RV EducationalInstitutions®

RV Institute of Technology and Managemet

(AffiliatedtoVTU,Belagavi)

JPNagar 8ᶦPhase, Bengaluru-560076

CSE CLUSTER

Department of Information Science and Engineering

Course Name: Parallel Computing Lab

Course Code: BCS702

VII Semester

2022 Scheme

PreparedBy:

Abhayakumar S Inchal Professor, Dept of ISE, RVITM

About the Course:

Course Title :Parallel Computing Laboratory	Course Code :BCS702
Teaching Hours/Week (L:T:P: S) 3:0:2:0	Credits : 04
SEE Marks :50	CIE Marks :50
Semester :7	Academic Year :2024-25
Lesson Plan Author :	Date :27-06-2025

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

Course Outcomes:

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.

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.

Continuous Internal Evaluation (CIE):

IPCC means practical portion integrated with the theory of the course.

- CIE marks for the theory component are **25 marks** and that for the practical component is **25 marks**.

- 25 marks for the theory component are split into **15 marks** for two Internal Assessment Tests (Two Tests, each of **15 Marks** with 01-hour duration, are to be conducted) and **10 marks** for

other assessment methods mentioned in 22OB4.2. The first test at the end of 40-50% coverage of the syllabus and the second test after covering 85-90% of the syllabus.

● Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for the theory component of IPCC (that is for **25 marks**).

● The student has to secure 40% of **25 marks** to qualify in the CIE of the theory component of IPCC.

Semester End Evaluation (SEE):

● 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

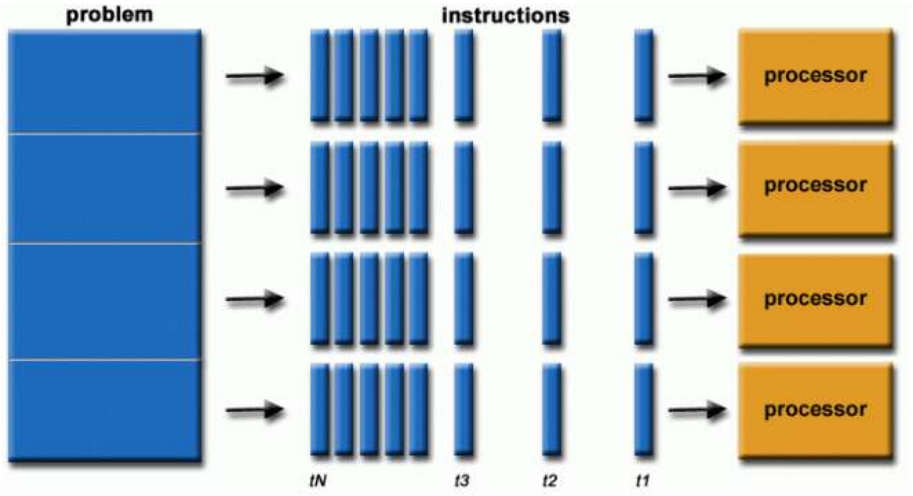
INDEX

II	SAMPLE VIVA QUESTIONS AND ANSWERS	39-43

Introduction *parallel computing*

In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem:

- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed



Parallel computing generic example

Introduction to Open MP

OpenMP is an Application Program Interface (API) that may be used to explicitly direct *multi-threaded, shared memory parallelism* in C/C++ programs. It is not intrusive on the original serial code in that the OpenMP instructions are made in pragmas interpreted by the compiler.

OpenMP uses the fork-join model of parallel execution. All OpenMP programs begin with a single master thread which executes sequentially until a parallel region is encountered, when it creates a team of parallel threads (FORK). When the team threads complete the parallel region, they synchronize and terminate, leaving only the master thread that executes sequentially (JOIN).

