COURSE OUTCOMES

Course Outcomes: At the end of this course, students are able to:
CO1 - Explain the need for parallel programming
CO2 - Demonstrate parallelism in MIMD system
CO3 - Apply MPI library to parallelize the code to solve the given problem.
CO4 - Apply OpenMP pragma and directives to parallelize the code to solve the given problem
CO5 - Design a CUDA program for the given problem

COs and POs Mapping of lab Component

COURSE OUTCOMES	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	3	2	-	-	-	-	-	-	-	-	-	2	2	-	-
CO2	3	3	2	2	2	-	-	-	-	-	-	1	3	2	-
CO3	3	3	3	-	3	-	-	-	1	-	2	2	3	3	2
CO4	3	3	3	-	3	-	-	-	1	-	2	1	3	3	2
CO5	3	3	3	-	3	-	-	-	1	-		2	3	2	3

CO-PO Mapping Justifications:

- **CO1 -** Understanding the fundamentals of parallel programming equips students to analyze the limitations of sequential processing. It also promotes lifelong learning as they explore the evolution and relevance of parallel computing models.

- **CO2 -** Students gain insight into multi-core architecture by analyzing and implementing MIMD-based parallelism. This helps them apply theoretical knowledge to real-world hardware configurations and improve performance understanding

- **CO3 -** Using MPI, students learn to structure and implement distributed applications that communicate through message passing. It strengthens their ability to design scalable solutions in distributed-memory environments**.**

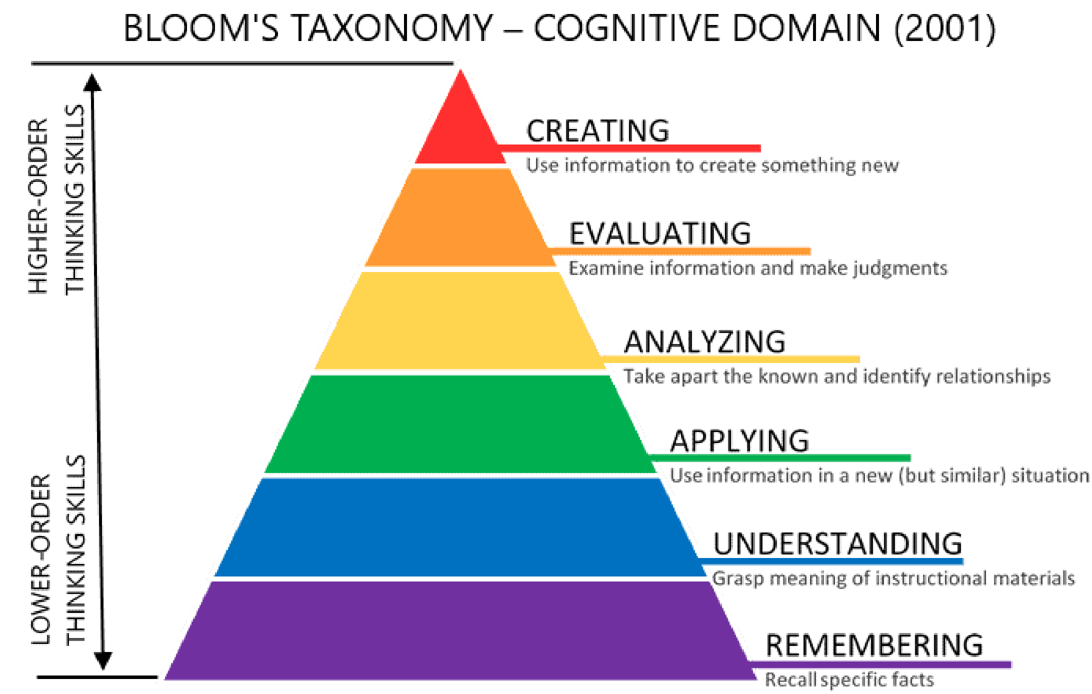
- **CO4 -** OpenMP allows students to parallelize code for shared-memory systems using compiler directives, improving execution speed. This fosters skills in identifying parallel sections and managing synchronization efficiently.

