VISVESVARAYA TECHNOLOGICAL UNIVERSITY

Department of Computer Science and Engineering

LAB MANUAL

Subject: Parallel Computing

Course Code: BCS702

Semester: 7th

Academic Year: 2025-2026

Prepared By: Prof. SUSHANT PATIL

College Name: MARATHA MANDAL’S ENGINEERING COLLEGE BELGAUM

VTU BCS702 – Parallel Computing

Lab Manual with Code
7th Semester – Computer Science & Engineering
Course Code: BCS702

Lab Objectives

- Understand the fundamentals of parallel computing using OpenMP and MPI.
- Implement parallel algorithms and analyze performance.
- Gain hands-on experience with shared and distributed memory models.
- Apply parallelism in real-world computational problems.

Experiment 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>


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

  int i = l, j = m + 1, k = 0;

  int temp[r - l + 1];


  while (i <= m && j <= r) {

    if (arr[i] <= arr[j])

      temp[k++] = arr[i++];

    else

      temp[k++] = arr[j++];

  }

  while (i <= m) temp[k++] = arr[i++];
```

```
    while (j <= r) temp[k++] = arr[j++];

    for (i = l, k = 0; i <= r; i++, k++)
        arr[i] = temp[k];
}

void sequentialMergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        sequentialMergeSort(arr, l, m);
        sequentialMergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

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

        #pragma omp parallel sections
        {
            #pragma omp section
            parallelMergeSort(arr, l, m);

            #pragma omp section
            parallelMergeSort(arr, m + 1, r);
        }

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

int main() {
    int n = 100000;
    int arr1[n], arr2[n];

    for (int i = 0; i < n; i++) {
        arr1[i] = rand() % 1000;
        arr2[i] = arr1[i];
    }

    double start, end;

    start = omp_get_wtime();
    sequentialMergeSort(arr1, 0, n - 1);
    end = omp_get_wtime();
    printf("Sequential Merge Sort Time: %f seconds\n", end - start);

    start = omp_get_wtime();
    parallelMergeSort(arr2, 0, n - 1);
    end = omp_get_wtime();
    printf("Parallel Merge Sort Time: %f seconds\n", end - start);

    return 0;
}
```

Output :

```
Sequential Merge Sort Time: 0.013000 seconds
Parallel Merge Sort Time: 0.169000 seconds
```

Experiment 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

```
#include <stdio.h>

#include <omp.h>


int main() {

   int n;

   printf("Enter number of iterations: ");

   scanf("%d", &n);


   int thread_start[100], thread_end[100];

   int i;


   for (i = 0; i < 100; i++) {

      thread_start[i] = -1;

      thread_end[i] = -1;

   }


   #pragma omp parallel for schedule(static,2)

   for (i = 0; i < n; i++) {

      int tid = omp_get_thread_num();


      if (thread_start[tid] == -1)

         thread_start[tid] = i;
```

```
        thread_end[tid] = i;
    }

    for (i = 0; i < 100; i++) {
        if (thread_start[i] != -1) {
            printf("Thread %d : Iterations %d -- %d\n", i, thread_start[i], thread_end[i]);

        }
    }

    return 0;
}
```

OUTPUT:

```
Enter number of iterations: 8
Thread 0 : Iterations 0 -- 1
Thread 1 : Iterations 2 -- 3
Thread 2 : Iterations 4 -- 5
Thread 3 : Iterations 6 -- 7
```

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

```
#include <stdio.h>

#include <omp.h>


int fib(int n) {

   int x, y;


   if (n < 2)

      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 number of Fibonacci terms: ");

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

```
    printf("Fibonacci Series:\n");

    for (int i = 0; i < n; i++) {
        int result;

        #pragma omp parallel
        {
            #pragma omp single
            {
                result = fib(i);
            }
        }

        printf("%d ", result);
    }

    printf("\n");
    return 0;
}
```

