odified

CBCS SCHEME

USN | | | | | | | | | | |

BCS702

Seventh Semester B.E./B.Tech. Degree Examination, Dec.2025/Jan.2026

Parallel Computing

Time: 3 hrs. Max. Marks: 100

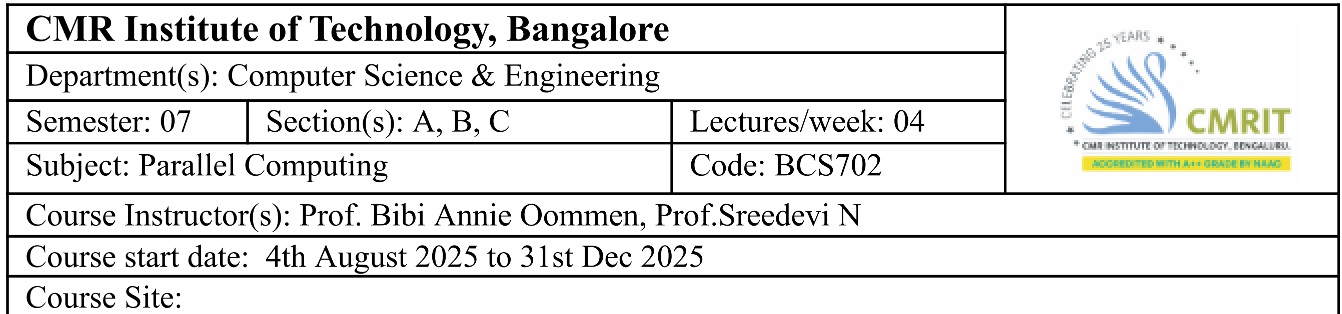
Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

		Module – 1	M	L	C
Q.1	a.	Explain in detail the classification of parallel computers according to Flynn's taxonomy. Compare SIMD and MIMD systems.	10	L1	CO1
	b.	Discuss shared memory and distributed memory architectures. Explain their working principle advantages and disadvantages.	10	L2	CO1
		OR			
Q.2	a.	What is cache coherence? Explain snooping and directory based coherence mechanisms with suitable example.	10	L3	CO1
	b.	Explain non-determinism and race conditions in shared memory programs. How can they be avoided?	10	L3	CO1
		Module – 2			
Q.3	a.	Explain GPU programming in detail.	10	L2	CO2
	b.	Describe input and output handling in MIMD and GPU systems.	10	L2	CO2
		OR			
Q.4	a.	Explain Amdahl's law with example and discuss it significance.	10	L3	CO2
	b.	Write a short note on timing and performance measurement of parallel programs.	10	L3	CO2
		Module – 3			
Q.5	a.	Define and explain the following MPI functions with syntax and purpose : i) MPI_Init( ) ii) MPI_Finalize( ) iii) MPI_Comm_Size( ) iv) MPI_Comm_rank( )	10	L3	CO3
	b.	Explain the concept of point to point communication in MPI with suitable example.	10	L2	CO3
		OR			
Q.6	a.	Explain the working of trapezoidal rule program in MPI.	10	L3	CO3
	b.	Compare the traditional global sum using process 0 as collector with tree structured global sum	10	L3	CO3

Module – 4					
Q.7	a.	Explain the structure and working of an OpenMP "Hello world" program.	6	L2	CO4
	b.	Explain the purpose of the reduction clause in OpenMP with example.	6	L2	CO4
	c.	Write a OpenMP program to calculate n-Fibonacci using tasks.	8	L2	CO4
OR					
Q.8	a.	Define OpenMP. Explain the key features of OpenMP and its advantages over Pthreads.	6	L1	CO4
	b.	Explain the concept of variable scope in OpenMP with suitable example.	6	L2	CO4
	c.	Estimate the value of π.	8	L3	CO4
Module – 5					
Q.9	a.	Define GPU and GPGPU. Explain the need for GPGPU.	6	L1	CO5
	b.	Explain thread, block and grid in CUDA.	6	L2	CO5
	c.	Explain CUDA vector addition program with suitable example.	8	L3	CO5
OR					
Q.10	a.	Compare CUDA and OpenCL.	6	L1	CO5
	b.	Explain Kernel with shared memory.	6	L2	CO5
	c.	Explain Heterogeneous computing in detail.	8	L3	CO5

* * * * *

VTU EXAMS DEC-JAN -2026



CMR Institute of Technology, Bangalore		
Department(s): Computer Science & Engineering		
Semester: 07	Section(s): A, B, C	Lectures/week: 04
Subject: Parallel Computing		Code: BCS702
Course Instructor(s): Prof. Bibi Annie Oommen, Prof.Sreedevi N		
Course start date: 4th August 2025 to 31st Dec 2025		
Course Site:		

Seventh Semester BE/BTech Degree Examination, Dec 2025/Jan 2026

VTU QP Solution

Q1.a. Explain in detail the classification of Parallel Computers according to Flynn's taxonomy. Compare SIMD and MIMD systems. (10 marks)

Classification of Parallel Computers according to Flynn's taxonomy- 5 marks

No of Instruction Streams

No of Data Streams	Single (Instruction)	Many (Instruction)
Single	**SISD** Traditional Single CPU computer (Sequential computer)	**MISD** Multiple Processing units, working on different instructions but on the same data element
Many	**SIMD** Multiple processing units, all of which execute the same instruction, but each of which can act on a distinct data element	**MIMD** Multiple processing units, each of which can execute a different Instruction on a different data element

Comparison of SIMD and MIMD systems- 5 marks

SIMD	MIMD
Executes *one instruction* across multiple data streams at the same time.	Executes *different instructions* across multiple data streams simultaneously.
Data-level parallelism.	Task-level parallelism.

