PARALLEL COMPUTING

Dr.Lokesh M R
Professor,
Department of Information end Engineering,
A J Institute of Engineering and Technology, Mangaluru

Syllabus

MODULE-1: Introduction to parallel programming, Parallel hardware and parallel software –

1. Classifications of Parallel Computers,
2. MIMD systems,
3. Interconnection networks,
4. Cache coherence,
5. Shared-memory vs. distributed-memory,
6. Coordinating the processes/threads,
7. Shared-memory,
8. Distributed-memory.

MODULE-2: GPU programming, Programming hybrid systems, MIMD systems, GPUs, Performance

1. Speedup and efficiency in MIMD systems,
2. Amdahl's law,
3. Scalability in MIMD systems,
4. Taking timings of MIMD programs,
5. GPU performance.

MODULE-3: Distributed memory programming with MPI –

1. MPI functions,
2. The trapezoidal rule in MPI,
3. Dealing with I/O,
4. Collective communication,
5. MPI-derived datatypes,
6. Performance evaluation of MPI programs,
7. A parallel sorting algorithm.

MODULE-4: Shared-memory programming with OpenMP –

1. openmp pragmas and directives,
2. The trapezoidal rule,
3. Scope of variables,
4. The reduction clause,
5. loop carried dependency,
6. scheduling,
7. producers and consumers,
8. Caches,
9. cache coherence and false sharing in openmp,
10. tasking,
11. thread safety.

MODULE-5: GPU programming with CUDA

1. GPUs and GPGPU,
2. GPU architectures,
3. Heterogeneous computing,
4. Threads,
5. blocks, and grids
6. Nvidia compute capabilities and device architectures,
7. Vector addition,
8. Returning results from CUDA kernels,
9. CUDA trapezoidal rule I,
10. CUDA trapezoidal rule II: improving performance,
11. CUDA trapezoidal rule III: blocks with more than one warp.

PRACTICAL COMPONENT OF IPCC: Experiments

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

gedit mergesort_sequential_parallel_openmp.c

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

// Function to merge two sorted subarrays into a single sorted array
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

    // Copy data to temporary arrays L[] and R[]
    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];

    // Merge the temporary arrays back into arr[]
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }

    free(L);
    free(R);
}

// Sequential merge sort
void sequential_merge_sort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
```

```
        sequential_merge_sort(arr, left, mid);
        sequential_merge_sort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

// Parallel merge sort using OpenMP sections
void parallel_merge_sort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        // Use OpenMP sections to parallelize the two recursive calls
        #pragma omp parallel sections
        {
            #pragma omp section
            {
                parallel_merge_sort(arr, left, mid);
            }
            #pragma omp section
            {
                parallel_merge_sort(arr, mid + 1, right);
            }
        }

        // Merge the sorted halves (sequential merge for simplicity)
        merge(arr, left, mid, right);
    }
}

// Function to print the array
void print_array(int arr[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    // Allocate and initialize the arrays with random values
    int *arr_seq = (int *)malloc(n * sizeof(int));
    int *arr_par = (int *)malloc(n * sizeof(int));
    srand(time(0));
    for (int i = 0; i < n; i++) {
        arr_seq[i] = arr_par[i] = rand() % 1000; // Random numbers between 0 and 999
    }

    printf("Original array: ");
    print_array(arr_seq, n);

    // Measure sequential merge sort time
    double start_seq = omp_get_wtime();
    sequential_merge_sort(arr_seq, 0, n - 1);
    double end_seq = omp_get_wtime();
    double seq_time = end_seq - start_seq;

    printf("Sorted array (sequential): ");
    print_array(arr_seq, n);
```

```
    // Measure parallel merge sort time
    double start_par = omp_get_wtime();
    parallel_merge_sort(arr_par, 0, n - 1);
    double end_par = omp_get_wtime();
    double par_time = end_par - start_par;

    printf("Sorted array (parallel): ");
    print_array(arr_par, n);

    // Print execution times
    printf("Sequential Merge Sort Time: %f seconds\n", seq_time);
    printf("Parallel Merge Sort Time: %f seconds\n", par_time);
    printf("Time Difference (Sequential - Parallel): %f seconds\n", seq_time - par_time);

    // Free allocated memory
    free(arr_seq);
    free(arr_par);

    return 0;
}
```

How to Compile and Run:

1. **Compile the program**: Use a compiler that supports OpenMP, such as gcc. On a Unix-like system, you can compile with:

   gcc -fopenmp mergesort_sequential_parallel_openmp.c -o mergesort

   The -fopenmp flag enables OpenMP support.
2. **Run the program**:

   ./mergesort

   When prompted, enter the number of elements (n). The program will generate a random array, sort it using both sequential and parallel merge sort, and display the execution times.
3. **Set the number of threads** (optional): You can control the number of threads used by OpenMP by setting the environment variable OMP_NUM_THREADS before running the program. For example:

   export OMP_NUM_THREADS=4
   ./mergesort

   This sets the number of threads to 4.

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

openmp_iteration_chunks_formatted.c

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

int main() {
    int num_iterations;

    // Input: Number of iterations
    printf("Enter the number of iterations: ");
```

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

    // Ensure non-negative iterations
    if (num_iterations < 0) {
        printf("Number of iterations must be non-negative.\n");
        return 1;
    }

    // Set the number of threads (optional, can be controlled via environment variable)
    // For example output, we assume 2 threads; user can set via OMP_NUM_THREADS
    printf("Note: Set the number of threads using OMP_NUM_THREADS (e.g., export
OMP_NUM_THREADS=2)\n");

    // Parallel for loop with static scheduling, chunk size of 2
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        #pragma omp for schedule(static, 2)
        for (int i = 0; i < num_iterations; i++) {
            // Use a critical section to avoid race conditions in output
            #pragma omp critical
            {
                printf("Thread %d: Iteration %d\n", thread_id, i);
            }
        }
    }

