VISVESVARAYA TECHNOLOGICAL UNIVERSITY

JNANA SANGAMA, BELGAVI-590018, KARNATAKA

A J INSTITUTE OF ENGINEERING & TECHNOLOGY

(A unit of Laxmi Memorial Education Trust. (R))
NH - 66, Kottara Chowki, Kodical Cross, Mangalore- 575 006.

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

MASTER MANUAL

Course: Parallel Computing

(SubjectCode:BIS702)

VII-SEMESTER

Prepared by:
Dr. Lokesh M R
Professor
Department of Information Science & Engineering, AJIET, Mangalore

ACADEMIC YEAR: 2025-26

VISVESVARAYATECHNOLOGICALUNIVERSITY,BELAGAVI

B.E. in Information Science and Engineering
Scheme of Teaching and Examinations 2022

Outcome-Based Education (OBE) and Choice Based Credit System (CBCS)

(Effective from the academic year 2022 - 23)

Course: Parallel Computing

Course Code: BIS702

TABLE OF CONTENTS

VISION OF THE INSTITUTE

"To produce top-quality engineers who are groomed for attaining excellence in their profession and competitive enough to help in the growth of nation and global society."

MISSION OF THE INSTITUTE

M1: To offer affordable high-quality graduate program in engineering with value education and make the students socially responsible.

M2: To support and enhance the institutional environment to attain research excellence in both faculty and students and to inspire them to push the boundaries of knowledge base.

M3: To identify the common areas of interest amongst the individuals for the effective industry- institute partnership in a sustainable way by systematically working together.

M4: To promote the entrepreneurial attitude and inculcate innovative ideas among the engineering professionals.

VISION OF THE DEPARTMENT

"To be a center of excellence in Information Science & Engineering education, research and training to meet the growing needs of the industry and society."

MISSION OF THE DEPARTMENT

M1: To impart theoretical and practical knowledge through the concepts and technologies in Information Science and Engineering.

M2: To foster research, collaboration and higher education with premier institutions and industries.

M3: Promote innovation and entrepreneurship to fulfill the needs of the society and industry.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

After 4 years of graduation, graduates will be able to

PEO1: Analyse, design and implement solutions to the real-world problems in the field of Information Science and Engineering with multidisciplinary setup

PEO2: Keep abreast with the technology, innovation and pursue higher education with high standards of social and professional ethics

PEO3: Develop professional and entrepreneurship skills to work effectively as an individual and in a team to meet the ever-changing goals of the organization

PROGRAM OUTCOMES (POs)

PO1: Engineering Knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: Problem Analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3: Design/Development of Solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4: Conduct Investigations of Complex Problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5: Modern Tool Usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO6: The Engineer and Society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7: Environment and Sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

PO9: Individual and Team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11: Project Management and Finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-Long Learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSOs)

At the end of the program, graduates will be able to

PSO1: Design, implement and maintain the information systems that fulfill the current needs of the industry.

PSO2: Apply computational theory, storage, and networking concepts to address the societal problems.

GENERAL LAB GUIDELINES

Do's

1. Maintain discipline in the Laboratory.
2. Before entering the Laboratory, keep the footwear on the shoe rack.
3. Proper dress code has to be maintained while entering the Laboratory.
4. Students should carry a lab observation book, student manual and record book completed in all aspects.
5. Read and understand the logic of the program thoroughly before coming to the laboratory.
6. Enter the login book before switching on the computer.
7. Enter your batch member names and other details in the slips for hardware kits.
8. Students should be at their concerned places; unnecessary movement is restricted.
9. Students should maintain the same computer until the end of the semester.
10. Report any problems in computers/hardware kits to the faculty member in-charge/laboratory technician immediately.
11. The practical result should be noted down into their observation and the result must be shown to the faculty member in-charge for verification.
12. After completing the experiments, students should switch off the computers, enter logout time, return the hardware kits and keep the chairs properly.

