SRI SAIRAM COLLEGE OF ENGINEERING, ANEKAL,

BENGALURU DEPARTMENT OF COMPUTER SCIENCE AND

ENGINEERING

Subject Code: BCS702

Subject: Parallel Computing Laboratory

Prepared by

K.Karthika
Assistant professor/CSE

V.Yamuna
Assistant Professor /CSE

EXPERIMENT NO 1

WRITE A OPENMP PROGRAM TO SORT AN ARRAY ON N ELEMENTS USING BOTH SEQUENTIAL AND PARALLELMERGESORT (USING SECTION). RECORD THE DIFFERENCE IN EXECUTION TIME.

What is OpenMp?
OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments. OpenMP identifies parallel regions as blocks of code that may run in parallel. Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run-time library to execute the region in parallel.

What is merge sort?
Merge sort is a divide and conquer algorithm that sorts the input array by breaking it into subarrays until they have 1 element, and then merging the results back into a sorted array. The time and space complexities are O(N log N) and O(n) respectively.

Sequential Merge Sort
Sequential merge sort in C is a sorting algorithm that follows the divide-and-conquer paradigm. It recursively divides an array into two halves until the sub-arrays contain only one element (which is considered sorted). Then, it merges these sub-arrays in a sorted manner to produce new sorted sub-arrays, repeating this process until the entire array is sorted.

Parallel Merge Sort
Parallel merge sort is an algorithm that sorts a list by recursively dividing it into sublists, sorting them independently using multiple threads or processes, and then merging the sorted sublists. This approach leverages parallel processing to improve sorting speed, especially for large datasets.

Section:
Sections are independent blocks of code, able to be assigned to separate threads if they are available.

CODE:
```
#include  <stdio.h>
#include <stdlib.h>
#include <omp.h>
void merge(int arr[], int left, int mid, int right) {
int n1 = mid - left + 1;
int n2 = right - mid;
int L[n1], R[n2];
for (int i = 0; i < n1; i++)
L[i] = arr[left + i];
for (int j = 0; j < n2; j++)
R[j] = arr[mid + 1 + j];
int i = 0, j = 0, k = left;
while (i < n1 && j < n2) {
if (L[i] <= R[j])
arr[k++] = L[i++];
else
arr[k++] = R[j++];
}
while (i < n1)
arr[k++] = L[i++];
while (j < n2)
arr[k++] = R[j++];
}
// Sequential Merge Sort
void mergeSortSequential(int arr[], int left, int right) {
if (left < right) {
int mid = left + (right - left) / 2;
mergeSortSequential(arr, left, mid);
mergeSortSequential(arr, mid + 1, right);
merge(arr, left, mid, right);
}
}
// Parallel Merge Sort
void mergeSortParallel(int arr[], int left, int right) {
if (left < right) {
int mid = left + (right - left) / 2;
#pragma omp parallel sections
{
#pragma omp section
mergeSortParallel(arr, left, mid);
#pragma omp section
mergeSortParallel(arr, mid + 1, right);
}
merge(arr, left, mid, right);
}
}
int main() {
int n;
printf("Enter number of elements: ");
scanf("%d", &n);
int *arr1 = (int *)malloc(n * sizeof(int));
int *arr2 = (int *)malloc(n * sizeof(int));
printf("Enter %d elements: ", n);
for (int i = 0; i < n; i++) {
scanf("%d", &arr1[i]);
arr2[i] = arr1[i]; // Copy for parallel sorting
}
double start, end;
// Sequential Sort
start = omp_get_wtime();
mergeSortSequential(arr1, 0, n - 1);
end = omp_get_wtime();
printf("Sequential Merge Sort Time: %f seconds\n",end-start);
// Parallel Sort
start = omp_get_wtime();
mergeSortParallel(arr2, 0, n - 1);
end = omp_get_wtime();
```

```
printf("Parallel Merge Sort Time: %f seconds\n",end-start);
printf("Sorted array: ");
for (int i = 0; i < n; i++)
printf("%d ", arr1[i]);
printf("\n");
free(arr1);
free(arr2);
return 0;
}
```

