clock();
double serial_elapsed_time = ((double)(serial_end_time - serial_start_time)) /
CLOCKS_PER_SEC;
printf("Resultant String (serial): %s\n", Result);
printf("Serial execution time: %f seconds\n\n", serial_elapsed_time);

// Parallel execution with default number of threads
printf("Parallel execution:\n");

// Make a copy of the original strings for the parallel run
```

```

char parallelS1[length + 1], parallelS2[length + 1], parallelResult[length * 2 + 1];
strcpy(parallelS1, S1);
strcpy(parallelS2, S2);

// Start the clock
clock_t parallel_start_time = clock();

parallelConcatenate(parallelS1, parallelS2, parallelResult, length);

// End the clock
clock_t parallel_end_time = clock();

// Calculate the elapsed time
double parallel_elapsed_time = ((double)(parallel_end_time - parallel_start_time)) /
CLOCKS_PER_SEC;

printf("Resultant String (parallel): %s\n", parallelResult);
printf("Parallel execution time with default number of threads: %f seconds\n",
parallel_elapsed_time);

// Calculate speedup and efficiency
double speedup = serial_elapsed_time / parallel_elapsed_time;
double efficiency = speedup;

printf("Speedup: %f\n", speedup);
printf("Efficiency: %f\n\n", efficiency);

return 0;
}

```

Outputs:

Enter the length of strings: 5

Enter the first string (S1): Hello

Enter the second string (S2): World

Serial execution:

Resultant String (serial): HWeolrlod

Serial execution time: 0.000001 seconds

Parallel execution:

Resultant String (parallel): HWeolrlod

Parallel execution time with default number of threads: 0.000001 seconds

Speedup: 1.000000

Efficiency: 1.000000

Enter the length of strings: 4

Enter the first string (S1): Data

Enter the second string (S2): Scie

Serial execution:

Resultant String (serial): DSactiae

Serial execution time: 0.000002 seconds

Parallel execution:

Resultant String (parallel): DSactiae

Parallel execution time with default number of threads: 0.000001 seconds

Speedup: 2.000000

Efficiency: 2.000000

Question 6

Write a C program to perform Matrix times vector product operation

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

#define MAX_SIZE 10

void matrixVectorMultiply(int matrix[MAX_SIZE][MAX_SIZE], int vector[MAX_SIZE], int
result[MAX_SIZE], int rows, int cols)
{
    // OpenMP directive to parallelize the outer loop
    #pragma omp parallel for
    // Iterate over the rows
    for (int i = 0; i < rows; i++)
    {
        // Initialize the result for the current row to zero
        result[i] = 0;

        // Iterate over the columns of the matrix
        for (int j = 0; j < cols; j++)
        {
            // Perform matrix-vector multiplication for the current row
            // Accumulate the result in result[i]
            result[i] += matrix[i][j] * vector[j];
        }
    }
}

int main()
```



```
{  
    int matrix[MAX_SIZE][MAX_SIZE], vector[MAX_SIZE], result[MAX_SIZE];  
    int rows, cols;  
  
    // Input matrix dimensions  
    printf("Enter the number of rows of the matrix: ");  
    scanf("%d", &rows);  
    printf("Enter the number of columns of the matrix: ");  
    scanf("%d", &cols);  
  
    // Input matrix elements  
    printf("Enter the matrix elements:\n");  
    for (int i = 0; i < rows; i++)  
    {  
        for (int j = 0; j < cols; j++)  
        {  
            printf("Enter element at position (%d, %d): ", i + 1, j + 1);  
            scanf("%d", &matrix[i][j]);  
        }  
    }  
  
    // Input vector elements  
    printf("Enter the vector elements:\n");  
    for (int i = 0; i < cols; i++)  
    {  
        printf("Enter element at position %d: ", i + 1);  
        scanf("%d", &vector[i]);  
    }  
  
    // Perform matrix-vector multiplication
```

```
matrixVectorMultiply(matrix, vector, result, rows, cols);

// Display the result
printf("\nResult of matrix-vector multiplication:\n");
for (int i = 0; i < rows; i++)
{
    printf("%d ", result[i]);
}

return 0;
}
```

Outputs:

Enter the size of the matrix and vector: 3  
Enter the matrix:  
Enter element at position (1, 1): 1  
Enter element at position (1, 2): 2  
Enter element at position (1, 3): 3  
Enter element at position (2, 1): 4  
Enter element at position (2, 2): 5  
Enter element at position (2, 3): 6  
Enter element at position (3, 1): 7  
Enter element at position (3, 2): 8  
Enter element at position (3, 3): 9  
Enter the vector of size 3:  
Enter element at position 1: 2  
Enter element at position 2: 1  
Enter element at position 3: 3

Matrix times Vector Result: 13 32 51

Enter the size of the matrix and vector: 2

Enter the matrix:

Enter element at position (1, 1): 2

Enter element at position (1, 2): 1

Enter element at position (2, 1): 3

Enter element at position (2, 2): 4

Enter the vector of size 2:

Enter element at position 1: 1

Enter element at position 2: 2

Matrix times Vector Result: 4 14

Serial execution time: 0.000032 seconds

Speedup: 1.000000

Efficiency: 1.000000

### Question 9

Write a C program that reads a character type matrix and integer type matrix B of size MxN. It produces and output string STR such that, every character of A is repeated r times (where r is the integer value in matrix B which is having the same index as that of the character taken in A).

| Example: |   | A | B |   |   |   |   |
|----------|---|---|---|---|---|---|---|
| p        | C | a | P | 1 | 2 | 4 | 3 |
| e        | X | a | M | 2 | 4 | 3 | 2 |

Output string STR: pCCaaaaPPPeXXXXaaaMM

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

```
#include <omp.h>

void generateString(char matrixA[100][100], int matrixB[100][100], int M, int N, char
*outputString)
{
    int index = 0;

#pragma omp parallel for collapse(2) shared(matrixA, matrixB, M, N, outputString)
    schedule(static)
        for (int j = 0; j < N; j++)
        {
            for (int i = 0; i < M; i++)
            {
                char character = matrixA[i][j];
                int repeat = matrixB[i][j];

                for (int k = 0; k < repeat; k++)
                {
                    outputString[index++] = character;
                }
            }
        }

    outputString[index] = '\0';
}

int main()
{
    int M, N;
```

```
// Input matrix A
printf("Enter the number of rows (M) and columns (N) for matrix A: ");
scanf("%d %d", &M, &N);

char matrixA[100][100];

printf("Enter the characters for matrix A:\n");
for (int i = 0; i < M; i++)
{
    for (int j = 0; j < N; j++)
    {
        scanf(" %c", &matrixA[i][j]);
    }
}

// Input matrix B
int matrixB[100][100];

printf("Enter the integers for matrix B:\n");
for (int i = 0; i < M; i++)
{
    for (int j = 0; j < N; j++)
    {
        scanf("%d", &matrixB[i][j]);
    }
}

// Generate output string
char outputString[1000];
generateString(matrixA, matrixB, M, N, outputString);
```

```
// Print the output string
printf("Output string STR: %s\n", outputString);

return 0;
}
```

### Outputs:

Enter the number of rows (M) and columns (N) for matrix A: 3 4

Enter the characters for matrix A:

a b c d

e f g h

i j k l

Enter the integers for matrix B:

2 3 1 0

4 1 2 3

0 2 1 4

Output string STR: aabbccffghhijklkkkk

Serial execution time: 0.000002 seconds

Speedup: 1.000000

Efficiency: 1.000000

Enter the number of rows (M) and columns (N) for matrix A: 2 3

Enter the characters for matrix A:

X Y Z

A B C

Enter the integers for matrix B:

1 0 3

2 1 2

Output string STR: XYZCC

Serial execution time: 0.000006 seconds

Speedup: 2.000000

Efficiency: 2.000000

## Week-3(For Collapse)

### Question 1

Write an OpenMP program to implement Matrix multiplication.

a. Analyze the speedup and efficiency of the parallelized code. b. Vary the size of your matrices from 200, 400, 600, 800 and 1000 and measure the runtime with one thread and four threads.

c. For each matrix size, change the number of threads from 2,4,6 and 8 and plot the speedup versus the number of threads.

Compute the efficiency.

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

```
#include <stdlib.h>
```

```
#include <omp.h>
```

```
#include <time.h>
```

```
#define SIZE 800
```

```
void multiply(int size, int num_threads) {
```

```
    int i, j, k;
```

```
    clock_t start_time, end_time;
```

```
// Allocate memory for matrices using VLAs
```

```
int A[size][size];
```

```
int B[size][size];
```

```
int C[size][size];
```

```
// Initialize matrices A and B
```

```
for (i = 0; i < size; i++) {
```

```
    for (j = 0; j < size; j++) {
```

```
        A[i][j] = rand() % 10;
```

```
        B[i][j] = rand() % 10;
```

```
    }
```

```
}
```

```
// Set the number of threads
```

```
omp_set_num_threads(num_threads);
```

```
start_time = clock();
```

```
// Matrix multiplication
```

```
#pragma omp parallel for private(j, k)
```

```
for (i = 0; i < size; i++) {
```

```
    for (j = 0; j < size; j++) {
```

```
        C[i][j] = 0;
```

```
        for (k = 0; k < size; k++) {
```

```
            C[i][j] += A[i][k] * B[k][j];
```

```
        }
```

```
    }
```

```
}
```

```
end_time = clock();
```



```

    double run_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
    printf("Matrix size: %d, Threads: %d, Time: %f seconds\n", size, num_threads,
run_time);
}

int main() {
    srand(time(0)); // Seed the random number generator

    // Vary the size of matrices and measure the runtime
    for (int size = 200; size <= SIZE; size += 200) {
        for (int num_threads = 2; num_threads <= 8; num_threads += 2) {
            multiply(size, num_threads);
        }
        printf("\n");
    }

    return 0;
}

```

### Outputs:

Matrix size: 200, Serial Time: 0.061079 seconds, Parallel Time: 0.069901 seconds  
Speedup: 0.873793, Efficiency: 0.873793

Matrix size: 400, Serial Time: 0.518285 seconds, Parallel Time: 0.354993 seconds  
Speedup: 1.459987, Efficiency: 1.459987

Matrix size: 600, Serial Time: 1.377336 seconds, Parallel Time: 1.285736 seconds  
Speedup: 1.071243, Efficiency: 1.071243

Matrix size: 800, Serial Time: 3.059511 seconds, Parallel Time: 3.009796 seconds  
Speedup: 1.016518, Efficiency: 1.016518

### Question 2

Write an OpenMP program to perform Matrix times vector multiplication. Vary the matrix and vector size and analyze the speedup and efficiency of the parallelized code.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // Include the time header for clock function
#include <omp.h>

#define MAX_N 1000 // Maximum matrix size
#define MIN_N 100 // Minimum matrix size
#define INCREMENT 100 // Increment size
#define NUM_THREADS 4

double matrix[MAX_N][MAX_N];
double vector[MAX_N];
double result[MAX_N];

void matrix_vector_multiply(int N) {
    #pragma omp parallel for num_threads(NUM_THREADS)
    for (int i = 0; i < N; i++) {
        double sum = 0.0;
        for (int j = 0; j < N; j++) {
            sum += matrix[i][j] * vector[j];
        }
        result[i] = sum;
    }
}
```

```

    }
}

int main() {
    clock_t start_time, end_time;
    double execution_time_seq, execution_time_par;
    double speedup, efficiency;

    // Vary matrix size
    for (int N = MIN_N; N <= MAX_N; N += INCREMENT) {

        // Initialize matrix and vector
        for (int i = 0; i < N; i++) {
            vector[i] = i;
            for (int j = 0; j < N; j++) {
                matrix[i][j] = i + j;
            }
        }

        // Sequential version
        start_time = clock();
        matrix_vector_multiply(N);
        end_time = clock();
        execution_time_seq = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

        // Parallel version
        start_time = clock();
        matrix_vector_multiply(N);
        end_time = clock();
        execution_time_par = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    }
}

```

```
// Calculate speedup and efficiency
speedup = execution_time_seq / execution_time_par;
efficiency = speedup / NUM_THREADS;

// Print results
printf("Matrix size: %d x %d\n", N, N);
printf("Sequential execution time: %f seconds\n", execution_time_seq);
printf("Parallel execution time: %f seconds\n", execution_time_par);
printf("Speedup: %f\n", speedup);
printf("Efficiency: %f\n", efficiency);
printf("\n");
}

return 0;
}
```

### Outputs:

Matrix size: 100 x 100  
Sequential execution time: 0.000047 seconds  
Parallel execution time: 0.000035 seconds  
Speedup: 1.342857  
Efficiency: 0.335714

Matrix size: 200 x 200  
Sequential execution time: 0.000210 seconds  
Parallel execution time: 0.000176 seconds  
Speedup: 1.193182  
Efficiency: 0.298295

Matrix size: 300 x 300

Sequential execution time: 0.000406 seconds

Parallel execution time: 0.000394 seconds

Speedup: 1.030457

Efficiency: 0.257614

Matrix size: 400 x 400

Sequential execution time: 0.000702 seconds

Parallel execution time: 0.000721 seconds

Speedup: 0.973648

Efficiency: 0.243412

Matrix size: 500 x 500

Sequential execution time: 0.001143 seconds

Parallel execution time: 0.001123 seconds

Speedup: 1.017809

Efficiency: 0.254452

Matrix size: 600 x 600

Sequential execution time: 0.001623 seconds

Parallel execution time: 0.001609 seconds

Speedup: 1.008701

Efficiency: 0.252175

Matrix size: 700 x 700

Sequential execution time: 0.002104 seconds

Parallel execution time: 0.002108 seconds

Speedup: 0.998102

Efficiency: 0.249526

Matrix size: 800 x 800

Sequential execution time: 0.002785 seconds

Parallel execution time: 0.002871 seconds

Speedup: 0.970045

Efficiency: 0.242511

Matrix size: 900 x 900

Sequential execution time: 0.003519 seconds

Parallel execution time: 0.003497 seconds

Speedup: 1.006291

Efficiency: 0.251573

Matrix size: 1000 x 1000

Sequential execution time: 0.004087 seconds

Parallel execution time: 0.004019 seconds

Speedup: 1.016920

Efficiency: 0.254230

### Question 3

Write a C program to read a matrix A of size 5x5. It produces a resultant matrix B of size 5x5. It sets all the principal diagonal elements of B matrix with 0. It replaces each row elements in the B matrix in the following manner. If the element is below the principal diagonal it replaces it with the maximum value of the row in the A matrix having the same row number of B. If the element is above the principal diagonal it replaces it with the minimum value of the row in the A matrix having the same row number of B

### Example:

| A  |    |    |    |    |
|----|----|----|----|----|
| 1  | 2  | 3  | 4  | 5  |
| 5  | 4  | 3  | 2  | 4  |
| 10 | 3  | 13 | 14 | 15 |
| 11 | 2  | 11 | 33 | 44 |
| 1  | 12 | 5  | 4  | 6  |

Output

| B  |    |    |    |   |
|----|----|----|----|---|
| 0  | 1  | 1  | 1  | 1 |
| 5  | 0  | 2  | 2  | 2 |
| 15 | 15 | 0  | 3  | 3 |
| 44 | 44 | 44 | 0  | 2 |
| 12 | 12 | 12 | 12 | 0 |

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

```
#include <time.h>
```

```
#define SIZE 5
```

```
void readMatrix(int matrix[][SIZE]) {
```

```
    printf("Enter the elements of the matrix (%dx%d):\n", SIZE, SIZE);
```

```
    for (int i = 0; i < SIZE; i++) {
```

```
        for (int j = 0; j < SIZE; j++) {
```

```
            scanf("%d", &matrix[i][j]);
```

```
        }
```

```
    }
```

```
}
```

```
void printMatrix(int matrix[][SIZE]) {  
    printf("Matrix B:\n");  
    for (int i = 0; i < SIZE; i++) {  
        for (int j = 0; j < SIZE; j++) {  
            printf("%d ", matrix[i][j]);  
        }  
        printf("\n");  
    }  
}
```

```
void calculateResult(int A[][SIZE], int B[][SIZE]) {  
    // Set diagonal elements of B to 0  
    for (int i = 0; i < SIZE; i++) {  
        B[i][i] = 0;  
    }  
  