    return 0;
}
```

Updated Compilation and Execution:

1. **Compile**:

   ```
   gcc -fopenmp openmp_iteration_chunks_formatted.c -o iteration_chunks_formatted
   ```

2. **Set Threads and Schedule**:

   ```
   export OMP_NUM_THREADS=2
   export OMP_SCHEDULE="static,2"
   ```

3. **Run**:

   ```
   ./iteration_chunks_formatted
   ```

Example Output (Updated Program):

For 2 threads and 4 iterations:

Enter the number of iterations: 4

Thread 0: Iterations 0 −− 1

Thread 1: Iterations 2 −− 3

For 2 threads and 6 iterations:

Enter the number of iterations: 6

Thread 0: Iterations 0 −− 1

Thread 1: Iterations 2 −− 3

Thread 0: Iterations 4 −− 5

Explanation of the Updated Program:

- **Input**: The program takes the number of iterations as user input.
- **Scheduling**:
  - The #pragma omp for schedule(static, 2) directive ensures that iterations are divided into chunks of 2, distributed statically among threads.

    - With static,2, if there are 2 threads and 4 iterations, Thread 0 gets iterations 0–1, and Thread 1 gets iterations 2–3.
- **Tracking Iterations**:
    - Arrays start_iterations and end_iterations track the first and last iteration executed by each thread.
    - The critical section ensures thread-safe updates to these arrays.
- **Output Formatting**:
    - After the loop, the program prints the range of iterations for each thread in the format "Thread X: Iterations A −− B".
    - Only threads that executed iterations (i.e., count_iterations[t] > 0) are printed.
- **Memory Management**: Dynamically allocated arrays are freed to prevent memory leaks.

Notes:

- **Static Scheduling**: The static,2 schedule ensures that iterations are divided into chunks of 2 and assigned to threads in a round-robin manner. For 4 iterations and 2 threads, Thread 0 gets iterations 0–1, Thread 1 gets 2–3.
- **Scalability**: If the number of iterations is not evenly divisible by the chunk size or number of threads, some threads may handle more chunks (e.g., for 6 iterations, Thread 0 gets 0–1 and 4–5, Thread 1 gets 2–3).
- **Environment Variables**: The OMP_NUM_THREADS and OMP_SCHEDULE variables must be set before running the program to control the number of threads and scheduling policy.
- **Alternative Approach**: You could also set the schedule dynamically in the code using omp_set_schedule(omp_sched_static, 2), but the environment variable approach aligns with the requirement (OMP_SCHEDULE=static,2).

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

fibonacci_openmp_tasks.c

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

// Function to calculate the nth Fibonacci number using OpenMP tasks
long long fibonacci(int n) {
   if (n <= 1) return n;

   long long fib_n_minus_1, fib_n_minus_2;

   // Create a task for computing fib(n-1)
   #pragma omp task shared(fib_n_minus_1)
   fib_n_minus_1 = fibonacci(n - 1);

   // Create a task for computing fib(n-2)
   #pragma omp task shared(fib_n_minus_2)
   fib_n_minus_2 = fibonacci(n - 2);

   // Wait for both tasks to complete
   #pragma omp taskwait

   return fib_n_minus_1 + fib_n_minus_2;
}

int main() {
   int n;

   // Input: Number of Fibonacci numbers to compute
   printf("Enter the number of Fibonacci numbers to compute: ");
   scanf("%d", &n);

   // Validate input
```

```
    if (n <= 0) {
        printf("Number of Fibonacci numbers must be positive.\n");
        return 1;
    }
    if (n > 92) { // long long can handle up to fib(92) before overflow
        printf("Input too large; long long can compute up to 92 Fibonacci numbers.\n");
        return 1;
    }

    // Array to store Fibonacci numbers
    long long *fib = (long long *)malloc(n * sizeof(long long));
    if (fib == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    // Compute Fibonacci numbers using OpenMP tasks
    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < n; i++) {
                fib[i] = fibonacci(i);
            }
        }
    }

    // Print the Fibonacci numbers
    printf("First %d Fibonacci numbers:\n", n);
    for (int i = 0; i < n; i++) {
        printf("Fib(%d) = %lld\n", i, fib[i]);
    }

    // Free allocated memory
    free(fib);

    return 0;
}
```

How to Compile and Run:

1. **Compile the Program**: Use a compiler that supports OpenMP, such as gcc. Include the -fopenmp flag to enable OpenMP support:
   bash
   CollapseWrapRun
   Copy
   gcc -fopenmp fibonacci_openmp_tasks.c -o fibonacci_tasks
2. **Set the Number of Threads** (Optional): You can control the number of threads using the OMP_NUM_THREADS environment variable. For example, to use 4 threads:
   bash
   CollapseWrapRun
   Copy
   export OMP_NUM_THREADS=4
3. **Run the Program**:
   bash
   CollapseWrapRun
   Copy
   ./fibonacci_tasks
   When prompted, enter the number of Fibonacci numbers to compute (e.g., 10).

Example Output:

For n = 10 with 2 threads:
Enter the number of Fibonacci numbers to compute: 10
First 10 Fibonacci numbers:
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

Example with Larger Input:
For n = 15:
Enter the number of Fibonacci numbers to compute: 15
First 15 Fibonacci numbers:
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
Fib(10) = 55
Fib(11) = 89
Fib(12) = 144
Fib(13) = 233
Fib(14) = 377

Explanation of the Code:

- **Fibonacci Function**:
  - The fibonacci function computes the nth Fibonacci number recursively.
  - Base cases (n <= 1) return n directly.
  - For n > 1, it computes fib(n-1) and fib(n-2) and returns their sum.
- **OpenMP Tasks**:
  - The #pragma omp task directive creates a task for computing fib(n-1) and another for fib(n-2).
  - The shared clause ensures that the variables fib_n_minus_1 and fib_n_minus_2 are accessible to the tasks.
  - The #pragma omp taskwait directive ensures that both tasks complete before the sum is computed, avoiding race conditions.
- **Main Function**:
  - Takes the number of Fibonacci numbers (n) as input.
  - Validates the input: Ensures n is positive and not too large (to avoid overflow with long long).
  - Allocates an array fib to store the Fibonacci numbers.
  - Uses a parallel region with a single thread (#pragma omp single) to generate tasks for computing each Fibonacci number.
  - The tasks are executed by available threads in the parallel region.
- **Parallelization**:
  - The #pragma omp parallel directive creates a team of threads.
  - The #pragma omp single ensures that only one thread executes the loop that generates tasks, but the tasks themselves are distributed across all threads.
  - This approach allows recursive calls to be parallelized, as each recursive step creates new tasks that can be executed concurrently.
- **Data Type**:
  - Uses long long to handle larger Fibonacci numbers (up to fib(92) before overflow, as fib(93) exceeds the range of a 64-bit long long).

Notes on Performance and Scalability:

- **Task Overhead**: For small n (e.g., n < 30), the overhead of creating tasks may outweigh the benefits of parallelization, making the program slower than a sequential version. OpenMP tasks are more beneficial for larger n or deeper recursion.
- **Scalability**: The program scales well for moderate n (e.g., n = 40), as the recursive nature of Fibonacci creates many tasks that can be distributed across threads. However, for very large n, the number of tasks can become excessive, leading to overhead.
- **Optimization**: For better performance, you could add a task cutoff (e.g., switch to sequential computation for small n within the fibonacci function) to reduce task creation overhead:
  c
  CollapseWrap
  Copy
  if (n < 20) return sequential_fibonacci(n); *// Sequential for small n*
- **Overflow**: The program limits n to 92 because fib(93) exceeds the maximum value of a long long (approximately 2^63 - 1). For larger n, you would need a big integer library.

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.

prime_numbers_openmp.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include <omp.h>

// Function to find primes using Sieve of Eratosthenes (Serial version)
void serial_sieve(int n, bool *is_prime, int *primes, int *num_primes) {
    // Initialize all numbers as prime
    for (int i = 0; i <= n; i++) {
        is_prime[i] = true;
    }
    is_prime[0] = is_prime[1] = false;

    // Sieve of Eratosthenes
    for (int i = 2; i <= (int)sqrt(n); i++) {
        if (is_prime[i]) {
            for (int j = i * i; j <= n; j += i) {
                is_prime[j] = false;
            }
        }
    }

    // Collect primes into the primes array
    *num_primes = 0;
    for (int i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes[*num_primes] = i;
            (*num_primes)++;
        }
    }
}

// Function to find primes using Sieve of Eratosthenes (Parallel version with OpenMP)
void parallel_sieve(int n, bool *is_prime, int *primes, int *num_primes) {
    // Initialize all numbers as prime
    #pragma omp parallel for
    for (int i = 0; i <= n; i++) {
```

```
        is_prime[i] = true;
    }
    is_prime[0] = is_prime[1] = false;

    // Parallel Sieve of Eratosthenes
    for (int i = 2; i <= (int)sqrt(n); i++) {
        if (is_prime[i]) {
            #pragma omp parallel for
            for (int j = i * i; j <= n; j += i) {
                is_prime[j] = false;
            }
        }
    }

