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>
#define MAX 100000
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);
}
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);
    }
}
void copyArray(int *src, int *dest, int n) {
    for (int i = 0; i < n; i++) dest[i] = src[i];
```

```
}
int main() {
    int n;
    int *arr_seq, *arr_par;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    if (n > MAX) {
        printf("Max limit exceeded (%d)\n", MAX);
        return 1;
    }
    arr_seq = (int *)malloc(n * sizeof(int));
    arr_par = (int *)malloc(n * sizeof(int));
    // Generate random data
    for (int i = 0; i < n; i++) {
        arr_seq[i] = rand() % 10000;
    }
    copyArray(arr_seq, arr_par, n);
    double start, end;
    // Sequential sort
    start = omp_get_wtime();
    sequentialMergeSort(arr_seq, 0, n - 1);
    end = omp_get_wtime();
    printf("Sequential MergeSort Time: %.6f seconds\n", end - start);
    // Parallel sort
    start = omp_get_wtime();
    #pragma omp parallel
    {
        #pragma omp single
        parallelMergeSort(arr_par, 0, n - 1);
    }
    end = omp_get_wtime();
    printf("Parallel MergeSort Time: %.6f seconds\n", end - start);
    free(arr_seq); free(arr_par);
    return 0;
}
```

Step 1: Install GCC with OpenMP Support

```
sudo apt update
sudo apt install gcc
```

Ensure OpenMP is supported:

```
gcc –version
```

Step 2: Save the Program: Save the code above into a file named mergesort_openmp.c.

Step 3: Compile with OpenMP: `gcc -fopenmp -o mergesort_openmp mergesort_openmp.c`

Step 4: Run the Program: `./mergesort_openmp`

Output:

Enter number of elements: 100000
Sequential MergeSort Time: 0.092385 seconds
Parallel MergeSort Time: 0.039472 seconds

Viva Questions:

- **What is the primary goal of this program?**
  The program aims to compare the execution time of merge sort implemented sequentially and in parallel using OpenMP to highlight the performance benefits of parallel processing.

- **Why is merge sort suitable for parallelization?**
  Merge sort is a divide-and-conquer algorithm that naturally splits the array into independent subproblems, which can be sorted concurrently before merging, making it highly suitable for parallelization.

- **How is parallelism achieved in this program?**
  Using #pragma omp parallel sections, the two recursive calls of merge sort (left and right sub-arrays) are executed in parallel sections when performing parallelMergeSort.

- **What is the purpose of the #pragma omp single directive in the main function?**
  It ensures that only a single thread initially invokes the recursive parallel merge sort function, preventing redundant recursive calls by all threads.

- **How is memory managed during the merge operation?**
  Temporary arrays (L and R) are dynamically allocated using malloc during each merge and freed after use to prevent memory leaks.

- **What would happen if #pragma omp parallel sections were not used in parallelMergeSort?**
  The program would behave like a sequential merge sort since both recursive calls would execute serially, defeating the purpose of parallelization.

- **Why is array copying done before performing parallel sort?**
  To ensure a fair comparison between sequential and parallel executions by using the same input data, as sorting modifies the array.

- **How does OpenMP handle task creation internally for the sections?**
  Each #pragma omp section is treated as a separate task, potentially executed by different threads from the OpenMP thread pool concurrently.

- **What are the limitations of parallel merge sort in this implementation?**
  Excessive thread creation for small sub-arrays can cause overhead. Also, if the number of recursive levels exceeds available threads, parallelism may not yield significant performance gains.

- **How does the use of omp_get_wtime() aid in evaluating performance?**
  It provides accurate wall-clock time measurement, allowing objective comparison of execution durations for sequential and parallel versions.

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 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;
}
```

Step 1: Save the Program : Save the code above into a file named static_schedule.c
Step 2: Compile with OpenMP: gcc -fopenmp -o static_schedule static_schedule.c
Step 3: Run the Program: ./static_schedule

Output:

```
Enter the number of iterations: 6

Using schedule(static,2):

Thread 0 : Iteration 0
Thread 0 : Iteration 1
Thread 1 : Iteration 2
Thread 1 : Iteration 3
Thread 0 : Iteration 4
Thread 0 : Iteration 5
```

Viva Questions:

- **What does OMP_SCHEDULE=static,2 mean?**
  It assigns fixed-size chunks of 2 iterations to each thread in order.