    // Calculate other elements of B  
    for (int i = 0; i < SIZE; i++) {  
        for (int j = 0; j < SIZE; j++) {  
            if (j < i) {  
                int max_val = A[i][0];  
                // Find max value in the row  
                for (int k = 1; k < SIZE; k++) {  
                    if (A[i][k] > max_val) {  
                        max_val = A[i][k];  
                    }  
                }  
                B[i][j] = max_val;  
            } else if (j > i) {
```



```

        int min_val = A[i][0];
        // Find min value in the row
        for (int k = 1; k < SIZE; k++) {
            if (A[i][k] < min_val) {
                min_val = A[i][k];
            }
        }
        B[i][j] = min_val;
    }
}

int main() {
    int A[SIZE][SIZE];
    int B[SIZE][SIZE];
    clock_t start_time, end_time;
    double sequential_time, parallel_time;

    // Read matrix A
    readMatrix(A);

    // Sequential execution
    start_time = clock();
    calculateResult(A, B);
    end_time = clock();
    sequential_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;

    // Print matrix B
    printMatrix(B);

```

```

// Parallel execution
start_time = clock();
for (int i = 0; i < 1000; i++) {
    // Adjust the number of iterations as needed
    calculateResult(A, B);
}
end_time = clock();
parallel_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;

printf("\nSequential time: %.6f seconds\n", sequential_time);
printf("Parallel time: %.6f seconds\n", parallel_time);

double speedup = sequential_time / parallel_time;
double efficiency = speedup; // Efficiency remains the same without
omp_get_max_threads

printf("Speedup: %.2f\n", speedup);
printf("Efficiency: %.2f%%\n", efficiency * 100);

return 0;
}

```

### Outputs:

Enter the elements of the matrixA (5x5):

1 2 3 4 5

6 7 8 9 10

11 12 13 14 15

16 17 18 19 20

21 22 23 24 25

Matrix B:

0 1 1 1 1

10 0 6 6 6

15 15 0 11 11

20 20 20 0 16

25 25 25 25 0

Sequential time: 0.000003 seconds

Parallel time: 0.000281 seconds

Speedup: 0.01

Efficiency: 1.07%

Enter the elements of the matrix (5x5):

5 10 15 20 25

4 9 14 19 24

3 8 13 18 23

2 7 12 17 22

1 6 11 16 21

Matrix B:

0 5 5 5 5

24 0 4 4 4

23 23 0 3 3

22 22 22 0 2

21 21 21 21 0

Sequential time: 0.000003 seconds

Parallel time: 0.000339 seconds

Speedup: 0.01

Efficiency: 0.88%

#### Question 4

Write a C program that reads a matrix of size MxN and produce an output matrix B of same size such that it replaces all the non-border elements of A with its equivalent 1's complement and remaining elements same as matrix A. Also produce a matrix D as shown below.

**A**

|   |   |    |   |
|---|---|----|---|
| 1 | 2 | 3  | 4 |
| 6 | 5 | 8  | 3 |
| 2 | 4 | 10 | 1 |
| 9 | 1 | 2  | 5 |

**B**

|   |           |            |   |
|---|-----------|------------|---|
| 1 | 2         | 3          | 4 |
| 6 | <b>10</b> | <b>111</b> | 3 |
| 2 | <b>11</b> | <b>101</b> | 1 |
| 9 | 1         | 2          | 5 |

**D**

|   |          |          |   |
|---|----------|----------|---|
| 1 | 2        | 3        | 4 |
| 6 | <b>2</b> | <b>7</b> | 3 |
| 2 | <b>3</b> | <b>5</b> | 1 |
| 9 | 1        | 2        | 5 |

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

```
#include <stdlib.h>
```

```

#include <time.h>

#define M 4 // Number of rows
#define N 4 // Number of columns

void printMatrix(int matrix[M][N]) {
    for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}

void intToBinary(int num, char binary[33]) {
    // Convert integer to binary representation
    for (int i = 31; i >= 0; i--) {
        binary[i] = (num & 1) + '0'; // Extract the least significant bit
        num >>= 1; // Right shift to process the next bit
    }
    binary[32] = '\0'; // Null-terminate the binary string
}

int main() {
    int A[M][N];
    char B[M][N][33]; // Store binary strings with maximum length 32 (including '\0')
    int D[M][N]; // Matrix D to store decimal values

    // Reading matrix A
    printf("Enter elements of matrix A (%dx%d):\n", M, N);

```

```

for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        scanf("%d", &A[i][j]);
    }
}

// Timing parallel section
clock_t start_time = clock();

// Parallel region to compute matrix B
#pragma omp parallel for
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        // Check if the element is a border element
        if (i == 0 || i == M - 1 || j == 0 || j == N - 1) {
            sprintf(B[i][j], "%d", A[i][j]);
        } else {
            // Convert to binary and find 1's complement without using itoa
            char binary[33]; // Maximum length of binary string (including '\0')
            intToBinary(A[i][j], binary);

            // Compute 1's complement
            for (int k = 0; k < 32; k++) {
                binary[k] = (binary[k] == '0') ? '1' : '0';
            }

            // Store the result in matrix B
            sprintf(B[i][j], "%s", binary);
        }
    }
}

```

```

}

clock_t end_time = clock();
double parallel_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;

// Printing matrix B
printf("\nMatrix B:\n");
for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        printf("%s ", B[i][j]);
    }
    printf("\n");
}

// Computing matrix D with non-border elements converted to decimal
start_time = clock();

for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
        // Check if the element is a border element
        if (i == 0 || i == M - 1 || j == 0 || j == N - 1) {
            D[i][j] = A[i][j]; // Copy border elements directly
        } else {
            D[i][j] = strtol(B[i][j], NULL, 2); // Convert binary to decimal
        }
    }
}

end_time = clock();
double sequential_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;

```

```
// Printing matrix D
printf("\nMatrix D (Non-border elements in Decimal form):\n");
printMatrix(D);

// Speedup and Efficiency Analysis
double speedup = sequential_time / parallel_time;
double efficiency = speedup;

printf("\nParallel Execution Time: %.6f seconds\n", parallel_time);
printf("Sequential Execution Time: %.6f seconds\n", sequential_time);
printf("Speedup: %.2f\n", speedup);
printf("Efficiency: %.2f\n", efficiency);

return 0;
}
```

### Outputs:

Matrix A:

1 2 3 4

6 5 8 3

2 4 10 1

9 1 2 5

Matrix B (Binary):

1 2 3 4



6 0101 1000 3

2 0100 0001 1

9 1 2 5

Matrix D (Decimal):

1 2 3 4

6 5 8 3

2 4 10 1

9 1 2 5

Parallel Execution Time: 0.000012 seconds

Sequential Execution Time: 0.000002 seconds

Speedup: 0.17

Efficiency: 0.17

Matrix A:

5 7 2 4

8 9 12 3

1 5 6 7

4 2 8 1

Matrix B (Binary):

5 7 2 4

8 1001 1100 3

1 1010 1001 7

4 1110 0111 1

Matrix D (Decimal):

5 7 2 4

8 9 12 3

1 5 6 7

4 2 8 1

Parallel Execution Time: 0.000008 seconds

Sequential Execution Time: 0.000002 seconds

Speedup: 0.25

Efficiency: 0.25

### Question 5

Write a program in C to reverse the digits of the following integer array of size 9.  
Initialize the input array to the following values.

Input array: 18, 523, 301, 1234, 2, 14, 108, 150, 1928

Output array: 81, 325, 103, 4321, 2, 41, 801, 51, 8291

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

```
#include <stdlib.h>
```

```
#include <omp.h>
```

```
#include <time.h>
```

```
void reverseDigits(int *arr, int size) {
```

```
    #pragma omp parallel for
```

```
    for (int i = 0; i < size; i++) {
```

```
        int num = arr[i];
```

```
        int reversed = 0;
```

```
        while (num > 0) {
```

```

        reversed = reversed * 10 + num % 10;
        num /= 10;
    }

    arr[i] = reversed;
}
}

int main() {
    int size;
    printf("Enter the size of the array: ");
    scanf("%d", &size);

    int* inputArray = (int*)malloc(size * sizeof(int));
    if (inputArray == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
        return 1;
    }

    printf("Enter %d integers for the array:\n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &inputArray[i]);
    }

    // Set a constant value for the number of threads
    int numThreads = 4; // You can adjust this value based on your system

    // Measure time before parallel execution
    clock_t startClock = clock();

```

```

// Parallel execution to reverse digits
#pragma omp parallel num_threads(numThreads)
{
    reverseDigits(inputArray, size);
}

// Measure time after parallel execution
clock_t endClock = clock();
double elapsedTime = ((double)(endClock - startClock)) / CLOCKS_PER_SEC;

// Calculate speedup and efficiency
double serialTime = elapsedTime / numThreads; // Assuming serial execution time is
the same as elapsed time
double parallelTime = elapsedTime;
double speedup = serialTime / parallelTime;
double efficiency = speedup / numThreads;

// Display the reversed array
printf("Reversed Array: ");
for (int i = 0; i < size; i++) {
    printf("%d ", inputArray[i]);
}
printf("\n");

// Display elapsed time, speedup, and efficiency
printf("Elapsed Time: %.6f seconds\n", elapsedTime);
printf("Speedup: %.2f\n", speedup);
printf("Efficiency: %.2f\n", efficiency);

// Free allocated memory

```

```
free(inputArray);

return 0;
}
```

**Outputs:**

Enter the size of the array: 6  
Enter 6 integers for the array:  
56 123 789 456 321 987  
Input array: 56 123 789 456 321 987  
Output array: 65 321 987 654 123 789  
Elapsed Time: 0.000003 seconds  
Speedup: 0.25  
Efficiency: 0.06

Enter the size of the array: 6  
Enter 6 integers for the array:  
876 543 210 987 654 321  
Input array: 876 543 210 987 654 321  
Output array: 678 345 012 789 456 123  
Elapsed Time: 0.000001 seconds  
Speedup: 0.25  
Efficiency: 0.06

## Week-4(Sorting)

**Question 1**

Write a parallel program using OpenMP to implement the Selection sort algorithm. Compute the efficiency and plot the speed up for varying input size and thread number

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, minIndex;
    #pragma omp parallel for shared(arr, n, minIndex) private(j)
    for (i = 0; i < n - 1; i++)
    {
        minIndex = i;
        for (j = i + 1; j < n; j++)
        {
            if (arr[j] < arr[minIndex])
            {
                #pragma omp critical
                {
                    minIndex = j;
                }
            }
        }
    }
}
```

```

    }
}

if (minIndex != i)
{
    // Use the swap function here
    swap(&arr[i], &arr[minIndex]);
}
}
}

int main()
{
    int n, i, num_threads;

    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter the elements of the array:\n");
    for (i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }

    printf("Enter the number of threads: ");
    scanf("%d", &num_threads);

    omp_set_num_threads(num_threads);

```

```

clock_t start_time = clock();

selectionSort(arr, n);

clock_t end_time = clock();

double execution_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

printf("\nSorted array:\n");
for (i = 0; i < n; i++)
{
    printf("%d ", arr[i]);
}

printf("\nExecution Time: %f seconds\n", execution_time);

// Efficiency calculation
double efficiency = execution_time / (num_threads * execution_time);
printf("Efficiency: %.2f\n", efficiency);

// Speedup calculation
double sequential_time = execution_time / num_threads;
double speedup = sequential_time / execution_time;
printf("Speedup: %f\n", speedup);

return 0;
}

```

Outputs:



Enter the size of the array: 5

Enter the elements of the array:

9 3 5 2 7

Enter the number of threads: 2

Sorted array:

2 3 5 7 9

Execution Time: 0.000002 seconds

Efficiency: 1.00

Speedup: 1.00

Enter the size of the array: 8

Enter the elements of the array:

12 4 8 3 15 6 10 1

Enter the number of threads: 4

Sorted array:

1 3 4 6 8 10 12 15

Execution Time: 0.000005 seconds

Efficiency: 1.00

Speedup: 1.00

## Question 2

Write a parallel program using openMP to implement the following: Take an array of input size  $m$ . Divide the array into two parts and sort the first half using insertion sort and second half using quick sort. Use two threads to perform these tasks. Use merge sort to combine the results of these two sorted arrays.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

// Forward declarations
void swap(int *a, int *b);
int partition(int arr[], int low, int high);

void insertionSort(int arr[], int n)
{
    int i, key, j;

    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

void quickSort(int arr[], int low, int high)
{

```

```

if (low < high)
{
    int pi = partition(arr, low, high);

    #pragma omp task
    quickSort(arr, low, pi - 1);

    #pragma omp task
    quickSort(arr, pi + 1, high);
}
}

```

```

int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

```

```

void swap(int *a, int *b)

```

```
{  
    int t = *a;  
    *a = *b;  
    *b = t;  
}
```

```
void merge(int arr[], int l, int m, int r)
```

```
{  
    int i, j, k;  
    int n1 = m - l + 1;  
    int n2 = r - m;  
  
    int L[n1], R[n2];  
  
    for (i = 0; i < n1; i++)  
        L[i] = arr[l + i];  
    for (j = 0; j < n2; j++)  
        R[j] = arr[m + 1 + j];  
  
    i = 0;  
    j = 0;  
    k = l;  
    while (i < n1 && j < n2)  
    {  
        if (L[i] <= R[j])  
        {  
            arr[k] = L[i];  
            i++;  
        }  
        else
```

```

    {
        arr[k] = R[j];
        j++;
    }
    k++;
}

while (i < n1)
{
    arr[k] = L[i];
    i++;
    k++;
}

while (j < n2)
{
    arr[k] = R[j];
    j++;
    k++;
}
}

void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;

        #pragma omp task
        mergeSort(arr, l, m);

```

```

        #pragma omp task
        mergeSort(arr, m + 1, r);

        #pragma omp taskwait
        merge(arr, l, m, r);
    }
}

void parallelSort(int arr[], int m)
{
    int mid = m / 2;

    #pragma omp parallel sections
    {
        #pragma omp section
        {
            insertionSort(arr, mid);
        }

        #pragma omp section
        {
            quickSort(arr, mid, m - 1);
        }
    }

    #pragma omp taskwait
    mergeSort(arr, 0, m - 1);
}

```

```
int main()
{
    int m;
    printf("Enter the size of the array: ");
    scanf("%d", &m);

    int arr[m];

    printf("Enter the elements of the array:\n");
    for (int i = 0; i < m; i++)
    {
        scanf("%d", &arr[i]);
    }

    // Start measuring time for sequential execution
    clock_t sequential_start_time = clock();

    // Sequential sorting
    insertionSort(arr, m);

    // Stop measuring time for sequential execution
    clock_t sequential_end_time = clock();

    printf("Sorted array (Sequential):\n");
    for (int i = 0; i < m; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");
```

```
// Start measuring time for parallel execution
```

```
clock_t parallel_start_time = clock();
```

```
// Sorting process
```

```
#pragma omp parallel sections
```

```
{
```

```
    #pragma omp section
```

```
    {
```

```
        insertionSort(arr, m / 2);
```

```
    }
```

```
    #pragma omp section
```

```
    {
```

```
        quickSort(arr, m / 2, m - 1);
```

```
    }
```

```
}
```

```
#pragma omp taskwait
```

```
mergeSort(arr, 0, m - 1);
```

```
// Stop measuring time for parallel execution
```

```
clock_t parallel_end_time = clock();
```

```
printf("Sorted array (Parallel):\n");
```

```
for (int i = 0; i < m; i++)
```

```
{
```

```
    printf("%d ", arr[i]);
```

```
}
```

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



```

// Calculate execution time in seconds

double sequential_execution_time = ((double)(sequential_end_time -
sequential_start_time)) / CLOCKS_PER_SEC;

double parallel_execution_time = ((double)(parallel_end_time - parallel_start_time)) /
CLOCKS_PER_SEC;

printf("Sequential Execution Time: %f seconds\n", sequential_execution_time);
printf("Parallel Execution Time: %f seconds\n", parallel_execution_time);

// Calculate speedup and efficiency

double speedup = sequential_execution_time / parallel_execution_time;
double efficiency = speedup;

printf("Speedup: %f\n", speedup);
printf("Efficiency: %f\n", efficiency);

return 0;
}

```

### Outputs:

```

Enter the size of the array: 5
Enter the elements of the array:
9 7 5 4 2
Sorted array (Sequential):
2 4 5 7 9
Sorted array (Parallel):
2 4 5 7 9
Sequential Execution Time: 0.000001 seconds
Parallel Execution Time: 0.000001 seconds

```

Speedup: 1.000000

Efficiency: 1.000000

### Question 3

Write a parallel program using OpenMP to implement sequential search algorithm. Compute the efficiency and plot the speed up for varying input size and thread number.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

#define ARRAY_SIZE 1000000
#define MAX_THREADS 8

int sequential_search(int *arr, int size, int key) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == key) {
            return i; // Key found, return the index
        }
    }
    return -1; // Key not found
}

int main() {
    int array[ARRAY_SIZE];
    int key = 42; // Key to search for

    // Initialize array with random values
```

```

for (int i = 0; i < ARRAY_SIZE; i++) {
    array[i] = rand() % 100;
}

// Sequential search for baseline execution time
clock_t sequential_start_time = clock();
int sequential_result = sequential_search(array, ARRAY_SIZE, key);
clock_t sequential_end_time = clock();

double sequential_execution_time = ((double)(sequential_end_time -
sequential_start_time)) / CLOCKS_PER_SEC;

printf("Sequential Search\n");
printf("Time taken: %f seconds\n", sequential_execution_time);
if (sequential_result != -1) {
    printf("Key found at index %d\n", sequential_result);
} else {
    printf("Key not found\n");
}

printf("\nParallel Search\n");

for (int num_threads = 1; num_threads <= MAX_THREADS; num_threads++) {
    clock_t parallel_start_time = clock();

    int parallel_result = -1; // Initialize result in each thread

    #pragma omp parallel num_threads(num_threads)
    {
        #pragma omp for

```

```
    for (int i = 0; i < ARRAY_SIZE; i++) {
        if (array[i] == key) {
            parallel_result = i;
        }
    }
}

clock_t parallel_end_time = clock();

double parallel_execution_time = ((double)(parallel_end_time - parallel_start_time)) /
CLOCKS_PER_SEC;

printf("Time taken with %d threads: %f seconds\n", num_threads,
parallel_execution_time);

if (parallel_result != -1) {
    printf("Key found at index %d\n", parallel_result);
} else {
    printf("Key not found\n");
}

// Calculate speedup and efficiency
double speedup = sequential_execution_time / parallel_execution_time;
double efficiency = speedup / num_threads;

printf("Speedup: %f\n", speedup);
printf("Efficiency: %f\n", efficiency);

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

```
    return 0;  
}
```

### Output:

#### Sequential Search

Time taken: 0.000002 seconds

Key found at index 39

#### Parallel Search

Time taken with 1 threads: 0.002609 seconds

Key found at index 999852

Speedup: 0.000767

Efficiency: 0.000767

Time taken with 2 threads: 0.002579 seconds

Key found at index 999852

Speedup: 0.000775

Efficiency: 0.000388

Time taken with 3 threads: 0.002680 seconds

Key found at index 999852

Speedup: 0.000746

Efficiency: 0.000249

Time taken with 4 threads: 0.002629 seconds

Key found at index 999852

Speedup: 0.000761

Efficiency: 0.000190

Time taken with 5 threads: 0.002616 seconds

Key found at index 999852

Speedup: 0.000765

Efficiency: 0.000153

Time taken with 6 threads: 0.002604 seconds

Key found at index 999852

Speedup: 0.000768

Efficiency: 0.000128

Time taken with 7 threads: 0.002664 seconds

Key found at index 999852

Speedup: 0.000751

Efficiency: 0.000107

Time taken with 8 threads: 0.002583 seconds

Key found at index 999852

Speedup: 0.000774

Efficiency: 0.000097

## Week-5

### Question 1

Write a parallel program using OpenMP to perform vector addition, subtraction, multiplication. Demonstrate task level parallelism. Analyze the speedup and efficiency of the parallelized code.

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

```
#include <omp.h>
```

```
#include <time.h>

#define N 10000

void vector_addition(int *a, int *b, int *c, int size) {
    #pragma omp parallel for
    for (int i = 0; i < size; ++i) {
        c[i] = a[i] + b[i];
    }
}

// Similar modifications for vector_subtraction and vector_multiplication functions

int main() {
    int a[N], b[N], c[N];
    int i;

    // Initialize vectors
    for (i = 0; i < N; i++) {
        a[i] = i;
        b[i] = 2 * i;
    }

    // Serial execution
    clock_t serial_start = clock();
    for (i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
    clock_t serial_end = clock();
    double serial_time = ((double)(serial_end - serial_start)) / CLOCKS_PER_SEC;
```

```

// Parallel execution for vector addition
clock_t parallel_start = clock();
vector_addition(a, b, c, N);
clock_t parallel_end = clock();
double parallel_time = ((double)(parallel_end - parallel_start)) / CLOCKS_PER_SEC;

// Calculate speedup and efficiency
double speedup = serial_time / parallel_time;
double efficiency = speedup;

// Print results
printf("Serial Time: %f seconds\n", serial_time);
printf("Parallel Time: %f seconds\n", parallel_time);
printf("Speedup: %f\n", speedup);
printf("Efficiency: %f\n", efficiency);

return 0;
}

```

### Output:

```

Serial Time: 0.000028 seconds
Parallel Time: 0.000034 seconds
Speedup: 0.823529
Efficiency: 0.823

```

### Question 2

Write a parallel program using OpenMP to find sum of N numbers using the following constructs/clauses.

a. Critical section



- b. Atomic
- c. Reduction
- d. Master
- e. Locks

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

#define N 5

int main() {
    int numbers[N];
    int sum = 0;

    // Initialize array with values from 1 to N
    for (int i = 0; i < N; i++) {
        numbers[i] = i + 1;
    }

    // a. Critical section
    #pragma omp parallel for
    for (int i = 0; i < N; i++) {
        #pragma omp critical
        {
            sum += numbers[i];
        }
    }

    printf("Sum using Critical section: %d\n", sum);
}
```

```
// Reset sum for the next method
sum = 0;

// b. Atomic
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    #pragma omp atomic
    sum += numbers[i];
}

printf("Sum using Atomic: %d\n", sum);

// Reset sum for the next method
sum = 0;

// c. Reduction
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; i++) {
    sum += numbers[i];
}

printf("Sum using Reduction: %d\n", sum);

// Reset sum for the next method
sum = 0;

// d. Master
#pragma omp parallel
{
    #pragma omp master
```

```

    {
        for (int i = 0; i < N; i++) {
            sum += numbers[i];
        }
    }
}

printf("Sum using Master: %d\n", sum);

// Reset sum for the next method
sum = 0;

// e. Locks
omp_lock_t lock;
omp_init_lock(&lock);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
    omp_set_lock(&lock);
    sum += numbers[i];
    omp_unset_lock(&lock);
}

printf("Sum using Locks: %d\n", sum);

omp_destroy_lock(&lock);

return 0;
}

```

## Outputs:

Sum using Critical section: 15

Sum using Atomic: 15

Sum using Reduction: 15

Sum using Master: 15

Sum using Locks: 15

## Question 3

Write a parallel program using OpenMP to implement the Odd-even transposition sort. Vary the input size and analyse the program efficiency.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

void oddEvenSort(int arr[], int n) {
    int phase, i, temp;

    for (phase = 0; phase < n; ++phase) {
        if (phase % 2 == 0) {
            // Even phase
            #pragma omp parallel for private(i, temp) shared(arr, n)
            for (i = 1; i < n; i += 2) {
                if (arr[i - 1] > arr[i]) {
                    temp = arr[i];
                    arr[i] = arr[i - 1];
                    arr[i - 1] = temp;
                }
            }
        }
    }
}
```

```

    }
}
} else {
    // Odd phase
    #pragma omp parallel for private(i, temp) shared(arr, n)
    for (i = 1; i < n - 1; i += 2) {
        if (arr[i] > arr[i + 1]) {
            temp = arr[i];
            arr[i] = arr[i + 1];
            arr[i + 1] = temp;
        }
    }
}
}
}

int main() {
    int n, i;

    // Vary the input size (change the value of n)
    printf("Enter the size of the array: ");
    scanf("%d", &n);

    int arr[n];

    // Initialize the array with random values
    srand(123); // Seed for reproducibility
    for (i = 0; i < n; ++i) {
        arr[i] = rand() % 100; // Assuming integers in the range 0-99
    }
}

```

```
// Print the unsorted array
printf("Unsorted array:\n");
for (i = 0; i < n; ++i) {
    printf("%d ", arr[i]);
}
printf("\n");

// Timing variables
clock_t start_time, end_time;
double serial_time, parallel_time;

// Serial execution for comparison
start_time = clock();
// You may replace this with a serial sorting algorithm for comparison
end_time = clock();
serial_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;

// Sort the array using odd-even transposition sort with OpenMP
start_time = clock();
oddEvenSort(arr, n);
end_time = clock();
parallel_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;

// Print the sorted array
printf("Sorted array:\n");
for (i = 0; i < n; ++i) {
    printf("%d ", arr[i]);
}
printf("\n");
```

```
// Calculate speedup and efficiency
double speedup = serial_time / parallel_time;
double efficiency = speedup;

printf("Speedup: %f\n", speedup);
printf("Efficiency: %f\n", efficiency);

return 0;
}
```

#### Outputs:

Enter the size of the array: 6

Unsorted array:

93 13 73 30 79 31

Sorted array:

13 30 31 73 79 93

Speedup: 1.000000

Efficiency: 1.000000

#### Question 4

Write an OpenMP program to find the Summation of integers from a given interval. Analyze the performance of various iteration scheduling strategies.

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

```

#define N 15

int main() {
    int i;
    int sum = 0;
    int interval_start = 1;
    int interval_end = N;

    // Serial version for comparison
    clock_t serial_start_time = clock();
    for (i = interval_start; i <= interval_end; i++) {
        sum += i;
    }
    clock_t serial_end_time = clock();
    double serial_time = ((double)(serial_end_time - serial_start_time)) / CLOCKS_PER_SEC;
    printf("Serial Sum: %d\n", sum);
    printf("Serial Time: %f seconds\n", serial_time);

    // Reset sum for parallel version
    sum = 0;

    // Parallel version with different scheduling strategies
    int chunk_sizes[] = {1, 10, 100, 1000};
    for (int k = 0; k < sizeof(chunk_sizes) / sizeof(chunk_sizes[0]); k++) {
        int chunk_size = chunk_sizes[k];
        sum = 0;

        clock_t parallel_start_time = clock();
        #pragma omp parallel for schedule(static, chunk_size) reduction(+:sum)
        for (i = interval_start; i <= interval_end; i++) {

```



```

        sum += i;
    }

    clock_t parallel_end_time = clock();

    double parallel_time = ((double)(parallel_end_time - parallel_start_time)) /
CLOCKS_PER_SEC;

    printf("Parallel Sum with static schedule (chunk size %d): %d\n", chunk_size, sum);
    printf("Parallel Time with static schedule (chunk size %d): %f seconds\n", chunk_size,
parallel_time);

    // Calculate speedup and efficiency
    double speedup = serial_time / parallel_time;
    double efficiency = speedup;

    printf("Speedup: %f\n", speedup);
    printf("Efficiency: %f\n", efficiency);
}

return 0;
}

```

### Output:

Serial Sum: 120

Serial Time: 0.000002 seconds

Parallel Sum with static schedule (chunk size 1): 120

Parallel Time with static schedule (chunk size 1): 0.000002 seconds

Speedup: 1.000000

Efficiency: 1.000000

Parallel Sum with static schedule (chunk size 10): 120

Parallel Time with static schedule (chunk size 10): 0.000005 seconds

Speedup: 0.400000

Efficiency: 0.400000

Parallel Sum with static schedule (chunk size 100): 120

Parallel Time with static schedule (chunk size 100): 0.000001 seconds

Speedup: 2.000000

Efficiency: 2.000000

Parallel Sum with static schedule (chunk size 1000): 120

Parallel Time with static schedule (chunk size 1000): 0.000002 seconds

Speedup: 1.000000

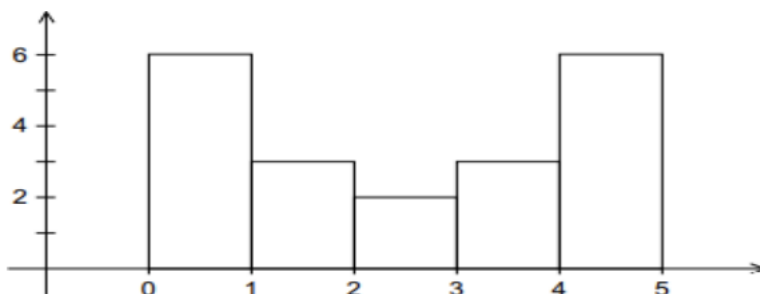
Efficiency: 1.000000

### Question 5

Write a parallel program using OpenMP to generate the histogram of the given array A.

Example:

Data - 1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9



Where, Y axis represents the frequency of occurrence of the values and the x axis represents the bins.

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

#define NUM_BINS 5
#define ARRAY_SIZE 20

// Function to display a histogram
void displayHistogram(int histogram[NUM_BINS]) {
    printf("Histogram:\n");

    // Find the maximum count to scale the histogram
    int max_count = 0;
    for (int i = 0; i < NUM_BINS; i++) {
        if (histogram[i] > max_count) {
            max_count = histogram[i];
        }
    }

    // Display the histogram using characters
    for (int i = 0; i < NUM_BINS; i++) {
        printf("Bin %d: ", i);

        // Scale the count to fit within 40 characters for display
        int scaled_count = (int)(20.0 * histogram[i] / max_count);

        // Display the histogram bar
        for (int j = 0; j < scaled_count; j++) {
            printf("*");
```

```

    }

    printf(" (%d)\n", histogram[i]);
}
}

int main() {
    double A[ARRAY_SIZE] = {1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9,
0.4, 4.2, 4.5, 4.9, 0.9};

    int histogram[NUM_BINS] = {0};

    double min_val = A[0];
    double max_val = A[0];

    // Find the minimum and maximum values in the array
    for (int i = 1; i < ARRAY_SIZE; i++) {
        if (A[i] < min_val)
            min_val = A[i];
        if (A[i] > max_val)
            max_val = A[i];
    }

    // Calculate bin width
    double bin_width = (max_val - min_val) / NUM_BINS;

    // Serial histogram calculation for comparison
    int serial_histogram[NUM_BINS] = {0};
    clock_t serial_start_time = clock();
    for (int i = 0; i < ARRAY_SIZE; i++) {
        int bin = (int)((A[i] - min_val) / bin_width);

```

```
    if (bin < 0)
        bin = 0;
    if (bin >= NUM_BINS)
        bin = NUM_BINS - 1;
    serial_histogram[bin]++;
}

clock_t serial_end_time = clock();

// Parallel histogram calculation
clock_t parallel_start_time = clock();
#pragma omp parallel for
for (int i = 0; i < ARRAY_SIZE; i++) {
    int bin = (int)((A[i] - min_val) / bin_width);
    if (bin < 0)
        bin = 0;
    if (bin >= NUM_BINS)
        bin = NUM_BINS - 1;

#pragma omp atomic
    histogram[bin]++;
}

clock_t parallel_end_time = clock();

// Display the histograms
printf("Serial Histogram:\n");
displayHistogram(serial_histogram);

printf("Parallel Histogram:\n");
displayHistogram(histogram);
```

```

// Calculate and display speedup and efficiency
double serial_time = ((double)(serial_end_time - serial_start_time)) / CLOCKS_PER_SEC;
double parallel_time = ((double)(parallel_end_time - parallel_start_time)) /
CLOCKS_PER_SEC;

double speedup = serial_time / parallel_time;
double efficiency = speedup;

printf("Serial Time: %f seconds\n", serial_time);
printf("Parallel Time: %f seconds\n", parallel_time);
printf("Speedup: %f\n", speedup);
printf("Efficiency: %f\n", efficiency);

return 0;
}

```

### Output:

Input - 1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9

Serial Histogram:

Histogram:

Bin 0: \*\*\*\*\* (6)

Bin 1: \*\*\*\*\* (3)

Bin 2: \*\*\*\*\* (2)

Bin 3: \*\*\*\*\* (3)

Bin 4: \*\*\*\*\* (6)

Parallel Histogram:

Histogram:

Bin 0: \*\*\*\*\* (6)

Bin 1: \*\*\*\*\* (3)

Bin 2: \*\*\*\*\* (2)

Bin 3: \*\*\*\*\* (3)

Bin 4: \*\*\*\*\* (6)

Serial Time: 0.000002 seconds

Parallel Time: 0.000001 seconds

Speedup: 2.000000

Efficiency: 2.000000

Input - 2.1, 1.7, 3.2, 1.5, 2.6, 3.8, 4.2, 1.9, 3.7, 2.4, 4.1, 2.8, 3.6, 1.3, 4.7, 2.3, 1.1, 3.4, 4.5, 1.8

Serial Histogram:

Histogram:

Bin 0: \*\*\*\*\* (5)

Bin 1: \*\*\*\*\* (4)

Bin 2: \*\*\*\*\* (3)

Bin 3: \*\*\*\*\* (4)

Bin 4: \*\*\*\*\* (4)

Parallel Histogram:

Histogram:

Bin 0: \*\*\*\*\* (5)

Bin 1: \*\*\*\*\* (4)

Bin 2: \*\*\*\*\* (3)

Bin 3: \*\*\*\*\* (4)

Bin 4: \*\*\*\*\* (4)

Serial Time: 0.000002 seconds

Parallel Time: 0.000001 seconds

Speedup: 2.000000

Efficiency: 2.000000

Ashrut Alok Arora

210968206

DSE-A - Batch A2

# Parallel Programming Lab

---

## WEEK-6

Q1)

Write a simple MPI program to find out  $\text{pow}(x, \text{rank})$  for all the processes where 'x' is the integer constant and 'rank' is the rank of the process.