Synchronous (lockstep)	Execution can be synchronous or asynchronous. MIMD systems are usually asynchronous, that is, the processors can operate at their own pace. In many MIMD systems, there is no global clock, and there may be no relation between the system times on two different processors. In fact, unless the programmer imposes some synchronization, even if the processors are executing exactly the same sequence of instructions, at any given instant they may be executing different statements.
deterministic execution	Execution can be deterministic or non-deterministic
Best suited for specialized problems characterized by a high degree of regularity, such as image processing. Two varieties: Processor Arrays and Vector Pipelines	Examples: most current supercomputers, networked parallel computer grids

Q1.b. Discuss shared memory and distributed memory architectures. Explain their working principle, advantages and disadvantages. (10 marks)

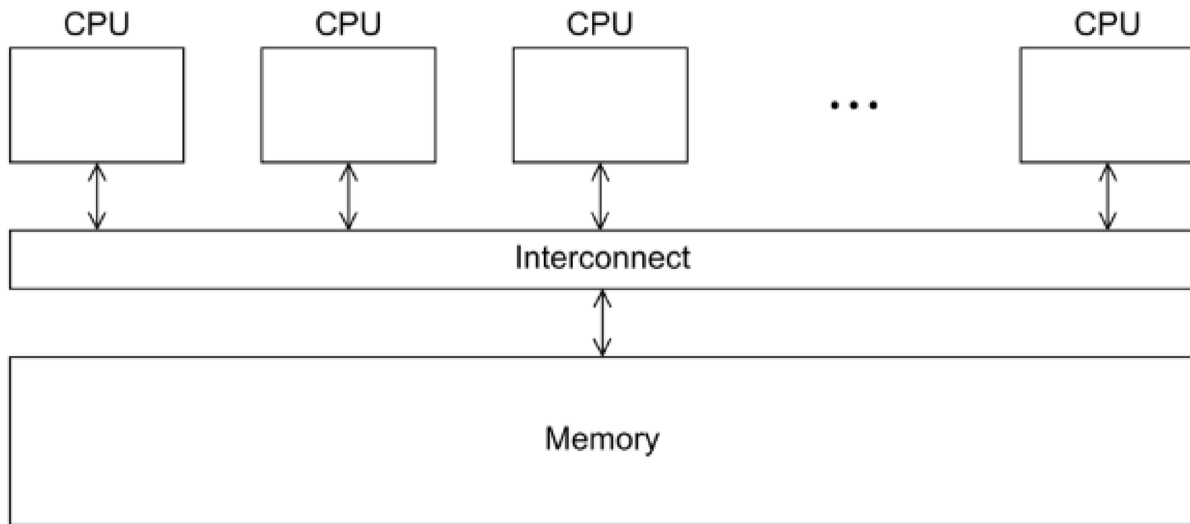
Definition of shared memory and distributed memory – 2 marks

Diagrams- 4 marks

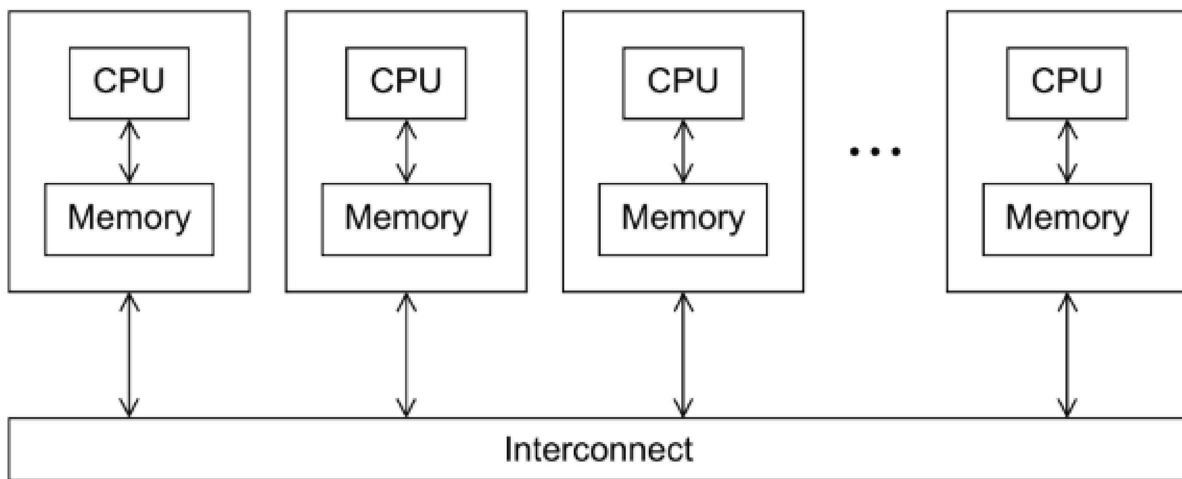
working principle- 2 marks

advantages and disadvantages- 2 marks

A shared-memory system consists of a collection of cores connected to a globally accessible memory, in which each core can have access to any memory location.



A distributed-memory system consists of a collection of core-memory pairs connected by a network, and the memory associated with a core is directly accessible only to that core.



Working principle-

Shared Memory: All processors access a common global memory space, communicating by reading and writing to the same locations.

Distributed Memory: Each processor has its own local memory, and communication occurs through explicit message passing between processors.

Shared Memory Systems

Advantages

- Fast communication: All processors access a common memory space, reducing latency.
- Ease of programming: Programmers can use threads within a single address space (e.g., OpenMP).
- Efficient for small-scale parallelism: Works well when the number of processors is limited.
- Direct data sharing: No need for explicit message passing; variables are accessible to all threads.

Disadvantages

- Scalability limits: As the number of processors grows, contention for memory increases.
- Synchronization overhead: Requires locks, semaphores, or barriers to avoid race conditions.
- Memory bottleneck: Shared bus or memory controller can become a performance bottleneck.
- Hardware cost: Maintaining cache coherence across many processors is complex and expensive.