Don'ts

1. Do not come late to the Laboratory.
2. Do not enter the laboratory without an ID card, lab dress code, observation book and record.
3. Do not leave the laboratory without the permission of the faculty in-charge.
4. Never eat, drink while working in the laboratory.
5. Do not handle any equipment before reading the instructions/instruction manuals.
6. Do not exchange the computers with others and hardware kits also.
7. Do not misbehave in the laboratory.
8. Do not alter computer settings/software settings.
9. External Disk/drives should not be connected to computers without permission, doing so will attract fines.
10. Do not remove anything from the kits/experimental set up without permission. Doing so will attract fines.
11. Do not mishandle the equipment / Computers.
12. Do not leave the laboratory without verification of hardware kits by the lab instructor.
13. Usage of Mobile phones, tablets and other portable devices are not allowed in restricted places.

INSTRUCTIONS TO STUDENTS

- Students must bring Observation book, record and manual along with pen, pencil, and eraser etc., no borrowing from others.
- Students must handle the trainer kit and other components carefully, as they are expensive.
- Before switch on the trainer kit, must show the connections to one of the faculties or instructors.
- After the completion of the experiment should return the components to the respective lab instructors.
- Before leaving the lab, should check whether they have switch off the power supplies and keep their chairs properly.
- Be regular to the Lab Do not come late to the Lab
- Do not throw connecting wires on the Floor
- Wear your College ID card Do not operate the IC trainer kits without permission
- Avoid unnecessary talking while doing the experiment
- Avoid loose connection and short circuits
- Take the signature of the lab in charge before taking the components
- Do not interchange the ICs while doing the experiment
- Handle the trainer kit properly
- Do not panic if you do not get the output
- Keep your work area clean after completing the experiment.
- After completion of the experiment switch off the power and return the components Arrange your chairs and tables before leaving.

RULES FOR MAINTAINING LABORATORY RECORD

- Put your name, USN and subject on the outside front cover of the record. Put that same information on the first page inside.
- Update Table of Contents every time you start each new experiment or topic
- Always use pen and write neatly and clearly
- Start each new topic (experiment, notes, calculation, etc.) on a right-side (odd numbered) page
- Obvious care should be taken to make it readable, even if you have bad handwriting
- Date to be written every page on the top right side corner
- On each right-side page
  - Title of experiment
  - Aim/Objectives
  - Components Required
  - Theory
  - Procedure described clearly in steps
  - Result
- On each left side page
  - Pin diagrams
  - Circuit diagram
  - Tables
  - Graphs
- Use labels and captions for figures and tables
- Attach printouts and plots of data as needed. Stick printouts (A4 Size) on the right side of the lab record

Strictly observe the instructions given by the Teacher/ Lab Instructor

SYLLABUS

Course Code	BIS702	Semester	VII
Teaching Hours/ Week (L:T:P:S)	3:0:2:0	CIE MARKS	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	SEE MARKS	50
Credits	04	Total Marks	100
Examination nature (SEE)	Theory/Practical	Exam Hours	03

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

Reference Books:

1. Calvin Lin, Lawrence Snyder – Principles of Parallel Programming, Pearson
2. Barbara Chapman – Using OpenMP: Portable Shared Memory Parallel Programming, Scientific and Engineering Computation
3. William Gropp, Ewing Lusk – Using MPI:Portable Parallel Programing, Third edition, Scientific and Engineering Computation

COURSE OUTCOMES

Course outcomes (Course Skill Set):

At the end of the course, the student will be able to:

- Explain the need for parallel programming
- Demonstrate parallelism in MIMD system.
- Apply MPI library to parallelize the code to solve the given problem.
- Apply OpenMP pragma and directives to parallelize the code to solve the given problem
- Design a CUDA program for the given problem.