- **What is the purpose of omp parallel for?**
  It parallelizes the for loop across multiple threads.

- **How does static scheduling work in OpenMP?**
  Iterations are divided before execution and assigned evenly to threads.

- **What is a "chunk"?**
  A group of consecutive loop iterations assigned to a thread.

- **What happens if the number of iterations is not a multiple of the chunk size?**
  The remaining iterations are still assigned, possibly unevenly.

- **What is omp_get_thread_num() used for?**
  It returns the ID of the thread executing the current block.

- **How many threads are created if we use -np 2?**
  Two threads will execute the parallel region.

- **Why is output sometimes unordered in parallel loops?**
  Because threads execute concurrently and write to output at different times.

- **How does this program demonstrate scheduling clearly?**
  It prints which iterations each thread is executing, showing chunk assignments.

- **Can chunk size affect performance?**
  Yes, inappropriate chunk sizes can lead to load imbalance or overhead.

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++) {
        #pragma omp task firstprivate(i)
        {
        int 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;
}
```

Step 1: Save the Program : Save the code above into a file named **fib_tasks.c**
Step 2: Compile with OpenMP: gcc -fopenmp -o fib_tasks fib_tasks.c
Step 3: Run the Program: ./ fib_tasks.c

Enter the number of Fibonacci numbers to calculate: 10
First 10 Fibonacci numbers using OpenMP tasks:

Fib(0) = 0
Fib(1) = 1
Fib(2) = 1
Fib(3) = 2
Fib(4) = 3
Fib(5) = 5
Fib(6) = 8
Fib(7) = 13
Fib(8) = 21
Fib(9) = 34
Execution time: 0.001234 seconds

Viva Questions

- **What is the purpose of using omp task in this Fibonacci program?**
  omp task is used to create separate tasks for recursive Fibonacci calls so that they can be executed in parallel, exploiting task-level parallelism.

- **Why is omp taskwait used in the fib function?**
  omp taskwait ensures that both recursive tasks (fib(n-1) and fib(n-2)) complete before computing the final result x + y.

- **What is the role of the firstprivate(i) clause in the task creation?**
  firstprivate(i) ensures each task receives its own private copy of the loop variable i, preserving its value at the time the task was created.

- **Why is omp single used in the main function?**
  omp single ensures that only one thread (from the parallel region) creates the set of tasks in the loop, avoiding redundant task creation.

- **What happens if we remove the omp parallel directive in main()?**
  Without omp parallel, no threads will be available to execute the created tasks in parallel; the program will execute serially.

- **What is the purpose of omp critical in this program?**
  omp critical prevents concurrent writes to the output stream (printf), avoiding mixed or corrupted output from multiple threads.

- **Is this program efficient for large values of n? Why or why not?**
  No, it is inefficient for large n due to redundant recursive calls and exponential time complexity. It demonstrates tasking, not optimized computation.

- **How does task parallelism differ from data parallelism in OpenMP?**
  Task parallelism divides the program into independent tasks (e.g., recursive calls), while data parallelism splits data into chunks and applies the same operation.

- **How many tasks are generated when calculating fib(n) using this method?**
  Approximately 2^n tasks are generated because each Fibonacci call spawns two subtasks, leading to exponential task creation.

- **What optimization could you apply to improve this Fibonacci task-based program?**
  Memoization (caching results) or switching to an iterative Fibonacci approach would significantly reduce redundant computations.

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 <stdlib.h>
#include <omp.h>
#include <math.h>

// Function to check if a number is prime
int is_prime(int num) {
    if (num < 2) return 0;
    if (num == 2) return 1;
    if (num % 2 == 0) return 0;
    for (int i = 3; i <= sqrt(num); 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 less than %d.\n", n);
        return 0;
    }

    printf("\n--- Serial Execution ---\n");
    double start_serial = omp_get_wtime();
    int serial_count = 0;
    for (int i = 2; i <= n; i++) {
        if (is_prime(i)) {
            serial_count++;
            // printf("%d ", i); // Optional: Uncomment to display primes
        }
    }
    double end_serial = omp_get_wtime();
    printf("Total primes found (serial): %d\n", serial_count);
    printf("Serial Execution Time: %.6f seconds\n", end_serial - start_serial);

    printf("\n--- Parallel Execution (OpenMP) ---\n");

    // Set number of threads explicitly
    omp_set_num_threads(4);

    double start_parallel = omp_get_wtime();
    int parallel_count = 0;
```

```
    #pragma omp parallel for reduction(+:parallel_count)
    for (int i = 2; i <= n; i++) {
        if (is_prime(i)) {
            parallel_count++;
            // #pragma omp critical
            // printf("%d ", i); // Optional: Uncomment to display primes
        }
    }
    double end_parallel = omp_get_wtime();
    printf("Total primes found (parallel): %d\n", parallel_count);
    printf("Parallel Execution Time: %.6f seconds\n", end_parallel - start_parallel);

    return 0;
}
```