Distributed Memory Systems

Advantages

- High scalability: Each processor has its own memory, making it easier to scale to thousands of nodes.
- No contention: Independent memory avoids bottlenecks from shared access.
- Flexibility: Suitable for large-scale scientific simulations and high-performance computing clusters.
- Cost-effective scaling: Commodity hardware can be networked together to build large systems.

Disadvantages

- Complex programming: Requires explicit message passing (e.g., MPI), which is harder to implement.
- Higher communication overhead: Data exchange between nodes is slower compared to shared memory.
- Debugging difficulty: Errors in message passing or synchronization are harder to trace.

- Latency issues: Network delays can affect performance in tightly coupled tasks.

Q2. a. What is cache coherence? Explain snooping and directory based coherence mechanisms with suitable example. (10 marks)

cache coherence- 3 marks

snooping and directory based coherence mechanisms- 6 marks

example- *1 mark*

cache coherence

- It refers to the challenge of keeping multiple caches consistent when processors read and write shared data.
- If one processor updates a value in its cache, other processors may still have outdated copies of that value in their own caches.
- This can lead to incorrect program behavior, such as using stale data or violating expected memory ordering.
- There are two main approaches to ensuring cache coherence:
    - snooping cache coherence
    - directory-based cache coherence.

snooping cache coherence

- snooping protocol: When the cores share a bus, any signal transmitted on the bus can be "seen" by all the cores connected to the bus. Thus when core 0 updates the copy of x stored in its cache, if it also broadcasts this information across the bus, and if core 1 is "snooping" the bus, it will see that x has been updated, and it can mark its copy of x as invalid.
- the broadcast informs the other cores that the cache line containing x has been updated, not that x has been updated.
- snooping works with both write-through and write-back caches.
- if the interconnect is shared—as with a bus—with write-through caches, there's no need for additional traffic on the interconnect, since each core can simply "watch" for writes.
- With write-back caches, an extra communication is necessary, since updates to the cache don't get immediately sent to memory.
- snooping cache coherence requires a broadcast every time a variable is updated. So snooping cache coherence isn't scalable, because for larger systems it will cause performance to degrade.

directory-based cache coherence.

- Directory-based cache coherence protocols attempt to solve this problem through the use of a data structure called a directory.

- The directory stores the status of each cache line.
- this data structure is distributed; each core/memory pair might be responsible for storing the part of the structure that specifies the status of the cache lines in its local memory.
- when a line is read into, say, core 0's cache, the directory entry corresponding to that line would be updated, indicating that core 0 has a copy of the line.
- When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable's cache line in their caches will invalidate those lines.
- additional storage is required for the directory, but when a cache variable is updated, only the cores storing that variable need to be contacted.

Q2.b. Explain non-determinism and race conditions in shared memory programs. How can they be avoided? . (10 marks)

non-determinism and race conditions- 5 marks

How can they be avoided- 5 marks

Non-Determinism

- Definition: Non-determinism occurs when a program can produce different outputs or behaviors across runs, even with the same input, due to unpredictable scheduling of concurrent threads/processes.
- Cause in Shared Memory: Multiple threads accessing and modifying shared variables without strict ordering. The operating system's scheduler decides which thread runs when, leading to different execution paths.

Race Conditions

- Definition: A race condition happens when the program's correctness depends on the timing or order of thread execution.
- Cause in Shared Memory: Two or more threads simultaneously read/write shared data, and the final outcome depends on which thread "wins the race."

Ways to ensure that an operation is atomic:

- Mutual exclusion lock or mutex,or lock.
- Busy-waiting
- Semaphores
- Monitor

Q3. a. Explain GPU programming in detail. (10 marks)

GPU Programming- 10 marks

- GPUs are usually not "standalone" processors. They don't ordinarily run an operating system and system services, such as direct access to secondary storage.
- So programming a GPU also involves writing code for the CPU "host" system, which runs on an ordinary CPU.
- The memory for the CPU host and the GPU memory are usually separate. So the code that runs on the host typically allocates and initializes storage on both the CPU and the GPU. It will start the program on the GPU, and it is responsible for the output of the results of the GPU program.
- Thus GPU programming is really heterogeneous programming, since it involves programming two different types of processors.

b. Describe input and output handling in MIMD and GPU systems.(10 marks)

input and output handling in MIMD systems - 5 marks

input and output handling in GPU systems- 5 marks

When multiple processes/threads can access input or output, the distribution of the input and the sequence of the output are usually nondeterministic.

For output, the data will probably appear in a different order each time the program is run, or, even worse, the output of one process/thread may be broken up by the output of another process/thread. For input, the data read by each process/thread may be different on each run, even if the same input is used.

To partially address these issues, following rules may help when our parallel programs need to do I/O:

• In distributed-memory programs, only process 0 will access input. In shared-memory programs, only the master thread or thread 0 will access input.

• because of the nondeterministic order of output, in most cases

only a single process/thread will be used for all output.

• Only a single process/thread will attempt to access any single file,

. So,each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.

• Debug output should always include the rank or ID of the process/thread that's generating the output.

GPU's

In most cases, the host code in GPU programs will carry out all I/O. Since we'll only be running one process/thread on the host, the standard C I/O functions should behave as they do in ordinary serial C programs. The exception to the rule that we use the host for I/O is that when we are debugging our GPU code, we'll want to be able to write to stdout and/or stderr. In the systems we use, each thread can write to stdout, and, as with MIMD programs, the order of the output is nondeterministic. Also in the systems we use, no GPU thread has access to stderr, stdin, or secondary storage.