Course Outcomes (COs)	Program Outcomes (POs)												Program Specific Outcomes (PSOs)	
	1	2	3	4	5	6	7	8	9	10	11	12	1	2
CO1	2	2	-	-	-	-	-	-	-	-	-	-	2	2
CO2	1	2	3	-	-	-	-	-	-	-	-	-	2	2
CO3	1	2	2	-	2	-	-	-	-	-	-	-	2	2
CO4	1	2	2	-	3	-	-	-	-	-	-	-	1	1
CO5	1	2	3	-	3	-	-	-	-	-	-	-	1	1
Average	1	2	3	-	3	-	-	-	-	-	-	-	2	2

ASSESSMENT DETAILS (BOTH CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

CIE for the practical component of the IPCC

- 15 marks for the conduction of the experiment and preparation of laboratory record, and 10 marks for the test to be conducted after the completion of all the laboratory sessions.
- On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day
- The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to 15 marks.
- The laboratory test (duration 02/03 hours) after completion of all the experiments shall be conducted for 50 marks and scaled down to 10 marks.
- Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for 25 marks.
- The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

SEE for IPCC

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course (duration 03 hours)

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), should have a mix of topics under that module.
3. The students have to answer 5 full questions, selecting one full question from each module.
4. Marks scored by the student shall be proportionally scaled down to 50 Marks

The theory portion of the IPCC shall be for both CIE and SEE, whereas the practical portion will have a CIE component only. Questions mentioned in the SEE paper may include questions from the practical component.

RUBRICS FOR PRACTICAL SESSIONS

Parallel Computing **BIS702**

Practical Sessions- Continuous Evaluation (CE) (15marks)				
Evaluation Parameter	**Level of Achievement**			
#R1: Observation/ Conduction **(07 Marks)**	**Excellent (07)**	**Good (06-05)**	**Average (05-04)**	**Poor (04-0)**
	Student has written properly With perfect logic and code snippets	Student is able to document properly but has made minor mistakes in snippet	Student has not studied properly and presentation of psudocode and code snippets are inaccurate	No correct information is present
#R2: Record **(5 Marks)**	**Excellent (05)**	**Good (04-03)**	**Average (03-02)**	**Poor (01-0)**
	Student has completed the record with correct results and indexes maintained	Student has completed the record but documented partial results and indexes	Student has several details pending and not submitted record on time	Student has several details pending and not submitted record on time
#R3: Performance, Viva (03 Marks)	**Excellent, Good (03)**	**Average (02-01)**	**Poor (01-0)**	
	Student has correctly gained knowledge and presented	Student has not understood the concept and average presentation	Student has not presented correctly	

RUBRICS FOR PRACTICAL SESSIONS

Practical Sessions- Practical Test(10marks)				
#R1: **Write-Up (03 Marks)**	**Excellent (03)**	**Good (02)**	**Average (01)**	**Poor (0)**
	Program neatly written. Handwriting is clear. Programs written with no mistakes.	Program neatly written. Handwriting is clear. Programs written with very few mistakes.	Program is written in unclear manner. Handwriting is not very clear. Programs written with fewer mistakes.	Program is written in unclear manner. Handwriting is not clear, Programs written with lot of mistakes.
#R2: Conduction/ Execution (04 Marks)	**Excellent (04)**	**Good (03)**	**Average(`02)**	**Poor (01-0)**
	Execution of the program done as per the procedure. Programs had less than 10 errors. The errors were debugged without any help. The Result was tabulated for all the cases.	Execution of the program done as per the procedure. Programs had less than 20 errors. The errors were debugged with a little help. The Result was tabulated for almost all the cases	Execution of the program done as per the procedure. Programs had more than 20 errors. The errors were debugged with the help of instructor. The Result was tabulated for few of the cases	Execution of the program was not done as per the procedure. Programs was full of syntax and logical error. The errors were resolved by the instructor. The Result was tabulated only for 1 or 2 Cases
#R3: **Viva (03 Marks)**	**Excellent (03)**	**Good (02)**	**Average (01)**	**Poor (0)**
	Answered all questions with elaboration has excellent understanding of the topic.	Answered most of the questions Failed to elaborate some of the concepts	Answered a few questions. Subject knowledge is not adequate	Not able to answer any of the questions. Subject knowledge not adequate