- **CO5 -** Students apply CUDA to solve data-parallel problems on GPUs, learning to optimize memory and thread usage. This enhances their understanding of heterogeneous computing and modern accelerator-based systems.

Mapping of 'Graduate Attributes' (GAs) and 'Program Outcomes' (POs)

Graduate Attributes (GAs) (As per Washington Accord Accreditation)	**Program Outcomes (POs) (As per NBA New Delhi)**
Engineering Knowledge	Apply the knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems
Problem Analysis	Identify, formulate, review research literature and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences.
Design/Development of solutions	Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate considerations for the public health and safety and the cultural, societal and environmental consideration.
Conduct Investigation of complex problems	Use research – based knowledge and research methods including design of experiments, analysis and interpretation of data and synthesis of the information to provide valid conclusions.
Modern Tool Usage	Create, select and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
The engineer and society	Apply reasoning informed by the contextual knowledge to assess society, health, safety, legal and cultural issues and the consequential responsibilities relevant to the professional engineering practice.
Environment and sustainability	Understand the impact of the professional engineering solutions in societal and environmental context and demonstrate the knowledge of and need for sustainable development.
Ethics	Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
Individual and team work	Function effectively as an individual and as a member or leader in diverse teams and in multidisciplinary settings.
Communication	Communicate effectively on complex engineering activities with the engineering community and with society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations and give and receive clear instructions.
Project management & finance	Demonstrate knowledge and understanding of the engineering and management principles and apply these to ones won work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
Life Long Learning	Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

REVISED BLOOMS TAXONOMY (RBT)



LAB EVALUATION PROCESS

WEEK WISE EVALUATION OF EACH PROGRAM PART A		
SL.NO	**ACTIVITY**	**MARKS**
1	Observation Book	10
2	Record and Viva	15+5
	TOTAL	**30**

INTERNAL ASSESSMENT PART B		
SL.NO	**ACTIVITY**	**MARKS**
1	Procedure	5
2	Conduction	10
3	Viva -Voce	5
TOTAL		**20**
PART A + PART B		**50**

PROGRAM LIST

Program 1: Write a OpenMP program to sort an array on n elements using both sequential and parallel mergesort (using Section). Record the difference in execution time.

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

// Function to merge two halves of the array
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 = (int*)malloc(n1 * sizeof(int));
    int* R = (int*)malloc(n2 * sizeof(int));

    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) {
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    }
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];

    free(L);
    free(R);
}

// Sequential MergeSort
void mergeSortSequential(int* arr, int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSortSequential(arr, l, m);
        mergeSortSequential(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

// Parallel MergeSort using OpenMP sections
void mergeSortParallel(int* arr, int l, int r, int depth) {
    if (l < r) {
        int m = (l + r) / 2;

        if (depth <= 0) {
            // Fallback to sequential at depth limit
            mergeSortSequential(arr, l, m);
            mergeSortSequential(arr, m + 1, r);
        } else {
```

```
            #pragma omp parallel sections
            {
                #pragma omp section
                mergeSortParallel(arr, l, m, depth - 1);

                #pragma omp section
                mergeSortParallel(arr, m + 1, r, depth - 1);
            }
        }

        merge(arr, l, m, r);
    }
}

// Helper to check if array is sorted
int isSorted(int* arr, int n) {
    for (int i = 1; i < n; i++) {
        if (arr[i - 1] > arr[i]) return 0;
    }
    return 1;
}

int main() {
    int n = 1000000;
    int* arrSeq = (int*)malloc(n * sizeof(int));
    int* arrPar = (int*)malloc(n * sizeof(int));

    // Seed for reproducibility
    srand(42);
    for (int i = 0; i < n; i++) {
        arrSeq[i] = rand() % 100000;
        arrPar[i] = arrSeq[i];
    }

    // Time sequential sort
    double start = omp_get_wtime();
    mergeSortSequential(arrSeq, 0, n - 1);
    double end = omp_get_wtime();
    double timeSeq = end - start;

    // Time parallel sort
    start = omp_get_wtime();
    mergeSortParallel(arrPar, 0, n - 1, 4);  // You can tune depth
    end = omp_get_wtime();
    double timePar = end - start;