Q4.a. Explain Amdahl's law with example and discuss its significance

Statement and formula – 4 marks

Example – 2 marks

Significance - marks

Back in the 1960s, Gene Amdahl made an observation that's become known as Amdahl's law. It says, that unless virtually all of a serial program is parallelized, the possible speedup is going to be very limited—regardless of the number of cores available. Amdahl's Law states that the maximum speedup of a program using multiple processors is limited by the portion of the program that cannot be parallelized. Even with infinite processors, the sequential part of the workload caps performance gains

Let:

P = fraction of the program that can be parallelized

1−P = fraction that must remain sequential

N = number of processors

The speedup formula is

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

Suppose, for example, that we're able to parallelize 90% of a serial program. Furthermore, suppose that the parallelization is "perfect," that is,

regardless of the number of cores p we use, the speedup of this part of the program will be p.

If the serial run-time is Tserial = 20 seconds,

then the run-time of the parallelized part will be 0.9 × Tserial/p = 18/p and

the run-time of the "unparallelized" part will be 0.1×Tserial = 2.

The overall parallel run-time will be

$$T_{\text{parallel}} = 0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}} = 18/p + 2,$$

and the speedup will be

$$S = \frac{T_{\text{serial}}}{0.9 \times T_{\text{serial}}/p + 0.1 \times T_{\text{serial}}} = \frac{20}{18/p + 2}.$$

Now as p gets larger and larger, 0.9 × Tserial/p = 18/p gets closer and closer to 0, so the total parallel run-time can't be smaller than 0.1 × Tserial = 2.

That is, the denominator in S can't be smaller than 0.1 × Tserial = 2.

The fraction S must therefore satisfy the inequality

$$S \leq \frac{T_{\text{serial}}}{0.1 \times T_{\text{serial}}} = \frac{20}{2} = 10.$$

S ≤ 10 => even though we've done a perfect job in parallelizing 90% of the program, and even if we have, say, 1000 cores, we'll never get a speedup better than 10.

if a fraction r of our serial program remains unparallelized, then Amdahl's law says we can't get a speedup better than 1/r.

In our example, r = 1 − 0.9 = 1/10, so we couldn't get a speedup better than 10.

If fraction r of our serial program is "inherently serial," that is, cannot possibly be parallelized, then we can't possibly get a speedup better than 1/r. Thus even if r is quite small—say, 1/100—and we have a system with thousands of cores, we can't possibly get a speedup better than 100.

b. Write a short note on timing and performance measurement of parallel programs. (10 marks)

5 points- 2 mars for each point

- Consider the reasons for taking timings: During program development, we may take timings to determine if the program is behaving as we intend. once we've completed development of the program, we're often interested in determining how good its performance is.
- Consider which portion of the program needs to be timed we're usually not interested in the time that elapses between the program's start and the program's finish. We're usually interested only in some part of the program. For example, if we write a program that implements bubble sort, we're probably only interested in the time it takes to sort the keys, not the time it takes to read them in and print them out.
- 3. Identify if clock time or CPU time is required we're usually not interested in "CPU time." This is the time reported by the standard C function clock. It's the total time the program spends in code executed as part of the program. It would not include time the program was idle, and this could be a problem. For example, in a distributed-memory program, a process that calls a receive function may have to wait for the sending process to execute the matching send, and the operating system might put the receiving process to sleep while it waits. This idle time wouldn't be counted as CPU time, since no function that's been called by the process is active. However, it should count in our evaluation of the overall run-time, since it may be a real cost in our program. If each time the program is run, the process has to wait, ignoring the time it spends waiting would give a misleading picture of the actual run-time of the program.
- Parallel programs need special handling while timing; When we're timing parallel programs, we need to be a little more careful about how the timings are taken. In our example, the code that we want to time is probably being executed by multiple processes or threads, and our original timing will result in the output of p elapsed times: However, what we're usually interested in is a single time: the time that has elapsed from when the first process/thread began execution of the code to the time the last process/thread finished execution of the code-the largest of the elapsed times
- Consider the variability in timings for each run: When we run a program several times, it's extremely likely that the elapsed time will be different for each run. This will be true, even if each time we run the program we use the same input and the same systems. It might seem that the best way to deal with this would be to report either a mean or a median run-time. However, it's unlikely that some outside event could actually make our program run faster than its best possible run-time. So instead of reporting the mean or median time, we usually report the minimum time.

Q5. a. Define and explain the following MPI functions with syntax and purpose:

MPI_Init(), MPI_Finalize(), MPI_Comm_size(), MPI_Comm_rank(). (10 marks)

2.5 marks for each function

Purpose – 1 mark each

Syntax- 1.5 marks each

MPI_Init tells the MPI system to do all of the necessary setup. For example, it might allocate storage for message buffers, and it might decide which process gets which rank. No other MPI functions should be called before the program calls MPI_Init.

Its syntax is:

int MPI_Init (

int ∗ argc_p /∗ in / out ∗/ ,

char ∗∗∗ argv_p /∗ in / out ∗/) ;

MPI_Finalize tells the MPI system that we're done using MPI, and that any resources allocated for MPI can be freed.

Syntax:

int MPI_Finalize (void) ;

In general, no MPI functions should be called after the call to MPI_Finalize.

MPI_Comm_size and MPI_Comm_rank:For both functions, the first argument is a communicator and has the special type defined by MPI for communicators, MPI_Comm. MPI_Comm_size returns in its second argument the number of processes in the communicator,

MPI_Comm_rank returns in its second argument the calling process's rank in the communicator.

```
int MPI_Comm_size(
      MPI_Comm   comm          /* in  */,
      int*       comm_sz_p     /* out */);

int MPI_Comm_rank(
      MPI_Comm   comm          /* in  */,
      int*       my_rank_p     /* out */);
```

Q5.b. Explain the concept of point to point communication in MPI with suitable example. (10 marks)

point to point communication- 2 marks

APIs like MPI_Send, MPI_Receive- 3 marks

Example MPI program – 5 marks

Point-to-point communication involves sending and receiving messages between two specific MPI processes.

MPI_Send and MPI_Recv are often called point-to-point communications.