LIST OF MAJOR EQUIPMENT

Name Of the Laboratory: DATA PROCESSING LABORATORY

Sl. No	Equipment	Specification	Quantity
1	Desktops Computers	Intel core i3-6100/4170, 4.00/8.00GB RAM, 1TB/500GB HDD, 64-bit Operating System, 18.5" Acer Monitor, Acer Keyboard & Mouse.	35
2	UPS	20 KVA	1
3	Switches	24 Port Giga	2
4	Internet	100mbps	1
5	Projector	EPSON Projector with LCD Screen HDMI Port	1
6	Softwares Installed	Oracle 12c, Apache Tomcat Server , MySQL, Gcc Compiler, Turbo C++, VS Code, Jupyter Notebook, Eclipse, Apache Netbeans, Pycharm,Xamp Server	

Room Number : **A-308**

Total Area of the laboratory : 115 Sq. Meters

Total Amount Spent : Rs. 9,55,688.00 /-

Name of the HOD : Dr. John Prakash Veigas

Name of the lab in charge : Mrs. R Sahaya Shamini

Name of the lab instructor : Ms. Daksha

CONTENT

Program 1

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

Program

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;
}
```

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.

Output:

```
(base) cnlab@cnlab-Veriton-M200-H610:~/PC$ gedit merge.c
(base) cnlab@cnlab-Veriton-M200-H610:~/PC$ gcc -fopenmp merge.c -o merge
(base) cnlab@cnlab-Veriton-M200-H610:~/PC$ ./merge
Enter the number of elements: 4
Original array: 893 726 550 864
Sorted array (sequential): 550 726 864 893
Sorted array (parallel): 550 726 864 893
Sequential Merge Sort Time: 0.000004 seconds
Parallel Merge Sort Time: 0.000257 seconds
Time Difference (Sequential - Parallel): -0.000253 seconds
(base) cnlab@cnlab-Veriton-M200-H610:~/PC$
```

VIVA

1. **What is the purpose of using OpenMP in this program?**
   OpenMP allows the merge sort algorithm to execute parts of its recursion in parallel, which reduces runtime by distributing work across multiple CPU cores.

2. **Why do we use #pragma omp parallel sections instead of parallel for?**
   Parallel sections is used because merge sort divides work into two independent recursive calls, not a loop.

3. **Why is there a max_depth parameter in the parallel merge sort?**
   prevent unlimited thread creation during deep recursion. After a certain depth, the algorithm switches to sequential mode to avoid overhead.

4. **What is the time complexity of merge sort?**
   Both sequential and parallel merge sort have time complexity O(n log n) Parallel merge sort improves the constant factor by running halves in parallel.

5. **Why do we measure time using omp_get_wtime()?**
   omp_get_wtime() provides high-resolution, thread-safe timing suitable for parallel programs, unlike **clock()**, which measures only CPU time.

Program 2

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

a. Thread 0 : Iterations 0 −− 1
b. Thread 1 : Iterations 2 – 3

Theory

- **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).

Program

```
#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;
}
```

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

Output :

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

VIVA

1. **What does** schedule(static,2) **mean?**
   It tells OpenMP to divide the loop into chunks of 2 iterations each and assign these chunks statically (fixed assignment) to threads.

2. **What is the difference between static and dynamic scheduling?**

   Static: pre-determined assignment of iterations; minimal overhead; good for uniform workloads.

   Dynamic: threads request work at runtime; handles irregular workloads better but has more overhead.

3. **What does omp_get_thread_num() do?**

   It returns the ID of the thread currently executing the code inside the parallel region.

4. **Why do we use #pragma omp parallel for?**
   It tells OpenMP to parallelize the loop so that multiple threads execute iterations concurrently.

5. **What happens if the number of iterations is not divisible by the chunk size?**

   The last chunk may contain fewer iterations, but OpenMP still assigns it correctly to a thread.

Program 3

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

Theory

- **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.

Program