Hello World Example

Here is a basic example showing how to parallelize a hello world program. First, the serial version:

```
#include <stdio.h>
int main() {
   printf( "Hello, World from just me!\n" );
   return 0;
}
```

To do this in parallel (have a series of threads print out a “Hello World!” statement), we would do the following:

```
#include <stdio.h>
#include <omp.h>
int main() {
   int thread_id;
   #pragma  om parallel private(thread_id)
   {
      thread_id = omp_get_thread_num();
      printf( "Hello, World from thread %d!\n", thread_id );
   }
   return 0;
}
```

Running an OpenMP program

To compile and run the above omphello.c program:
wget https://carleton.ca/rcs/wp-content/uploads/omphello.c
gcc -o omphello omphello.c -fopenmp
export OMP_NUM_THREADS=4./omphello

OpenMP General Code Structure

The snippet below shows the general structure of a C/C++ program using Open MP.

```
#include <omp.h>
main () {
  int var1, var2, var3;
Serial code
  . . .
  //Beginning of parallel section.
  #pragma omp parallel private(var1, var2) shared(var3)
  {
    /* Parallel section executed by all threads */
    . . .
    /* All threads join master thread and disband*/
  }
Resume serial code
  . . .
  return 0;
}
```

When looking at this example you should notice a few things. First, we need to include the OpenMP header (omp.h) .Second, we notice a few variables that are declared outside of the parallel region of the code. If these variables are used within the parallel region we will need to know if they
are *public* or *private* variables . A variable being private means that every thread will have their own copy of this variable and that changes to that variable by one thread will not be seen by other threads.

A variable defined within the parallel region will be private. On the other hand, a public variable is one that is shared between all of the threads and any changes made by one thread will be seen by all of the threads. Any read-only variables can be shared. Caution must be taken when having multiple threads read and write to the same variable. Ensuring that this is done in the proper order avoids what are called “race conditions”.

Parallel For Loops

OpenMP can be used to easily parallelize for loops. This can only be done when the loop iterations are independent (ie. the running of one iteration of the loop does not depend on the result of previous iterations). Here is an example of a serial for loop:

```
for( i=0; i < 25; i++ ) {
   printf(“Foo”);
}
```

The parallel version of this loop is:

```
#pragma omp parallel for
for( i=0; i < 25; i++ ) {
  printf(“Foo”);
```

```
}
```

OpenMP Directives

In the previous sections examples of OpenMP directives have been given. The general format of these directives are:

```
#pragma omp directive-name [clause,..] newline
```

The scope of a directive is a block of statements surrounded by { }. A variety of clauses are available, including:

- if (expression): only in parallel if expression evaluates to true
- private(list): variables private and local to a thread
- shared(list): data accessed by all threads
- default (none|shared)
- reduction (operator: list)

  The reduction clause is used when the result of a parallel region is single value. For example, imagine we have an array of integers we would like the sum of. We can do this in parallel as follows:

```
int sum = 0;
#pragma omp parallel default(none) shared (n, x) \
  private (i) reduction(+ : sum)
{
   for(i = 0; i < n; i++)
       sum = sum + x(I);
}
```

  Since sum is a shared variable, we must be careful to avoid race conditions surrounding it. Using a reduction clause ensures that the generated code avoids such situations.

Introduction to MPI

The Message Passing Interface (MPI) is an Application Program Interface that defines a model of parallel computing where each parallel process has its own local memory, and data must be explicitly shared by passing messages between processes. Using MPI allows programs to scale beyond the processors and shared memory of a single compute server, to the *distributed memory* and processors of multiple compute servers combined together.

An MPI parallel code requires some changes from serial code, as MPI function calls to communicate data are added, and the data must somehow be divided across processes.