    // Validate and output
    printf("Sequential sort time: %.6f seconds\n", timeSeq);
    printf("Parallel sort time  : %.6f seconds\n", timePar);
    printf("Speedup             : %.2fx\n", timeSeq / timePar);
    if (!isSorted(arrSeq, n)) printf("Sequential sort failed!\n");
```

```
    if (!isSorted(arrPar, n)) printf("Parallel sort failed!\n");

    free(arrSeq);
    free(arrPar);

    return 0;
}
```

Output

```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gcc -fopenmp mergesort_openmp
.c -o mergesort
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ ./mergesort
Sequential sort time: 0.188905 seconds
Parallel sort time   : 0.174429 seconds
Speedup              : 1.08x
```

Program 2: **Write an OpenMP program that divides the Iterations into chunks containing 2 iterations, respectively (OMP_SCHEDULE=static,2). Its input should be the number of iterations, and its output should be which iterations of a parallelized for loop are executed by which thread. For example, if there are two threads and four iterations, the output might be the following:**

a. Thread 0: Iterations 0 –– 1

b. Thread 1: Iterations 2 –– 3

Program 2

```
#include <stdio.h>
#include <omp.h>
int main() {
    int num_iterations;

    printf("Enter the number of iterations: ");
    scanf("%d", &num_iterations);

    // Optional: Set number of threads (or use OMP_NUM_THREADS)
    // omp_set_num_threads(2);

    printf("\nUsing schedule(static,2):\n\n");

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();

        #pragma omp for schedule(static, 2)
        for (int i = 0; i < num_iterations; i++) {
            printf("Thread %d : Iteration %d\n", tid, i);
        }
    }

    return 0;
}
```

Output

```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit omp_static_chunks.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ export OMP_NUM_THREADS=2
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gcc -fopenmp omp_static_chunk
s.c -o omp_static_chunks
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ ./omp_static_chunks
Enter the number of iterations: 5

Using schedule(static,2):

Thread 1 : Iteration 2
Thread 0 : Iteration 0
Thread 0 : Iteration 1
Thread 0 : Iteration 4
Thread 1 : Iteration 3
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$
```

Program 3: Write a OpenMP program to calculate n Fibonacci numbers using tasks.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
// Recursive Fibonacci using OpenMP tasks
int fib(int n) {
    int x, y;
    if (n <= 1) return n;
    #pragma omp task shared(x)
    x = fib(n - 1);
    #pragma omp task shared(y)
    y = fib(n - 2);
    #pragma omp taskwait
    return x + y;
}
int main() {
    int n;
    printf("Enter the number of Fibonacci numbers to calculate: ");
    scanf("%d", &n);
    if (n <= 0) {
        printf("Please enter a positive integer.\n");
        return 0;
    }
    printf("First %d Fibonacci numbers using OpenMP tasks:\n", n);
    double start = omp_get_wtime();
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < n; i++) {
                int result;
                #pragma omp task shared(result)
                {
                    result = fib(i);
                    #pragma omp critical
                    printf("Fib(%d) = %d\n", i, result);
```

```
                }
            }
        }
    }
    double end = omp_get_wtime();
    printf("Execution time: %.6f seconds\n", end - start);
    return 0;
}
```

Output

```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gcc -fopenmp fibonacci_tasks.
c -o fibonacci_tasks
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ ./fibonacci_tasks
Enter the number of Fibonacci numbers to calculate: 12
First 12 Fibonacci numbers using OpenMP tasks:
Fib(0) = 0
Fib(1) = 1
Fib(3) = 2
Fib(4) = 3
Fib(5) = 5
Fib(6) = 8
Fib(7) = 13
Fib(8) = 21
Fib(9) = 34
Fib(10) = 55
Fib(11) = 89
Fib(2) = 1
Execution time: 0.000562 seconds
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$
```