```
#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;
}
```

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

VIVA

1. **What is the purpose of using tasks in this Fibonacci program?**
   OpenMP tasks allow each recursive Fibonacci call to run concurrently, enabling parallel execution of independent subproblems and improving performance for large ***n***.

2. **Why is #pragma omp single used before calling fib()?**
   To ensure that only one thread creates the initial task tree. Without it, multiple threads ould redundantly start their own Fibonacci computations.

3.What does #pragma omp taskwait do?
forces the parent task to wait until all child tasks it created (e.g., fib(n−1) and fib(n−2**)**) are completed before proceeding.

4. **Why is Fibonacci a good example for task parallelism?**
   Because Fibonacci's recursive structure naturally forms a **task tree**, where each call branches into two independent subtasks, making it ideal for demonstrating OpenMP tasking.

5.Why does recursive Fibonacci perform poorly for large n, even with tasks?
Because recursive Fibonacci has exponential complexity ($O(2^n)$) and creates many redundant computations. Tasks help, but do not eliminate algorithmic inefficiency.

Program 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.

Theory

- **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.

Program

```
#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;
}
```

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).

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

VIVA

1. **Why is the parallel version faster?**
   Because the work of checking each number for primality is divided among multiple threads, reducing total execution time.
2. **Why is a critical section used in the parallel loop?**
   Multiple threads may find primes at the same time; the **critical section** ensures only one thread writes to the shared **parallel_primes** array at a time, preventing data races.
3. **What directive is used to parallelize the for loop?**
   #pragma omp parallel for This splits loop iterations among available threads.

4. **Why can prime-checking be easily parallelized?**
   Each number can be checked independently. There are **no dependencies** between iterations, making it ideal for parallel execution.
5. **What is the complexity of the prime checking used here?**

   For each number n, we check up to √n, so the complexity is:
   O(n√n) for the full loop

Program 5

Write a MPI Program to demonstration of MPI_Send and MPI_Recv.

Theory

- **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.

Program

```
#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;
}
```

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.

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

VIVA

1. **What is the purpose of MPI_Send and MPI_Recv?**
   MPI_Send is used to send data from one process to another, and MPI_Recv is used to receive that data. Together, they form the basic mechanism of point-to-point communication in MPI.

2. **What are the important parameters in MPI_Send?**

   - Address of data
   - Number of elements
   - Data type (e.g., MPI_INT)
   - Destination rank
   - Message tag
   - Communicator (usually MPI_COMM_WORLD)

3. **What does MPI_COMM_WORLD represent?**
   It is the default communicator that includes **all processes** in the MPI program. It defines the group of processes that can communicate with each other.

4. **What happens if MPI_Recv receives a message with a different data type or count?**
   It may result in a runtime error, incorrect data, or program crash. Sender and receiver must agree on data type, count, and tag.

5. **Why does MPI require at least two processes for this program?**
   Because one process must perform the send operation and another must perform the receive. With only one process, communication cannot occur.

Program 6

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

Theory

- **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

Program

```
#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;
}
```

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

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.

VIVA

1. **What causes a deadlock in MPI point-to-point communication?**
   Deadlock occurs when all processes are waiting for each other—typically when every process calls **MPI_Send** first, so no process is available to execute the matching **MPI_Recv**.

2. **How can deadlock be avoided in MPI?**
   - Change communication order (one sends first, the other receives first)
   - Use non-blocking communication (MPI_Isend, MPI_Irecv)
   - Use buffered sends (MPI_Bsend)

3. **What is the difference between blocking and non-blocking sends?**
   - Blocking send **(**MPI_Send**)** waits until the message buffer is safe to modify.
   - Non-blocking send **(**MPI_Isend**)** returns immediately, allowing computation to proceed.

4. **Why does MPI_Send sometimes wait indefinitely?**
   Because it may wait for the matching **MPI_Recv** to be posted. If the receiver never reaches **MPI_Recv**, the sender blocks forever → deadlock.

5. **Can deadlock occur even with only two MPI processes?**

   Deadlock can occur with just **two processes** if both attempt to send first and neither posts a receive.

Program 7

Write a MPI Program to demonstration of Broadcast operation.

Theory

- **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).

Program