Suppose process q calls MPI_Send with

MPI_Send (send_buf_p , send_buf_sz , send_type , dest , send_tag ,send_comm) ;

Also suppose that process r calls MPI_Recv with

MPI_Recv (recv_buf_p , recv_buf_sz , recv_type , src , recv_tag ,recv_comm , &status);

Then the message sent by q with the above call to MPI_Send can be received by r with

the call to MPI_Recv if

- recv_comm = send_comm,
- recv_tag = send_tag,
- dest = r, and
- src = q.

```
// a MPI Program to demonstration of MPI_Send and MPI_Recv.

#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 frfrom Process 0
            MPI_Recv(&number, 1, MPI_INT, 0, 0, PI_COMM_WORLD,MPI_STATUS_IGNORE);
            printf("Process 1 received number %d from Process 0\n",number);
        }
        // Finalize the MPI environment
        MPI_Finalize();

        return 0;
}
```

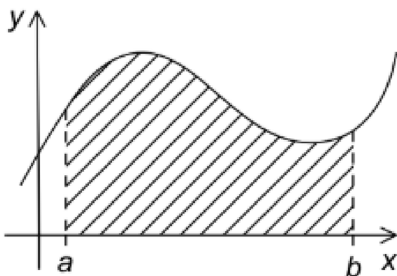
Q6.a. Explain the working of trapezoidal rule in MPI

Trapezoidal rule – 2 marks

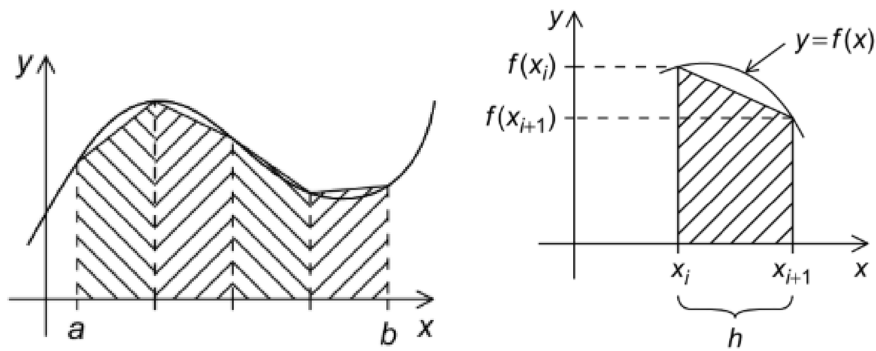
Approach used for parallelization- 2 marks

MPI implementation – 6 marks

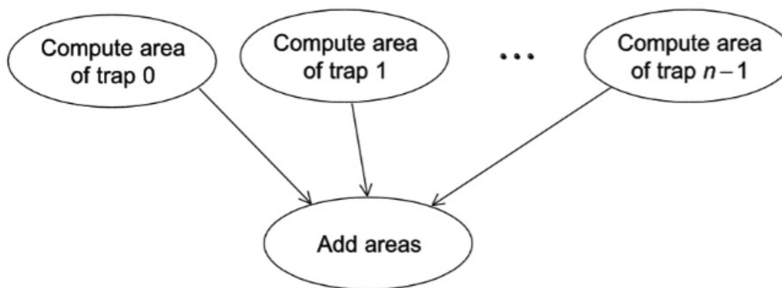
The trapezoidal rule is used to approximate the area between the graph of a function, y = f (x), two vertical lines, and the x-axis.



The basic idea is to divide the interval on the x-axis into n equal subintervals.



Then we approximate the area lying between the graph and each subinterval by a trapezoid, whose base is the subinterval, whose vertical sides are the vertical lines through the end-points of the subinterval, and whose fourth side is the secant line joining the points where the vertical lines cross the graph.



Parallelizing the trapezoidal rule-pseudocode

```
1     Get a, b, n;
2     h = (b-a)/n;
3     local_n = n/comm_sz;
4     local_a = a + my_rank*local_n*h;
5     local_b = local_a + local_n*h;
6     local_integral = Trap(local_a, local_b, local_n, h);
7     if (my_rank != 0)
8        Send local_integral to process 0;
9     else /* my_rank == 0 */
10       total_integral = local_integral;
11       for (proc = 1; proc < comm_sz; proc++) {
12          Receive local_integral from proc;
13          total_integral += local_integral;
14       }
15    }
16    if (my_rank == 0)
17       print result;
```

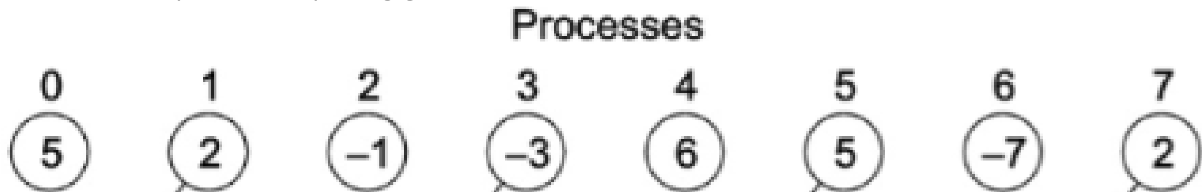
b. Compare the traditional global sum using process 0 as collector with tree structured global sum.

approach used for the traditional global sum using process 0 as collector-4 marks

approach used for the tree structured global sum- 4 marks

comparison of approaches in terms of performance – 2 marks

Consider example of computing global sum



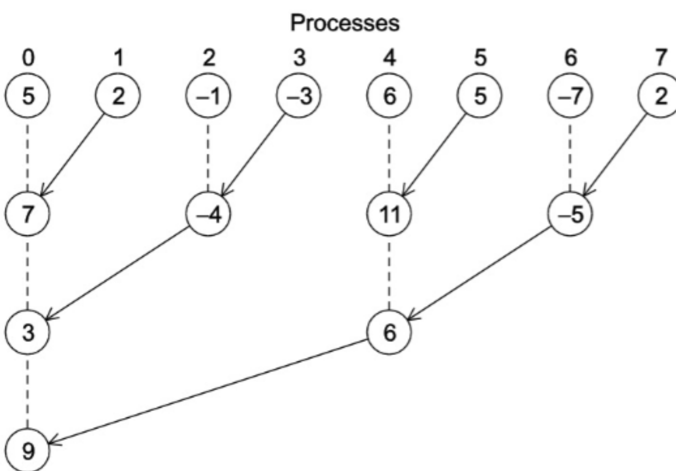
Each process with rank greater than 0 is, in effect, saying "add this number into the total."