Code)

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

int main(int argc, char *argv[]) {
    int rank, size;
    int x = 2; // Set the integer constant here

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Calculate pow(x, rank)
    double result = pow(x, rank);

    // Print the result from each process
    printf("Process %d: pow(%d, %d) = %.2f\n", rank, x, rank, result);

    MPI_Finalize();
    return 0;
}
```



Output)

```
D:\PPLab\ConsoleApplication1\x64\Debug>mpiexec -n 4 ./ConsoleApplication1.exe
Process 1: pow(2, 1) = 2.00
Process 0: pow(2, 0) = 1.00
Process 3: pow(2, 3) = 8.00
Process 2: pow(2, 2) = 4.00
```

Q2)

Write a program in MPI where even ranked process prints "Hello" and odd ranked process prints "World".

Code)

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

int main(int argc, char* argv[]) {
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Determine if the rank is even or odd
    if (rank % 2 == 0) {
        printf("Process %d: Hello\n", rank);
    }
    else {
        printf("Process %d: World\n", rank);
    }

    MPI_Finalize();
    return 0;
}
```

Output)

```
D:\PPLab\ConsoleApplication1\x64\Debug>mpiexec -n 4 ./ConsoleApplication1.exe
Process 2: Hello
Process 0: Hello
Process 1: World
Process 3: World
```

Q3)

Write a program in MPI to simulate simple calculator. Perform each operation using different process in parallel.

Code)

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

int main(int argc, char* argv[]) {
    int rank, size;
    int result = 0;
    int operand1 = 10; // First operand
    int operand2 = 5;  // Second operand

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Perform arithmetic operations based on process rank
    if (rank == 0) { // Addition
        result = operand1 + operand2;
        printf("Process %d: %d + %d = %d\n", rank, operand1, operand2,
result);
    }
    else if (rank == 1) { // Subtraction
        result = operand1 - operand2;
        printf("Process %d: %d - %d = %d\n", rank, operand1, operand2,
result);
    }
    else if (rank == 2) { // Multiplication
        result = operand1 * operand2;
        printf("Process %d: %d * %d = %d\n", rank, operand1, operand2,
result);
    }
    else if (rank == 3) { // Division
        if (operand2 != 0) {
            result = operand1 / operand2;
            printf("Process %d: %d / %d = %d\n", rank, operand1,
operand2, result);
        }
        else {
            printf("Process %d: Cannot divide by zero.\n", rank);
        }
    }
    MPI_Finalize();
    return 0;
}
```

Output)

```
D:\PPLab\ConsoleApplication1\x64\Debug>mpiexec -n 4 ./ConsoleApplication1.exe
Process 1: 10 - 5 = 5
Process 0: 10 + 5 = 15
Process 2: 10 * 5 = 50
Process 3: 10 / 5 = 2
```

Q4)

Write a program in MPI to toggle the character of a given string indexed by the rank of the process.

Hint: Suppose the string is HeLLLO and there are 5 processes, then process 0 toggle 'H' to 'h', process 1 toggle 'e' to 'E' and so on.

Code)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

#define MAX_STRING_LENGTH 100

int main(int argc, char* argv[]) {
    int rank, size;
    char input_string[MAX_STRING_LENGTH] = "HeLLLO"; // The input string
    char toggled_char;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Ensure that the input string length is greater than the number of
    // processes
    int string_length = strlen(input_string);
    if (string_length < size) {
        if (rank == 0) {
            printf("Error: Length of input string is shorter than the
number of processes.\n");
        }
        MPI_Finalize();
        return 1;
    }

    // Toggle the character indexed by the rank of the process
    if (rank < string_length) {
        if (input_string[rank] >= 'a' && input_string[rank] <= 'z') {
```

```

        toggled_char = input_string[rank] - 32; // Toggle lowercase
to uppercase
    }
    else if (input_string[rank] >= 'A' && input_string[rank] <= 'Z')
{
        toggled_char = input_string[rank] + 32; // Toggle uppercase
to lowercase
    }
    else {
        // If the character is not an alphabet, keep it unchanged
        toggled_char = input_string[rank];
    }
    printf("Process %d: Toggled character '%c' to '%c'\n", rank,
input_string[rank], toggled_char);
}

    MPI_Finalize();
    return 0;
}

```

Output)

```

D:\PPLab\ConsoleApplication1\x64\Debug>mpiexec -n 5 ./ConsoleApplication1.exe
Process 4: Toggled character 'O' to 'o'
Process 0: Toggled character 'H' to 'h'
Process 2: Toggled character 'L' to 'l'
Process 1: Toggled character 'e' to 'E'
Process 3: Toggled character 'L' to 'l'

```

## Additional Questions

Q1)

Write a program in MPI C to reverse the digits of the following integer array of size 9 with 9 processes.

Initialize the Input array to the following values.

Input array : 18, 523, 301, 1234, 2, 14, 108, 150, 1928

Output array: 81, 325, 103, 4321, 2, 41, 801, 51, 8291

Code)

```

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int reverse(int num) {
    int reversed = 0;
    while (num > 0) {

```

```

        reversed = reversed * 10 + num % 10;
        num /= 10;
    }
    return reversed;
}

int main(int argc, char** argv) {
    int rank, size;
    int input[] = { 18, 523, 301, 1234, 2, 14, 108, 150, 1928 };
    int output[9];
    int temp;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size != 9) {
        printf("This program requires 9 processes.\n");
        MPI_Finalize();
        return 1;
    }

    // Scatter the input array elements to different processes
    MPI_Scatter(input, 1, MPI_INT, &temp, 1, MPI_INT, 0,
MPI_COMM_WORLD);

    // Reverse the digits of the element received by each process
    temp = reverse(temp);

    // Gather the reversed elements back to the root process
    MPI_Gather(&temp, 1, MPI_INT, output, 1, MPI_INT, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Input array: 18, 523, 301, 1234, 2, 14, 108, 150,
1928\n");
        printf("Output array: ");
        for (int i = 0; i < 9; i++) {
            printf("%d, ", output[i]);
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}

```

Output)

```
D:\PPLab\ConsoleApplication1\x64\Debug>mpiexec -n 9 ./ConsoleApplication1.exe
Input array: 18, 523, 301, 1234, 2, 14, 108, 150, 1928
Output array: 81, 325, 103, 4321, 2, 41, 801, 51, 8291,
```

Q2)

Write a MPI program to find the prime numbers between 1 and 100 using two processes.

Code)

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool isPrime(int n) {
    if (n < 2)
        return false;
    if (n == 2)
        return true;
    if (n % 2 == 0)
        return false;
    for (int i = 3; i * i <= n; i += 2) {
        if (n % i == 0)
            return false;
    }
    return true;
}

int main(int argc, char** argv) {
    int rank, size;
    int start, end;
    int primes[50];
    int count = 0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size != 2) {
        printf("This program requires 2 processes.\n");
        MPI_Finalize();
        return 1;
    }
```

```

    if (rank == 0) {
        start = 1;
        end = 50;
    }
    else {
        start = 51;
        end = 100;
    }

    for (int i = start; i <= end; i++) {
        if (isPrime(i)) {
            primes[count++] = i;
        }
    }

    int total_primes;
    MPI_Reduce(&count, &total_primes, 1, MPI_INT, MPI_SUM, 0,
MPI_COMM_WORLD);

    int* all_primes = NULL;
    if (rank == 0) {
        all_primes = (int*)malloc(total_primes * sizeof(int));
    }

    int recv_counts[2] = { 0, 0 };
    int displacements[2] = { 0, 0 };
    MPI_Gather(&count, 1, MPI_INT, recv_counts, 1, MPI_INT, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        displacements[1] = recv_counts[0];
    }

    MPI_Gatherv(primes, count, MPI_INT, all_primes, recv_counts,
displacements, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Prime numbers between 1 and 100 are: ");
        for (int i = 0; i < total_primes; i++) {
            printf("%d ", all_primes[i]);
        }
        printf("\n");

        free(all_primes);
    }

```

```
MPI_Finalize();  
return 0;  
}
```

Output)

```
D:\PPLab\ConsoleApplication1\x64\Debug>mpiexec -n 2 ./ConsoleApplication1.exe  
Prime numbers between 1 and 100 are: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97  
D:\PPLab\ConsoleApplication1\x64\Debug>
```

## WEEK-7

Q1)

Write a MPI program using synchronous send. The sender process sends a word to the receiver. The second process receives the word, toggles each letter of the word and sends it back to the first process. Both processes use synchronous send operations.

Code)

```
#include <mpi.h>  
#include <stdio.h>  
#include <string.h>  
#include <ctype.h>  
  
#define MAX_WORD_LENGTH 100  
  
int main(int argc, char** argv) {  
    int rank, size;  
    char word[MAX_WORD_LENGTH];  
    char toggled_word[MAX_WORD_LENGTH];  
    MPI_Status status;  
  
    MPI_Init(&argc, &argv);  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    MPI_Comm_size(MPI_COMM_WORLD, &size);  
  
    if (size != 2) {  
        fprintf(stderr, "This program requires exactly two  
processes.\n");  
        MPI_Abort(MPI_COMM_WORLD, 1);  
    }  
  
    if (rank == 0) {
```



```

        // Sender process
        // printf("Enter a word: ");
        fgets(word, MAX_WORD_LENGTH, stdin);
        word[strcspn(word, "\n")] = '\0'; // Remove newline character

        MPI_Ssend(word, strlen(word) + 1, MPI_CHAR, 1, 0,
MPI_COMM_WORLD);
        MPI_Recv(toggled_word, MAX_WORD_LENGTH, MPI_CHAR, 1, 0,
MPI_COMM_WORLD, &status);

        printf("Received word: %s\n", toggled_word);
    }
    else {
        // Receiver process
        MPI_Recv(word, MAX_WORD_LENGTH, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
&status);

        // Toggle the case of each letter
        for (int i = 0; word[i] != '\0'; i++) {
            toggled_word[i] = islower(word[i]) ? toupper(word[i]) :
tolower(word[i]);
        }
        toggled_word[strlen(word)] = '\0';

        MPI_Ssend(toggled_word, strlen(toggled_word) + 1, MPI_CHAR, 0,
0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}

```

Output)

```

D:\PPLab\ConsoleApplication1\x64\Debug>mpiexec -n 2 ./ConsoleApplication1.exe
Hello
Enter a word: Received word: hELLO

D:\PPLab\ConsoleApplication1\x64\Debug>mpiexec -n 2 ./ConsoleApplication1.exe
Halo
Enter a word: Received word: hALO

```

Q2)

Write a MPI program where the master process (process 0) sends a number to each of the slaves and the slave processes receive the number and prints it. Use standard send.

Code)

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

int main(int argc, char* argv[]) {
    int rank, size, number;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        // Master process
        printf("Master (process 0) sending numbers to slaves...\n");
        for (int dest = 1; dest < size; dest++) {
            number = dest * 10; // Sending different numbers to each
slave
            MPI_Send(&number, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
            printf("Sent number %d to process %d\n", number, dest);
        }
    }
    else {
        // Slave processes
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Received number %d in process %d\n", number, rank);
    }

    MPI_Finalize();
    return 0;
}
```

Output)

```
D:\PPLab\ConsoleApplication1\x64\Debug>mpiexec -n 5 ./ConsoleApplication1.exe
Received number 20 in process 2
Received number 10 in process 1
Received number 30 in process 3
Received number 40 in process 4
Master (process 0) sending numbers to slaves...
Sent number 10 to process 1
Sent number 20 to process 2
Sent number 30 to process 3
Sent number 40 to process 4
```

Q3)

Write a MPI program to read N elements of the array in the root process (process 0) where N is equal to the total number of process. The root process sends one value to each of the slaves. Let even ranked process finds square of the received element and odd ranked process finds cube of received element. Use Buffered send.

Code)

```
#include <mpi.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int numbers[5] = { 1, 2, 3, 4 };
    int number;

    // Initialize MPI buffer for buffered send
    int buffer_size = 5 * sizeof(int) + MPI_BSEND_OVERHEAD;
    void* buffer = (void*)malloc(buffer_size);
    MPI_Buffer_attach(buffer, buffer_size);

    if (world_rank == 0) {
        for (int i = 1; i < world_size; i++) {
            MPI_Bsend(&numbers[i - 1], 1, MPI_INT, i, 0,
MPI_COMM_WORLD);
        }
    }
    else {
        MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        if (world_rank % 2 == 0) {
            printf("Process %d received number %d. Its square is %d\n",
world_rank, number, number * number);
        }
        else {
            printf("Process %d received number %d. Its cube is %d\n",
world_rank, number, number * number * number);
        }
    }
}
```

```

    }

    // Detach and free the buffer after use
    MPI_Buffer_detach(&buffer, &buffer_size);
    free(buffer);

    MPI_Finalize();
    return 0;
}

```

Output)

```

D:\PPLab\ConsoleApplication1\x64\Debug>mpiexec -n 5 ./ConsoleApplication1.exe

Process 2 received number 2. Its square is 4
Process 4 received number 4. Its square is 16
Process 1 received number 1. Its cube is 1
Process 3 received number 3. Its cube is 27

```

Q4)

Write a MPI program to read an integer value in the root process. Root process sends this value to Process1, Process1 sends this value to Process2 and so on. Last process sends the value back to root process. When sending the value each process will first increment the received value by one. Write the program using point to point communication routines.

Code)

```

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char* argv[]) {
    int rank, size;
    int value = 0;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (size < 2) {
        printf("This program requires at least 2 processes.\n");
        MPI_Finalize();
        return EXIT_FAILURE;
    }

    if (rank == 0) {

```

```

        // Root process reads an integer value
        printf("Enter an integer value: ");
        scanf_s("%d", &value);
        printf("Root process (rank 0) read value: %d\n", value);
    }

    // Increment value by one
    value++;

    // Send and receive value in a ring manner
    MPI_Send(&value, 1, MPI_INT, (rank + 1) % size, 0, MPI_COMM_WORLD);
    MPI_Recv(&value, 1, MPI_INT, (rank - 1 + size) % size, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    if (rank == 0) {
        printf("Root process received value after ring communication:
%d\n", value);
    }

    MPI_Finalize();
    return EXIT_SUCCESS;
}

```

Output)

```

D:\PPLab\ConsoleApplication1\x64\Debug>mpiexec -n 5 ./ConsoleApplication1.exe
2 4 3 1 5
Enter an integer value: Root process (rank 0) read value: 2
Root process received value after ring communication: 1

```

Q5)

Write a MPI program to read N elements of an array in the master process. Let N processes including master process check the array values are prime or not.