```
#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;
}
```

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
```

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

VIVA

1. **What is the purpose of MPI_Bcast?**
   MPI_Bcast sends the same data from a root process to all other processes in a communicator. It is used to share common input or configuration data.
2. **What are the parameters of MPI_Bcast?**

   - Address of data buffer
   - Number of elements
   - Data type (e.g., MPI_INT)
   - Root process rank
   - Communicator (e.g., MPI_COMM_WORLD

3. **How is MPI_Bcast different from multiple MPI_Send() calls?**
   MPI_Bcast internally optimizes communication (tree-based algorithms) making it faster and scalable**,** unlike manually sending data to each process using **MPI_Send**

4. **What happens if non-root processes modify the buffer before calling MPI_Bcast?**
   Their changes are ignored. Only the root's buffer value is broadcast and overwritten into all other processes.

5. **Can MPI_Bcast be used for large data arrays?**
   Yes. MPI_Bcast works for any size of data, as long as the buffer size matches across processes. It is efficient for broadcasting large arrays or matrices**.**

Program 8

Write a MPI Program demonstration of MPI_Scatter and MPI_Gather

Theory

- **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.

Program

```
#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;
}
```

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

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

VIVA

1. **What is the purpose of MPI_Scatter?**
   MPI_Scatter distributes chunks of data from the **root process** to **all processes**, including the root itself.

2. **What is the purpose of MPI_Gather?**
   MPI_Gather collects data from all processes and stores it in a buffer on the root process.

3. **What must be equal across all processes when calling MPI_Scatter or MPI_Gather?**
   The number of elements each process sends/receives and the data types (e.g., MPI_INT) must match.

4. **Why is only the root process responsible for providing the send buffer in MPI_Scatter?**
   Because the root is the one distributing (scattering) the data. Other processes only need a receive buffer.

5. **Can MPI_Scatter and MPI_Gather be used for arrays larger than the number of processes?**
   Yes. They work for any data size, as long as the data

Program 9

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

Theory

- **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.

Program

```
#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;
}
```

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

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

VIVA

1. **What is the purpose of MPI_Reduce?**
   MPI_Reduce collects data from all processes, applies a reduction operation (SUM, MAX, MIN, PROD, etc.), and stores the result in the root process**.**

2. **What is the difference between MPI_Reduce and MPI_Allreduce?**

   - MPI_Reduce stores the result only in the root process.
   - MPI_Allreduce stores the result in all processes**,** making it available everywhere.

3. **Can you use MPI_Reduce for non-numeric operations?**
   Yes, if a user-defined reduction operation is provided, but common pre-defined operations include SUM, PROD, MAX, MIN.

4. **Why is MPI_Allreduce useful in parallel programming?**
   It allows all processes to have consistent summary information (e.g., global sum or max) without additional communication. Useful for collective decision-making.

5. **What happens if the data types in MPI_Reduce/Allreduce are mismatched across processes?**
   It may lead to **incorrect results or runtime errors**. All processes must agree on the **data type**, **count**, and **operation**

Parallel Computing Example (Vector Addition)

Program

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

// CUDA kernel: each thread computes one element of the sum
__global__ void vectorAdd(const float *A, const float *B, float *C, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < n) {
        C[idx] = A[idx] + B[idx];
    }
}

int main() {
    int n = 1 << 20; // 1M elements
    size_t size = n * sizeof(float);

    // Allocate host memory
    float *h_A = (float*)malloc(size);
    float *h_B = (float*)malloc(size);
    float *h_C = (float*)malloc(size);

