Internal Assessment Test II – DECEMBER 2025

Sub:	PARALLEL COMPUTING					Sub Code:	BCS702	Branch:	ISE	
Date:	02-12-2025	Duration:	90 min's	Max Marks:	50	Sem/Sec:	Vii- A,B,C			OBE

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1	Explain in detail why MPI_Sendrecv is preferred in the MPI implementation of the Odd–Even Transposition Sort algorithm. Discuss how it prevents deadlocks. **Solutions:** Odd–Even Transposition Sort is a parallel sorting algorithm that works as follows: Each process holds a local portion of the array. Sorting proceeds in phases**:** **Odd phase:** Each process exchanges data with its neighbour at an odd distance (process i exchanges with i+1 if i is odd). **Even phase:** Each process exchanges data with its neighbor at an even distance (process i exchanges with i+1 if i is even). 1. After each exchange, processes compare elements and keep the appropriate values (smaller or larger, depending on ordering). 2. The process continues for P phases, where P is the total number of processes. During each phase: Every process must send its data to a neighbor **and** receive data from the neighbor**.** The communication **is** bidirectional**,** i.e., both neighbors are simultaneously sending and receiving. If we use separate `MPI_Send` and `MPI_Recv` calls: MPI_Send(&my_data, 1, MPI_INT, neighbor, 0, MPI_COMM_WORLD); MPI_Recv(&neighbor_data, 1, MPI_INT, neighbor, 0, MPI_COMM_WORLD); • Suppose two neighboring processes both call `MPI_Send` first. • Each process waits for the other to receive**,** causing a deadlock**.** • This happens because standard blocking **`MPI_Send`** may block until the matching **`MPI_Recv`** is posted**.** `MPI_Sendrecv` is a single call that performs **a** send and receive simultaneously**:** MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, MPI_COMM_WORLD, &status); How MPI_Sendrecv Prevents Deadlocks • Blocking operations in MPI (**MPI_Send**) can deadlock if two processes wait for each other. • MPI_Sendrecv is implemented such that the send and receive happen logically at the same time**:** o MPI can internally buffer the data or match receives while sending.	10	CO3	L2