Step 1: Save the Program : prime_parallel.c
Step 2: Compile with OpenMP: gcc -fopenmp -o prime_parallel prime_parallel.c –lm
Step 3: Run the Program: ./ prime_parallel.c

Output:

Enter the upper limit (n): 100000

--- Serial Execution ---

Total primes found (serial): 9592
Serial Execution Time: 0.188745 seconds

--- Parallel Execution (OpenMP) ---

Total primes found (parallel): 9592
Parallel Execution Time: 0.062341 seconds

Viva Questions:

- **What is the primary objective of this OpenMP program?**
  The objective is to compare the performance of serial and parallel implementations for counting prime numbers up to a given limit n using OpenMP.

- **Why is sqrt(num) used in the is_prime() function?**
  To optimize prime checking by reducing the number of iterations. A number greater than 1 is non-prime if it has a factor less than or equal to its square root.

- **What is the purpose of the directive #pragma omp parallel for reduction (+:parallel_count)?**
  It parallelizes the loop and uses a reduction clause to safely accumulate the number of primes (parallel_count) across threads without race conditions.

- **Why is the critical section commented out in the parallel version?**

It was optionally used to safely print prime numbers during parallel execution. However, printing inside parallel regions can slow down performance, so it's often disabled.

- **What are the advantages of using omp_get_wtime() in this program?**
omp_get_wtime() provides high-resolution wall-clock timing to accurately measure and compare execution time for serial and parallel blocks.

- **What scheduling type is used in this program's for loop, and why?**
The default static scheduling is applied here. It evenly divides iterations among threads, which works well when each iteration has nearly equal computational load, as in this case.

- **What is the purpose of omp_set_num_threads(4)?**
It explicitly sets the number of OpenMP threads to 4 for the parallel region. This helps control resource usage and test performance scalability.

- **How does the reduction clause avoid race conditions?**
Each thread maintains a private copy of the variable (parallel_count), and after the loop, these are combined in a thread-safe manner into a single final result.

- **Could this program benefit from dynamic scheduling? Why or why not?**
Not significantly, because the computation per iteration (checking if a number is prime) is relatively uniform. Dynamic scheduling is more beneficial for irregular workloads.

- **What is the observed performance gain when using OpenMP parallelization in this program?**
The performance gain depends on the hardware and input size. Typically, parallel execution reduces time compared to serial execution, especially for large n, by utilizing multiple cores.

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

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

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

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

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

   if (rank == 0) {
      char message[] = "Hello from Process 0 to Process 1";
      printf("Process %d sending message: %s\n", rank, message);
      MPI_Send(message, strlen(message) + 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
   } else if (rank == 1) {
      char received_message[100];
      MPI_Recv(received_message, 100, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
&status);
      printf("Process %d received message: %s\n", rank, received_message);
   }