    // Collect primes into the primes array (sequential for simplicity)
    *num_primes = 0;
    for (int i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes[*num_primes] = i;
            (*num_primes)++;
        }
    }
}

int main() {
    int n;

    // Input: Upper limit for finding primes
    printf("Enter the upper limit to find prime numbers (1 to n): ");
    scanf("%d", &n);

    // Validate input
    if (n < 1) {
        printf("Upper limit must be at least 1.\n");
        return 1;
    }

    // Allocate arrays for both serial and parallel versions
    bool *is_prime_serial = (bool *)malloc((n + 1) * sizeof(bool));
    bool *is_prime_parallel = (bool *)malloc((n + 1) * sizeof(bool));
    int *primes_serial = (int *)malloc((n + 1) * sizeof(int));
    int *primes_parallel = (int *)malloc((n + 1) * sizeof(int));
    int num_primes_serial = 0, num_primes_parallel = 0;

    if (!is_prime_serial || !is_prime_parallel || !primes_serial || !primes_parallel) {
        printf("Memory allocation failed.\n");
        free(is_prime_serial);
        free(is_prime_parallel);
        free(primes_serial);
        free(primes_parallel);
        return 1;
    }

    // Serial execution
    double start_serial = omp_get_wtime();
    serial_sieve(n, is_prime_serial, primes_serial, &num_primes_serial);
    double end_serial = omp_get_wtime();
    double serial_time = end_serial - start_serial;

    // Parallel execution
```

```
    double start_parallel = omp_get_wtime();
    parallel_sieve(n, is_prime_parallel, primes_parallel, &num_primes_parallel);
    double end_parallel = omp_get_wtime();
    double parallel_time = end_parallel - start_parallel;

    // Print the prime numbers (from serial version)
    printf("Prime numbers from 1 to %d:\n", n);
    for (int i = 0; i < num_primes_serial; i++) {
        printf("%d ", primes_serial[i]);
    }
    printf("\nTotal number of primes: %d\n", num_primes_serial);

    // Print execution times
    printf("\nSerial Execution Time: %f seconds\n", serial_time);
    printf("Parallel Execution Time: %f seconds\n", parallel_time);
    printf("Time Difference (Serial - Parallel): %f seconds\n", serial_time - parallel_time);

    // Free allocated memory
    free(is_prime_serial);
    free(is_prime_parallel);
    free(primes_serial);
    free(primes_parallel);

    return 0;
}
```

How to Compile and Run:

1. **Compile the Program**: Use a compiler that supports OpenMP, such as gcc. Include the -fopenmp flag to enable OpenMP support:
   gcc -fopenmp prime_numbers_openmp.c -o prime_numbers -lm
   - The -lm flag links the math library for sqrt (used in the Sieve algorithm).
   - The -fopenmp flag enables OpenMP support.
2. **Set the Number of Threads** (Optional): You can control the number of threads using the OMP_NUM_THREADS environment variable. For example, to use 4 threads:
   export OMP_NUM_THREADS=4
3. **Run the Program**:
   ./prime_numbers
   When prompted, enter the upper limit n (e.g., 100).

Example Output:

For n = 100 with 4 threads:
Enter the upper limit to find prime numbers (1 to n): 100
Prime numbers from 1 to 100:
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
Total number of primes: 25

Serial Execution Time: 0.000012 seconds
Parallel Execution Time: 0.000018 seconds
Time Difference (Serial - Parallel): -0.000006 seconds
For a larger n = 1000000 with 4 threads:
Enter the upper limit to find prime numbers (1 to n): 1000000
Prime numbers from 1 to 1000000:
2 3 5 7 11 ... 999961 999979 999983 999989 999997
Total number of primes: 78498

Serial Execution Time: 0.008214 seconds
Parallel Execution Time: 0.003892 seconds
Time Difference (Serial - Parallel): 0.004322 seconds

Explanation of the Code:

- **Sieve of Eratosthenes Algorithm**:
  - The algorithm initializes a boolean array is_prime marking all numbers as prime.

    - It sets 0 and 1 as non-prime.
    - For each number i from 2 to sqrt(n), if i is prime, it marks all multiples of i starting from i*i as non-prime.
    - Finally, it collects all numbers marked as prime into an array.
- **Serial Version (serial_sieve)**:
    - Implements the Sieve of Eratosthenes algorithm sequentially.
    - The outer loop (up to sqrt(n)) and inner loop (marking multiples) are executed by a single thread.
- **Parallel Version (parallel_sieve)**:
    - Uses OpenMP to parallelize both the initialization of the is_prime array and the inner loop of the Sieve algorithm.
    - #pragma omp parallel for is applied to:
        - The initialization loop (for (int i = 0; i <= n; i++)).
        - The inner loop marking multiples (for (int j = i * i; j <= n; j += i)), which is parallelized for each i.
    - The collection of primes into the primes array is kept sequential to avoid race conditions, as parallelizing this step would require additional synchronization (e.g., using a critical section), which might reduce performance gains.
- **Timing**:
    - omp_get_wtime() is used to measure the wall-clock time for both serial and parallel executions.
    - The difference in execution time (serial_time - parallel_time) is printed to compare performance.
- **Memory Management**:
    - Allocates arrays for both serial and parallel versions to ensure independent execution.
    - Frees all allocated memory to prevent leaks.

Notes on Performance:

- **Small n**: For small values of n (e.g., n = 100), the parallel version may be slower due to the overhead of thread creation and synchronization in OpenMP. This is evident in the first example output, where the parallel version takes slightly longer.
- **Large n**: For larger values of n (e.g., n = 1000000), the parallel version shows better performance as the computational work outweighs the thread overhead. The second example output demonstrates a speedup (serial time is 0.008214 seconds, parallel time is 0.003892 seconds).
- **Scalability**: The parallelization of the inner loop (j loop) provides good scalability for large n, as marking multiples is computationally intensive and benefits from parallel execution. However, the sequential collection of primes limits further speedup; this could be parallelized with additional synchronization if needed.
- **Optimization**: The Sieve algorithm is already efficient for finding primes, but further optimization could include:
    - Using a bit array instead of a boolean array to reduce memory usage.
    - Parallelizing the prime collection step with a reduction operation or critical section, though this might introduce overhead.

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

mpi_send_recv_demo.c

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

int main(int argc, char *argv[]) {
   int rank, size;
   int message = 42; // Message to be sent (can be modified)
   double start_time, end_time;

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

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

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

    // Ensure there are at least 2 processes
    if (size < 2) {
        if (rank == 0) {
            printf("This program requires at least 2 processes to run.\n");
        }
        MPI_Finalize();
        return 1;
    }

    // Record start time for communication
    start_time = MPI_Wtime();

    if (rank == 0) {
        // Process 0: Send the message to process 1
        printf("Process 0 sending message %d to process 1\n", message);
        MPI_Send(&message, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        // Process 1: Receive the message from process 0
        int received_message;
        MPI_Recv(&received_message, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process 1 received message %d from process 0\n", received_message);
    }

    // Record end time for communication
    end_time = MPI_Wtime();

    // Print the communication time on process 0
    if (rank == 0) {
        printf("Communication time: %f seconds\n", end_time - start_time);
    }

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

How to Compile and Run:

1. **Compile the Program**: Use an MPI-enabled compiler, such as mpicc (part of MPICH or OpenMPI). Ensure MPI is installed on your system.
   bash
   CollapseWrapRun
   Copy
   mpicc -o mpi_send_recv mpi_send_recv_demo.c
2. **Run the Program**: Use mpirun or mpiexec to execute the program with a specified number of processes. For this example, we need at least 2 processes:
   bash
   CollapseWrapRun
   Copy
   mpirun -np 2 ./mpi_send_recv
   - -np 2 specifies that the program should run with 2 processes.
   - ./mpi_send_recv is the compiled executable.

Example Output:
Running with 2 processes:
text

CollapseWrap
Copy
Process 0 sending message 42 to process 1
Process 1 received message 42 from process 0
Communication time: 0.000123 seconds

Explanation of the Code:

- **MPI Initialization and Setup**:
  - MPI_Init(&argc, &argv) initializes the MPI environment.
  - MPI_Comm_rank(MPI_COMM_WORLD, &rank) gets the rank (ID) of the current process.
  - MPI_Comm_size(MPI_COMM_WORLD, &size) gets the total number of processes.
  - The program checks if there are at least 2 processes; otherwise, it exits with an error message.
- **MPI Communication**:
  - **Process 0 (Sender)**:
    - Uses MPI_Send(&message, 1, MPI_INT, 1, 0, MPI_COMM_WORLD) to send the integer message to process 1.
    - Parameters:
      - &message: Address of the data to send.
      - 1: Number of elements to send (1 integer).
      - MPI_INT: Data type of the elements.
      - 1: Destination rank (process 1).
      - 0: Message tag (used to identify the message).
      - MPI_COMM_WORLD: Communicator (group of processes).
  - **Process 1 (Receiver)**:
    - Uses MPI_Recv(&received_message, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE) to receive the message from process 0.
    - Parameters:
      - &received_message: Buffer to store the received data.
      - 1: Number of elements to receive.
      - MPI_INT: Data type of the elements.
      - 0: Source rank (process 0).
      - 0: Message tag.
      - MPI_COMM_WORLD: Communicator.
      - MPI_STATUS_IGNORE: Ignores the status of the receive operation.
- **Timing**:
  - MPI_Wtime() is used to measure the wall-clock time for the communication.
  - The time difference (end_time - start_time) is printed by process 0 to show the communication latency.
- **MPI Finalization**:
  - MPI_Finalize() cleans up the MPI environment before the program exits.

Notes:

- **Communication Pattern**:
  - This program demonstrates a simple point-to-point communication where process 0 sends a single integer to process 1.
  - MPI_Send and MPI_Recv are blocking calls, meaning the sender waits until the message is sent, and the receiver waits until the message is received.
- **Scalability**:
  - The program is designed for exactly 2 processes to keep the example simple. To extend it for more processes, you could modify it to have process 0 send messages to multiple processes (e.g., using a loop) and have other processes receive messages accordingly.
- **Performance**:
  - The communication time depends on the system and network latency. For local processes on the same machine, the time will be very small (e.g., microseconds). On a distributed system, the time may increase due to network delays.
  - For small messages (like a single integer), the overhead of MPI setup might dominate the actual data transfer time.
- **Error Handling**:
  - The program checks for the minimum number of processes required (2).

    - Additional error handling (e.g., checking the return values of MPI_Send and MPI_Recv) could be added for robustness.
- **Extensions**:
    - To make the program more interesting, you could:
        - Send an array of integers instead of a single integer.
        - Implement a ping-pong pattern where process 1 sends a reply back to process 0.
        - Measure communication time for varying message sizes to analyze performance.

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

mpi_deadlock_demo.c

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

void demonstrate_deadlock(int rank) {
   int message_out = rank; // Message to send (process rank)
   int message_in;         // Buffer for received message

   printf("Process %d: Attempting to send message %d (Deadlock scenario)\n", rank, message_out);

   // Both processes send before receiving, causing a deadlock
   MPI_Send(&message_out, 1, MPI_INT, 1 - rank, 0, MPI_COMM_WORLD);
   MPI_Recv(&message_in, 1, MPI_INT, 1 - rank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

   printf("Process %d: Received message %d (This won't print in deadlock)\n", rank, message_in);
}

void avoid_deadlock(int rank) {
   int message_out = rank; // Message to send (process rank)
   int message_in;         // Buffer for received message

   printf("Process %d: Attempting communication (Deadlock avoidance scenario)\n", rank);

   // Alter the call sequence to avoid deadlock
   if (rank == 0) {
      // Process 0 receives first, then sends
      MPI_Recv(&message_in, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Send(&message_out, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
      printf("Process %d: Received message %d\n", rank, message_in);
   } else if (rank == 1) {
      // Process 1 sends first, then receives
      MPI_Send(&message_out, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
      MPI_Recv(&message_in, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Process %d: Received message %d\n", rank, message_in);
   }
}

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

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

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

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

