USN										

CMR INSTITUTE OF TECHNOLOGY, BENGALURU.

Internal Assessment Test 1 – Sept. 2025

Subject/Code : Parallel computing/ BCS702				
Date: 26/9/25	Duration: 90 mins	Max. Marks:50	Semester: 07	Branch: CSE

Sl.	**Answer any FIVE FULL Questions**	Marks	CO	RBT
1a)	Explain race condition in the context of OpenMP programming.	[02]	4	L2
b)	Illustrate any one approach to avoid race condition using an Open MP program that finds the sum of all elements in an array of 1000 elements	[08]	4	L3

2a)	For the following code segment, use OpenMP pragmas to make the loop parallel
	for (int i = 0; i < (int) sqrt(x); i++) {
	    a[i] = 2.3 * i;
	    if (i < 10)
	        b[i] = a[i];
	}
		[02]	4	L3

b)	Explain the usage of the pragmas- omp task and omp barrier	[04]	4	L2
c)	Explain cache coherence and the problem of false sharing in Open MP programs.	[04]	4	L2
3a)	Illustrate and explain the difference between static and dynamic scheduling.	[04]	4	L3

b)	Describe how loop iterations are mapped to threads in the OpenMP trapezoidal rule implementation. Also write Open MP parallel code for the following serial version of trapezoidal rule.
	h = (b-a)/n;
	approx = (f(a)+f(b))/2.0;
	for(i=1;i<= n-1; i++){
	    x_i = a + i*h;
	    approx +=f(x_i);
	}
	approx. = h*approx;
		[06]	4	L3

4a)	Explain the purpose and syntax of MPI Scatter and MPI Gather.	[06]	3	L2
b)	Explain the purpose and syntax of MPI_Conn_rank and MPI_Comm_size	[04]	3	L2
5a)	Write a MPI program to broadcast an integer value 50 from process 1 in a communicator with size of 5.	[05]	3	L3
b)	Explain the need for derived data types in MPI communication. Describe the function and its arguments used to create a derived data type.	[05]	3	L2
6a)	Illustrate the approach used in Odd-Even Transposition Sort using a small list of 5 elements.	[05]	3	L3
b)	Explain how the Odd-Even Transposition Sort approach can be parallelized and why bubble sort cannot be parallelized.	[05]	3	L2

Answer Key and Scheme

1a) Explain race condition in the context of OpenMP programming. (2 marks)
When threads attempt to simultaneously access a shared resource (memory location), and the accesses can result in an error, we often say the program has a race condition, because the threads are in a "race" to carry out an operation. That is, the outcome of the computation can be unpredictable.
explanation of race condition- 1 mark
consequence of race condition- 1 mark

1b) Illustrate any one approach to avoid race condition using an Open MP program that finds the sum of all elements in an array of 1000 elements. (8 marks)
Approach can be either using omp atomic or omp critical or reduction clause
correct syntax for pragmas- 5 marks
mentioning header file mpi.h – 1 mark
calculating sum of all elements in an array of 1000 elements – 2 marks

A sample program osing reduction clause is as follows

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

int main() {
   int A[1000];
   for (int i = 0; i < 1000; i++) A[i] = i;

   int sum = 0;

   #pragma omp parallel for reduction(+:sum)
   for (int i = 0; i < 1000; i++) {
      sum += A[i];
   }

   printf("Sum = %d\n", sum);
   return 0;
}
```

2a) For the following code segment, use OpenMP pragmas to make the loop parallel

```
for (int i = 0; i < (int) sqrt(x); i++) {
   a[i] = 2.3 * i;
   if (i < 10)
      b[i] = a[i];
}
```

(2 marks)

```
# pragma omp parallel for
for (int i = 0; i < (int) sqrt(x); i++) {
   a[i] = 2.3 * i;
   if (i < 10)
      b[i] = a[i];
}
```

2b) Explain the usage of the pragmas- omp task and omp barrier. (4 marks)
Explanation of each pragma- 2 marks
Syntax/usage of each pragma – 2 marks
pragma omp task: This directive is used to define a task—a unit of work that can be executed independently and potentially in parallel.
```
#pragma omp parallel
{
   #pragma omp single
   {
      for (int i = 0; i < 5; i++) {
         #pragma omp task
         {
            printf("Task %d executed by thread %d\n", i, omp_get_thread_num());
         }
      }
   }
}
```

pragma omp barrier: This directive creates a synchronization point where all threads must wait until every thread reaches the barrier.
```
#pragma omp parallel
{
   printf("Thread %d before barrier\n", omp_get_thread_num());