   MPI_Finalize();
   return 0;
}
```

Step 1: Install MPI : **sudo apt update**

sudo apt install openmpi-bin openmpi-common libopenmpi-dev

Step 2: Check MPI installation:

mpirun –version

Step 3: Save the MPI Program : mpi_send_recv.c

Step 4: Compile the MPI Program: mpicc mpi_send_recv.c -o mpi_send_recv

Step 5: Run the MPI Program: mpirun -np 2 ./mpi_send_recv

Output:

Process 0 sending message: Hello from Process 0 to Process 1
Process 1 received message: Hello from Process 0 to Process 1

Viva Questions:

- **What is the purpose of MPI_Send and MPI_Recv?**
  They are used for point-to-point communication to send and receive messages between processes in MPI.

- **What does the tag parameter signify in MPI_Send and MPI_Recv?**
  It identifies the message and helps match send and receive operations correctly.

- **What is MPI_COMM_WORLD?**
  It is the default communicator that includes all MPI processes in a given program.

- **What is the role of MPI_Status?**
  It provides information about a received message, such as the source and tag.

- **What is the output if the program is run with only one process?**
  It prints a message that at least two processes are required and exits.

- **Why do we use strlen(message) + 1 in MPI_Send?**
  To include the null-terminator \0 so the string can be correctly received and printed.

- **Can MPI_Send and MPI_Recv be used in a non-blocking way?**
  Not directly—non-blocking communication uses MPI_Isend and MPI_Irecv.

- **What happens if the receiver buffer size is smaller than the message?**
  It can cause a buffer overflow or program crash due to insufficient memory.

- **What ensures synchronization between sender and receiver?**
  Blocking behavior of MPI_Send and MPI_Recv ensures synchronization.

- **Can multiple processes use the same tag for different messages?**
  Yes, as long as the source and tag combination is correctly matched.

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

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[]) {
int rank, size;
int msg_send = 100, msg_recv;
MPI_Status status;
int cause_deadlock = 0;
// Initialize MPI
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get process rank
MPI_Comm_size(MPI_COMM_WORLD, &size); // Get number of processes
// Check for mode argument
if (argc != 2) {
if (rank == 0)
printf("Usage: %s --deadlock | --avoid\n", argv[0]);
MPI_Finalize();
return 1;
}
if (strcmp(argv[1], "--deadlock") == 0) {
cause_deadlock = 1;
} else if (strcmp(argv[1], "--avoid") == 0) {
cause_deadlock = 0;
} else {
if (rank == 0)
printf("Invalid argument. Use --deadlock or --avoid\n");
MPI_Finalize();
return 1;
}
if (size != 2) {
if (rank == 0)
printf("This program requires exactly 2 processes.\n");
MPI_Finalize();
return 1;
}
// Begin communication
if (cause_deadlock) {
// DEADLOCK: both send first, then receive
if (rank == 0) {
printf("Process 0 sending to Process 1...\n");
MPI_Send(&msg_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
printf("Process 0 receiving from Process 1...\n");
MPI_Recv(&msg_recv, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
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);
printf("Process 1 receiving from Process 0...\n");
MPI_Recv(&msg_recv, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
printf("Process 1 received from Process 0: %d\n", msg_recv);
}
} else {
// DEADLOCK AVOIDANCE: rank 0 sends first, rank 1 receives first
if (rank == 0) {
printf("Process 0 sending to Process 1...\n");
MPI_Send(&msg_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
printf("Process 0 receiving from Process 1...\n");
MPI_Recv(&msg_recv, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
printf("Process 0 received from Process 1: %d\n", msg_recv);
} else if (rank == 1) {
printf("Process 1 receiving from Process 0...\n");
MPI_Recv(&msg_recv, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
printf("Process 1 received from Process 0: %d\n", msg_recv);
printf("Process 1 sending to Process 0...\n");
MPI_Send(&msg_send, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}
}
MPI_Finalize();
return 0;
}
```

Step 1: Save the MPI Program : mpi_deadlock_demo.c
Step 2: Compile the MPI Program:
mpicc mpi_deadlock_demo.c -o mpi_deadlock_demo
Step 3: Run the MPI Program: mpirun -np 2 ./mpi_deadlock_demo –deadlock
Step 4: Run the MPI Program: mpirun -np 2 ./mpi_deadlock_demo --avoid

Output 1:

Process 0 sending to Process 1...
Process 1 sending to Process 0...

Output 2:

Process 0 sending to Process 1...
Process 1 receiving from Process 0...
Process 1 received from Process 0: 100
Process 1 sending to Process 0...
Process 0 receiving from Process 1...
Process 0 received from Process 1: 100

Viva Questions:

- **What is deadlock in MPI?**
  Deadlock occurs when two or more processes wait indefinitely for each other to complete communication.

- **What MPI functions are used in point-to-point communication?**
  MPI_Send() and MPI_Recv().

- **How can deadlock occur using MPI_Send() and MPI_Recv()?**
  If both processes use blocking MPI_Send() before MPI_Recv(), they can wait indefinitely.

- **How can deadlock be avoided in point-to-point communication?**
  By changing the order of send/receive operations or using non-blocking functions like MPI_Isend() and MPI_Irecv().