```
    // Ensure exactly 2 processes are used
    if (size != 2) {
        if (rank == 0) {
            printf("This program requires exactly 2 processes to run.\n");
        }
        MPI_Finalize();
        return 1;
    }

    // Uncomment the following line to demonstrate deadlock (program will hang)
    // printf("=== Demonstrating Deadlock ===\n");
    // demonstrate_deadlock(rank);

    // Demonstrate deadlock avoidance
    printf("=== Demonstrating Deadlock Avoidance ===\n");
    avoid_deadlock(rank);

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

How to Compile and Run:

1. **Compile the Program**: Use an MPI-enabled compiler, such as mpicc (part of MPICH or OpenMPI). Ensure MPI is installed on your system.
   bash
   CollapseWrapRun
   Copy
   mpicc -o mpi_deadlock mpi_deadlock_demo.c
2. **Run the Program**: Use mpirun or mpiexec to execute the program with exactly 2 processes:
   bash
   CollapseWrapRun
   Copy
   mpirun -np 2 ./mpi_deadlock

Example Output:

Deadlock Scenario (Uncomment demonstrate_deadlock call):

If you uncomment the demonstrate_deadlock(rank) call and comment out the avoid_deadlock(rank) call, the program will hang:

text
CollapseWrap
Copy
=== Demonstrating Deadlock ===
Process 0: Attempting to send message 0 (Deadlock scenario)
Process 1: Attempting to send message 1 (Deadlock scenario)
[Program hangs here]

- **Explanation**: Both processes call MPI_Send before MPI_Recv. Since MPI_Send is blocking (in most implementations, unless buffering is available), each process waits for the other to receive its message, resulting in a deadlock.

Deadlock Avoidance Scenario (Default Execution):

With the avoid_deadlock(rank) call active:

text
CollapseWrap
Copy
=== Demonstrating Deadlock Avoidance ===
Process 0: Attempting communication (Deadlock avoidance scenario)
Process 1: Attempting communication (Deadlock avoidance scenario)
Process 0: Received message 1
Process 1: Received message 0

- **Explanation**: The deadlock is avoided by altering the call sequence:
    - Process 0 calls MPI_Recv first, waiting for a message from Process 1.
    - Process 1 calls MPI_Send first, sending its message to Process 0.
    - This ensures that Process 1's send matches Process 0's receive, allowing communication to proceed. Then, Process 0 sends its message, which Process 1 receives.