OUTPUT:

```
Enter number of Fibonacci terms: 9
Fibonacci Series:
0 1 1 2 3 5 8 13 21
```

Experiment 4: 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 <math.h>

#include <omp.h>


int is_prime(int num) {

   if (num < 2) return 0;

   for (int i = 2; i <= sqrt(num); i++) {

      if (num % i == 0)

         return 0;

   }

   return 1;

}


int main() {

   int n;

   printf("Enter the value of n: ");

   scanf("%d", &n);


   printf("\nPrime numbers from 1 to %d:\n", n);

   for (int i = 1; i <= n; i++) {

      if (is_prime(i))

         printf("%d ", i);

   }

   printf("\n");
```

```
    double start_time, end_time;


    start_time = omp_get_wtime();
    for (int i = 1; i <= n; i++) {
        is_prime(i);
    }
    end_time = omp_get_wtime();
    printf("Serial Time: %f seconds\n", end_time - start_time);


    start_time = omp_get_wtime();
    #pragma omp parallel for
    for (int i = 1; i <= n; i++) {
        is_prime(i);
    }
    end_time = omp_get_wtime();
    printf("Parallel Time: %f seconds\n", end_time - start_time);


    return 0;
}
```

OUTPUT:

```
Enter the value of n: 500

Prime numbers from 1 to 500:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233
239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359
367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487
491 499

Serial Time: 0.000000 seconds

Parallel Time: 0.002000 seconds
```

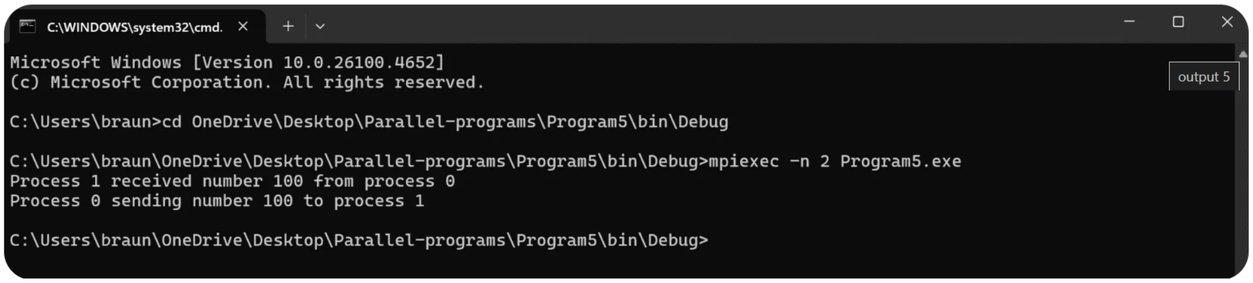
Experiment 5: Write a MPI Program to demonstration of MPI_Send and MPI_Recv.

PROGRAM:

```
#include <mpi.h>

#include <stdio.h>


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

  int rank, size;

  int number;


  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("Please run with at least 2 processes.\n");

    MPI_Finalize();

    return 0;

  }


  if (rank == 0) {

    number = 100;

    printf("Process %d sending number %d to process 1\n", rank, 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 %d received number %d from process 0\n", rank, number);
    }


    MPI_Finalize();
    return 0;
}
```

OUTPUT:



```
Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.

C:\Users\braun>cd OneDrive\Desktop\Parallel-programs\Program5\bin\Debug

C:\Users\braun\OneDrive\Desktop\Parallel-programs\Program5\bin\Debug>mpiexec -n 2 Program5.exe
Process 1 received number 100 from process 0
Process 0 sending number 100 to process 1