OUTPUT:

```
ubantu@ubantu-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ gcc -fopenmp
mergesort.c
ubantu@ubantu-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ ./a.out
Enter number of elements: 6
Enter 6 elements: 2 9 1 5 10 4
Sequential Merge Sort Time: 0.000001 seconds
Parallel Merge Sort Time: 0.000244 seconds
Sorted array: 1 2 4 5 9 10
```

EXPERIMENT NO 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

What is Scheduling in OpenMP?
Scheduling is a method in OpenMP to distribute iterations to different threads in for loop. Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.
Static:
Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads. If you do not specify chunk-size variable, OpenMP will divides iterations into chunks that are approximately equal in size and it distributes chunks to threads **in order.**
CODE:

```
#include<stdio.h>
#include<omp.h>
int main() {
int num_iterations;
printf("Enter the number of iterations: ");
scanf("%d", &num_iterations);
#pragma omp parallel
{
#pragma omp for schedule(static,2)
for (int i = 0; i < num_iterations; i++) {
printf("Thread %d: Iteration %d\n", omp_get_thread_num(), i);
}
}
return 0;
}
```

OUTPUT:
```
ubantu@ubantu-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ gcc -fopenmp
iterations.c
ubantu@ubantu-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ ./a.out
Enter the number of iterations: 6
Thread 1: Iteration 2
Thread 1: Iteration 3
Thread 2: Iteration 4
Thread 2: Iteration 5
Thread 0: Iteration 0
Thread 0: Iteration 1
```

EXPERIMENT NO 3

WRITE A OPENMP PROGRAM TO CALCULATE N FIBONACCI NUMBERS USING TASKS.

Tasks:
An OpenMP task is a single line of code or a structured block which is immediately "written down" in a list of tasks. The new task can be executed immediately, or it can be deferred. If the if clause is used and the argument evaluates to 0, then the task is executed immediately, superseding whatever else that thread is doing. There has to be an existing parallel thread team for this to work. Otherwise one thread ends up doing all tasks and you don't get any contribution to parallelism.
Fibonacci sequence:
The Fibonacci sequence is a sequence where the next term is the sum of the previous two terms. The first two terms of the Fibonacci sequence are 0 followed by 1. Ex: Fibonacci Sequence is : 0,1,1,2,3,5,8,13………

CODE:

```
#include <stdio.h>
#include <omp.h>
int fib(int n)
{
int i, j;
if (n<2)
return n;
else
{
#pragma omp task shared(i) firstprivate(n)
i=fib(n-1);

#pragma omp task shared(j) firstprivate(n)
j=fib(n-2);

#pragma omp taskwait
return i+j;
}
}

int main()
{
int n = 10;

omp_set_dynamic(0);
omp_set_num_threads(4);

#pragma omp parallel shared(n)
{
#pragma omp single
printf ("fib(%d) = %d\n", n, fib(n));
}
}
```

OUTPUT:

```
bantu@ubantu-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ gcc -fopenmp fibonacci.c
ubantu@ubantu-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ ./a.out fib(10) = 55
```

EXPERIMENT NO 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.

Parallel Directive:
The omp parallel directive explicitly instructs the compiler to parallelize the chosen block of code.
For Directive:
Causes the work done in a for loop inside a parallel region to be divided among threads.
Prime Number:
A prime number is a positive integer that is divisible only by 1 and itself. For example: 2, 3, 5, 7, 11, 13, 17…..

CODE:
```
#include<stdio.h>
#include <omp.h>
void main()
{
int prime[1000], i, j, n;

// Prompt user for input
printf("\n In order to find prime numbers from 1 to n, enter the value of n:");
scanf("%d", &n);

// Initialize all numbers as prime (set all to 1)
for(i = 1; i <= n; i++)
{
prime[i] = 1;
}

// 1 is not a prime number
prime[1]= 0;

// Sieve of Eratosthenes with parallelization
for(i = 2; i * i <= n; i++) {
#pragma omp parallel for
for(j = i * i; j <= n; j = j + i) {
if(prime[j] == 1) {
prime[j] = 0;
}
}
}

// Print prime numbers
printf("\n Prime numbers from 1 to %d are\n", n);
for(i = 2; i <= n; i++) {
```

```
if(prime[i] == 1) {
printf("%d\t", i);
}
}
printf("\n");
}
```