Explanation of the Code:

- **Deadlock Scenario (demonstrate_deadlock)**:
    - Both processes (rank 0 and rank 1) attempt to send a message to each other using MPI_Send before calling MPI_Recv.
    - Since MPI_Send is blocking (in standard mode, it waits until the message is received or buffered), both processes wait indefinitely for the other to receive, causing a deadlock.
    - The message each process sends is its rank (0 or 1), and the tag is 0.
- **Deadlock Avoidance Scenario (avoid_deadlock)**:
    - The call sequence is altered to break the deadlock cycle:
        - Process 0 calls MPI_Recv first, then MPI_Send.
        - Process 1 calls MPI_Send first, then MPI_Recv.
    - This ensures that when Process 1 sends, Process 0 is ready to receive, and vice versa, allowing communication to complete successfully.
    - The messages sent and received are the same as in the deadlock scenario (each process sends its rank).
- **MPI Functions Used**:
    - MPI_Send(&message_out, 1, MPI_INT, dest, tag, MPI_COMM_WORLD): Sends a message to the destination process.
    - MPI_Recv(&message_in, 1, MPI_INT, src, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE): Receives a message from the source process.
    - Parameters:
        - 1: Number of elements (1 integer).
        - MPI_INT: Data type of the elements.
        - dest/src: Destination/source rank (1 - rank, since rank 0 sends to 1, and rank 1 sends to 0).
        - tag: Message tag (0 in this case).
        - MPI_COMM_WORLD: Communicator.
        - MPI_STATUS_IGNORE: Ignores the status of the receive operation.
- **Program Structure**:
    - The program ensures exactly 2 processes are used, as the example is designed for a simple two-process deadlock scenario.
    - The deadlock demonstration is commented out by default to prevent the program from hanging during normal execution.
    - The deadlock avoidance scenario is executed by default to show a working solution.

Notes:

- **Deadlock Cause**:
    - Deadlock occurs because both processes use blocking sends (MPI_Send) before receiving. In MPI, a blocking send typically waits until the message is received by the destination or copied to a system buffer. If both processes send first, neither can proceed to the receive step, resulting in a deadlock.
    - The deadlock depends on the MPI implementation. Some implementations may buffer small messages, avoiding the deadlock, but this is not guaranteed (e.g., for large messages or in standard mode).
- **Deadlock Avoidance Strategy**:
    - The avoidance strategy ensures that at least one process is ready to receive when the other sends, breaking the cyclic dependency.
    - This is achieved by ordering the operations: Process 0 receives first, while Process 1 sends first. This ensures that the communication can proceed without waiting indefinitely.
- **Alternative Avoidance Strategies**:
    - Use non-blocking communication (MPI_Isend and MPI_Irecv) to initiate sends and receives without blocking, followed by MPI_Wait to ensure completion.
    - Use MPI_Sendrecv, a combined send-receive operation that avoids deadlock by handling both operations in a single call.

    - Introduce buffering by using MPI_Bsend (buffered send), though this requires setting up a buffer with MPI_Buffer_attach.
- **Performance**:
    - The program does not measure execution time for simplicity, but you can add timing using MPI_Wtime() (as in previous examples) to compare the deadlock avoidance scenario with other strategies.
- **Scalability**:
    - This example is designed for 2 processes to keep the deadlock scenario simple. For more processes, you could create a ring communication pattern where each process sends to the next and receives from the previous, which can also lead to deadlock if not managed properly.

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

mpi_broadcast_demo.c

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

int main(int argc, char *argv[]) {
   int rank, size;
   int message = 0; // Message to be broadcast
   double start_time, end_time;

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

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

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

   // Process 0 sets the message to broadcast
   if (rank == 0) {
      message = 42; // Value to broadcast (can be modified)
      printf("Process 0 broadcasting message: %d\n", message);
   }

   // Record start time for the broadcast
   start_time = MPI_Wtime();

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

   // Record end time for the broadcast
   end_time = MPI_Wtime();

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

   // Process 0 prints the broadcast time
   if (rank == 0) {
      printf("Broadcast time: %f seconds\n", end_time - start_time);
   }

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

How to Compile and Run:

1. **Compile the Program**: Use an MPI-enabled compiler, such as mpicc (part of MPICH or OpenMPI). Ensure MPI is installed on your system.

   mpicc -o mpi_broadcast mpi_broadcast_demo.c

2. **Run the Program**: Use mpirun or mpiexec to execute the program with a specified number of processes. For example, to run with 4 processes:

   mpirun -np 4 ./mpi_broadcast

Example Output:
Running with 4 processes:
text
CollapseWrap
Copy
Process 0 broadcasting message: 42
Process 0 received broadcast message: 42
Process 1 received broadcast message: 42
Process 2 received broadcast message: 42
Process 3 received broadcast message: 42
Broadcast time: 0.000089 seconds

Explanation of the Code:

- **MPI Initialization and Setup**:
  - MPI_Init(&argc, &argv) initializes the MPI environment.
  - MPI_Comm_rank(MPI_COMM_WORLD, &rank) gets the rank (ID) of the current process.
  - MPI_Comm_size(MPI_COMM_WORLD, &size) gets the total number of processes.
- **Broadcast Operation**:
  - Process 0 sets the message to 42 (this can be any integer value).
  - MPI_Bcast(&message, 1, MPI_INT, 0, MPI_COMM_WORLD) broadcasts the message from the root process (rank 0) to all processes in MPI_COMM_WORLD.
    - Parameters:
      - &message: Address of the data to broadcast (also the receive buffer for non-root processes).
      - 1: Number of elements to broadcast (1 integer).
      - MPI_INT: Data type of the elements.
      - 0: Rank of the root process (process 0).
      - MPI_COMM_WORLD: Communicator (group of processes).
  - After the broadcast, all processes (including the root) have the same value of message.
- **Timing**:
  - MPI_Wtime() is used to measure the wall-clock time for the broadcast operation.
  - The time difference (end_time - start_time) is printed by process 0 to show the broadcast latency.
- **Output**:
  - Each process prints the message it received from the broadcast.
  - Process 0 also prints the time taken for the broadcast operation.
- **MPI Finalization**:
  - MPI_Finalize() cleans up the MPI environment before the program exits.

Notes:

- **Broadcast Operation**:
  - MPI_Bcast is a collective operation, meaning all processes in the communicator must call it.
  - It ensures that the data from the root process is copied to all other processes.
  - In this example, the root process is rank 0, but any process can be the root by changing the root parameter in MPI_Bcast.
- **Performance**:
  - The broadcast time depends on the system, network latency, and the number of processes. For a small message (1 integer) on a local machine, the time is typically very small (e.g.,

microseconds). On a distributed system with many processes, the time may increase due to network communication.
    - The efficiency of MPI_Bcast depends on the MPI implementation, which often uses optimized algorithms (e.g., tree-based broadcasting) for large-scale systems.
- **Scalability**:
    - The program works with any number of processes (at least 1). Increasing the number of processes will increase the broadcast time, especially in a distributed environment.
    - For large messages or many processes, you might observe more significant performance differences.
- **Extensions**:
    - To make the program more interesting, you could:
        - Broadcast an array instead of a single integer (e.g., int message[100]).
        - Measure the broadcast time for varying message sizes to analyze performance.
        - Compare MPI_Bcast with a manual implementation using MPI_Send and MPI_Recv in a loop (though this would be less efficient).