Hello World Example

Here is an example MPI program called mpihello.c:

```
#include <mpi.h>                                    //line 1
#include <stdio.h>                                  //line 2
int main( int argc, char **argv ) {                 //line 3
   int rank, size;                                  //line 4
   MPI_Init( &argc, &argv );                        //line 5
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );          //line 6
   MPI_Comm_size( MPI_COMM_WORLD, &size );          //line 7
   printf( "Hello from process %d/%d\n", rank, size ); //line 8
   MPI_Finalize( );                                 //line 9
   return 0;                                        //line 10
}
```

This hello world program has a considerable amount added to it from the standard C example. There are a number of things to point out:

- line 1: We include the MPI header here to have access to the various MPI functions.
- line 5: Here we initialize the MPI execution environment. This must be done at the start of the program.
- line 6: Each MPI process is assigned a unique integer ID starting at 0. This function retrieves the local ID and stores it in the local variable rank.
- line 7: This function retrieves the total number of MPI processes running and stores it in the local variable size.
- line 8: Here we print hello world with the environment information we gathered in the previous lines.
- line 9: Here we finalize the MPI execution environment. This must be done before the end of the program.

What Is CUDA C Programming

CUDA is a parallel computing platform and programming model developed by NVIDIA that enables dramatic increases in computing performance by harnessing the power of the GPU. It allows developers to accelerate compute-intensive applications using C, C++, and Fortran, and is widely adopted in fields such as deep learning, scientific computing, and high-performance computing (HPC).

CUDA C++ provides a simple path for users familiar with the C++ programming language to easily write programs for execution by the device.

It consists of a minimal set of extensions to the C++ language and a runtime library.

The core language extensions have been introduced in **Programming Model.** They allow programmers to define a kernel as a C++ function and use some new syntax to specify the grid and block dimension each time the function is called. A complete description of all extensions can be found in **C++ Language Extensions.** Any source file that contains some of these extensions must be compiled with nvcc as outlined in **Compilation with NVCC.**

The runtime is introduced in **CUDA Runtime**. It provides C and C++ functions that execute on the host to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc. A complete description of the runtime can be found in the CUDA reference manual.

The runtime is built on top of a lower-level C API, the CUDA driver API, which is also accessible by the application. The driver API provides an additional level of control by exposing lower-level concepts such as CUDA contexts - the analogue of host processes for the device - and CUDA modules - the analogue of dynamically loaded libraries for the device. Most applications do not use the driver API as they do not need this additional level of control and when using the runtime, context and module management are implicit, resulting in more concise code. As the runtime is interoperable with the driver API, most applications that need some driver API features can default to use the runtime API and only use the driver API where needed. The driver API is introduced in **Driver API** and fully described in the reference manual.

Kernels can be written using the CUDA instruction set architecture, called *PTX*, which is described in the PTX reference manual. It is however usually more effective to use a high-level programming language such as C++. In both cases, kernels must be compiled into binary code by **nvcc** to execute on the device.

Program's

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

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#define SIZE 100000
// ---------------------- MERGE FUNCTION ----------------------
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int *L = (int *)malloc(n1 * sizeof(int));
    int *R = (int *)malloc(n2 * sizeof(int));

    for (i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    i = 0; j = 0; k = left;

    while (i < n1 && j < n2)
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

    while (i < n1)
        arr[k++] = L[i++];

    while (j < n2)
        arr[k++] = R[j++];

    free(L);
    free(R);
}
```

```
// ---------------------- SERIAL MERGE SORT ----------------------
void serialMergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        serialMergeSort(arr, left, mid);
        serialMergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

// ---------------------- PARALLEL MERGE SORT ----------------------
void parallelMergeSort(int arr[], int left, int right, int depth) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        if (depth <= 4) {
            #pragma omp parallel sections
            {
                #pragma omp section
                parallelMergeSort(arr, left, mid, depth + 1);

                #pragma omp section
                parallelMergeSort(arr, mid + 1, right, depth + 1);
            }
        } else {
            // Switch to serial to avoid too many threads
            serialMergeSort(arr, left, mid);
            serialMergeSort(arr, mid + 1, right);
        }

        merge(arr, left, mid, right);
    }
}

// ---------------------- MAIN FUNCTION ----------------------
int main() {
    int *arr_serial = (int *)malloc(SIZE * sizeof(int));
    int *arr_parallel = (int *)malloc(SIZE * sizeof(int));