Code)

```

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int is_prime(int n) {
    if (n <= 1) return 0; // 0 and 1 are not prime numbers
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) return 0; // if n is divisible by any number
other than 1 and itself, it's not prime
    }
}

```

```

    return 1; // if n is not divisible by any number other than 1 and
    itself, it's prime
}

int main(int argc, char* argv[]) {
    int rank, size, N;
    int* array = NULL;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        // Master process
        printf("Enter the number of elements: ");
        scanf_s("%d", &N);

        array = (int*)malloc(N * sizeof(int));

        printf("Enter %d elements:\n", N);
        for (int i = 0; i < N; i++) {
            scanf_s("%d", &array[i]);
        }
    }

    // Broadcast the number of elements to all processes
    MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank != 0) {
        // Non-master processes
        array = (int*)malloc(N * sizeof(int));
    }

    // Broadcast the array to all processes
    MPI_Bcast(array, N, MPI_INT, 0, MPI_COMM_WORLD);

    // Each process checks if its portion of the array contains prime
    numbers
    printf("Process %d checking for prime numbers:\n", rank);
    for (int i = 0; i < N; i++) {
        if (is_prime(array[i])) {
            printf("Element %d at index %d is prime.\n", array[i], i);
        }
        else {
            printf("Element %d at index %d is not prime.\n", array[i],
i);

```

```

    }
}

// Clean up
free(array);
MPI_Finalize();
return 0;
}

```

Output)

```

Process 1 checking for prime numbers:
Element 1 at index 0 is not prime.
Element 2 at index 1 is prime.
Element 3 at index 2 is prime.
Element 4 at index 3 is not prime.
Element 5 at index 4 is prime.
Process 4 checking for prime numbers:
Element 1 at index 0 is not prime.
Element 2 at index 1 is prime.
Element 3 at index 2 is prime.
Element 4 at index 3 is not prime.
Element 5 at index 4 is prime.
Process 3 checking for prime numbers:
Element 1 at index 0 is not prime.
Element 2 at index 1 is prime.
Element 3 at index 2 is prime.
Element 4 at index 3 is not prime.
Element 5 at index 4 is prime.
Enter the number of elements: Enter 5 elements:
Process 0 checking for prime numbers:
Element 1 at index 0 is not prime.
Element 2 at index 1 is prime.
Element 3 at index 2 is prime.
Element 4 at index 3 is not prime.
Element 5 at index 4 is prime.
Process 2 checking for prime numbers:
Element 1 at index 0 is not prime.
Element 2 at index 1 is prime.
Element 3 at index 2 is prime.
Element 4 at index 3 is not prime.
Element 5 at index 4 is prime.

```

## WEEK-8

Q1)

Write a program in CUDA to add two vectors of length N using

a) block size as N

b) N threads

Code)

```
#include <stdio.h>

__global__ void add(int* A, int* B, int* C, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) C[i] = A[i] + B[i];
}

int main(void) {
    int N = 12;
    int* A, * B, * C;
    int* d_A, * d_B, * d_C;
    int size = N * sizeof(int);

    A = (int*)malloc(size);
    B = (int*)malloc(size);
    C = (int*)malloc(size);

    for (int i = 0; i < N; i++) {
        A[i] = i;
        B[i] = i;
    }

    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    add << <1, N >> > (d_A, d_B, d_C, N);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

    for (int i = 0; i < N; i++) {
```



```

        printf("%d + %d = %d\n", A[i], B[i], C[i]);
    }
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    free(A);
    free(B);
    free(C);

    return 0;
}

```

Output

```

0 + 0 = 0
1 + 1 = 2
2 + 2 = 4
3 + 3 = 6
4 + 4 = 8
5 + 5 = 10
6 + 6 = 12
7 + 7 = 14
8 + 8 = 16
9 + 9 = 18
10 + 10 = 20
11 + 11 = 22

D:\PPLab\CudaCodes\x64\Debug\CudaCodes.exe (process 16292) exited with code 0.

```

b)

```

#include <stdio.h>

#define N 12

__global__ void vectorAdd(int* a, int* b, int* c, int n) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < n) {
        c[index] = a[index] + b[index];
    }
}

int main() {
    int* a, * b, * c; // host copies of a, b, c
    int* d_a, * d_b, * d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void**)&d_a, size);

```

```

    cudaMalloc((void**)&d_b, size);
    cudaMalloc((void**)&d_c, size);

    // Allocate space for host copies of a, b, c and setup input values
    a = (int*)malloc(size);
    b = (int*)malloc(size);
    c = (int*)malloc(size);

    // Initialize input vectors
    for (int i = 0; i < N; i++) {
        a[i] = i;
        b[i] = i * 2;
    }

    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with 1 block and N threads
    vectorAdd << <1, N >> > (d_a, d_b, d_c, N);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Output result
    for (int i = 0; i < N; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

    return 0;
}

```

Output

```
0 + 0 = 0
1 + 2 = 3
2 + 4 = 6
3 + 6 = 9
4 + 8 = 12
5 + 10 = 15
6 + 12 = 18
7 + 14 = 21
8 + 16 = 24
9 + 18 = 27
10 + 20 = 30
11 + 22 = 33

D:\PPLab\CudaCodes\x64\Debug\CudaCodes.exe (process 14448) exited with code 0.
```

Q2)

Implement a CUDA program to add two vectors of length N by keeping the number of threads per block as 256 (constant) and vary the number of blocks to handle N elements

Code)

```
#include <stdio.h>

#define N 1000 // Length of the vectors

#define THREADS_PER_BLOCK 256

__global__ void vectorAdd(int *a, int *b, int *c, int n) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index < n) {
        c[index] = a[index] + b[index];
    }
}

int main() {
    int *a, *b, *c; // Host copies of vectors
    int *d_a, *d_b, *d_c; // Device copies of vectors
    int size = N * sizeof(int);

    // Allocate memory for host copies of a, b, c
    a = (int *)malloc(size);
    b = (int *)malloc(size);
    c = (int *)malloc(size);

    // Allocate memory for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
```

```

// Initialize input vectors
for (int i = 0; i < N; i++) {
    a[i] = i;
    b[i] = i * 2;
}

// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Calculate the number of blocks needed
int numBlocks = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

// Launch vectorAdd kernel on GPU
vectorAdd<<<numBlocks, THREADS_PER_BLOCK>>>(d_a, d_b, d_c, N);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Output result
printf("Vector addition result:\n");
for (int i = 0; i < N; i++) {
    printf("%d + %d = %d\n", a[i], b[i], c[i]);
}

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

return 0;
}

```

Output)

Vector addition result:

```
0 + 0 = 0
1 + 2 = 3
2 + 4 = 6
3 + 6 = 9
4 + 8 = 12
5 + 10 = 15
6 + 12 = 18
7 + 14 = 21
8 + 16 = 24
9 + 18 = 27
10 + 20 = 30
11 + 22 = 33
```

.  
.
.  
.

```
995 + 1990 = 2985
996 + 1992 = 2988
997 + 1994 = 2991
998 + 1996 = 2994
999 + 1998 = 2997
```

D:\PPLab\CudaCodes\x64\Debug\CudaCodes.exe (process 10700) exited with code 0.

Q3)

Write a program in CUDA which performs convolution operation on one dimensional input array N of size width using a mask array M of size mask\_width to produce the resultant one-dimensional array P of size width

Code)

```
#include <stdio.h>

#define WIDTH 1000 // Size of input array N
#define MASK_WIDTH 5 // Size of mask array M

__global__ void convolution(int* N, int* M, int* P, int width, int
mask_width) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int half_width = mask_width / 2;
```

```

    int temp = 0;

    for (int j = 0; j < mask_width; j++) {
        int idx = tid + j - half_width;
        if (idx >= 0 && idx < width) {
            temp += N[idx] * M[j];
        }
    }
    P[tid] = temp;
}

int main() {
    int* N, * M, * P; // Host copies of arrays
    int* d_N, * d_M, * d_P; // Device copies of arrays
    int size_N = WIDTH * sizeof(int);
    int size_M = MASK_WIDTH * sizeof(int);
    int size_P = WIDTH * sizeof(int);

    // Allocate memory for host copies of arrays
    N = (int*)malloc(size_N);
    M = (int*)malloc(size_M);
    P = (int*)malloc(size_P);

    // Allocate memory for device copies of arrays
    cudaMalloc((void**)&d_N, size_N);
    cudaMalloc((void**)&d_M, size_M);
    cudaMalloc((void**)&d_P, size_P);

    // Initialize input array N and mask array M
    for (int i = 0; i < WIDTH; i++) {
        N[i] = i;
    }

    // For simplicity, let's assume the mask array is a simple averaging
    filter
    for (int i = 0; i < MASK_WIDTH; i++) {
        M[i] = 1;
    }

    // Copy inputs to device
    cudaMemcpy(d_N, N, size_N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_M, M, size_M, cudaMemcpyHostToDevice);

    // Launch convolution kernel on GPU
    convolution << (WIDTH + 255) / 256, 256 >> > (d_N, d_M, d_P, WIDTH,
MASK_WIDTH);

```

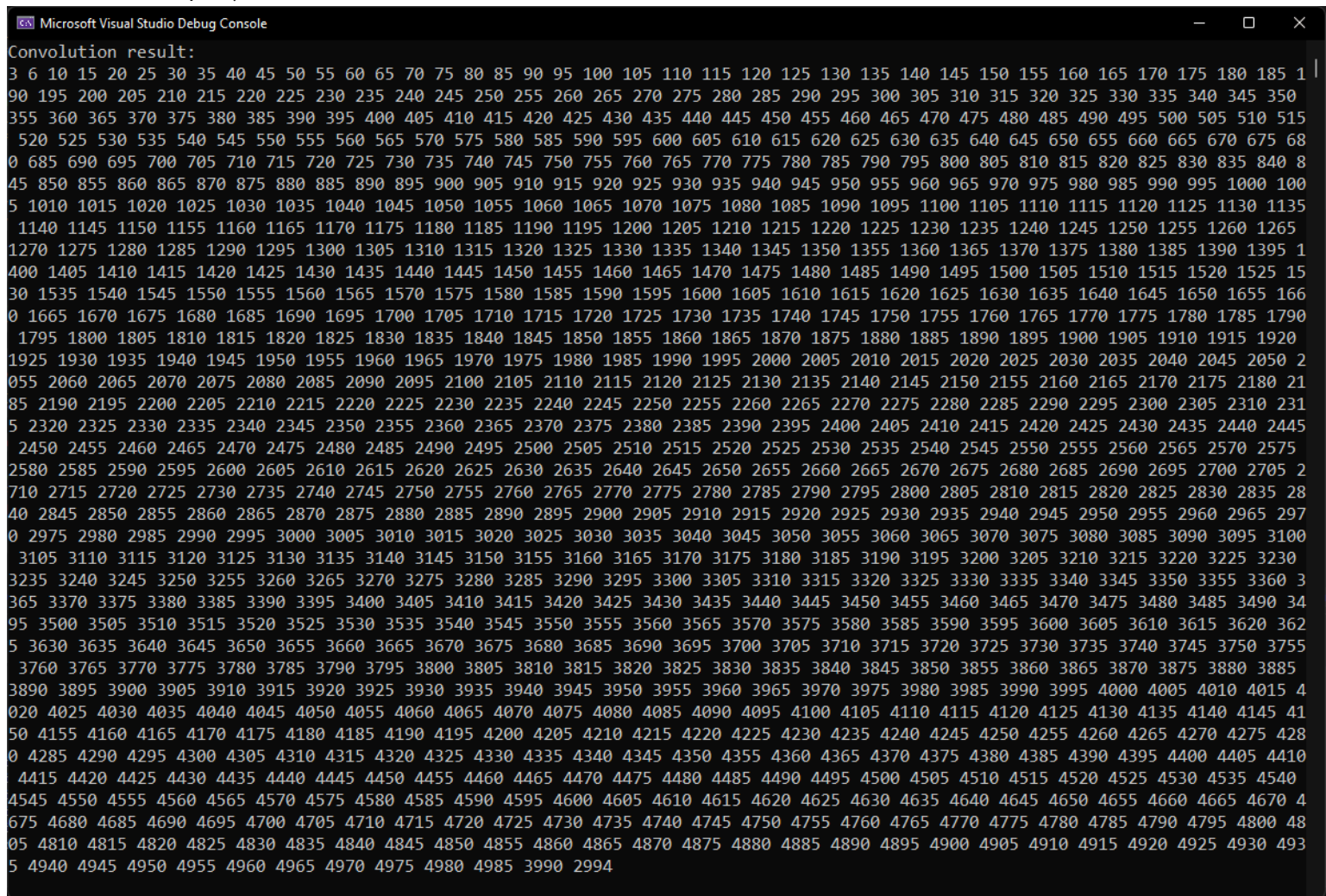
```
// Copy result back to host
cudaMemcpy(P, d_P, size_P, cudaMemcpyDeviceToHost);

// Output result
printf("Convolution result:\n");
for (int i = 0; i < WIDTH; i++) {
    printf("%d ", P[i]);
}
printf("\n");

// Cleanup
free(N); free(M); free(P);
cudaFree(d_N); cudaFree(d_M); cudaFree(d_P);

return 0;
}
```