Process 0 is doing nearly all the work in computing the global sum, while the other processes are doing almost nothing

Tree-structured communication:

Tree-structured communication arranges processes in a binary tree pattern to perform collective operations like reduce, broadcast, or gather.

Instead of all processes communicating directly with a root, data flows through intermediate nodes, reducing communication overhead.



Suppose we have 8 processes with ranks 0 to 7:

1. Level 1:
   - Processes 1, 3, 5, 7 send data to 0, 2, 4, 6 respectively.
   - These receivers combine their own data with the received data.
2. Level 2:
   - Processes 2 and 6 send their results to 0 and 4.
   - Again, receivers combine the data.
3. Level 3:
   - Process 4 sends its result to process 0.
   - Process 0 now has the final reduced result.

Tree-Structured Reduction — Efficiency Summary:

Work Distribution:

- Half the processes (1, 3, 5, 7) perform the same work as in the linear scheme.
- Process 0 now does only 3 receives and 3 additions, compared to 7 of each in the original linear method.

Concurrency:

- Multiple processes (0, 2, 4, 6) perform receives and additions simultaneously in the first phase.
- This parallelism significantly reduces the overall computation time.

Performance Gain:

- Since the final result depends only on process 0's reduced workload, the total time drops by more than 50% compared to the linear approach.

Q7.a. Explain the structure and working of an OpenMP "Hello world" program. (6 marks)

Header file- 1 mark

Use of pragma and display of thread id– 2 marks

Explanation- 3 marks

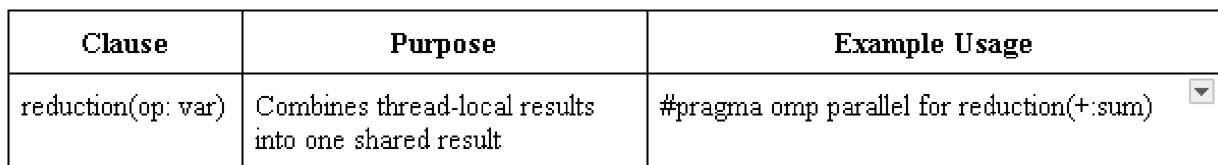
```
#include <stdio.h>

#include <omp.h>

int main()

{
```

```
 int n = 20;
omp_set_num_threads(n);
#pragma omp parallel
   printf("\n Hello World OMP from thread %d", omp_get_thread_num());
 return 0;
}
```

Q7.b. Explain the purpose of reduction clause in OpenMP with example. (6 marks)

purpose of reduction clause- 3 marks

example- 3 marks



Clause	Purpose	Example Usage
reduction(op: var)	Combines thread-local results into one shared result	#pragma omp parallel for reduction(+:sum)

```
#include <omp.h>
#include <stdio.h>
int main() {
  int A[10];
  for (int i = 0; i < 10; i++) A[i] = i;
  int sum = 0;
  #pragma omp parallel for reduction(+:sum)
  for (int i = 0; i < 10; i++) {
    sum += A[i];
  }
  printf("Sum = %d\n", sum);
  return 0;
```

}

Each thread gets a private copy of sum. After the loop, OpenMP automatically combines all partial sums.Thread-safe, no race conditions.

Q7.c. Write an OpenMP program to calculate n fibanocci using tasks. (8 marks)

Header file omp.h – 1 mark

Use of pragma omp task- 3 marks

Use of pragma omp taskwait- 2 marks

Fibonacci logic- 1 marks

Use of pragma omp single- 1 marks

```
#include <stdio.h>

#include <omp.h>

#include <stdio.h>

#include <omp.h>

#include <time.h>


// Serial Fibonacci calculation function

long int ser_fib (long int n)

{

        if (n < 2) return n;

        long int x, y;

        x = ser_fib(n - 1);

        y = ser_fib(n - 2);

        return x + y;

}
```

```
// parallel Fibonacci calculation function
int fib(long int n)
{
        int x, y;
        if (n < 2) return n;
        else
        {
        #pragma omp task shared(x) firstprivate(n)
                x = fib(n - 1);
        #pragma omp task shared(y) firstprivate(n)
                y = fib(n - 2);
        //printf("\nParallel %ld %ld",x, y);
        #pragma omp taskwait
                return x + y;
    }
}


int main()
{
        long int n1 = 10, n2= 10, result1, result2;
        clock_t start, end;
        double cpu_time;
        omp_set_dynamic(0);
        omp_set_num_threads(4);
        printf("\n enter the value of n");
```

```
        scanf("%ld",&n1);
        n2= n1;
        start=clock();
        result1 = ser_fib(n1);
        end=clock();
        cpu_time = ((double) (end - start)) / CLOCKS_PER_SEC;

        printf("\nSerial Fibonacci(%ld) = %ld",n1, result1);
        printf("\n the time used to execute the program in sequential mode= %f\n",cpu_time);

        start=clock();
        #pragma omp parallel shared(n2)
        {
                #pragma omp single
                        result2 = fib(n2);
        }
        end=clock();
        cpu_time = ((double) (end - start)) / CLOCKS_PER_SEC;
        printf("\nParallel Fibonacci(%ld) = %ld\n", n2, result2);
        printf("\n the time used to execute the program in parallel mode= %f\n",cpu_time);
        return 0;
}
```

Q8.a. Define OpenMP. Explain the key features of OpenMP and its advantages over Pthreads.(6 marks)

OpenMP definition- 2 marks

key features of OpenMP – 2 marks

advantages over Pthreads- 2 marks

OpenMP is an API for shared-memory MIMD programming. The "MP" in OpenMP stands for "multiprocessing," a term that is synonymous with shared-memory MIMD computing. OpenMP is designed for systems in which each thread or process can potentially have access to all available memory. With OpenMP, our system is a collection of autonomous cores or CPUs, all of which have access to main memory.