    // Initialize both arrays with the same random values
    for (int i = 0; i < SIZE; i++) {
        int val = rand() % 100000;
```

```
    arr_serial[i] = val;
    arr_parallel[i] = val;
  }

  // ---------- SERIAL MERGE SORT ----------
  clock_t start_serial = clock();
  serialMergeSort(arr_serial, 0, SIZE - 1);
  clock_t end_serial = clock();
  double time_serial = (double)(end_serial - start_serial) / CLOCKS_PER_SEC;

  // ---------- PARALLEL MERGE SORT ----------
  clock_t start_parallel = clock();
  parallelMergeSort(arr_parallel, 0, SIZE - 1, 0);
  clock_t end_parallel = clock();
  double time_parallel = (double)(end_parallel - start_parallel) / CLOCKS_PER_SEC;

  // ---------- OUTPUT ----------
  printf("Serial Merge Sort Time   : %.6f seconds\n", time_serial);
  printf("Parallel Merge Sort Time : %.6f seconds\n", time_parallel);

  // Optional: Verify correctness
  /*
  for (int i = 0; i < SIZE; i++) {
    if (arr_serial[i] != arr_parallel[i]) {
      printf("Mismatch at index %d\n", i);
      break;
    }
  }
  */

  free(arr_serial);
  free(arr_parallel);
  return 0;
}
```

OUTPUT

Case:1

The number of elements in the array ar 100000

Serial Merge Sort Time : 0.010000 seconds

Parallel Merge Sort Time : 0.002000 seconds

Process returned 0 (0x0) execution time : 0.161 s

Case:2

The number of elements in the array ar 1000000

Serial Merge Sort Time : 0.176000 seconds

Parallel Merge Sort Time : 0.100000 seconds

Process returned 0 (0x0) execution time : 0.472 s

Case:3

The number of elements in the array ar 10000000

Serial Merge Sort Time : 1.907000 seconds

Parallel Merge Sort Time : 1.131000 seconds

Process returned 0 (0x0) execution time : 3.318 s

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

a. Thread 0 : Iterations 0 – 1

b. Thread 1 : Iterations 2 – 3

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

int main() {
   int n = 16,thread;
   printf("\n Enter the number of tasks");
   scanf("%d",&n);
   printf("\n Enter the number of threads");
   scanf("%d",&thread);
   omp_set_num_threads(thread);
   printf("\n -------------------------------------\n");
   #pragma omp parallel for schedule(static, 2)
   for (int i = 0; i < n; i++) {
      printf("Thread %d executes iteration %d\n", omp_get_thread_num(), i);
```

```
    }

    return 0;
}
```

OUTPUT

```
Enter the number of tasks24

 Enter the number of threads12

 ------------------------------------
Thread 1 executes iteration 2
Thread 1 executes iteration 3
Thread 4 executes iteration 8
Thread 4 executes iteration 9
Thread 6 executes iteration 12
Thread 6 executes iteration 13
Thread 9 executes iteration 18
Thread 9 executes iteration 19
Thread 7 executes iteration 14
Thread 7 executes iteration 15
Thread 10 executes iteration 20
Thread 10 executes iteration 21
Thread 0 executes iteration 0
Thread 0 executes iteration 1
Thread 2 executes iteration 4
Thread 2 executes iteration 5
```

Thread 3 executes iteration 6

Thread 3 executes iteration 7

Thread 5 executes iteration 10

Thread 5 executes iteration 11

Thread 11 executes iteration 22

Thread 11 executes iteration 23

Thread 8 executes iteration 16

Thread 8 executes iteration 17

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

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

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

// Serial Fibonacci calculation function
int ser_fib(long int n) {
   if (n < 2) return n;
   long int x, y;
   x = fib(n - 1);
   y = fib(n - 2);
   return x + y;
}