- **What is a blocking communication?**
  The function waits until the operation completes before moving forward.

- **What is a tag in MPI communication?**
  A message identifier used to match sends and receives.

- **Why is it necessary to use MPI_COMM_WORLD?**
  It defines the default communicator including all processes.

- **What is MPI_STATUS_IGNORE used for?**
  It ignores the status return of a receive operation.

- **What is the role of MPI_Init and MPI_Finalize?**
  They initialize and clean up the MPI environment.

- **How many processes are required to demonstrate deadlock?**
  At least two.

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

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char *argv[]) {
int rank, size;
int data;

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

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

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

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

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

// All processes (including root) receive the broadcasted data
printf("Process %d received data = %d\n", rank, data);

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

Step 1: Save the MPI Program : mpi_broadcast_demo.c

Step 2: Compile the MPI Program:

mpicc mpi_broadcast_demo.c -o mpi_broadcast_demo

Step 3: Run the MPI Program: mpirun -np 4 ./mpi_broadcast_demo

Output:

Process 0 (Root) broadcasting data = 42
Process 0 received data = 42
Process 1 received data = 42
Process 2 received data = 42
Process 3 received data = 42

Viva Questions:

.

- **What is the purpose of MPI_Bcast()?**
  It broadcasts data from one process (root) to all other processes.

- **Which process initiates the broadcast?**
  The root process.

- **Can the root process receive the data as well?**
  Yes, it also participates in the broadcast.

- **What kind of data can be broadcasted?**
  Any datatype supported by MPI, such as MPI_INT, MPI_FLOAT, etc.

- **What arguments are required by MPI_Bcast()?**
  Buffer, count, datatype, root rank, and communicator.

- **Does MPI_Bcast use point-to-point internally?**
  Yes, it is implemented using multiple send/receive operations internally.

- **Is MPI_Bcast blocking?**
  Yes, it blocks until the broadcast is complete.

- **How many times is MPI_Bcast called in a program?**
  Once by each process for a broadcast operation.

- **Can we have multiple broadcasts in one program?**
  Yes.

- **What happens if a non-root process calls MPI_Bcast with a different buffer size?**
  It leads to undefined behavior or runtime errors.

8. 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;
int send_data[100];                                    // Array for root to scatter
int recv_data; // Each process receives one element
int gathered_data[100];                                // Root will gather data here
MPI_Init(&argc, &argv);                                // Initialize MPI
MPI_Comm_rank(MPI_COMM_WORLD, &rank);                  // Get process rank
MPI_Comm_size(MPI_COMM_WORLD, &size);                  // Get total processes
if (rank == 0)
{
// Initialize data to scatter (only root does this)
for (int i = 0; i < size; i++)
{
send_data[i] = i * 10;                                 // Example data: 0, 10, 20, ...
}
printf("Process 0 (Root): Data prepared for scattering:\n");
for (int i = 0; i < size; i++) {
printf("%d ", send_data[i]);
}
printf("\n");
}
// Scatter: each process receives one element of the send_data array
MPI_Scatter(send_data, 1, MPI_INT, &recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Each process prints what it received
printf("Process %d received value: %d\n", rank, recv_data);

// Each process modifies its data (optional)
recv_data += 1;

// Gather: all modified recv_data values are sent back to root
MPI_Gather(&recv_data, 1, MPI_INT, gathered_data, 1, MPI_INT, 0,
MPI_COMM_WORLD);