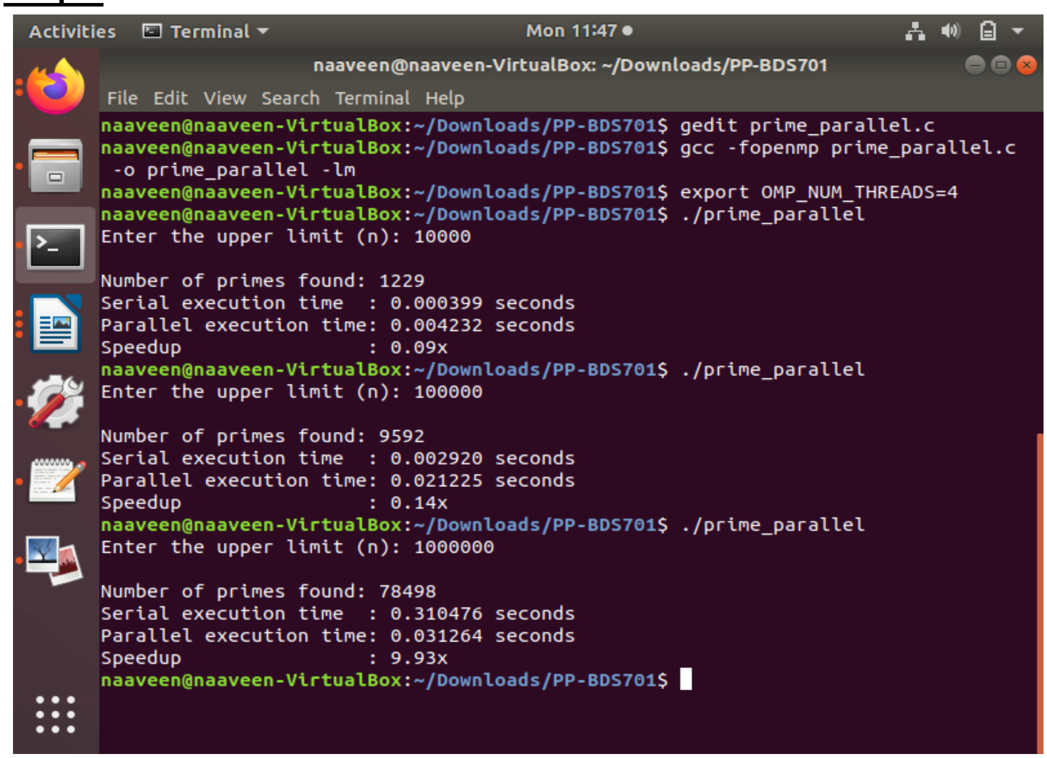
<u>Program 4:</u> Write a OpenMP program to find the prime numbers from 1 to n employing parallel for directive. Record both serial and parallel execution times.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
// Check if a number is prime
int is_prime(int num) {
   if (num <= 1) return 0;
   if (num == 2) return 1;
   if (num % 2 == 0) return 0;
  int limit = (int)sqrt(num);
   for (int i = 3; i <= limit; i += 2) {
      if (num % i == 0) return 0;
   }
   return 1;
}
int main() {
   int n;
   printf("Enter the upper limit (n): ");
   scanf("%d", &n);
   if (n < 2) {
      printf("There are no prime numbers <= %d\n", n);
      return 0;
   }
   // SERIAL VERSION
   double start_serial = omp_get_wtime();
   int* primes_serial = (int*)malloc((n + 1) * sizeof(int));
   int count_serial = 0;
   for (int i = 2; i <= n; i++) {
      if (is_prime(i)) {
```

```
        primes_serial[count_serial++] = i;
    }
}
double end_serial = omp_get_wtime();
double time_serial = end_serial - start_serial;
// PARALLEL VERSION
double start_parallel = omp_get_wtime();
int* primes_parallel = (int*)malloc((n + 1) * sizeof(int));
int count_parallel = 0;
#pragma omp parallel
{
    int* local_primes = (int*)malloc((n / omp_get_num_threads() + 1) * sizeof(int));
    int local_count = 0;
    #pragma omp for nowait
    for (int i = 2; i <= n; i++) {
        if (is_prime(i)) {
            local_primes[local_count++] = i;
        }
    }
    // Combine local results into global array (critical section)
    #pragma omp critical
    {
        for (int i = 0; i < local_count; i++) {
            primes_parallel[count_parallel++] = local_primes[i];
        }
    }
    free(local_primes);
}
double end_parallel = omp_get_wtime();
double time_parallel = end_parallel - start_parallel;
// Output
```