Key Features of OpenMP

- Directive-based model: Uses compiler directives (#pragma omp) to parallelize code, making it easier to add/remove parallelism without rewriting large portions of code.
- Shared-memory parallelism: Designed for systems where threads share memory space, avoiding explicit message passing.
- Incremental parallelization: Developers can start with sequential code and gradually add parallel directives.
- Portability: Supported by most major compilers (GCC, Intel, Clang, MSVC).
- Scalability: Automatically manages thread creation, scheduling, and workload distribution.
- Synchronization constructs: Provides high-level constructs like barriers, critical sections, locks, and atomic operations.
- Tasking model: Supports dynamic task creation and execution, useful for irregular workloads.
- Environment control: Runtime library routines and environment variables allow tuning (e.g., number of threads, scheduling policy).

OpenMP is higher-level, easier, and faster for shared-memory parallel programming, especially when you want to parallelize existing sequential code incrementally.

Pthreads gives fine-grained control but at the cost of complexity, verbosity, and potential for bugs. It's better suited when you need explicit control over threads, synchronization, or portability across non-OpenMP environments.

Q8.b. Explain the concept of variable scope in OpenMP with suitable example. (6 marks)

scope of a variable in OpenMP- 2 marks

explanation of different scopes- shared, private, firstprivate – 4 marks

In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.

A variable that can be accessed by all the threads in the team has shared scope, while a variable that can only be accessed by a single thread has private scope.

Variables that have been declared before a parallel directive have shared scope among the threads in the team, while variables declared in the block (e.g., local variables in functions) have private scope.

private(varlist):Each thread gets its own uninitialized copy of the listed variables

shared(varlist):Variables are shared among all threads

firstprivate(varlist):Each thread gets its own copy, initialized from the master thread

```
#include <stdio.h>

#include <omp.h>

int main() {

    int shared_var = 10;   // Shared by default

    int private_var = 5;   // Will be privatized

    int sum = 0;           // Used in reduction

    #pragma omp parallel private(private_var) firstprivate(shared_var) reduction(+:sum)

    {

        int tid = omp_get_thread_num();

        // private_var is unique per thread, not initialized

        private_var = tid;

        // shared_var is copied into each thread (firstprivate)

        printf("Thread %d: private_var=%d, shared_var=%d\n", tid, private_var, shared_var);

        // reduction variable accumulates across threads

        sum += tid;

    }
```

```
    printf("Final sum = %d\n", sum);

    return 0;

}
```

Q8.c. Estimate the value of Pi. (8 marks)

Formula for Pi- 3 marks

Parallelization logic using pragma- 2 marks

Avoidance of race condition and loop carried dependency- 3 marks

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
    sum += factor/(2*k+1);
    factor = -factor;
}
pi_approx = 4.0*sum;
```

OpenMP solution

```
double sum = 0.0;
pragma omp parallel for num_threads(thread_count) \
    reduction(+:sum) private(factor)
for (k = 0; k < n; k++) {
    if (k % 2 == 0)
       factor = 1.0;
    else
       factor = -1.0;
    sum += factor/(2*k+1);
}
```

Insures factor has private scope.

Q9. a. Define GPU and GPGPU. Explain need for GPGPU. (6 marks)

Definition of GPU – 2 marks

Definition of GPGPU– 2 marks

need for GPGPU– 2 marks

In response to growing demand for realistic video games and animations in the late 1990s and early 2000s, the computer industry developed powerful Graphics Processing Units (GPUs)

These specialized processors were designed to enhance the performance of programs that require rendering large numbers of detailed images efficiently

By the early 2000s programmers were trying to apply the power of GPUs to solving general computational problems, problems such as searching and sorting, rather than graphics.

GPGPU transforms the GPU from a graphics-only engine into a high-performance parallel processor.

Feature	CPU	GPU
Core Count	Few powerful cores	Thousands of lightweight cores
Processing Style	Sequential	Massively parallel
Ideal For	Control logic, branching	Data-parallel tasks

Q9. b. Explain thread, block and grid in CUDA. (6 marks)

Thread– 2 marks

Block– 2 marks

grid– 2 marks

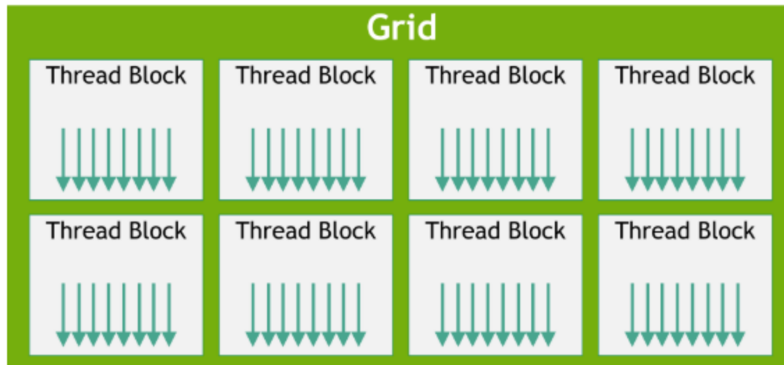
CUDA organizes threads into blocks and grids.

A thread block is a collection of threads that run on a single SM.

When the kernel is started, each block is assigned to an SM, and the threads in the block are then run on that SM.

A grid is the collection of thread blocks started by a kernel.

thread block is composed of threads, and a grid is composed of thread blocks.



There are several built-in variables that a thread can use to get information on the grid started by the kernel.

The following four variables are structs that are initialized in each thread's memory when a kernel begins execution:

threadIdx: the rank or index of the thread in its thread block

blockDim: the dimensions, shape, or size of the thread blocks

blockIdx: the rank or index of the block within the grid

gridDim: the dimensions,shape,or size of the grid

Q9. c. Explain cuda vector addition program with suitable example. (8 marks)

Kernel definition- 3 marks

Kernel invocation- 2 marks

Vector addition logic- 3 marks