OUTPUT:

```
ubantu@ubantu-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ gcc -fopenmp prime.c
ubantu@ubantu-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ ./a.out
In order to find prime numbers from 1 to n, enter the value of n: 100 Prime
numbers from 1 to 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
```

2)

```
#include <stdio.h>
#include <omp.h>
int main()
{
int prime[1000], i, j, n;
// Prompt user for input
printf("\n In order to find prime numbers from 1 to n, enter the value of n:");
scanf("%d", &n);
// Initialize all numbers as prime (set all to 1)
for(i = 1; i <= n; i++) {
prime[i] = 1;
}

// 1 is not a prime number prime[1]
= 0;

// Sieve of Eratosthenes with parallelization for(i = 2; i
* i <= n; i++) {
#pragma omp parallel for for(j = i *
i; j <= n; j = j + i) { if(prime[j] == 1)
{
prime[j] = 0;
}
}
}

// Print prime numbers
printf("\n Prime numbers from 1 to %d are\n", n);
for(i = 2; i <= n; i++) {
if(prime[i] == 1) {
printf("%d\t", i);
}
}
double start, end;
// Sequential
start = omp_get_wtime();
prime[i];
end = omp_get_wtime();
printf("Sequential Time: %f seconds\n",end-start);
// Parallel
start = omp_get_wtime();
prime[i];
end = omp_get_wtime();
printf("Parallel Time: %f seconds\n",end-start);
printf("\n");
}
```

OUTPUT:

```
ubantu@ubantu-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ gcc -fopenmp prime.c
ubantu@ubantu-HP-Pro-Tower-280-G9-PCI-Desktop-PC:~$ ./a.out
In order to find prime numbers from 1 to n, enter the value of n:50 Prime
numbers from 1 to 50 are
2       3       5       7       11      13      17      19      23      29      31
        37      41      43      47
Sequential Time: 0.000001 seconds Parallel Time: 0.000000 seconds
```

EXPERIMENT NO 5

WRITE A MPI PROGRAM TO DEMONSTRATION OF MPI_SEND AND MPI_RECV.

MESSAGE-PASSING INTERFACE (MPI)

Message-Passing is a communication model used on distributed-memory architecture. MPI is a standard that specifies the message-passing libraries supporting parallel programming in C/C++ or Fortran. MPI_Send and MPI_Recv are the basic building blocks for essentially all of the more specialized MPI commands in MPI. They are also the basic communication tools in MPI application. Since MPI_Send and MPI_Recv involve two ranks, they are called "point-to-point" communication. The process of communicating data follows a standard pattern. Rank A decides to send data to rank B. It first packs the data into a buffer. This avoids sending multiple messages, which would take more time. Rank A then calls MPI_Send to create a message for rank B. The communication device is then given the responsibility of routing the message to the correct destination. Rank B must know that it is about to receive a message and acknowledge this by calling MPI_Recv.

MPI_RANK:

"Rank" refers to a unique identifier assigned to each process within a communication group. It's a logical way of numbering processes to facilitate communication between them. MPI automatically assigns ranks in the MPI_COMM_WORLD group when the MPI environment is initialized (MPI_Init).

CODE:

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

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

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

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

    char message[100];

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

    if (rank == 0) {
        // Process 0 sends a message
        strcpy(message, "Hello from Process 0");
        MPI_Send(message, strlen(message) + 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
        printf("Process 0 sent message: %s\n", message);
    } else if (rank == 1) {
        // Process 1 receives the message
        MPI_Recv(message, 100, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status);
        printf("Process 1 received message: %s\n", message);
    }

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

OUTPUT:

```
ssce@ssce:~/Desktop/openmpi-4.1.6$ mpicc send_recv_demo.c -o send_recv_demo
ssce@ssce:~/Desktop/openmpi-4.1.6$ mpirun -np 4 ./send_recv_demo
Process 0 sent message: Hello from Process 0
Process 1 received message: Hello from Process 0
```

EXPERIMENT NO 6

WRITE A MPI PROGRAM TO DEMONSTRATION OF DEADLOCK USING POINT TO POINT COMMUNICATION AND AVOIDANCE OF DEADLOCK BY ALTERING THE CALL SEQUENCE.