// Root process prints the gathered data
if (rank == 0) {
printf("Process 0 (Root): Data gathered after modification:\n");
for (int i = 0; i < size; i++) {
printf("%d ", gathered_data[i]);
}
printf("\n");
}
MPI_Finalize(); // Finalize MPI
return 0;
}
```

Step 1: Save the MPI Program : mpi_scatter_gather_demo.c
Step 2: Compile the MPI Program:
mpicc mpi_scatter_gather_demo.c -o mpi_scatter_gather_demo

Step 3: Run the MPI Program: mpirun -np 4 ./mpi_scatter_gather_demo

Output:

Process 0 (Root): Data prepared for scattering: 0 10 20 30
Process 0 received value: 0
Process 1 received value: 10
Process 2 received value: 20
Process 3 received value: 30
Process 0 (Root): Data gathered after modification: 1 11 21 31

Viva Questions:

- **What does MPI_Scatter do?**
  It divides data from the root and sends portions to all processes.
- **What is MPI_Gather used for?**
  It collects data from all processes and assembles it at the root.
- **Do all processes need to call MPI_Scatter/MPI_Gather?**
  Yes, all processes in the communicator must call them.
- **Is the data distributed equally in MPI_Scatter?**
  Yes, each process receives an equal portion of the total data.
- **What if the number of elements is not divisible by the number of processes?**
  It may cause errors or incomplete data distribution.
- **What is the role of the root in MPI_Scatter and MPI_Gather?**
  The root sends and receives the complete data buffer, respectively.
- **Are MPI_Scatter and MPI_Gather blocking?**
  Yes, both are blocking operations.
- **What is the signature of MPI_Scatter?**
  MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
- **Can these operations be performed without the root process?**
  No, the root is mandatory for data distribution and collection.
- **Can we use `MPI_Scatter` with different datatypes?**
  No, all processes must use the same datatype.

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

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char* argv[]) {
int rank, size;
int value, sum, product, max, min;
int all_sum, all_product, all_max, all_min;
MPI_Init(&argc, &argv); // Initialize the MPI environment
MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get the rank of the process
MPI_Comm_size(MPI_COMM_WORLD, &size); // Get the total number of processes

// Each process has its own value (for simplicity, we use rank + 1)
value = rank + 1;
printf("Process %d has value: %d\n", rank, value);

// ==============================
// Reduce operations (results gathered at root process - rank 0)
// ==============================

MPI_Reduce(&value, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
MPI_Reduce(&value, &product, 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--- MPI_Reduce Results (at Root Process 0) ---\n");
printf("Sum = %d\n", sum);
printf("Product = %d\n", product);
printf("Maximum = %d\n", max);
printf("Minimum = %d\n", min);
}

// ==============================
// Allreduce operations (results available at all processes)
// ==============================

MPI_Allreduce(&value, &all_sum, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(&value, &all_product, 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 - MPI_Allreduce Results:\n", rank);
printf("Sum = %d\n", all_sum);
printf("Product = %d\n", all_product);
printf("Maximum = %d\n", all_max);
printf("Minimum = %d\n", all_min);
MPI_Finalize();
return 0;
}
```

Step 1: Save the MPI Program : mpi_reduce_allreduce.c
Step 2: Compile the MPI Program:

mpicc mpi_reduce_allreduce.c -o mpi_reduce_allreduce

Step 3: Run the MPI Program: mpirun -np 4 ./mpi_reduce_allreduce

Output:

```
Process 0 has value: 1
Process 1 has value: 2
Process 2 has value: 3
Process 3 has value: 4

--- MPI_Reduce Results (at Root Process 0) ---
Sum = 10
Product  = 24
Maximum  = 4
Minimum  = 1

Process 0 - MPI_Allreduce Results:
Sum = 10
Product  = 24
Maximum  = 4
Minimum  = 1

Process 1 - MPI_Allreduce Results:
Sum = 10
Product  = 24
Maximum  = 4
Minimum  = 1

Process 2 - MPI_Allreduce Results:
Sum = 10
Product  = 24
Maximum  = 4
Minimum  = 1

Process 3 - MPI_Allreduce Results:
Sum  = 10
Product  = 24
Maximum  = 4
Minimum  = 1
```

Viva Questions:

- **What is the purpose of MPI_Reduce?**
  It performs a reduction operation (e.g., sum, max) and returns the result to the root.

- **What does MPI_Allreduce do?**
  It performs a reduction and distributes the result to all processes.

- **What are some common operations in MPI_Reduce?**
  MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD.

- **How does MPI_Reduce differ from MPI_Allreduce?**
  MPI_Reduce returns result to root; MPI_Allreduce to all.

- **Are these reduction operations commutative?**
  Yes, MPI reduction operations are assumed to be commutative and associative.

- **Can we define custom reduction operations?**
  Yes, using MPI_Op_create.

- **What is the significance of the root parameter in MPI_Reduce?**
  It specifies which process receives the final result.

- **Can MPI_Allreduce be used without a root?**
  Yes, it returns results to all processes.

- **Are these operations synchronous?**
  Yes, they are blocking and require participation from all processes.

- **What happens if datatypes mismatch in reduction?**
  The program leads to undefined behavior or runtime errors.