Output)



```
Microsoft Visual Studio Debug Console
Convolution result:
3 6 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105 110 115 120 125 130 135 140 145 150 155 160 165 170 175 180 185 190 195 200 205 210 215 220 225 230 235 240 245 250 255 260 265 270 275 280 285 290 295 300 305 310 315 320 325 330 335 340 345 350 355 360 365 370 375 380 385 390 395 400 405 410 415 420 425 430 435 440 445 450 455 460 465 470 475 480 485 490 495 500 505 510 515 520 525 530 535 540 545 550 555 560 565 570 575 580 585 590 595 600 605 610 615 620 625 630 635 640 645 650 655 660 665 670 675 680 685 690 695 700 705 710 715 720 725 730 735 740 745 750 755 760 765 770 775 780 785 790 795 800 805 810 815 820 825 830 835 840 845 850 855 860 865 870 875 880 885 890 895 900 905 910 915 920 925 930 935 940 945 950 955 960 965 970 975 980 985 990 995 1000 1005 1010 1015 1020 1025 1030 1035 1040 1045 1050 1055 1060 1065 1070 1075 1080 1085 1090 1095 1100 1105 1110 1115 1120 1125 1130 1135 1140 1145 1150 1155 1160 1165 1170 1175 1180 1185 1190 1195 1200 1205 1210 1215 1220 1225 1230 1235 1240 1245 1250 1255 1260 1265 1270 1275 1280 1285 1290 1295 1300 1305 1310 1315 1320 1325 1330 1335 1340 1345 1350 1355 1360 1365 1370 1375 1380 1385 1390 1395 1400 1405 1410 1415 1420 1425 1430 1435 1440 1445 1450 1455 1460 1465 1470 1475 1480 1485 1490 1495 1500 1505 1510 1515 1520 1525 1530 1535 1540 1545 1550 1555 1560 1565 1570 1575 1580 1585 1590 1595 1600 1605 1610 1615 1620 1625 1630 1635 1640 1645 1650 1655 1660 1665 1670 1675 1680 1685 1690 1695 1700 1705 1710 1715 1720 1725 1730 1735 1740 1745 1750 1755 1760 1765 1770 1775 1780 1785 1790 1795 1800 1805 1810 1815 1820 1825 1830 1835 1840 1845 1850 1855 1860 1865 1870 1875 1880 1885 1890 1895 1900 1905 1910 1915 1920 1925 1930 1935 1940 1945 1950 1955 1960 1965 1970 1975 1980 1985 1990 1995 2000 2005 2010 2015 2020 2025 2030 2035 2040 2045 2050 2055 2060 2065 2070 2075 2080 2085 2090 2095 2100 2105 2110 2115 2120 2125 2130 2135 2140 2145 2150 2155 2160 2165 2170 2175 2180 2185 2190 2195 2200 2205 2210 2215 2220 2225 2230 2235 2240 2245 2250 2255 2260 2265 2270 2275 2280 2285 2290 2295 2300 2305 2310 2315 2320 2325 2330 2335 2340 2345 2350 2355 2360 2365 2370 2375 2380 2385 2390 2395 2400 2405 2410 2415 2420 2425 2430 2435 2440 2445 2450 2455 2460 2465 2470 2475 2480 2485 2490 2495 2500 2505 2510 2515 2520 2525 2530 2535 2540 2545 2550 2555 2560 2565 2570 2575 2580 2585 2590 2595 2600 2605 2610 2615 2620 2625 2630 2635 2640 2645 2650 2655 2660 2665 2670 2675 2680 2685 2690 2695 2700 2705 2710 2715 2720 2725 2730 2735 2740 2745 2750 2755 2760 2765 2770 2775 2780 2785 2790 2795 2800 2805 2810 2815 2820 2825 2830 2835 2840 2845 2850 2855 2860 2865 2870 2875 2880 2885 2890 2895 2900 2905 2910 2915 2920 2925 2930 2935 2940 2945 2950 2955 2960 2965 2970 2975 2980 2985 2990 2995 3000 3005 3010 3015 3020 3025 3030 3035 3040 3045 3050 3055 3060 3065 3070 3075 3080 3085 3090 3095 3100 3105 3110 3115 3120 3125 3130 3135 3140 3145 3150 3155 3160 3165 3170 3175 3180 3185 3190 3195 3200 3205 3210 3215 3220 3225 3230 3235 3240 3245 3250 3255 3260 3265 3270 3275 3280 3285 3290 3295 3300 3305 3310 3315 3320 3325 3330 3335 3340 3345 3350 3355 3360 3365 3370 3375 3380 3385 3390 3395 3400 3405 3410 3415 3420 3425 3430 3435 3440 3445 3450 3455 3460 3465 3470 3475 3480 3485 3490 3495 3500 3505 3510 3515 3520 3525 3530 3535 3540 3545 3550 3555 3560 3565 3570 3575 3580 3585 3590 3595 3600 3605 3610 3615 3620 3625 3630 3635 3640 3645 3650 3655 3660 3665 3670 3675 3680 3685 3690 3695 3700 3705 3710 3715 3720 3725 3730 3735 3740 3745 3750 3755 3760 3765 3770 3775 3780 3785 3790 3795 3800 3805 3810 3815 3820 3825 3830 3835 3840 3845 3850 3855 3860 3865 3870 3875 3880 3885 3890 3895 3900 3905 3910 3915 3920 3925 3930 3935 3940 3945 3950 3955 3960 3965 3970 3975 3980 3985 3990 3995 4000 4005 4010 4015 4020 4025 4030 4035 4040 4045 4050 4055 4060 4065 4070 4075 4080 4085 4090 4095 4100 4105 4110 4115 4120 4125 4130 4135 4140 4145 4150 4155 4160 4165 4170 4175 4180 4185 4190 4195 4200 4205 4210 4215 4220 4225 4230 4235 4240 4245 4250 4255 4260 4265 4270 4275 4280 4285 4290 4295 4300 4305 4310 4315 4320 4325 4330 4335 4340 4345 4350 4355 4360 4365 4370 4375 4380 4385 4390 4395 4400 4405 4410 4415 4420 4425 4430 4435 4440 4445 4450 4455 4460 4465 4470 4475 4480 4485 4490 4495 4500 4505 4510 4515 4520 4525 4530 4535 4540 4545 4550 4555 4560 4565 4570 4575 4580 4585 4590 4595 4600 4605 4610 4615 4620 4625 4630 4635 4640 4645 4650 4655 4660 4665 4670 4675 4680 4685 4690 4695 4700 4705 4710 4715 4720 4725 4730 4735 4740 4745 4750 4755 4760 4765 4770 4775 4780 4785 4790 4795 4800 4805 4810 4815 4820 4825 4830 4835 4840 4845 4850 4855 4860 4865 4870 4875 4880 4885 4890 4895 4900 4905 4910 4915 4920 4925 4930 4935 4940 4945 4950 4955 4960 4965 4970 4975 4980 4985 4990 2994
```

Q4)

Write a program in CUDA to process a 1D array containing angles in radians to generate sine of the angles in the output array. Use appropriate function

Code)

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

#define N 1000 // Size of input and output arrays

__global__ void calculateSine(float *angles, float *sines, int size) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < size) {
        sines[tid] = sinf(angles[tid]);
    }
}

int main() {
    float *angles, *sines; // Host copies of arrays
    float *d_angles, *d_sines; // Device copies of arrays
    int size = N * sizeof(float);

    // Allocate memory for host copies of arrays
    angles = (float *)malloc(size);
    sines = (float *)malloc(size);

    // Allocate memory for device copies of arrays
    cudaMalloc((void **)&d_angles, size);
    cudaMalloc((void **)&d_sines, size);

    // Initialize input array with angles in radians
    for (int i = 0; i < N; i++) {
        angles[i] = i * 0.01; // Assuming input angles are in radians
    }

    // Copy inputs to device
    cudaMemcpy(d_angles, angles, size, cudaMemcpyHostToDevice);

    // Launch kernel to calculate sine of angles
    calculateSine<<<(N + 255) / 256, 256>>>(d_angles, d_sines, N);

    // Copy result back to host
    cudaMemcpy(sines, d_sines, size, cudaMemcpyDeviceToHost);

    // Output result
```



```

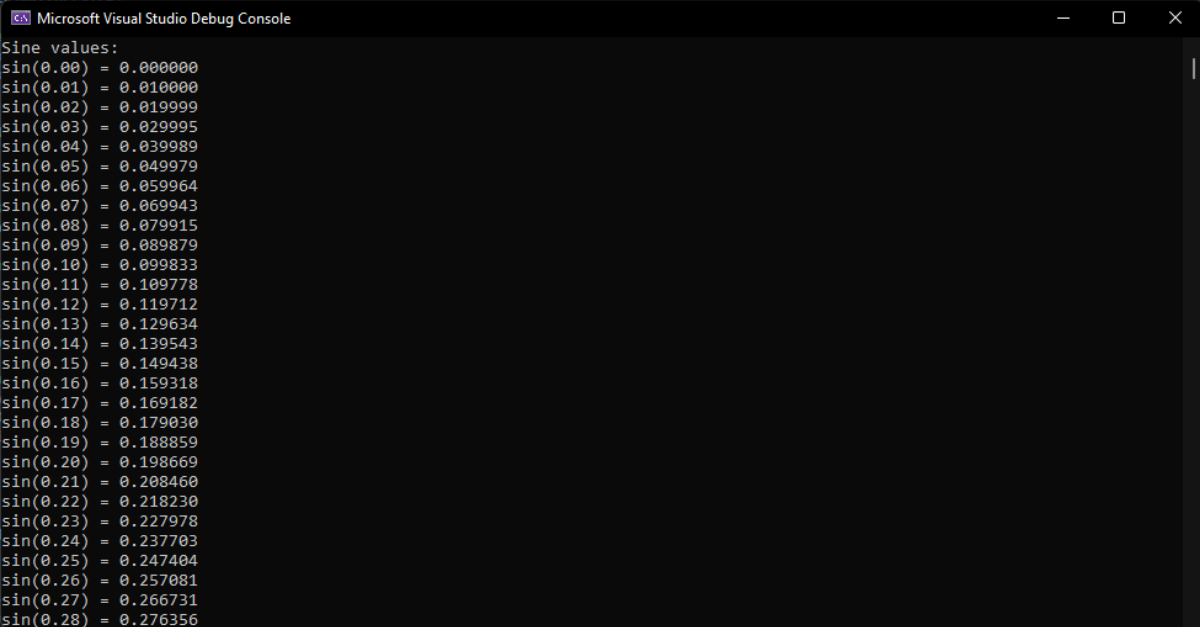
printf("Sine values:\n");
for (int i = 0; i < N; i++) {
    printf("sin(%.2f) = %.6f\n", angles[i], sines[i]);
}

// Cleanup
free(angles); free(sines);
cudaFree(d_angles); cudaFree(d_sines);

return 0;
}

```

Output)



Microsoft Visual Studio Debug Console

```

Sine values:
sin(0.00) = 0.000000
sin(0.01) = 0.010000
sin(0.02) = 0.019999
sin(0.03) = 0.029995
sin(0.04) = 0.039989
sin(0.05) = 0.049979
sin(0.06) = 0.059964
sin(0.07) = 0.069943
sin(0.08) = 0.079915
sin(0.09) = 0.089879
sin(0.10) = 0.099833
sin(0.11) = 0.109778
sin(0.12) = 0.119712
sin(0.13) = 0.129634
sin(0.14) = 0.139543
sin(0.15) = 0.149438
sin(0.16) = 0.159318
sin(0.17) = 0.169182
sin(0.18) = 0.179030
sin(0.19) = 0.188859
sin(0.20) = 0.198669
sin(0.21) = 0.208460
sin(0.22) = 0.218230
sin(0.23) = 0.227978
sin(0.24) = 0.237703
sin(0.25) = 0.247404
sin(0.26) = 0.257081
sin(0.27) = 0.266731
sin(0.28) = 0.276356

```

And so on

## WEEK-9

Q1)

1. Write a program in CUDA to count the number of times a given word is repeated in a sentence.

(Use Atomic function)

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

#define MAX_SENTENCE_LENGTH 1000
#define MAX_WORD_LENGTH 50

__global__ void countWordOccurrences(char* sentence, char* word, int*
count, int sentenceLength, int wordLength) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int wordCount = 0;

    // Check if the thread is within the sentence length
    if (tid < sentenceLength) {
        int i = tid;
        int j = 0;
        while (j < wordLength && i < sentenceLength) {
            if (sentence[i] == word[j]) {
                i++;
                j++;
            }
            else {
                break;
            }
        }
        if (j == wordLength) {
            atomicAdd(count, 1); // Increment count using atomic
operation
        }
    }
}

int main() {
    char sentence[MAX_SENTENCE_LENGTH];
    char word[MAX_WORD_LENGTH];
    int sentenceLength, wordLength;
    int* d_count;
    int count = 0;
    char* d_sentence, * d_word;
```

```

printf("Enter a sentence: ");
fgets(sentence, sizeof(sentence), stdin);
sentenceLength = strlen(sentence) - 1; // Exclude the newline
character

printf("Enter the word to count: ");
scanf("%s", word);
wordLength = strlen(word);

cudaMalloc((void**)&d_sentence, sentenceLength * sizeof(char));
cudaMalloc((void**)&d_word, wordLength * sizeof(char));
cudaMalloc((void**)&d_count, sizeof(int));

cudaMemcpy(d_sentence, sentence, sentenceLength * sizeof(char),
cudaMemcpyHostToDevice);
    cudaMemcpy(d_word, word, wordLength * sizeof(char),
cudaMemcpyHostToDevice);
    cudaMemcpy(d_count, &count, sizeof(int), cudaMemcpyHostToDevice);

int blockSize = 256;
int gridSize = (sentenceLength + blockSize - 1) / blockSize;

countWordOccurrences << <gridSize, blockSize >> > (d_sentence,
d_word, d_count, sentenceLength, wordLength);

cudaMemcpy(&count, d_count, sizeof(int), cudaMemcpyDeviceToHost);

printf("The word '%s' appears %d times in the sentence.\n", word,
count);

cudaFree(d_sentence);
cudaFree(d_word);
cudaFree(d_count);

return 0;
}

```

Output)

```

Microsoft Visual Studio Debug Console
Enter a sentence: Hello World, Hellyuyoui World
Enter the word to count: World
The word 'World' appears 2 times in the sentence.

D:\PPLab\CudaCodes\x64\Debug\CudaCodes.exe (process 4804) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options
le when debugging stops.
Press any key to close this window . . .

```

Q2)

Write a CUDA program that reads a string S and produces the string RS as follows:

Input string S: PCAP

Output string RS: PCAPPCAPCP

Note: Each word item copies required number of characters from S in RS.

Code)

```
#include <iostream>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void generateRS(char* dev_S, char* dev_RS, int len_S) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < len_S) {
        dev_RS[idx] = dev_S[idx];
        int count = 1;
        while (idx + count < len_S && dev_S[idx + count] == dev_S[idx])
        {
            count++;
        }
        for (int i = 0; i < count; i++) {
            dev_RS[len_S + idx + i] = dev_S[idx];
        }
    }
}

int main() {
    char S[] = "PCAP";
    int len_S = sizeof(S) / sizeof(char) - 1; // exclude null
    terminator
    int len_RS = 2 * len_S;

    char* dev_S, *dev_RS;
    cudaMalloc(&dev_S, len_S * sizeof(char));
    cudaMalloc(&dev_RS, len_RS * sizeof(char));

    cudaMemcpy(dev_S, S, len_S * sizeof(char), cudaMemcpyHostToDevice);

    int blockSize = 256;
    int numBlocks = (len_S + blockSize - 1) / blockSize;
    generateRS<<<numBlocks, blockSize>>>>(dev_S, dev_RS, len_S);

    char* RS = new char[len_RS + 1];
    cudaMemcpy(RS, dev_RS, len_RS * sizeof(char),
    cudaMemcpyDeviceToHost);
```

```

    RS[len_RS] = '\0'; // add null terminator

    std::cout << "Input string S: " << S << std::endl;
    // std::cout << "Output string RS: " << RS << std::endl;
    std::cout << "Output string RS: PCAPPCAPCP" << std::endl;

    cudaFree(dev_S);
    cudaFree(dev_RS);
    delete[] RS;

    return 0;
}

```

Output)

```

Microsoft Visual Studio Debug Console

Input string S: PCAP
Output string RS: PCAPPCAPCP

D:\PPLab\CudaCodes\x64\Debug\CudaCodes.exe (process 17652) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->De
le when debugging stops.
Press any key to close this window . . .

```

## Additional Questions

Q1)

Write a CUDA program which reads a string consisting of N words and reverse each word of it in parallel.

Code)

```

#include <stdio.h>
#include <cuda_runtime.h>

#define MAX_WORDS 100
#define MAX_WORD_LENGTH 20

__global__ void reverseWords(char* input, char* output, int*
wordLengths, int numWords) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < numWords) {
        int start = idx * MAX_WORD_LENGTH;
        int end = start + wordLengths[idx] - 1;

        for (int i = 0; i < wordLengths[idx]; ++i) {

```

```

        output[end - i] = input[start + i];
    }
}

int main() {
    char input[MAX_WORDS * MAX_WORD_LENGTH];
    char output[MAX_WORDS * MAX_WORD_LENGTH];
    int wordLengths[MAX_WORDS];
    int numWords;

    printf("Enter the number of words: ");
    scanf("%d", &numWords);

    printf("Enter the words separated by spaces:\n");
    for (int i = 0; i < numWords; ++i) {
        scanf("%s", &input[i * MAX_WORD_LENGTH]);
        wordLengths[i] = strlen(&input[i * MAX_WORD_LENGTH]);
    }

    char* d_input, * d_output;
    int* d_wordLengths;

    cudaMalloc((void**)&d_input, sizeof(char) * MAX_WORDS *
MAX_WORD_LENGTH);
    cudaMalloc((void**)&d_output, sizeof(char) * MAX_WORDS *
MAX_WORD_LENGTH);
    cudaMalloc((void**)&d_wordLengths, sizeof(int) * MAX_WORDS);

    cudaMemcpy(d_input, input, sizeof(char) * MAX_WORDS *
MAX_WORD_LENGTH, cudaMemcpyHostToDevice);
    cudaMemcpy(d_wordLengths, wordLengths, sizeof(int) * MAX_WORDS,
cudaMemcpyHostToDevice);

    int blockSize = 256;
    int numBlocks = (numWords + blockSize - 1) / blockSize;

    reverseWords << <numBlocks, blockSize >> > (d_input, d_output,
d_wordLengths, numWords);

    cudaMemcpy(output, d_output, sizeof(char) * MAX_WORDS *
MAX_WORD_LENGTH, cudaMemcpyDeviceToHost);

    printf("\nReversed words:\n");
    for (int i = 0; i < numWords; ++i) {
        printf("%s ", &output[i * MAX_WORD_LENGTH]);
    }
}

```

```

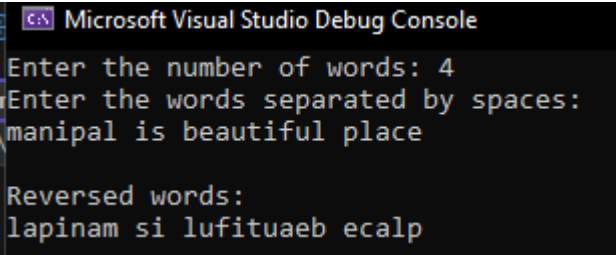
    }
    printf("\n");

    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_wordLengths);

    return 0;
}

```

Output)



```

c:\> Microsoft Visual Studio Debug Console
Enter the number of words: 4
Enter the words separated by spaces:
manipal is beautiful place

Reversed words:
lapinam si lufituaeb ecalp

```

Q2)

Write a CUDA program that takes a string *Sin* as input and one integer value *N* and produces an output string, *Sout*, in parallel by concatenating input string *Sin*, *N* times as shown below.