// parallel Fibonacci calculation function
int fib(long int n) {
   if (n < 2) return n;
   long int x, y;

   #pragma omp task shared(x)
```

```
    x = fib(n - 1);

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

    #pragma omp taskwait
    return x + y;
}

int main() {
    long int n = 10, result;
    clock_t start, end;
    double cpu_time;
    printf("\n enter the value of n");
    scanf("%ld",&n);

  start=clock();
    #pragma omp parallel
    {
       #pragma omp single
       result = fib(n);
    }
    end=clock();
    cpu_time = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("Fibonacci(%d) = %d\n", n, result);
    printf("\n the time used to execute the program in parallel mode= %f",cpu_time);

start=clock();
     result = ser_fib(n);
     end=clock();
     cpu_time = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("\nFibonacci(%d) = %d\n", n, result);
    printf("\n the time used to execute the program in sequential mode= %f",cpu_time);
    return 0;
}
```

Output:-

```
enter the value of n32
Fibonacci(32) = 2178309

 the time used to execute the program in parallel mode= 3.370000
Fibonacci(32) = 2178309

 the time used to execute the program in sequential mode= 0.157000
```

4. Write a OpenMP program to find the prime numbers from 1 to n employing parallel for directive. Record both serial and parallel execution times

```
#include <stdio.h>
#include <omp.h>
#include<time.h>
int is_prime(int n) {
    if (n < 2) return 0;
    for (int i = 2; i*i <= n; i++)
        if (n % i == 0) return 0;
    return 1;
}

int main() {
    long n = 10000000;
        time_t start,end;
    double cpu_time;
    printf("\n the range of numbers is 1 to %d\n",n);
    printf("\n----------------------------------\n");
    // Serial
    start=clock();
    for (int i = 1; i <= n; i++)
    {
        is_prime(i);
    }
    end = clock();
    cpu_time = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf(" Time to compute prime numbers serially: %f\n",cpu_time);
```

```
    // Parallel
    start=clock();
    #pragma omp parallel for
   for (int i = 1; i <= n; i++)
    {
       is_prime(i);


    }
    end = clock();
    cpu_time = ((double) (end - start)) / CLOCKS_PER_SEC;
      printf("Time to compute prime numbers Parallel: %f\n",cpu_time);
    return 0;
}
```

Output 1 :

the range of numbers is 1 to 10000000

Time to compute prime numbers serially: 2.403000

Time to compute prime numbers Parallel: 0.491000

Output 2:

the range of numbers is 1 to 1000000

Time to compute prime numbers serially: 0.095000

Time to compute prime numbers Parallel: 0.025000

Output 3:

the range of numbers is 1 to 100000

Time to compute prime numbers serially: 0.006000

Time to compute prime numbers Parallel: 0.000000

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

```
// Filename: mpi_send_recv_demo.c

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

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

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

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

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

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

   if (rank == 0) {
      // Process 0 sends a number to Process 1
      number = 42;
      printf("Process 0 is sending number %d to Process 1\n", number);
      MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
   } else if (rank == 1) {
      // Process 1 receives a number from Process 0
      MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      printf("Process 1 received number %d from Process 0\n", number);
   }
```

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

Output:

Process 0 is sending number 42 to Process 1

Process 1 received number 42 from Process 0

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

```
// deadlock_mpi.c
#include <stdio.h>
#include <mpi.h>

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

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

    data_send = rank;

    if (rank == 0) {
```

```
        MPI_Send(&data_send, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&data_recv, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    } else if (rank == 1) {
        MPI_Send(&data_send, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Recv(&data_recv, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    }

    printf("Process %d received %d\n", rank, data_recv);
    MPI_Finalize();
    return 0;
}
```

Output:

Process 1 received message: 0
Process 0 received message: 1

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

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

int main(int argc, char** argv) {
    int rank, data = 0;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        data = 100;
```

```
    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Process %d received data: %d\n", rank, data);

    MPI_Finalize();
    return 0;
}
```