8. Write a MPI Program demonstration of MPI_Scatter and MPI_Gather

mpi_scatter_gather_demo.c

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

int main(int argc, char *argv[]) {
   int rank, size;
   int *send_buffer = NULL;   // Buffer for data to scatter (used by root)
   int *recv_buffer = NULL;   // Buffer for data to gather (used by root)
   int local_data;            // Local data for each process after scatter
   double start_time, end_time;

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

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

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

   // Allocate buffers on the root process (rank 0)
   if (rank == 0) {
      send_buffer = (int *)malloc(size * sizeof(int));
      recv_buffer = (int *)malloc(size * sizeof(int));
      if (send_buffer == NULL || recv_buffer == NULL) {
         printf("Memory allocation failed on root process.\n");
         MPI_Finalize();
         return 1;
      }

      // Initialize the send buffer with data (e.g., 0, 1, 2, ..., size-1)
      for (int i = 0; i < size; i++) {
         send_buffer[i] = i;
      }

      printf("Root process (0) scattering data: ");
      for (int i = 0; i < size; i++) {
         printf("%d ", send_buffer[i]);
      }
      printf("\n");
   }
```

```
    // Record start time for scatter and gather operations
    start_time = MPI_Wtime();

    // Scatter the data: each process gets one element from send_buffer
    MPI_Scatter(send_buffer, 1, MPI_INT, &local_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Each process modifies its local data (e.g., doubles it)
    local_data *= 2;
    printf("Process %d: Received %d, Modified to %d\n", rank, local_data / 2, local_data);

    // Gather the modified data back to the root process
    MPI_Gather(&local_data, 1, MPI_INT, recv_buffer, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // Record end time
    end_time = MPI_Wtime();

    // Root process prints the gathered data
    if (rank == 0) {
        printf("Root process (0) gathered data: ");
        for (int i = 0; i < size; i++) {
            printf("%d ", recv_buffer[i]);
        }
        printf("\nScatter and Gather time: %f seconds\n", end_time - start_time);

        // Free allocated memory
        free(send_buffer);
        free(recv_buffer);
    }

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

How to Compile and Run:

1. **Compile the Program**: Use an MPI-enabled compiler, such as mpicc (part of MPICH or OpenMPI). Ensure MPI is installed on your system.

   ```
   mpicc -o mpi_scatter_gather mpi_scatter_gather_demo.c
   ```

2. **Run the Program**: Use mpirun or mpiexec to execute the program with a specified number of processes. For example, to run with 4 processes:

   ```
   mpirun -np 4 ./mpi_scatter_gather
   ```

Example Output:
Running with 4 processes:

```
Root process (0) scattering data: 0 1 2 3
Process 0: Received 0, Modified to 0
Process 1: Received 1, Modified to 2
Process 2: Received 2, Modified to 4
Process 3: Received 3, Modified to 6
Root process (0) gathered data: 0 2 4 6
Scatter and Gather time: 0.000145 seconds
```

Explanation of the Code:

- **MPI Initialization and Setup**:
  - MPI_Init(&argc, &argv) initializes the MPI environment.

    - MPI_Comm_rank(MPI_COMM_WORLD, &rank) gets the rank (ID) of the current process.
    - MPI_Comm_size(MPI_COMM_WORLD, &size) gets the total number of processes.
- **Data Preparation**:
    - The root process (rank 0) allocates two buffers:
        - send_buffer: Holds the data to be scattered (size = number of processes).
        - recv_buffer: Will hold the gathered data after processing.
    - The send_buffer is initialized with values 0, 1, 2, ..., size-1.
- **MPI_Scatter**:
    - MPI_Scatter(send_buffer, 1, MPI_INT, &local_data, 1, MPI_INT, 0, MPI_COMM_WORLD) scatters the data from the root process to all processes.
        - Parameters:
            - send_buffer: Data to scatter (on root).
            - 1: Number of elements to send to each process.
            - MPI_INT: Data type of the elements.
            - &local_data: Buffer to receive the scattered data (on all processes).
            - 1: Number of elements to receive.
            - MPI_INT: Data type of the received elements.
            - 0: Rank of the root process.
            - MPI_COMM_WORLD: Communicator.
    - Each process receives one integer from the send_buffer. For example, with 4 processes, process 0 gets 0, process 1 gets 1, process 2 gets 2, and process 3 gets 3.
- **Data Modification**:
    - Each process doubles the value it received (e.g., process 1 receives 1, modifies it to 2).
    - This step simulates a computation that each process might perform on its portion of the data.
- **MPI_Gather**:
    - MPI_Gather(&local_data, 1, MPI_INT, recv_buffer, 1, MPI_INT, 0, MPI_COMM_WORLD) gathers the modified data from all processes back to the root.
        - Parameters:
            - &local_data: Data to send from each process.
            - 1: Number of elements to send.
            - MPI_INT: Data type of the elements.
            - recv_buffer: Buffer to receive the gathered data (on root).
            - 1: Number of elements to receive per process.
            - MPI_INT: Data type of the received elements.
            - 0: Rank of the root process.
            - MPI_COMM_WORLD: Communicator.
    - The root process collects the modified values into recv_buffer. For example, with 4 processes, recv_buffer will contain [0, 2, 4, 6].
- **Timing**:
    - MPI_Wtime() measures the wall-clock time for the scatter and gather operations combined.
    - The time difference (end_time - start_time) is printed by the root process.
- **Output**:
    - The root process prints the original data before scattering and the gathered data after processing.
    - Each process prints the data it received and the modified value.
    - The root process prints the total time for the scatter and gather operations.
- **MPI Finalization**:
    - MPI_Finalize() cleans up the MPI environment before the program exits.

Notes:

- **Scatter and Gather Operations**:
    - MPI_Scatter distributes data evenly from the root to all processes. In this example, each process receives exactly 1 integer, but you can scatter larger chunks by adjusting the sendcount and recvcount parameters.
    - MPI_Gather collects data from all processes back to the root. The root process must allocate enough space in recv_buffer to hold the data from all processes (size * recvcount elements).
- **Performance**:
    - The scatter and gather time depends on the system, network latency, and the number of processes. For a small message (1 integer per process) on a local machine, the time is very

small (e.g., microseconds). In a distributed environment with many processes, the time may increase.
    - The MPI implementation typically uses optimized algorithms (e.g., tree-based or linear distribution) for these collective operations.
- **Scalability**:
    - The program works with any number of processes. The send_buffer and recv_buffer sizes scale with the number of processes (size).
    - For large datasets or many processes, the scatter and gather operations may become a bottleneck, especially in a distributed system.
- **Extensions**:
    - To make the program more interesting, you could:
        - Scatter and gather an array of integers for each process (e.g., each process receives 10 integers).
        - Perform a more complex computation on the scattered data (e.g., summing an array, matrix operations).
        - Measure the scatter and gather times separately to analyze their individual performance.