C:\Users\braun\OneDrive\Desktop\Parallel-programs\Program5\bin\Debug>
```

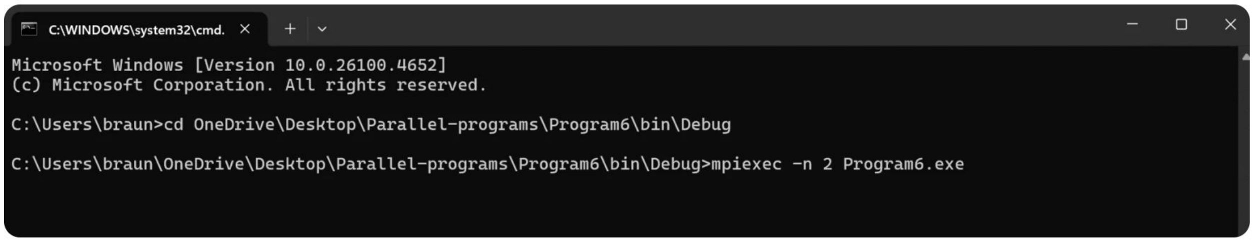
Experiment 6: Write a MPI program to demonstration of deadlock using point to point communication and avoidance of deadlock by altering the call sequence.

PROGRAM:

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

// Change this to 1 for deadlock, 2 for deadlock avoidance
#define DEADLOCK_PART 2

int main(int argc, char *argv[]) {
  int rank, size, num = 123;

  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 DEADLOCK_PART == 1
  // ----------- Part A: Deadlock -----------
  if (rank == 0) {
    printf("Process 0 waiting to receive from Process 1...\n");
```

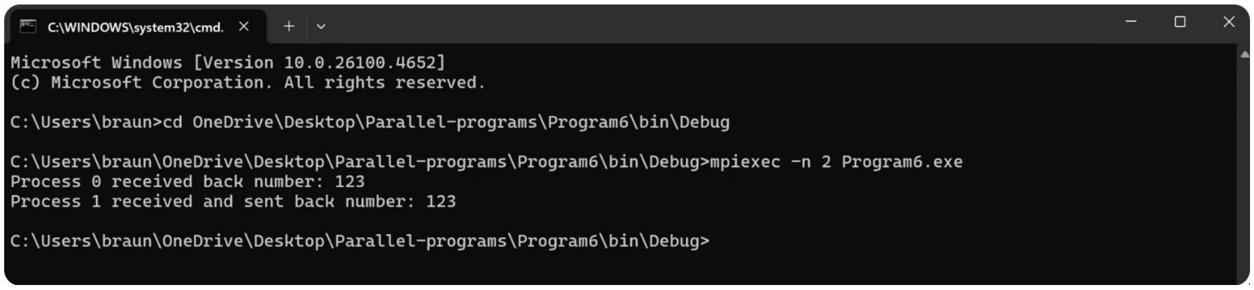
```
        MPI_Recv(&num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(&num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        printf("Process 1 waiting to receive from Process 0...\n");
        MPI_Recv(&num, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(&num, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

#elif DEADLOCK_PART == 2
    // ----------- Part B: Deadlock Avoidance -----------
    if (rank == 0) {
        MPI_Send(&num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 0 received back number: %d\n", num);
    } else if (rank == 1) {
        MPI_Recv(&num, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Send(&num, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        printf("Process 1 received and sent back number: %d\n", num);
    }
#endif

    MPI_Finalize();
    return 0;
}
```

OUTPUT 1:



```
Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.

C:\Users\braun>cd OneDrive\Desktop\Parallel-programs\Program6\bin\Debug

C:\Users\braun\OneDrive\Desktop\Parallel-programs\Program6\bin\Debug>mpiexec -n 2 Program6.exe
```

OUTPUT 2:

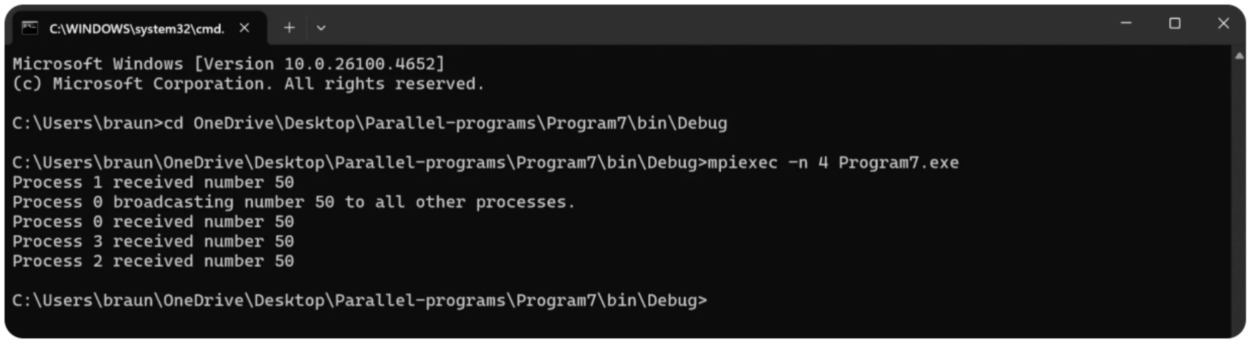


```
Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.

C:\Users\braun>cd OneDrive\Desktop\Parallel-programs\Program6\bin\Debug

C:\Users\braun\OneDrive\Desktop\Parallel-programs\Program6\bin\Debug>mpiexec -n 2 Program6.exe
Process 0 received back number: 123
Process 1 received and sent back number: 123

C:\Users\braun\OneDrive\Desktop\Parallel-programs\Program6\bin\Debug>
```

Expected Behavior:

- Process 0 sends a number to Process 123
- Process 1 receives and sends it back
- Both complete successfully

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

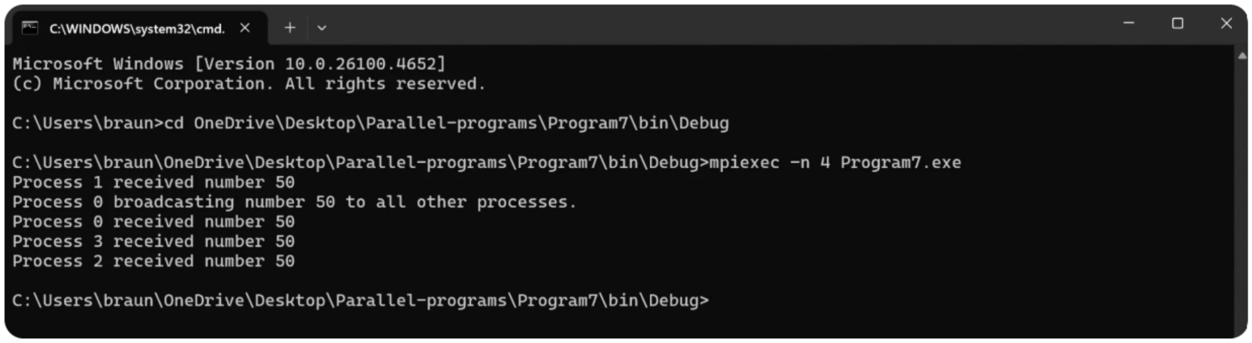


```
Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.

C:\Users\braun>cd OneDrive\Desktop\Parallel-programs\Program7\bin\Debug

C:\Users\braun\OneDrive\Desktop\Parallel-programs\Program7\bin\Debug>mpiexec -n 4 Program7.exe
Process 1 received number 50
Process 0 broadcasting number 50 to all other processes.
Process 0 received number 50
Process 3 received number 50
Process 2 received number 50

C:\Users\braun\OneDrive\Desktop\Parallel-programs\Program7\bin\Debug>
```

PROGRAM:

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


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


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


  if (rank == 0) {
    number = 50;
    printf("Process %d broadcasting number %d to all other processes.\n", rank, number);
  }


  MPI_Bcast(&number, 1, MPI_INT, 0, MPI_COMM_WORLD);
  printf("Process %d received number %d\n", rank, number);
  MPI_Finalize();

  return 0;

}
```

OUTPUT:



```
Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.