Output:

Process 0 received data: 100

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

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

int main(int argc, char** argv) {
    int rank, size, send_data[4] = {10, 20, 30, 40}, recv_data;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
    MPI_Scatter(send_data, 1, MPI_INT, &recv_data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Process %d received: %d\n", rank, recv_data);

    recv_data += 1;
    MPI_Gather(&recv_data, 1, MPI_INT, send_data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Gathered data: ");
        for (int i = 0; i < size; i++) printf("%d ", send_data[i]);
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

Output:

Process 0 received: 10
Process 1 received: 20
Process 2 received: 30
Process 3 received: 40
Gathered data: 11 21 31 41

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

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

int main(int argc, char** argv) {
    int rank, value, sum, max;
```

```
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    value = rank + 1;

    MPI_Reduce(&value, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) printf("Sum using Reduce: %d\n", sum);

    MPI_Allreduce(&value, &max, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    printf("Max using Allreduce (rank %d): %d\n", rank, max);

    MPI_Finalize();
    return 0;
}
```

Output:

Sum using Reduce: 10
Max using Allreduce (rank 0): 4
Max using Allreduce (rank 1): 4
Max using Allreduce (rank 2): 4
Max using Allreduce (rank 3): 4

Sample Questions

1. What is parallel computing and why do we use it?
2. Explain the difference between parallel and concurrent execution.
3. What are the types of parallelism you implemented in your lab?
4. How do you create threads in OpenMP? Write a sample pragma directive.
5. What is a race condition? How did you avoid it in your program?
6. Explain how you used synchronization constructs like critical sections or barriers.
7. What is the role of MPI in parallel programming?
8. How do you send and receive messages in MPI? Mention the functions used.
9. Explain how matrix multiplication was parallelized in your lab.
10. What is the difference between shared memory and distributed memory programming?
11. How did you measure the speedup or efficiency of your parallel program?
12. What happens if you don't synchronize threads properly?
13. What is a deadlock? Can you give an example?
14. What is load balancing and why is it important in parallel programs?
15. How does Amdahl's Law affect your program's performance?
16. How is work divided among processors in your parallel programs?
17. What tools or compilers did you use for running your parallel programs?
18. What challenges did you face while writing parallel code? How did you overcome them?
19. How would you debug a parallel program?
20. Explain the concept of barrier in parallel programming.
21. What is a thread and how is it different from a process?
22. Explain the concept of false sharing in shared memory systems.
23. What is the difference between strong scaling and weak scaling?
24. How do you implement parallel loops using OpenMP?
25. What are race conditions and data hazards? Give examples.
26. What is the use of the #pragma omp parallel for directive?
27. How do you handle critical sections in OpenMP?
28. Explain the concept of deadlock detection and prevention.
29. What is a communicator in MPI?
30. How does non-blocking communication work in MPI?
31. Describe the difference between blocking and non-blocking sends in MPI.

32. What is a reduction operation in parallel computing? How is it implemented in OpenMP?
33. What is granularity in parallel computing?
34. Explain what is meant by load imbalance and how can it affect performance.
35. What are collective communication operations in MPI? Give examples.
36. How do you debug MPI programs? Name any tools used.
37. What is task parallelism vs data parallelism?
38. How does GPU parallelism differ from CPU parallelism?
39. What is SIMD and how does it relate to parallel computing?
40. Explain thread affinity and why it matters.
41. What is the significance of synchronization primitives like mutex and semaphore?
42. How do barriers work in parallel programming?
43. What is a parallel reduction and how can it be optimized?
44. How do you ensure memory consistency in parallel programs?
45. What are the limitations of Amdahl's Law?
46. Can you explain Gustafson's Law? How does it differ from Amdahl's Law?
47. What is a parallel region in OpenMP?
48. How do you specify the number of threads in OpenMP?
49. What are the common causes of deadlock in parallel systems?
50. How do you test the correctness of parallel programs?