    // Initialize host arrays
    for (int i = 0; i < n; i++) {
        h_A[i] = i;
        h_B[i] = 2 * i;
    }

    // Allocate device memory
    float *d_A, *d_B, *d_C;
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    // Copy data from host to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Launch kernel: 256 threads per block
    int threadsPerBlock = 256;
    int blocks = (n + threadsPerBlock - 1) / threadsPerBlock;

    vectorAdd<<<blocks, threadsPerBlock>>>(d_A, d_B, d_C, n);

    // Copy results back
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Verify results
```

```
    for (int i = 0; i < 10; i++) {
        printf("C[%d] = %f\n", i, h_C[i]);
    }

    // Free memory
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    free(h_A); free(h_B); free(h_C);

    return 0;
}
```

Compile & Run

Compile:
nvcc vector_add.cu -o vector_add

Run:
./vector_add

Output :

C[0] = 0
C[1] = 3
C[2] = 6
...

Program Beyond syllabus 2

UDA Matrix Multiplication with Shared Memory (Tiled)

Theory

Without shared memory
Each thread reads many elements from global memory → **slow**.

With tiling + shared memory

- Threads cooperatively load blocks (tiles) of A and B.
- Each tile is reused *TILE_SIZE* times.
- Shared memory is ~100× faster than global memory.

Performance Notes

- A typical speedup: **3×–10×** compared with naive global-memory matrix multiplication.
- Best tiles often are 16×16 or 32×32 (depending on GPU architecture).

Compute:

C=A×BC = A \times BC=A×B

Using 2D CUDA blocks and **shared memory tiles** for much faster memory access.

Program

```
#include <stdio.h>

#include <cuda_runtime.h>

#define TILE_SIZE 16   // Tile width/height

// CUDA kernel using shared memory tiling

__global__ void matrixMulShared(float *A, float *B, float *C, int N) {

   __shared__ float tileA[TILE_SIZE][TILE_SIZE];

   __shared__ float tileB[TILE_SIZE][TILE_SIZE];

   int row = blockIdx.y * TILE_SIZE + threadIdx.y;

   int col = blockIdx.x * TILE_SIZE + threadIdx.x;

   float value = 0.0f;

   // Loop over all tiles

   for (int t = 0; t < N / TILE_SIZE; t++) {

      // Load tiles into shared memory
```

```
        tileA[threadIdx.y][threadIdx.x] = A[row * N + t * TILE_SIZE + threadIdx.x];
        tileB[threadIdx.y][threadIdx.x] = B[(t * TILE_SIZE + threadIdx.y) * N + col];
        __syncthreads(); // Wait for all threads to load tiles
        // Multiply tileA and tileB
        for (int k = 0; k < TILE_SIZE; k++) {
            value += tileA[threadIdx.y][k] * tileB[k][threadIdx.x];
        }
        __syncthreads(); // Wait before loading next tiles
    }
    // Store the result
    C[row * N + col] = value;
}
int main() {
    int N = 1024; // Matrix size N × N
    size_t size = N * N * sizeof(float);
    // Allocate host memory
    float *h_A = (float*)malloc(size);
    float *h_B = (float*)malloc(size);
    float *h_C = (float*)malloc(size);
    // Initialize host matrices
    for (int i = 0; i < N * N; i++) {
        h_A[i] = 1.0f;
        h_B[i] = 2.0f;
    }
    // Allocate device memory
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
```

```
    cudaMalloc(&d_C, size);
    // Copy host to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    // Launch kernel
    dim3 threads(TILE_SIZE, TILE_SIZE);
    dim3 blocks(N / TILE_SIZE, N / TILE_SIZE);
    matrixMulShared<<<blocks, threads>>>(d_A, d_B, d_C, N);
    // Copy back
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    printf("C[0] = %f\n", h_C[0]);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    free(h_A); free(h_B); free(h_C);
    return 0;
}
```