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

mpi_reduce_allreduce_demo.c

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

int main(int argc, char *argv[]) {
   int rank, size;
   int local_value;      // Value contributed by each process (rank + 1)
   int reduce_result;    // Result of MPI_Reduce (stored on root)
   int allreduce_result; // Result of MPI_Allreduce (stored on all processes)
   double start_time, end_time;

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

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

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

   // Each process contributes its rank + 1 as the local value
   local_value = rank + 1;
   printf("Process %d: Local value = %d\n", rank, local_value);

   // Record start time for reduction operations
   start_time = MPI_Wtime();

   // --- MPI_Reduce Demonstrations ---
   if (rank == 0) {
      printf("\n=== MPI_Reduce Results (Root Process Only) ===\n");
   }

   // MPI_Reduce with MPI_MAX
   MPI_Reduce(&local_value, &reduce_result, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
   if (rank == 0) {
      printf("MPI_Reduce (MPI_MAX): %d\n", reduce_result);
   }
```

```
    // MPI_Reduce with MPI_MIN
    MPI_Reduce(&local_value, &reduce_result, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("MPI_Reduce (MPI_MIN): %d\n", reduce_result);
    }

    // MPI_Reduce with MPI_SUM
    MPI_Reduce(&local_value, &reduce_result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("MPI_Reduce (MPI_SUM): %d\n", reduce_result);
    }

    // MPI_Reduce with MPI_PROD
    MPI_Reduce(&local_value, &reduce_result, 1, MPI_INT, MPI_PROD, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("MPI_Reduce (MPI_PROD): %d\n", reduce_result);
    }

    // --- MPI_Allreduce Demonstrations ---
    printf("\n=== MPI_Allreduce Results (All Processes) ===\n");

    // MPI_Allreduce with MPI_MAX
    MPI_Allreduce(&local_value, &allreduce_result, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    printf("Process %d: MPI_Allreduce (MPI_MAX): %d\n", rank, allreduce_result);

    // MPI_Allreduce with MPI_MIN
    MPI_Allreduce(&local_value, &allreduce_result, 1, MPI_INT, MPI_MIN, MPI_COMM_WORLD);
    printf("Process %d: MPI_Allreduce (MPI_MIN): %d\n", rank, allreduce_result);

    // MPI_Allreduce with MPI_SUM
    MPI_Allreduce(&local_value, &allreduce_result, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    printf("Process %d: MPI_Allreduce (MPI_SUM): %d\n", rank, allreduce_result);

    // MPI_Allreduce with MPI_PROD
    MPI_Allreduce(&local_value, &allreduce_result, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);
    printf("Process %d: MPI_Allreduce (MPI_PROD): %d\n", rank, allreduce_result);

    // Record end time
    end_time = MPI_Wtime();

    // Root process prints the total time for reductions
    if (rank == 0) {
        printf("\nTotal time for reductions: %f seconds\n", end_time - start_time);
    }

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

How to Compile and Run:

1. **Compile the Program**: Use an MPI-enabled compiler, such as mpicc (part of MPICH or OpenMPI). Ensure MPI is installed on your system.
   bash
   CollapseWrapRun
   Copy
   mpicc -o mpi_reduce_allreduce mpi_reduce_allreduce_demo.c
2. **Run the Program**: Use mpirun or mpiexec to execute the program with a specified number of processes. For example, to run with 4 processes:
   bash
   CollapseWrapRun

Copy
mpirun -np 4 ./mpi_reduce_allreduce

Example Output:
Running with 4 processes:

Process 0: Local value = 1
Process 1: Local value = 2
Process 2: Local value = 3
Process 3: Local value = 4

=== MPI_Reduce Results (Root Process Only) ===
MPI_Reduce (MPI_MAX): 4
MPI_Reduce (MPI_MIN): 1
MPI_Reduce (MPI_SUM): 10
MPI_Reduce (MPI_PROD): 24

=== MPI_Allreduce Results (All Processes) ===
Process 0: MPI_Allreduce (MPI_MAX): 4
Process 1: MPI_Allreduce (MPI_MAX): 4
Process 2: MPI_Allreduce (MPI_MAX): 4
Process 3: MPI_Allreduce (MPI_MAX): 4
Process 0: MPI_Allreduce (MPI_MIN): 1
Process 1: MPI_Allreduce (MPI_MIN): 1
Process 2: MPI_Allreduce (MPI_MIN): 1
Process 3: MPI_Allreduce (MPI_MIN): 1
Process 0: MPI_Allreduce (MPI_SUM): 10
Process 1: MPI_Allreduce (MPI_SUM): 10
Process 2: MPI_Allreduce (MPI_SUM): 10
Process 3: MPI_Allreduce (MPI_SUM): 10
Process 0: MPI_Allreduce (MPI_PROD): 24
Process 1: MPI_Allreduce (MPI_PROD): 24
Process 2: MPI_Allreduce (MPI_PROD): 24
Process 3: MPI_Allreduce (MPI_PROD): 24

Total time for reductions: 0.000214 seconds

Explanation of the Code:

- **MPI Initialization and Setup**:
  - MPI_Init(&argc, &argv) initializes the MPI environment.
  - MPI_Comm_rank(MPI_COMM_WORLD, &rank) gets the rank (ID) of the current process.
  - MPI_Comm_size(MPI_COMM_WORLD, &size) gets the total number of processes.
- **Local Value**:
  - Each process sets its local_value to rank + 1. For 4 processes, the values are 1, 2, 3, and 4 for ranks 0, 1, 2, and 3, respectively.
- **MPI_Reduce**:
  - MPI_Reduce(&local_value, &reduce_result, 1, MPI_INT, op, 0, MPI_COMM_WORLD) reduces the local_value from all processes to a single value on the root process (rank 0).
    - Parameters:
      - &local_value: Input data from each process.
      - &reduce_result: Output buffer for the result (on root).
      - 1: Number of elements to reduce.
      - MPI_INT: Data type of the elements.
      - op: Reduction operation (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD).
      - 0: Rank of the root process.
      - MPI_COMM_WORLD: Communicator.
  - The program performs four reductions:
    - MPI_MAX: Finds the maximum value (4).
    - MPI_MIN: Finds the minimum value (1).
    - MPI_SUM: Computes the sum (1 + 2 + 3 + 4 = 10).

        - MPI_PROD: Computes the product (1 * 2 * 3 * 4 = 24).
    - Results are printed only by the root process.
- **MPI_Allreduce**:
    - MPI_Allreduce(&local_value, &allreduce_result, 1, MPI_INT, op, MPI_COMM_WORLD) reduces the local_value from all processes and distributes the result to all processes.
        - Parameters:
            - &local_value: Input data from each process.
            - &allreduce_result: Output buffer for the result (on all processes).
            - 1: Number of elements to reduce.
            - MPI_INT: Data type of the elements.
            - op: Reduction operation (MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD).
            - MPI_COMM_WORLD: Communicator.
    - The same four reductions are performed as with MPI_Reduce, but the result is available to all processes, which print their results.
- **Timing**:
    - MPI_Wtime() measures the wall-clock time for all reduction operations combined.
    - The time difference (end_time - start_time) is printed by the root process.
- **Output**:
    - Each process prints its local value.
    - The root process prints the results of MPI_Reduce.
    - All processes print the results of MPI_Allreduce.
    - The root process prints the total time for all reductions.
- **MPI Finalization**:
    - MPI_Finalize() cleans up the MPI environment before the program exits.

Notes:

- **MPI_Reduce vs. MPI_Allreduce**:
    - MPI_Reduce collects the result only on the root process, making it suitable when only one process needs the result (e.g., for final output or decision-making).
    - MPI_Allreduce distributes the result to all processes, which is useful when all processes need the result for further computation (e.g., in iterative algorithms).
- **Reduction Operations**:
    - MPI_MAX: Returns the maximum value across all processes.
    - MPI_MIN: Returns the minimum value.
    - MPI_SUM: Returns the sum of values.
    - MPI_PROD: Returns the product of values.
    - These operations are commutative and associative, as required by MPI for reduction operations.
- **Performance**:
    - The reduction time depends on the system, network latency, and the number of processes. For a small dataset (1 integer per process) on a local machine, the time is very small (e.g., microseconds). In a distributed environment with many processes, the time may increase.
    - MPI implementations typically use optimized algorithms (e.g., tree-based or butterfly reduction) for these collective operations.
- **Scalability**:
    - The program works with any number of processes. The reduction operations scale logarithmically or linearly with the number of processes, depending on the MPI implementation.
    - For large datasets or many processes, the reduction operations may become a bottleneck, especially in a distributed system.
- **Extensions**:
    - To make the program more interesting, you could:
        - Reduce an array of values instead of a single integer (e.g., int local_values[10]).
        - Measure the time for each reduction operation separately to compare their performance.
        - Use other reduction operations like MPI_LAND (logical AND), MPI_LOR (logical OR), or user-defined operations.