```
    printf("\nNumber of primes found: %d\n", count_serial);
    printf("Serial execution time  : %.6f seconds\n", time_serial);
    printf("Parallel execution time: %.6f seconds\n", time_parallel);
    printf("Speedup                : %.2fx\n", time_serial / time_parallel);
    // Optional: Print the primes
    /*
    printf("\nPrimes:\n");
    for (int i = 0; i < count_parallel; i++) {
        printf("%d ", primes_parallel[i]);
    }
    printf("\n");
    */
    free(primes_serial);
    free(primes_parallel);
    return 0;
}
```

Output



<u>Program 5:</u> Write a MPI Program to demonstration of MPI_Send and MPI_Recv.

```c
#include <mpi.h>

#include <stdio.h>

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

    MPI_Init(&argc, &argv);              // Initialize MPI environment

    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get current process ID

    MPI_Comm_size(MPI_COMM_WORLD, &size); // Get number of processes

    if (size < 2) {

        if (rank == 0) {
            printf("This program requires at least 2 processes.\n");
        }

        MPI_Finalize();
        return 0;
    }
    if (rank == 0) {
        number = 100; // Example message
        printf("Process 0 sending number %d to Process 1\n", number);
        MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

    } else if (rank == 1) {

 MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        printf("Process 1 received number %d from Process 0\n", number);

    }

    MPI_Finalize(); // Clean up the MPI environment
    return 0;
}
```

<u>Output:</u>

```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit mpi_send_recv.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpicc mpi_send_recv.c -o mpi_
send_recv
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpirun -np 2 ./mpi_send_recv
Process 0 sending number 100 to Process 1
Process 1 received number 100 from Process 0
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$
```

Program 5

Program 6: **Write a MPI program to demonstration of deadlock using point to point communication and avoidance of deadlock by altering the call sequence.**

Program 6

```
#include <mpi.h>

#include <stdio.h>
int main(int argc, char** argv) {
    int rank, size;
    int msg_send = 100, msg_recv;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size < 2) {
        if (rank == 0)
            printf("Run with at least 2 processes.\n");
        MPI_Finalize();
        return 0;
    }

    if (rank == 0) {
        printf("Process 0 sending to Process 1...\n");
        MPI_Send(&msg_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD); // blocking send
        MPI_Recv(&msg_recv,1,MPI_INT,1,0,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 0 received from Process 1: %d\n", msg_recv);
    } else if (rank == 1) {
        printf("Process 1 sending to Process 0...\n");
        MPI_Send(&msg_send, 1, MPI_INT, 0, 0, MPI_COMM_WORLD); // blocking send
        MPI_Recv(&msg_recv,1,MPI_INT,0,0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received from Process 0: %d\n", msg_recv);
    }
    MPI_Finalize();
    return 0;
}
```

Output:

```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit mpi_deadlock.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpicc mpi_deadlock.c -o mpi_d
eadlock
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpirun -np 2 ./mpi_deadlock
Process 0 sending to Process 1...
Process 1 sending to Process 0...
Process 0 received from Process 1: 100
Process 1 received from Process 0: 100
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$
```

Program 7: **Write a MPI Program to demonstration of Broadcast operation.**

Program 7

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    int rank, size;
    int number;
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);

    // Get the rank and number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Process 0 gets the input
    if (rank == 0) {
        printf("Enter a number to broadcast: ");
        fflush(stdout); // Ensure prompt is printed before input
        scanf("%d", &number);
    }

    // Broadcast the number from process 0 to all other processes
    MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Each process prints the received number
    printf("Process %d received number: %d\n", rank, number);

    // Finalize the MPI environment
    MPI_Finalize();
    return 0;
}
```

Output:

```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit mpi_broadcast.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpicc mpi_broadcast.c -o mpi_
broadcast
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpirun -np 4 ./mpi_broadcast
Enter a number to broadcast: 42
Process 2 received number: 42
Process 0 received number: 42
Process 1 received number: 42
Process 3 received number: 42
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$
```