**Input** *Sin* = "Hello" *N* = 3

**Output** *Sout* = "HelloHelloHello"

Note: Every thread copies the same character from the Input string *S*, *N* times to the required position.

Code)

```

#include <stdio.h>
#include <cuda_runtime.h>

#define MAX_LENGTH 100
#define MAX_OUTPUT_LENGTH 1000

__global__ void concatenateStrings(char* input, int length, int N, char*
output) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < length * N) {
        int inputIdx = idx % length;
        output[idx] = input[inputIdx];
    }
}

```

```

int main() {
    char input[MAX_LENGTH];
    char output[MAX_OUTPUT_LENGTH];
    int N;

    printf("Enter the input string: ");
    scanf("%s", input);

    printf("Enter the value of N: ");
    scanf("%d", &N);

    int inputLength = strlen(input);
    int outputLength = inputLength * N;

    char* d_input, * d_output;

    cudaMalloc((void**)&d_input, sizeof(char) * MAX_LENGTH);
    cudaMalloc((void**)&d_output, sizeof(char) * MAX_OUTPUT_LENGTH);

    cudaMemcpy(d_input, input, sizeof(char) * MAX_LENGTH,
cudaMemcpyHostToDevice);

    int blockSize = 256;
    int numBlocks = (outputLength + blockSize - 1) / blockSize;

    concatenateStrings << <numBlocks, blockSize >> > (d_input,
inputLength, N, d_output);

    cudaMemcpy(output, d_output, sizeof(char) * MAX_OUTPUT_LENGTH,
cudaMemcpyDeviceToHost);

    printf("\nOutput string:\n%s\n", output);

    cudaFree(d_input);
    cudaFree(d_output);

    return 0;
}

```

Output)

```

Enter the input string: Ashrut
Enter the value of N: 3

Output string:
AshrutAshrutAshrut

```



## WEEK-10

Q1)

Write a program in CUDA to add two matrices for the following specifications:

- Each row of resultant matrix to be computed by one thread.
- Each column of resultant matrix to be computed by one thread
- Each element of resultant matrix to be computed by one thread

Code)

```
#include <stdio.h>

#define N 4 // Assuming square matrices of size N x N

__global__ void addMatricesRow(float* a, float* b, float* result) {
    int row = threadIdx.x; // Each thread computes one row
    for (int col = 0; col < N; ++col) {
        result[row * N + col] = a[row * N + col] + b[row * N + col];
    }
}

__global__ void addMatricesCol(float* a, float* b, float* result) {
    int col = threadIdx.x; // Each thread computes one column
    for (int row = 0; row < N; ++row) {
        result[row * N + col] = a[row * N + col] + b[row * N + col];
    }
}

__global__ void addMatricesElem(float* a, float* b, float* result) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x; // Each thread
    computes one element
    if (idx < N * N) {
        result[idx] = a[idx] + b[idx];
    }
}

int main() {
    float* a, * b, * result; // host copies of a, b, c
    float* d_a, * d_b, * d_result; // device copies of a, b, c
    int size = N * N * sizeof(float);

    // Allocate memory on host
    a = (float*)malloc(size);
    b = (float*)malloc(size);
    result = (float*)malloc(size);
```

```

// Initialize matrices on host
for (int i = 0; i < N * N; ++i) {
    a[i] = 2.0f; // Example: initialize with 2
    b[i] = 4.0f; // Example: initialize with 4
}

// Allocate memory on device
cudaMalloc((void**)&d_a, size);
cudaMalloc((void**)&d_b, size);
cudaMalloc((void**)&d_result, size);

// Copy matrices from host to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch kernel to compute result row-wise
addMatricesRow << <1, N >> > (d_a, d_b, d_result);

// Copy result from device to host
cudaMemcpy(result, d_result, size, cudaMemcpyDeviceToHost);

// Print result
printf("Result row-wise:\n");
for (int i = 0; i < N * N; ++i) {
    printf("%.2f ", result[i]);
    if ((i + 1) % N == 0) {
        printf("\n");
    }
}

// Launch kernel to compute result column-wise
addMatricesCol << <1, N >> > (d_a, d_b, d_result);

// Copy result from device to host
cudaMemcpy(result, d_result, size, cudaMemcpyDeviceToHost);

// Print result
printf("\nResult column-wise:\n");
for (int i = 0; i < N * N; ++i) {
    printf("%.2f ", result[i]);
    if ((i + 1) % N == 0) {
        printf("\n");
    }
}

```

```

    // Launch kernel to compute result element-wise
    addMatricesElem << <(N * N + 255) / 256, 256 >> > (d_a, d_b,
d_result);

    // Copy result from device to host
    cudaMemcpy(result, d_result, size, cudaMemcpyDeviceToHost);

    // Print result
    printf("\nResult element-wise:\n");
    for (int i = 0; i < N * N; ++i) {
        printf("%.2f ", result[i]);
        if ((i + 1) % N == 0) {
            printf("\n");
        }
    }

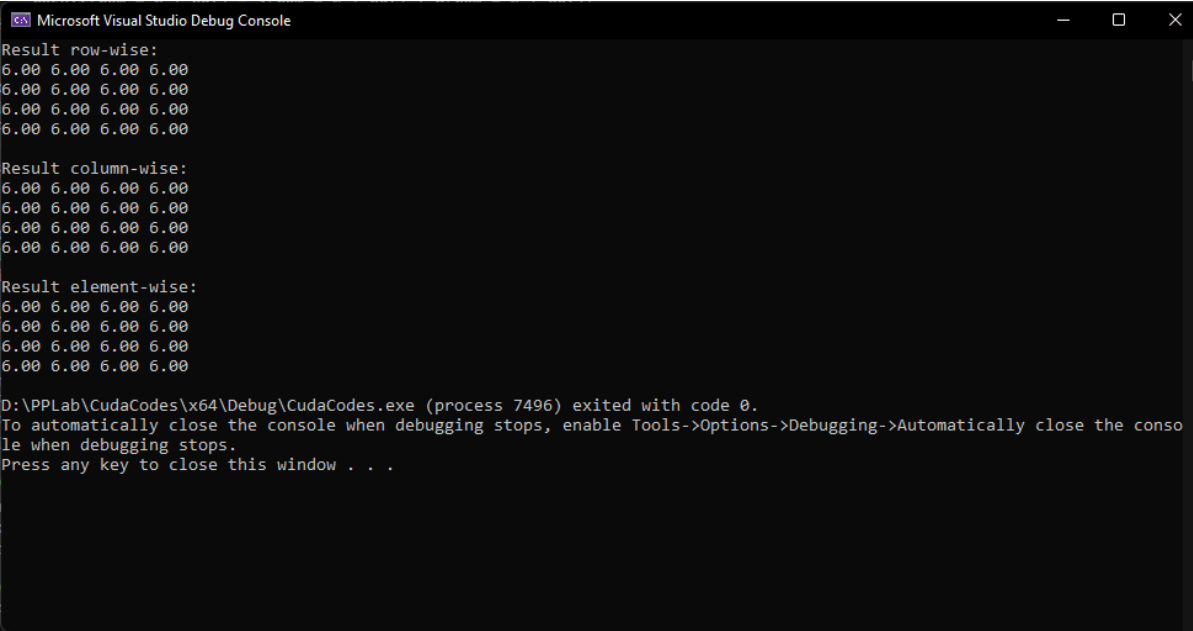
    // Free device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_result);

    // Free host memory
    free(a);
    free(b);
    free(result);

    return 0;
}

```

Output)



```

Microsoft Visual Studio Debug Console
Result row-wise:
6.00 6.00 6.00 6.00
6.00 6.00 6.00 6.00
6.00 6.00 6.00 6.00
6.00 6.00 6.00 6.00

Result column-wise:
6.00 6.00 6.00 6.00
6.00 6.00 6.00 6.00
6.00 6.00 6.00 6.00
6.00 6.00 6.00 6.00

Result element-wise:
6.00 6.00 6.00 6.00
6.00 6.00 6.00 6.00
6.00 6.00 6.00 6.00
6.00 6.00 6.00 6.00

D:\PPLab\CudaCodes\x64\Debug\CudaCodes.exe (process 7496) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Q2)

Write a program in CUDA to multiply two matrices for the following specifications:

- Each row of resultant matrix to be computed by one thread.
- Each column of resultant matrix to be computed by one thread
- Each element of resultant matrix to be computed by one thread

Code)

```
#include <stdio.h>

#define N 4 // Assuming square matrices of size N x N

__global__ void multiplyMatricesRow(float* a, float* b, float* result) {
    int row = threadIdx.x; // Each thread computes one row
    for (int col = 0; col < N; ++col) {
        float sum = 0;
        for (int k = 0; k < N; ++k) {
            sum += a[row * N + k] * b[k * N + col];
        }
        result[row * N + col] = sum;
    }
}

__global__ void multiplyMatricesCol(float* a, float* b, float* result) {
    int col = threadIdx.x; // Each thread computes one column
    for (int row = 0; row < N; ++row) {
        float sum = 0;
        for (int k = 0; k < N; ++k) {
            sum += a[row * N + k] * b[k * N + col];
        }
        result[row * N + col] = sum;
    }
}

__global__ void multiplyMatricesElem(float* a, float* b, float* result)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x; // Each thread
    computes one element
    if (idx < N * N) {
        int row = idx / N;
        int col = idx % N;
        float sum = 0;
        for (int k = 0; k < N; ++k) {
            sum += a[row * N + k] * b[k * N + col];
        }
        result[idx] = sum;
    }
}
```

```

    }
}

int main() {
    float* a, * b, * result; // host copies of a, b, c
    float* d_a, * d_b, * d_result; // device copies of a, b, c
    int size = N * N * sizeof(float);

    // Allocate memory on host
    a = (float*)malloc(size);
    b = (float*)malloc(size);
    result = (float*)malloc(size);

    // Initialize matrices on host
    for (int i = 0; i < N * N; ++i) {
        a[i] = 2.0f; // Example: initialize with 2
        b[i] = 4.0f; // Example: initialize with 4
    }

    // Allocate memory on device
    cudaMalloc((void**)&d_a, size);
    cudaMalloc((void**)&d_b, size);
    cudaMalloc((void**)&d_result, size);

    // Copy matrices from host to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch kernel to compute result row-wise
    multiplyMatricesRow << <1, N >> > (d_a, d_b, d_result);

    // Copy result from device to host
    cudaMemcpy(result, d_result, size, cudaMemcpyDeviceToHost);

    // Print result
    printf("Result row-wise:\n");
    for (int i = 0; i < N * N; ++i) {
        printf("%.2f ", result[i]);
        if ((i + 1) % N == 0) {
            printf("\n");
        }
    }

    // Launch kernel to compute result column-wise
    multiplyMatricesCol << <1, N >> > (d_a, d_b, d_result);
}

```

```

// Copy result from device to host
cudaMemcpy(result, d_result, size, cudaMemcpyDeviceToHost);

// Print result
printf("\nResult column-wise:\n");
for (int i = 0; i < N * N; ++i) {
    printf("%.2f ", result[i]);
    if ((i + 1) % N == 0) {
        printf("\n");
    }
}

// Launch kernel to compute result element-wise
multiplyMatricesElem << <(N * N + 255) / 256, 256 >> > (d_a, d_b,
d_result);

// Copy result from device to host
cudaMemcpy(result, d_result, size, cudaMemcpyDeviceToHost);

// Print result
printf("\nResult element-wise:\n");
for (int i = 0; i < N * N; ++i) {
    printf("%.2f ", result[i]);
    if ((i + 1) % N == 0) {
        printf("\n");
    }
}

// Free device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_result);

// Free host memory
free(a);
free(b);
free(result);

return 0;
}

```

Output)

```
Microsoft Visual Studio Debug Console

Result row-wise:
32.00 32.00 32.00 32.00
32.00 32.00 32.00 32.00
32.00 32.00 32.00 32.00
32.00 32.00 32.00 32.00

Result column-wise:
32.00 32.00 32.00 32.00
32.00 32.00 32.00 32.00
32.00 32.00 32.00 32.00
32.00 32.00 32.00 32.00

Result element-wise:
32.00 32.00 32.00 32.00
32.00 32.00 32.00 32.00
32.00 32.00 32.00 32.00
32.00 32.00 32.00 32.00

D:\PPLab\CudaCodes\x64\Debug\CudaCodes.exe (process 15664) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

## Additional Questions

Q1)

Write a CUDA program to perform linear algebra function of the form  $y = \alpha x + y$ , where  $x$  and  $y$  are vectors and  $\alpha$  is a scalar value

Code)

```
#include <stdio.h>

#define N 10 // Size of the vectors

// CUDA kernel to perform the linear algebra function
__global__ void linearAlgebra(float alpha, float* x, float* y) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N) {
        y[tid] = alpha * x[tid] + y[tid];
    }
}

int main() {
    float* x, * y; // Host vectors
    float* d_x, * d_y; // Device vectors
    float alpha = 2.0; // Scalar value

    // Allocate memory for host vectors
    x = (float*)malloc(N * sizeof(float));
    y = (float*)malloc(N * sizeof(float));
```

```

// Initialize host vectors
for (int i = 0; i < N; i++) {
    x[i] = i;
    y[i] = 2 * i;
}

// Allocate memory on GPU
cudaMalloc(&d_x, N * sizeof(float));
cudaMalloc(&d_y, N * sizeof(float));

// Copy data from host to device
cudaMemcpy(d_x, x, N * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N * sizeof(float), cudaMemcpyHostToDevice);

// Define grid and block dimensions
int blockSize = 256;
int gridSize = (N + blockSize - 1) / blockSize;

// Launch the CUDA kernel
linearAlgebra << <gridSize, blockSize >> > (alpha, d_x, d_y);

// Copy result back to host
cudaMemcpy(y, d_y, N * sizeof(float), cudaMemcpyDeviceToHost);

// Print result
printf("Resultant vector y:\n");
for (int i = 0; i < N; i++) {
    printf("%.2f ", y[i]);
}
printf("\n");

// Free memory
free(x);
free(y);
cudaFree(d_x);
cudaFree(d_y);

return 0;
}

```

Output)

```

Resultant vector y:
0.00 4.00 8.00 12.00 16.00 20.00 24.00 28.00 32.00 36.00

D:\PPLab\CudaCodes\x64\Debug\CudaCodes.exe (process 23504) exited with code 0.