C:\Users\braun>cd OneDrive\Desktop\Parallel-programs\Program7\bin\Debug

C:\Users\braun\OneDrive\Desktop\Parallel-programs\Program7\bin\Debug>mpiexec -n 4 Program7.exe
Process 1 received number 50
Process 0 broadcasting number 50 to all other processes.
Process 0 received number 50
Process 3 received number 50
Process 2 received number 50

C:\Users\braun\OneDrive\Desktop\Parallel-programs\Program7\bin\Debug>
```

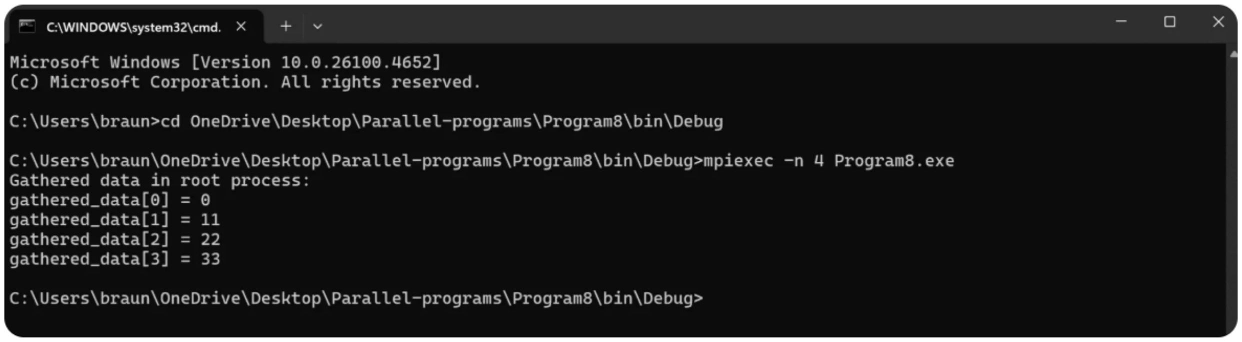
Experiment 8: Write a MPI Program demonstration of MPI_Scatter and MPI_Gather.

PROGRAM:

```
#include <mpi.h>

#include <stdio.h>


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

   int rank, size;

   int send_data[100], recv_data, gathered_data[100];


   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Comm_size(MPI_COMM_WORLD, &size);


   if (rank == 0) {

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

         send_data[i] = i * 10;

      }

   }


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

   recv_data = recv_data + rank;

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


   if (rank == 0) {

      printf("Gathered data in root process:\n");

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

         printf("gathered_data[%d] = %d\n", i, gathered_data[i]);
```

```
    }
  }

  MPI_Finalize();
  return 0;
}
```

OUTPUT:



```
Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.

C:\Users\braun>cd OneDrive\Desktop\Parallel-programs\Program8\bin\Debug

C:\Users\braun\OneDrive\Desktop\Parallel-programs\Program8\bin\Debug>mpiexec -n 4 Program8.exe
Gathered data in root process:
gathered_data[0] = 0
gathered_data[1] = 11
gathered_data[2] = 22
gathered_data[3] = 33