<u>Program 8:</u> Write a MPI Program demonstration of MPI_Scatter and MPI_Gather.

```
#include <stdio.h>

#include <mpi.h>

int main(int argc, char** argv)

{

int rank, size, send_data[4] = {10, 20, 30, 40},

recv_data; MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

MPI_Comm_size(MPI_COMM_WORLD, &size);

MPI_Scatter(send_data,  1,  MPI_INT,  &recv_data,  1,  MPI_INT,  0,  MPI_COMM_WORLD);
printf("Process %d received: %d\n", rank, recv_data);

recv_data += 1;

MPI_Gather(&recv_data, 1, MPI_INT, send_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (rank == 0)

{

printf("Gathered data: ");

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

printf("%d ", send_data[i]);

printf("\n");

}       MPI_Finalize();

        return 0;

}
```

Program 8

<u>Output</u>

```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit mpi_scatter_gather.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpicc mpi_scatter_gather.c -o
 mpi_scatter_gather
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpirun -np 4 ./mpi_scatter_ga
ther
Process 0 received: 10
Process 2 received: 30
Process 3 received: 40
Process 1 received: 20
Gathered data: 11 21 31 41
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$
```

<u>Program 9:</u> Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD)

Program 9

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {
    int rank, size;
    int value;
    int sum, prod, max, min;
    int sum_all, prod_all, max_all, min_all;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // Each process sets its own value (e.g., rank + 1)
    value = rank + 1;
    printf("Process %d has value %d\n", rank, value);
    // ---------- MPI_Reduce ----------
    MPI_Reduce(&value, &sum,  1, MPI_INT, MPI_SUM,  0, MPI_COMM_WORLD);
    MPI_Reduce(&value, &prod, 1, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);
    MPI_Reduce(&value, &max,  1, MPI_INT, MPI_MAX,  0, MPI_COMM_WORLD);
    MPI_Reduce(&value, &min,  1, MPI_INT, MPI_MIN,  0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("\n[Using MPI_Reduce at Root Process]\n");
        printf("Sum   = %d\n", sum);
        printf("Prod  = %d\n", prod);
        printf("Max   = %d\n", max);
        printf("Min   = %d\n", min);
    }
    // ---------- MPI_Allreduce ----------
    MPI_Allreduce(&value, &sum_all,  1, MPI_INT, MPI_SUM,  MPI_COMM_WORLD);
    MPI_Allreduce(&value, &prod_all, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);
```

```
    MPI_Allreduce(&value, &max_all,  1, MPI_INT, MPI_MAX,  MPI_COMM_WORLD);

    MPI_Allreduce(&value, &min_all,  1, MPI_INT, MPI_MIN,  MPI_COMM_WORLD);

    printf("\n[Process %d] MPI_Allreduce Results:\n", rank);

    printf("  Sum  = %d\n", sum_all);

    printf("  Prod = %d\n", prod_all);

    printf("  Max  = %d\n", max_all);

    printf("  Min  = %d\n", min_all);

    MPI_Finalize();

    return 0;

}
```

Output

```
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ gedit mpi_reduce_allreduce.c
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpicc mpi_reduce_allreduce.c -o mpi_reduce_allreduce
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$ mpirun -np 4 ./mpi_reduce_allreduce
Process 0 has value 1
Process 2 has value 3
Process 3 has value 4
Process 1 has value 2

[Using MPI_Reduce at Root Process]
Sum   = 10
Prod  = 24
Max   = 4
Min   = 1

[Process 3] MPI_Allreduce Results:
  Sum  = 10
  Prod = 24
  Max  = 4
  Min  = 1

[Process 1] MPI_Allreduce Results:
  Sum  = 10
  Prod = 24
  Max  = 4
  Min  = 1

[Process 2] MPI_Allreduce Results:
  Sum  = 10
  Prod = 24
  Max  = 4
  Min  = 1

[Process 0] MPI_Allreduce Results:
  Sum  = 10
  Prod = 24
  Max  = 4
  Min  = 1
naaveen@naaveen-VirtualBox:~/Downloads/PP-BDS701$
```