```
#include <stdio.h>

#include <cuda.h>


__global__ void vectorAdd(float *a, float *b, float *c, int n) {
```

```
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        c[i] = a[i] + b[i];
    }
}
int main() {
    int N = 1 << 20; // 1 million elements
    float *a, *b, *c;
    // Allocate Unified Memory
    cudaMallocManaged(&a, N * sizeof(float));
    cudaMallocManaged(&b, N * sizeof(float));
    cudaMallocManaged(&c, N * sizeof(float));
    // Initialize input vectors
    for (int i = 0; i < N; ++i) {
        a[i] = i;
        b[i] = 2*i ;
          }
    // Launch kernel with enough threads
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
     vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c, N);
    // Wait for GPU to finish
   cudaDeviceSynchronize();
    // Verify results
    for (int i = 0; i < 10; ++i) {
        printf("\n %f + %f = %f",a[i], b[i],c[i]);
```

```
    }
    // Free memory
    cudaFree(a);
    cudaFree(b);
    cudaFree(c);
    return 0;
}
```

Q10.a. Compare CUDA and OpenCL. (6 marks)

3 points- 2 marks each

- Performance: On NVIDIA GPUs, CUDA usually outperforms OpenCL because it is tightly integrated with NVIDIA's drivers and hardware optimizations.
- Portability: If your application must run on multiple platforms (e.g., AMD GPUs, CPUs, or Apple devices), OpenCL is the safer choice.
- Ecosystem: CUDA has a strong AI/ML ecosystem (TensorFlow, PyTorch, RAPIDS), making it the go-to for deep learning and HPC workloads.
- Learning Curve: CUDA is easier to learn due to its extensive documentation and libraries, while OpenCL requires more boilerplate code and vendor-specific tuning.
- CUDA Lock-in: Choosing CUDA ties you to NVIDIA hardware. If your deployment environment changes (e.g., AMD GPUs in cloud clusters), you'll face migration challenges.
- OpenCL Fragmentation: While portable, OpenCL's performance and tooling vary widely across vendors, leading to inconsistent developer experience.
- Longevity: CUDA evolves rapidly with NVIDIA's hardware releases, while OpenCL's updates are slower and depend on multiple vendors.

Q10.b. Explain kernel with shared memory. (6 marks)

Shared memory Approach using arrays- 3 marks

Explanation of accessing shared memory – 3 marks

__shared__: used to declared shared memory

Shared memory has much lower latency than global memory.

Vector Addition with Shared Memory

```
#include <stdio.h>

__global__ void vectorAddShared(int *a, int *b, int *c, int n) {

   // Declare shared memory for this block

   __shared__ int tempA[256];

   __shared__ int tempB[256];

   int tid = threadIdx.x + blockIdx.x * blockDim.x;

   if (tid < n) {

      // Each thread loads its data into shared memory

      tempA[threadIdx.x] = a[tid];

      tempB[threadIdx.x] = b[tid];

      // Synchronize to ensure all threads have finished loading

      __syncthreads();

      // Perform computation using shared memory

      c[tid] = tempA[threadIdx.x] + tempB[threadIdx.x];

   }

}


int main() {

   const int N = 256;

   int h_a[N], h_b[N], h_c[N];

   int *d_a, *d_b, *d_c;
```

```
// Initialize input arrays
for (int i = 0; i < N; i++) {
   h_a[i] = i;
   h_b[i] = i * 2;
}

// Allocate GPU memory
cudaMalloc((void**)&d_a, N * sizeof(int));
cudaMalloc((void**)&d_b, N * sizeof(int));
cudaMalloc((void**)&d_c, N * sizeof(int));
// Copy data to GPU
cudaMemcpy(d_a, h_a, N * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, N * sizeof(int), cudaMemcpyHostToDevice);
// Launch kernel with 1 block of 256 threads
vectorAddShared<<<1, 256>>>(d_a, d_b, d_c, N);
// Copy result back to CPU
cudaMemcpy(h_c, d_c, N * sizeof(int), cudaMemcpyDeviceToHost);
// Print results
for (int i = 0; i < 10; i++) {
   printf("%d + %d = %d\n", h_a[i], h_b[i], h_c[i]);
}
// Free GPU memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

```
    return 0;
}
```

Q10.c. Explain heterogeneous computing in detail. (8 marks)

heterogeneous computing-4 marks

why CUDA programming is heterogeneous- 4 marks

Heterogeneous computing refers to systems that use multiple types of processors (e.g., CPUs, GPUs, FPGAs, DSPs) working together to execute tasks. Instead of relying on a single type of processor, heterogeneous systems exploit the strengths of each to achieve better performance, efficiency, and flexibility.

The API we used for GPGPU programming is called CUDA,and it is an extension of C/C++. It assumes that the system has both a CPU and a GPU: the main function runs on the CPU a kernel is a CUDA function that is called by the host but that runs on the GPU or device. So a CUDA program ordinarily runs on both a CPU and a GPU, and since these have different architectures, writing a CUDA program is sometimes called heterogeneous programming.