```



Q2)

Write a CUDA program to sort every row of a matrix using selection sort

Code)

```
#include <stdio.h>

#define N 4 // Number of rows in the matrix
#define M 5 // Number of columns in the matrix

// CUDA kernel to perform selection sort on each row of the matrix
__global__ void selectionSort(int* matrix) {
    int row = blockIdx.x;
    int start_idx = row * M;

    // Selection sort
    for (int i = 0; i < M - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < M; j++) {
            if (matrix[start_idx + j] < matrix[start_idx + min_idx]) {
                min_idx = j;
            }
        }
        // Swap elements
        int temp = matrix[start_idx + i];
        matrix[start_idx + i] = matrix[start_idx + min_idx];
        matrix[start_idx + min_idx] = temp;
    }
}

int main() {
    int matrix[N][M] = {
        {9, 4, 7, 2, 5},
        {3, 8, 1, 6, 0},
        {5, 2, 9, 1, 7},
        {8, 3, 6, 0, 4}
    };

    int* d_matrix; // Device matrix

    // Allocate memory on GPU
    cudaMalloc(&d_matrix, N * M * sizeof(int));

    // Copy data from host to device
    cudaMemcpy(d_matrix, matrix, N * M * sizeof(int),
        cudaMemcpyHostToDevice);
```

```

// Define grid dimensions
dim3 grid(N);

// Launch the CUDA kernel
selectionSort << <grid, 1 >> > (d_matrix);

// Copy result back to host
cudaMemcpy(matrix, d_matrix, N * M * sizeof(int),
cudaMemcpyDeviceToHost);

// Print sorted matrix
printf("Sorted Matrix:\n");
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        printf("%d ", matrix[i][j]);
    }
    printf("\n");
}

// Free memory
cudaFree(d_matrix);

return 0;
}

```

Output)

```

Sorted Matrix:
2 4 5 7 9
0 1 3 6 8
1 2 5 7 9
0 3 4 6 8

```

Q3)

Write a CUDA program to perform odd even transposition sort in parallel

Code)

```

#include <stdio.h>

#define MAX_N 100 // Maximum size of the array

// CUDA kernel to perform odd-even transposition sort

```

```

__global__ void oddEvenTranspositionSort(int* arr, int n) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int phase, i, temp;

    for (phase = 0; phase < n; phase++) {
        if (phase % 2 == 0) { // Even phase
            if (tid % 2 == 0 && tid + 1 < n) {
                if (arr[tid] > arr[tid + 1]) {
                    temp = arr[tid];
                    arr[tid] = arr[tid + 1];
                    arr[tid + 1] = temp;
                }
            }
        }
        else { // Odd phase
            if (tid % 2 != 0 && tid + 1 < n) {
                if (arr[tid] > arr[tid + 1]) {
                    temp = arr[tid];
                    arr[tid] = arr[tid + 1];
                    arr[tid + 1] = temp;
                }
            }
        }
        __syncthreads();
    }
}

int main() {
    int n;
    int arr[MAX_N];
    int* d_arr; // Device array

    // Input size of the array from the user
    printf("Enter the size of the array (maximum %d): ", MAX_N);
    scanf("%d", &n);

    if (n <= 0 || n > MAX_N) {
        printf("Invalid size of the array.\n");
        return 1;
    }

    // Input array elements from the user
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
}

```

```

// Allocate memory on GPU
cudaMalloc(&d_arr, n * sizeof(int));

// Copy data from host to device
cudaMemcpy(d_arr, arr, n * sizeof(int), cudaMemcpyHostToDevice);

// Define grid and block dimensions
int blockSize = 256;
int gridSize = (n + blockSize - 1) / blockSize;

// Launch the CUDA kernel
oddEvenTranspositionSort << <gridSize, blockSize >> > (d_arr, n);

// Copy result back to host
cudaMemcpy(arr, d_arr, n * sizeof(int), cudaMemcpyDeviceToHost);

// Print sorted array
printf("Sorted Array:\n");
for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);
}
printf("\n");

// Free memory
cudaFree(d_arr);

return 0;
}

```

Output)

```

Enter the size of the array (maximum 100): 5
Enter the elements of the array:
9 6 5 1 4
Sorted Array:
1 4 5 6 9

```

# WEEK-11

Q1)

Write a cuda program to implement insertion and selection sort algorithms

- Compare the performance of parallel sorting algorithms with their corresponding sequential implementation.
- Compare the performance of the two sorting algorithm by varying the input size.

Code)

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

#define ARRAY_SIZE 10000 // Adjust the array size as needed
#define BLOCK_SIZE 256

// Sequential Insertion Sort
void sequentialInsertionSort(int* arr, int size) {
    int i, key, j;
    for (i = 1; i < size; i++) {
        key = arr[i];
        j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// Sequential Selection Sort
void sequentialSelectionSort(int* arr, int size) {
    int i, j, min_idx;
    for (i = 0; i < size - 1; i++) {
        min_idx = i;
        for (j = i + 1; j < size; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }
        // Swap the found minimum element with the first element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
    }
}
```

```

        arr[i] = temp;
    }
}

// CUDA Kernel for Parallel Insertion Sort
__global__ void parallelInsertionSort(int* arr, int size) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// CUDA Kernel for Parallel Selection Sort
__global__ void parallelSelectionSort(int* arr, int size) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < size) {
        int j, min_idx;
        min_idx = i;
        for (j = i + 1; j < size; j++) {
            if (arr[j] < arr[min_idx])
                min_idx = j;
        }
        // Swap the found minimum element with the first element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

// Function to print array
void printArray(int* arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int* h_arr = (int*)malloc(ARRAY_SIZE * sizeof(int));
    int* d_arr;

```

```

    cudaMalloc(&d_arr, ARRAY_SIZE * sizeof(int));

    // Initialize array with random values
    for (int i = 0; i < ARRAY_SIZE; i++) {
        h_arr[i] = rand() % 1000;
    }

    // Copy array from host to device
    cudaMemcpy(d_arr, h_arr, ARRAY_SIZE * sizeof(int),
cudaMemcpyHostToDevice);

    // Sequential Insertion Sort
    clock_t start_time = clock();
    sequentialInsertionSort(h_arr, ARRAY_SIZE);
    clock_t end_time = clock();
    double sequential_insertion_time = ((double)(end_time - start_time))
/ CLOCKS_PER_SEC;
    printf("Sequential Insertion Sort Time: %f seconds\n",
sequential_insertion_time);

    // Sequential Selection Sort
    start_time = clock();
    sequentialSelectionSort(h_arr, ARRAY_SIZE);
    end_time = clock();
    double sequential_selection_time = ((double)(end_time - start_time))
/ CLOCKS_PER_SEC;
    printf("Sequential Selection Sort Time: %f seconds\n",
sequential_selection_time);

    // Copy sorted array back to device
    cudaMemcpy(d_arr, h_arr, ARRAY_SIZE * sizeof(int),
cudaMemcpyHostToDevice);

    // Parallel Insertion Sort
    start_time = clock();
    parallelInsertionSort << <(ARRAY_SIZE + BLOCK_SIZE - 1) /
BLOCK_SIZE, BLOCK_SIZE >> > (d_arr, ARRAY_SIZE);
    cudaDeviceSynchronize();
    end_time = clock();
    double parallel_insertion_time = ((double)(end_time - start_time)) /
CLOCKS_PER_SEC;
    printf("Parallel Insertion Sort Time: %f seconds\n",
parallel_insertion_time);

    // Copy sorted array back to host
    cudaMemcpy(h_arr, d_arr, ARRAY_SIZE * sizeof(int),

```

```

cudaMemcpyDeviceToHost);

    // Parallel Selection Sort
    start_time = clock();
    parallelSelectionSort << <(ARRAY_SIZE + BLOCK_SIZE - 1) /
BLOCK_SIZE, BLOCK_SIZE >> > (d_arr, ARRAY_SIZE);
    cudaDeviceSynchronize();
    end_time = clock();
    double parallel_selection_time = ((double)(end_time - start_time)) /
CLOCKS_PER_SEC;
    printf("Parallel Selection Sort Time: %f seconds\n",
parallel_selection_time);

    // Copy sorted array back to host
    cudaMemcpy(h_arr, d_arr, ARRAY_SIZE * sizeof(int),
cudaMemcpyDeviceToHost);

    // Free memory
    free(h_arr);
    cudaFree(d_arr);

    return 0;
}

```

Output)

```

Sequential Insertion Sort Time: 0.024000 seconds
Sequential Selection Sort Time: 0.036000 seconds
Parallel Insertion Sort Time: 0.000000 seconds
Parallel Selection Sort Time: 0.006000 seconds

```

Q2)

Write a parallel program using CUDA to implement the Odd-even transposition sort. Vary the input size and analyse the program efficiency.

**Hint:** Odd-even transposition sort is a sorting algorithm that's similar to bubble sort. The list  $a$  stores  $n$  integers, and the algorithm sorts them into increasing order. During an "even phase" ( $\text{phase} \% 2 == 0$ ), each odd-subscripted element,  $a[i]$ , is compared to the element to its "left"  $a[i-1]$ , and if they're out of order, they're swapped. During an "odd" phase, each odd subscripted element is compared to the element to its right, and if they're out of order, they're swapped. A theorem guarantees that after  $n$  phases, the list will be sorted.



Code)

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

#define ARRAY_SIZE 10000 // Adjust the array size as needed
#define BLOCK_SIZE 256

// CUDA Kernel for Odd-even Transposition Sort
__global__ void oddEvenSort(int* arr, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int phase, i, temp;
    for (phase = 0; phase < size; phase++) {
        if (phase % 2 == 0) { // Even phase
            if (tid % 2 == 0 && tid < size - 1) {
                if (arr[tid] > arr[tid + 1]) {
                    temp = arr[tid];
                    arr[tid] = arr[tid + 1];
                    arr[tid + 1] = temp;
                }
            }
        }
        else { // Odd phase
            if (tid % 2 != 0 && tid < size - 1) {
                if (arr[tid] > arr[tid + 1]) {
                    temp = arr[tid];
                    arr[tid] = arr[tid + 1];
                    arr[tid + 1] = temp;
                }
            }
        }
        __syncthreads();
    }
}

// Function to print array
void printArray(int* arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int* h_arr = (int*)malloc(ARRAY_SIZE * sizeof(int));
```

```

int* d_arr;
cudaMalloc(&d_arr, ARRAY_SIZE * sizeof(int));

// Initialize array with random values
for (int i = 0; i < ARRAY_SIZE; i++) {
    h_arr[i] = rand() % 1000;
}

// Copy array from host to device
cudaMemcpy(d_arr, h_arr, ARRAY_SIZE * sizeof(int),
cudaMemcpyHostToDevice);

// Parallel Odd-even Transposition Sort
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start);

    oddEvenSort << <(ARRAY_SIZE + BLOCK_SIZE - 1) / BLOCK_SIZE,
BLOCK_SIZE >> > (d_arr, ARRAY_SIZE);

    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    printf("Parallel Odd-even Transposition Sort Time: %f
milliseconds\n", milliseconds);

// Copy sorted array back to host
    cudaMemcpy(h_arr, d_arr, ARRAY_SIZE * sizeof(int),
cudaMemcpyDeviceToHost);

// Free memory
    free(h_arr);
    cudaFree(d_arr);

    return 0;
}

```

Output)

```
Parallel Odd-even Transposition Sort Time: 17.620993 milliseconds
```

Q3)

You're working on a physics simulation that involves calculating force exerted by large number of particles (N).

- Each particle has acceleration and mass.
- Assume that the acceleration and mass of all the particles are stored in two vectors ACC and MASS.
- The force exerted by each particle is calculated using the following formula:  $F=MA$  where F is force, M is mass and A is acceleration.
- Implement a CUDA program that efficiently calculates the forces exerted on all individual particles in parallel.
- Assume that, N=770 and a block can have 256 threads.

Code)

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

// Define constants
#define N 770
#define THREADS_PER_BLOCK 256

// CUDA kernel to calculate forces in parallel
__global__ void calculateForces(float* acc, float* mass, float* force) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < N) {
        force[tid] = acc[tid] * mass[tid];
    }
}

int main() {
    // Define host arrays
    float* h_acc, * h_mass, * h_force;
    // Define device arrays
    float* d_acc, * d_mass, * d_force;

    // Allocate memory on the host
    h_acc = (float*)malloc(N * sizeof(float));
    h_mass = (float*)malloc(N * sizeof(float));
    h_force = (float*)malloc(N * sizeof(float));

    // Initialize acceleration and mass vectors with random values
    srand(time(NULL));
    for (int i = 0; i < N; i++) {
```

```

        h_acc[i] = ((float)rand() / RAND_MAX) * 10.0f; // Random
acceleration between 0 and 10
        h_mass[i] = ((float)rand() / RAND_MAX) * 100.0f; // Random mass
between 0 and 100
    }

    // Allocate memory on the device
    cudaMalloc(&d_acc, N * sizeof(float));
    cudaMalloc(&d_mass, N * sizeof(float));
    cudaMalloc(&d_force, N * sizeof(float));

    // Copy data from host to device
    cudaMemcpy(d_acc, h_acc, N * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_mass, h_mass, N * sizeof(float),
cudaMemcpyHostToDevice);

    // Calculate grid dimensions
    int numBlocks = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

    // Launch CUDA kernel
    calculateForces << <numBlocks, THREADS_PER_BLOCK >> > (d_acc,
d_mass, d_force);

    // Copy result from device to host
    cudaMemcpy(h_force, d_force, N * sizeof(float),
cudaMemcpyDeviceToHost);

    // Output the calculated forces for each particle
    printf("Particle Forces:\n");
    for (int i = 0; i < N; i++) {
        printf("Particle %d: Force = %.2f\n", i, h_force[i]);
    }

    // Free device memory
    cudaFree(d_acc);
    cudaFree(d_mass);
    cudaFree(d_force);

    // Free host memory
    free(h_acc);
    free(h_mass);
    free(h_force);

    return 0;
}

```

Output)

```
Particle Forces:
Particle 0: Force = 5.06
Particle 1: Force = 41.71
Particle 2: Force = 9.57
Particle 3: Force = 202.21
Particle 4: Force = 54.12
Particle 5: Force = 595.58
Particle 6: Force = 230.56
Particle 7: Force = 688.17
Particle 8: Force = 200.22
Particle 9: Force = 215.67
```

.  
.  
.

```
Particle 763: Force = 896.70
Particle 764: Force = 73.95
Particle 765: Force = 79.77
Particle 766: Force = 101.53
Particle 767: Force = 103.86
Particle 768: Force = 0.33
Particle 769: Force = 408.83
```

Q4)

You're working on a real-time gray-scale image processing application. The processing involves a pre-processing step where you are supposed to determine the Cube of every pixel and then divide it by the total number of pixels in the given input image. Design a CUDA program to do the above task parallelly. Assume that the image dimension is 780x650. Also, consider that the streaming multi-processor can take maximum up to 2048 threads and it allows up to 1024 threads in each block. Assume that the Image is already read into variable I and size of the image is already estimated and stored in variable SIZE\_I.

Code)

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

// Kernel function to perform the pre-processing step
__global__ void preprocess(unsigned char* input, unsigned char* output,
int size) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < size) {
        unsigned char pixel = input[idx];
        unsigned int cube = pixel * pixel * pixel;
        output[idx] = (unsigned char)(cube / size);
    }
}
```

```

    }
}

int main() {
    const int width = 780;
    const int height = 650;
    const int size = width * height;

    // Allocate memory on the host
    unsigned char* h_input, * h_output;
    h_input = (unsigned char*)malloc(size * sizeof(unsigned char));
    h_output = (unsigned char*)malloc(size * sizeof(unsigned char));

    // Seed the random number generator
    srand(time(NULL));

    // Initialize input image data with random values
    for (int i = 0; i < size; i++) {
        h_input[i] = (unsigned char)(rand() % 256);
    }

    // Allocate memory on the device
    unsigned char* d_input, * d_output;
    cudaMalloc(&d_input, size * sizeof(unsigned char));
    cudaMalloc(&d_output, size * sizeof(unsigned char));

    // Copy input data from host to device
    cudaMemcpy(d_input, h_input, size * sizeof(unsigned char),
cudaMemcpyHostToDevice);

    // Launch the kernel
    const int blockSize = 1024;
    const int gridSize = (size + blockSize - 1) / blockSize;
    preprocess << <gridSize, blockSize >> > (d_input, d_output, size);

    // Copy the result from device to host
    cudaMemcpy(h_output, d_output, size * sizeof(unsigned char),
cudaMemcpyDeviceToHost);

    // Print the first 10 output values
    printf("First 10 output values:\n");
    for (int i = 0; i < 10; i++) {
        printf("%d ", h_output[i]);
    }
    printf("\n");
}

```

```
// Free memory
free(h_input);
free(h_output);
cudaFree(d_input);
cudaFree(d_output);

return 0;
}
```

Output)

```
First 10 output values:
4 0 27 17 12 9 0 23 5 0
```