C:\Users\braun\OneDrive\Desktop\Parallel-programs\Program8\bin\Debug>
```

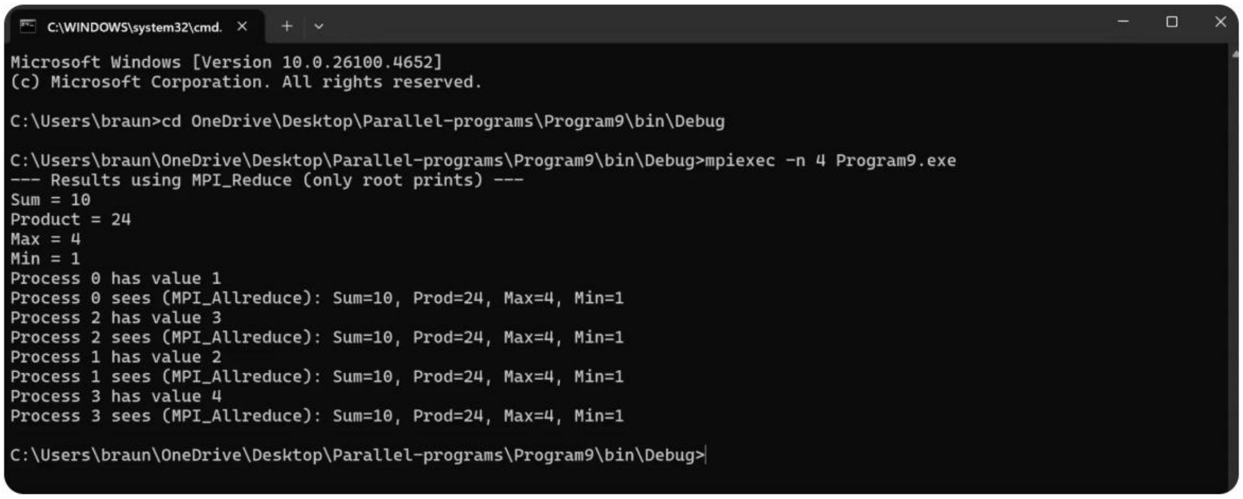
Experiment 9: Write a MPI Program to demonstration of MPI_Reduce and MPI_Allreduce (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD).

PROGRAM:

```
#include <mpi.h>

#include <stdio.h>


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

  int rank, size;

  int value, sum, prod, max, min;

  int all_sum, all_prod, all_max, all_min;


  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  MPI_Comm_size(MPI_COMM_WORLD, &size);


  value = rank + 1;


  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);


  MPI_Allreduce(&value, &all_sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

  MPI_Allreduce(&value, &all_prod, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);

  MPI_Allreduce(&value, &all_max, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);

  MPI_Allreduce(&value, &all_min, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
```

```
    if (rank == 0) {
        printf("--- Results using MPI_Reduce (only root prints) ---\n");
        printf("Sum = %d\n", sum);
        printf("Product = %d\n", prod);
        printf("Max = %d\n", max);
        printf("Min = %d\n", min);
    }

    printf("Process %d has value %d\n", rank, value);
    printf("Process %d sees (MPI_Allreduce): Sum=%d, Prod=%d, Max=%d, Min=%d\n",
           rank, all_sum, all_prod, all_max, all_min);

    MPI_Finalize();
    return 0;
}
```

OUTPUT:



```
Microsoft Windows [Version 10.0.26100.4652]
(c) Microsoft Corporation. All rights reserved.

C:\Users\braun>cd OneDrive\Desktop\Parallel-programs\Program9\bin\Debug

C:\Users\braun\OneDrive\Desktop\Parallel-programs\Program9\bin\Debug>mpiexec -n 4 Program9.exe
--- Results using MPI_Reduce (only root prints) ---
Sum = 10
Product = 24
Max = 4
Min = 1
Process 0 has value 1
Process 0 sees (MPI_Allreduce): Sum=10, Prod=24, Max=4, Min=1
Process 2 has value 3
Process 2 sees (MPI_Allreduce): Sum=10, Prod=24, Max=4, Min=1
Process 1 has value 2
Process 1 sees (MPI_Allreduce): Sum=10, Prod=24, Max=4, Min=1
Process 3 has value 4
Process 3 sees (MPI_Allreduce): Sum=10, Prod=24, Max=4, Min=1

C:\Users\braun\OneDrive\Desktop\Parallel-programs\Program9\bin\Debug>
```