COURSE OBJECTIVES: This course will enable to,

- Explore the need for parallel programming
- Explain how to parallelize on MIMD systems
- To demonstrate how to apply MPI library and parallelize the suitable programs
- To demonstrate how to apply OpenMP pragma and directives to parallelize the suitable programs
- To demonstrate how to design CUDA program

MODULE-1: Introduction to parallel programming, Parallel hardware and parallel software –

CLASSIFICATIONS OF PARALLEL COMPUTERS,

The **Introduction to Parallel Computing** slide (Slide 2) summarizes the importance of parallel systems in modern computing, such as multicore processors and clusters, and highlights Flynn's Taxonomy as the primary classification method, accompanied by a diagram of a parallel system. **Slide 3** provides an overview of Flynn's Taxonomy, detailing its four categories—SISD, SIMD, MISD, and MIMD—based on whether a system processes single or multiple instruction and data streams. A table or diagram visually distinguishes these categories.

Subsequent slides delve into each category. **Slide 4** describes **SISD (Single Instruction, Single Data)**, representing traditional serial computers like early von Neumann architectures, with a diagram of a single CPU processing one instruction and data stream, emphasizing its lack of parallelism. **Slide 5** covers **SIMD (Single Instruction, Multiple Data)**, which applies a single instruction to multiple data streams, ideal for data-parallel tasks like image processing, with a diagram illustrating GPU cores executing the same instruction on different data. **Slide 6** addresses **MISD (Multiple Instruction, Single Data)**, a less common category used in specialized applications like fault-tolerant systems, depicted with a diagram of multiple instructions processing a single data stream. **Slide 7** focuses on **MIMD (Multiple Instruction, Multiple Data)**, the most versatile and prevalent in modern systems like multicore CPUs and clusters, shown with a diagram of independent processors handling distinct instructions and data.

Slide 8 explores MIMD subclassifications—shared-memory and distributed-memory systems. Shared-memory MIMD, where processors access a common memory space (e.g., multicore processors), is contrasted with distributed-memory MIMD, where each processor has its own memory (e.g., clusters), using a comparative diagram to highlight programming and scalability differences. **Slide 9** introduces other classification approaches, such as **SPMD (Single Program, Multiple Data)**, common in MPI programs, and hybrid systems combining CPU and GPU architectures, illustrated with a diagram of SPMD or hybrid workflows. **Slide 10** discusses **Practical Implications**, linking architecture types to programming models (e.g., CUDA for SIMD, MPI/OpenMP for MIMD) and using a flowchart to guide hardware and API selection. **Slide 11**, the **Conclusion**, recaps Flynn's Taxonomy, emphasizes MIMD's dominance, and suggests further study topics like interconnection networks, paired with a summary graphic.

SIMD systems,

outlines Flynn's Taxonomy, a foundational framework for classifying parallel computers, with SIMD highlighted as a key category where a single instruction is executed simultaneously on multiple data streams. The response leverages this content to propose an 11-slide PowerPoint presentation, meticulously structured to explain SIMD systems for a technical audience familiar with basic computing but new to parallel architectures. The presentation, wrapped in an <xaiArtifact> tag with a unique UUID, is titled "SIMD_Systems_Presentation.pptx" and formatted as a Markdown artifact, detailing each slide's title, key points, explanations, suggested visuals, and presenter notes to ensure clarity and engagement.

The presentation begins with a **Title Slide** (Slide 1), introducing the topic of SIMD systems, crediting the textbook authors, and featuring a visual of a GPU or vector processor array to evoke parallel processing. **Slide 2**, "Introduction to SIMD," defines SIMD within Flynn's Taxonomy, emphasizing its role in data-parallel computing and its suitability for uniform operations on large datasets, accompanied by a diagram of one instruction applied to multiple data elements. **Slide 3**, "Characteristics of SIMD Systems," explains the architecture, where a single control unit broadcasts instructions to synchronized processing elements (PEs), each handling different data, illustrated by a schematic of a control unit connected to PEs. **Slide 4**, "Applications of SIMD," highlights practical uses such as image/video processing, scientific simulations, and machine learning, with a before/after image processing example to make the concept tangible. **Slide 5**, "SIMD Hardware Examples," covers vector processors (e.g., Cray-1), GPUs (e.g., NVIDIA CUDA cores), and CPU SIMD extensions (e.g., SSE/AVX), using a comparative diagram of vector processor and GPU architectures.

Slide 6, "Programming SIMD Systems," discusses programming interfaces like CUDA, OpenCL, and SIMD intrinsics, noting challenges like data alignment and thread divergence, with a concise CUDA kernel snippet for vector addition as a visual aid. **Slide 7**, "Advantages of SIMD," outlines benefits such as high throughput, energy efficiency, and scalability for data-intensive tasks, supported by a performance graph comparing GPU and CPU matrix multiplication. **Slide 8**, "Limitations of SIMD," addresses inflexibility for irregular tasks, efficiency losses due to divergence, and the need for specialized expertise, depicted with a diagram of stalled threads due to conditionals. **Slide 9**, "SIMD vs. Other Architectures," compares SIMD to SISD (sequential) and MIMD (flexible, multi-instruction), using a table to contrast flexibility, parallelism, and use cases. **Slide 10**, "Conclusion," recaps SIMD's critical role in modern computing, its prominence in GPUs and AI, and suggests further exploration of CUDA and GPU advancements, paired with a futuristic neural network graphic.

MODULE-1: Introduction to parallel programming, Parallel hardware and parallel software –

Classifications of Parallel Computers,
MIMD systems,
Interconnection networks,
Cache coherence,
Shared-memory vs. distributed-memory,
Coordinating the processes/threads,
Shared-memory,
Distributed-memory.

MODULE-2: GPU programming, Programming hybrid systems, MIMD systems, GPUs, Performance

Speedup and efficiency in MIMD systems,
Amdahl's law,
Scalability in MIMD systems,
Taking timings of MIMD programs,
GPU performance.

MODULE-3 : Distributed memory programming with MPI –

MPI functions,
The trapezoidal rule in MPI,
Dealing with I/O,
Collective communication,
MPI-derived datatypes,
Performance evaluation of MPI programs,
A parallel sorting algorithm.

MODULE-4: Shared-memory programming with OpenMP –

openmp pragmas and directives,
The trapezoidal rule,
Scope of variables,
The reduction clause,
loop carried dependency,
scheduling,
producers and consumers,
Caches,
cache coherence and false sharing in openmp,
tasking,
thread safety.

MODULE-5 : GPU programming with CUDA

GPUs and GPGPU,
GPU architectures,
Heterogeneous computing,
Threads,
blocks, and grids
Nvidia compute capabilities and device architectures,
Vector addition,
Returning results from CUDA kernels,
CUDA trapezoidal rule I,
CUDA trapezoidal rule II: improving performance,
CUDA trapezoidal rule III: blocks with more than one warp.

PRACTICAL COMPONENT OF IPCC: Experiments

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

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

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

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.

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

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

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

8. Write a MPI Program demonstration of MPI_Scatter and MPI_Gather

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