Deadlock:

Deadlock is an often-encountered situation in parallel processing. It results when two or more processes are in contention for the same set of resources. In communications, a typical scenario involves two processes wishing to exchange messages: each is trying to give a message to the other, but neither of them is ready to accept a message. **What is point-to-point communication in MPI?**

MPI processes communicate by explicitly sending and receiving messages. In point- to-point, messages are sent between two processes. Since MPI processes are independent, in order to coordinate work, they need to communicate by explicitly sending and receiving messages.

Demonstration of deadlock

CODE:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
    int rank, size, data = 100;
    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("This program requires exactly 2 processes.\n");
        MPI_Finalize();
        return 1;
    }

    printf("=== Running Deadlock Scenario ===\n");

    if (rank == 0) {
        // DEADLOCK: both processes send first
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Process 0 received data: %d\n", data);
    } else if (rank == 1) {
        MPI_Send(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
        printf("Process 1 received data: %d\n", data);
    }

    // Wait a bit and then run deadlock-free version
    MPI_Barrier(MPI_COMM_WORLD);
    printf("\n=== Running Deadlock-Free Version ===\n");

    if (rank == 0) {
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);  // send first
        MPI_Recv(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);  // then receive
        printf("Process 0 received data: %d\n", data);
    } else if (rank == 1) {
        MPI_Recv(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);  // receive first
        MPI_Send(&data, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);  // then send
        printf("Process 1 received data: %d\n", data);
    }

    MPI_Finalize();
    return 0;
}
```

OUTPUT:

```
ssce@ssce:~/Desktop/openmpi-4.1.6$ mpicc deadlockmerge.c -o deadlockmerge
ssce@ssce:~/Desktop/openmpi-4.1.6$ mpirun -np 2 ./deadlockmerge
=== Running Deadlock Scenario ===
=== Running Deadlock Scenario ===
Process 1 received data: 100
Process 0 received data: 100

=== Running Deadlock-Free Version ===

=== Running Deadlock-Free Version ===
Process 1 received data: 100
Process 0 received data: 100
```

EXPERIMENT NO 7

WRITE A MPI PROGRAM TO DEMONSTRATION OF BROADCAST OPERATION.

Broadcasting with MPI_Bcast:

A broadcast is one of the standard collective communication techniques. During a broadcast, one process sends the same data to all processes in a communicator. One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes. Basically process zero is the root process, and it has the initial copy of data. All of the other processes receive the copy of data.

CODE:

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

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

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

    // Get rank and size
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    if (rank == 0) {
        data = 42;  // Root process sets the data
        printf("Process %d broadcasting data = %d\n", rank, data);
    }

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

    // All processes print the received data
    printf("Process %d received data = %d\n", rank, data);

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

OUTPUT:

```
ssce@ssce:~/Desktop/openmpi-4.1.6$ mpicc broadcast_demo.c -o broadcast_demo
ssce@ssce:~/Desktop/openmpi-4.1.6$ mpirun -np 4 ./broadcast_demo
Process 0 broadcasting data = 42
Process 0 received data = 42
Process 2 received data = 42
Process 1 received data = 42
Process 3 received data = 42
```

EXPERIMENT NO 8

WRITE A MPI PROGRAM DEMONSTRATION OF MPI_SCATTER AND MPI_GATHER

MPI_Scatter:

MPI_Scatter is a collective routine that is very similar to MPI_Bcast. MPI_Scatter involves a designated root process sending data to all processes in a communicator. The primary difference between MPI_Bcast and MPI_Scatter is small but important. MPI_Bcast sends the same piece of data to all processes while MPI_Scatter sends chunks of an array to different processes.

MPI_Gather:

MPI_Gather is the inverse of MPI_Scatter. Instead of spreading elements from one process to many processes, MPI_Gather takes elements from many processes and gathers them to one single process. This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.

CODE:

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

int main(int argc, char *argv[]) {
   int rank, size;
   int send_data[4], recv_data, gathered_data[4];

   MPI_Init(&argc, &argv);                    // Initialize MPI
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);         // Get process rank
   MPI_Comm_size(MPI_COMM_WORLD, &size);         // Get number of processes

   if (size != 4) {
      if (rank == 0) {
         printf("Please run this program with 4 processes.\n");
      }
      MPI_Finalize();
      return 0;
   }

   // Only root process initializes the send_data array
   if (rank == 0) {
      send_data[0] = 10;
      send_data[1] = 20;
      send_data[2] = 30;
      send_data[3] = 40;
      printf("Root process sending data: 10, 20, 30, 40\n");
   }

   // Scatter: send one element of send_data to each process
   MPI_Scatter(send_data, 1, MPI_INT, &recv_data, 1, MPI_INT, 0,
MPI_COMM_WORLD);
   printf("Process %d received data = %d\n", rank, recv_data);
```

```
    // Do some operation (optional)
    recv_data += 1;  // Each process adds 1 to its data

    // Gather: collect updated data from all processes to root
    MPI_Gather(&recv_data, 1, MPI_INT, gathered_data, 1, MPI_INT, 0,
MPI_COMM_WORLD);

    // Root prints the gathered data
    if (rank == 0) {
        printf("Root process gathered data: ");
        for (int i = 0; i < 4; i++) {
            printf("%d ", gathered_data[i]);
        }
        printf("\n");
    }

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

OUTPUT:

```
ssce@ssce:~/Desktop/openmpi-4.1.6$ mpicc scatter_gather_demo.c -o scatter_gather_demo
ssce@ssce:~/Desktop/openmpi-4.1.6$ mpirun -np 4 ./scatter_gather_demo
Root process sending data: 10, 20, 30, 40
Process 0 received data = 10
Process 2 received data = 30
Process 3 received data = 40
Process 1 received data = 20
Root process gathered data: 11 21 31 41
```

EXPERIMENT NO 9

WRITE A MPI PROGRAM TO DEMONSTRATION OF MPI_REDUCE AND MPI_ALLREDUCE (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD)

MPI_Reduce:

Reduce is a classic concept from functional programming. Data reduction involves reducing a set of numbers into a smaller set of numbers via a function. For example, let's say we have a list of numbers [1, 2, 3, 4, 5]. Reducing this list of numbers with the sum function would produce sum([1, 2, 3, 4, 5]) = 15. Similarly, the multiplication reduction would yield multiply([1, 2, 3, 4, 5]) = 120. MPI_Reduce takes an array of input elements on each process and returns an array of output elements to the root process. The output elements contain the reduced result.

MPI_AllReduce:

Many parallel applications will require accessing the reduced results across all processes rather than the root process. In a similar complementary style of MPI_Allgather to MPI_Gather, MPI_Allreduce will reduce the values and distribute the results to all processes.

CODE:

```
#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);                          // Initialize MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);            // Get rank
    MPI_Comm_size(MPI_COMM_WORLD, &size);            // Get size

    // Each process sets its value as rank + 1 (just for demonstration)
    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--- Results from MPI_Reduce (only at root) ---\n");
        printf("Sum = %d\n", sum);
        printf("Product = %d\n", prod);
        printf("Maximum = %d\n", max);
        printf("Minimum = %d\n", min);
    }

    // ------------------- MPI_Allreduce -----------------------
    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);

    printf("\nProcess %d - Results from MPI_Allreduce:\n", rank);
    printf("All-Sum = %d, All-Prod = %d, All-Max = %d, All-Min = %d\n",
           all_sum, all_prod, all_max, all_min);

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

OUTPUT:

```
ssce@ssce:~/Desktop/openmpi-4.1.6$ mpicc reduce_allreduce_demo.c -o
reduce_allreduce_demo
ssce@ssce:~/Desktop/openmpi-4.1.6$ mpirun -np 4 ./reduce_allreduce_demo
Process 1 has value 2
Process 3 has value 4
Process 2 has value 3
Process 0 has value 1

--- Results from MPI_Reduce (only at root) ---
Sum = 10
Product = 24
Maximum = 4
Minimum = 1

Process 1 - Results from MPI_Allreduce:
All-Sum = 10, All-Prod = 24, All-Max = 4, All-Min = 1

Process 0 - Results from MPI_Allreduce:
All-Sum = 10, All-Prod = 24, All-Max = 4, All-Min = 1

Process 2 - Results from MPI_Allreduce:
All-Sum = 10, All-Prod = 24, All-Max = 4, All-Min = 1

Process 3 - Results from MPI_Allreduce:
All-Sum = 10, All-Prod = 24, All-Max = 4, All-Min = 1
```