	○ Neighboring processes never wait indefinitely, because MPI_Sendrecv ensures that data is exchanged in a coordinated manner. • In the Odd–Even Transposition Sort: ○ During odd and even phases, all processes call MPI_Sendrecv simultaneously with their neighbors. ○ Each process completes the send and receive atomically**,** so sorting proceeds without deadlocks.			
2	a) Explain speedup and efficiency of a parallel program and explain their ideal values **Solutions:** **<u>Speedup</u>** Speedup measures how much faster a parallel program runs compared to its corresponding sequential version. $$\text{Speedup}(S) = \frac{\text{Time taken by sequential program } (T_s)}{\text{Time taken by parallel program } (T_p)}$$ • T_s = execution time of the best sequential algorithm • T_p = execution time of the parallel algorithm using p processors **Interpretation:** • If $S = 1$, the parallel program is **as fast as sequential** → no gain. • If $S = p$ (number of processors), it is called **linear speedup** → ideal case. **Example:** • Sequential time = 100 seconds • Parallel time on 4 processors = 25 seconds $$S = 100/25 = 4 \quad \text{(linear speedup, ideal)}$$ **Efficiency** Efficiency measures how well the processors are utilized in a parallel program. It is the fraction of time processors are usefully working. $$\text{Efficiency}(E) = \frac{\text{Speedup}(S)}{\text{Number of processors } (p)} = \frac{T_s}{p \cdot T_p}$$ • $E = 1$ → 100% efficient, all processors fully utilized • $E < 1$ → some processors are idle or there is communication overhead **Interpretation:** • Efficiency decreases as the number of processors increases if the problem size is fixed. • Helps determine the **optimal number of processors** for a given problem. **Metric** — **Ideal Value** Speedup (S) — S = p Efficiency (E) — E = 1 b) Give an example MPI program that shows how one process can broadcast a value to all processes using MPI_Bcast **Solutions:** #include <stdio.h> #include <mpi.h> int main(int argc, char *argv[]) {	6+4	CO3	L2

```
    int rank, size;
    int value;

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

    if (rank == 0) {
        value = 100;                        // Process 0 initializes the value
        printf("Process %d broadcasting value = %d\n", rank, value);
    }

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

    // All processes now have the value
    printf("Process %d received value = %d\n", rank, value);

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

OUTPUT:

Process 0 broadcasting value = 100
Process 0 received value = 100
Process 1 received value = 100
Process 2 received value = 100
Process 3 received value = 100

3	a) Give a short description of OpenMP and how it is used in parallel programming.	6+4	CO4	L3

Solutions:

OpenMP (Open Multi-Processing) is an API for shared-memory parallel programming in C, C++, and Fortran. It provides a simple and flexible way to parallelize code on multi-core processors using compiler directives, library routines, and environment variables**.**

1. **Shared-memory parallelism**
   - All threads can access a common memory space.
   - No need for explicit message passing.
2. **Easy to use**
   - Parallelism is introduced via **pragmas/directives** in the source code, e.g., #pragma omp parallel.
3. **Flexible thread control**
   - Specify number of threads (omp_set_num_threads)
   - Use work-sharing constructs like for, sections, and single.
4. **Synchronization support**
   - Provides critical, barrier, atomic to prevent race conditions.

	b) Explain the difference between data parallelism and task parallelism with examples.
	Solutions:
	
	• **Data Parallelism:**
	  • Same operation applied simultaneously to different parts of the data**.**
	  • Focuses on splitting data across threads**.**
	  • Example: Adding two arrays element-wise in parallel.
	
	• **Task Parallelism:**
	  • Different tasks or operations executed in parallel**,** possibly on the same or different data.
	  • Focuses on splitting tasks across threads**.**
	  • Example: One thread reads input, another processes data, another writes output.
	
	• **Key Difference:**
	  • Data parallelism → parallelizes data
	  • Task parallelism → parallelizes tasks/operations			

4	a) Explain the importance of shared memory in CUDA when implementing the trapezoidal rule. **Solutions:** In CUDA, shared memory plays an important role in improving the performance of parallel numerical integration algorithms such as the trapezoidal rule. Shared memory is a small, user-managed memory space that is much faster than global memory and is accessible to all threads within the same block**.** When implementing the trapezoidal rule: $$\int_a^b f(x)\,dx \approx \frac{h}{2}(f(a)+f(b)) + h\sum_{i=1}^{n-1} f(x_i)$$ Each thread typically evaluates one or more values of f(xi)f(x_i)f(xi). The sum of all these values must then be accumulated. Shared memory helps in this process in several ways. Shared memory is important in CUDA implementations of the trapezoidal rule because: 1. It provides fast storage for intermediate function values computed by threads.	6+4	CO4	L2

	2. It enables efficient parallel reduction within each block to accumulate partial sums.
	3. It reduces global memory access**,** improving performance.
	4. It allows synchronization among threads in a block using __syncthreads().
	5. It reduces global memory writes to only one value per block.
	6. It improves scalability for large-scale numerical integration.
	
	b) Illustrate usage of reduction clause in openMP with a suitable Open MP program.
	
	Solutions:
	
	The reduction clause in OpenMP is used to combine the partial results computed by multiple threads into a single final result.
	It ensures that each thread has its own private copy of a variable, and at the end of the parallel region, these private copies are combined using the specified operator (e.g., +, *, max, min, etc.).
	
	#include <stdio.h>
	#include <omp.h>
	
	int main() {
	int i;
	int n = 10;
	int sum = 0;
	
	/* Parallel for loop with reduction */
	#pragma omp parallel for reduction(+:sum)
	for (i = 1; i <= n; i++) {
	sum += i; // each thread accumulates in its private copy
	}
	
	printf("Sum of first %d numbers = %d\n", n, sum);
	
	return 0;
	}
	OUTPUT:
	Sum of first 10 numbers = 55			

5	a) Explain the concepts of threads, blocks, and grids in CUDA. **Solutions:** A **thread** in CUDA is the smallest unit of execution that runs a kernel function independently. Each thread has its own register state, local memory, and a unique identifier (threadIdx) used to access its portion of data. A **thread block** is a group of threads that execute the same kernel and can cooperate with each other. Threads within a block can share data through **shared memory** and can synchronize using __syncthreads(). Each block is uniquely identified by blockIdx.	5+5	CO5	L3

	A **grid** is the collection of all thread blocks launched for a single kernel execution. All blocks in a grid run the same kernel but execute independently, without inter-block synchronization. The grid is identified by its dimensions (gridDim).
	
	b) Write a CUDA program that prints greetings from the threads.
	Solutions:
	`#include <stdio.h>`
	
	`__global__ void greet_from_threads() {`
	`  printf("Hello from thread %d in block %d!\n", threadIdx.x, blockIdx.x);`
	`}`
	
	`int main() {`
	`  int blocks = 2;  // number of blocks`
	`  int threads = 4;  // threads per block`
	
	`  greet_from_threads<<<blocks, threads>>>();`
	
	`  cudaDeviceSynchronize();  // wait for GPU to finish`
	`  return 0;`
	`}`
	Hello from thread 0 in block 0!
	Hello from thread 1 in block 0!
	Hello from thread 2 in block 0!
	Hello from thread 3 in block 0!
	Hello from thread 0 in block 1!
	Hello from thread 1 in block 1!
	Hello from thread 2 in block 1!
	Hello from thread 3 in block 1!			

6	Explain the trapezoidal rule for numerical integration and implement a serial C function to compute the integral of a given function using this rule. **Solutions:** The **trapezoidal rule** is a numerical method used to approximate the definite integral of a function $$\int_a^b f(x)\,dx$$ The idea is to divide the interval $[a, b]$ into **n equal subintervals**, each of width $$h = \frac{b-a}{n}$$ Over each subinterval $[x_i, x_{i+1}]$, the curve is approximated by a **straight line**, forming a **trapezoid**. The area of one trapezoid is: $$\text{Area}_i = \frac{f(x_i) + f(x_{i+1})}{2} \cdot h$$ Summing all trapezoids, the approximation becomes: $$\int_a^b f(x)\,dx \approx \frac{h}{2}\left(f(a) + f(b) + 2\sum_{i=1}^{n-1} f(x_i)\right)$$	10	CO5	L3

This method is frequently used in parallel programming because each function evaluation f(xi)f(x_i)f(xi) can be computed independently. The example is given below

Example:

```
#include <stdio.h>

/* Function prototype */
double f(double x);

/* Compute the integral using the trapezoidal rule */
double Trap(double a, double b, int n) {
   double h, integral, x;
   int i;

   h = (b - a) / n;
   integral = (f(a) + f(b)) / 2.0;
   for (i = 1; i <= n - 1; i++) {
      x = a + i * h;
      integral += f(x);
   }
   integral = integral * h;

   return integral;
}

/* Example function: f(x) = x*x */
double f(double x) {
   return x * x;
}

int main() {
   double a = 0.0, b = 1.0;
   int n = 1024;
   double result;

   result = Trap(a, b, n);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %f\n", a, b, result);

   return 0;
}
```