   #pragma omp barrier

   printf("Thread %d after barrier\n", omp_get_thread_num());
}
```

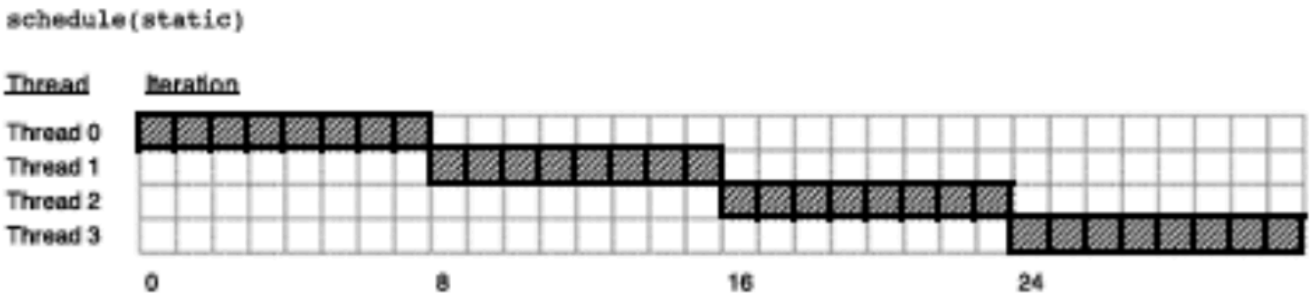
2c) Explain cache coherence and the problem of false sharing in Open MP programs. (4 marks)
Cache coherence – 2 marks
False sharing– 2 marks

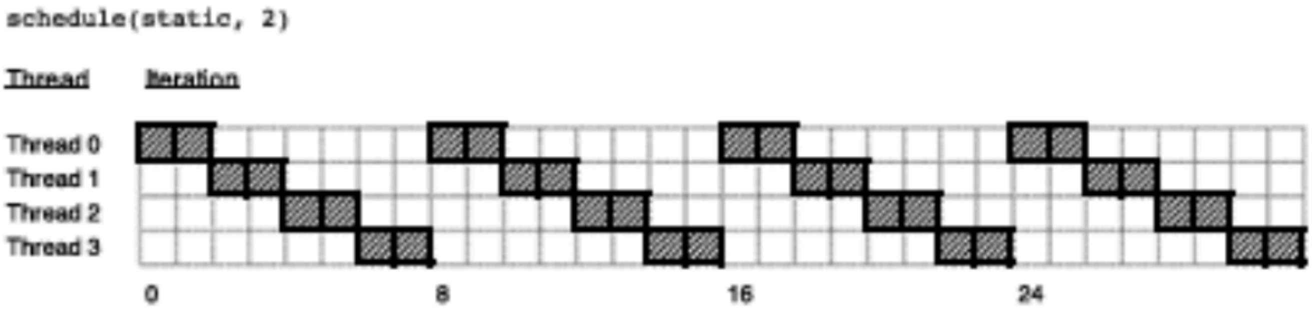
CPUs use caches to speed up memory access. In a multi-core system, each core typically has its own cache. Cache coherence ensures that all cores see a consistent view of memory. If Thread A updates a variable in its cache, Cache coherence ensures that Thread B must see that update if it accesses the same variable.

False sharing occurs when threads access different variables that happen to reside on the same cache line. Even though the variables are independent, the cache system treats them as shared. If arr[0] and arr[1] are on the same cache line, both threads will cause cache invalidations—even though they're not sharing data logically. This results in performance overhead due to unnecessary cache coherence traffic.

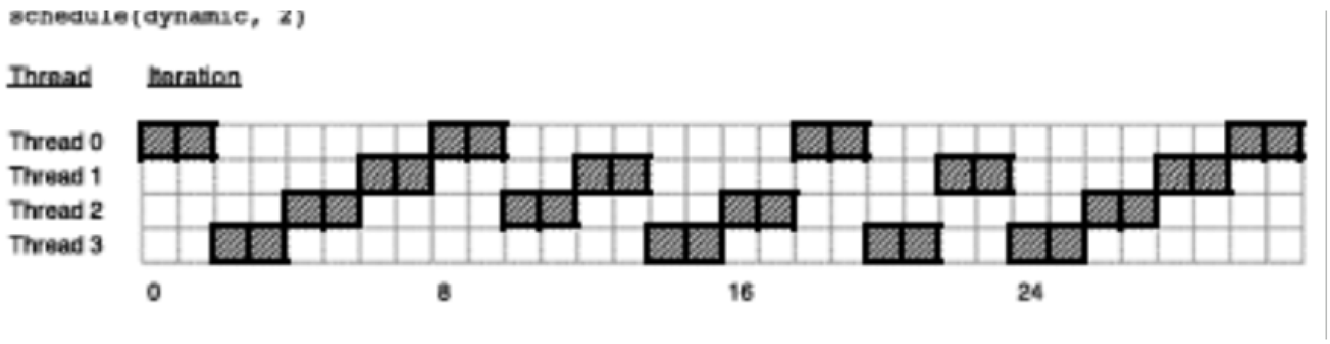
3 a) Illustrate and explain the difference between static and dynamic scheduling. (4 marks)
Explanation – 1 mark each
Illustration – 1 mark each

static scheduling: Iterations are divided among threads before execution
```
#pragma omp for schedule(static)
```





dynamic scheduling : Iterations are assigned during execution as threads become available
#pragma omp for schedule(dynamic)



3 b) Describe how loop iterations are mapped to threads in the OpenMP trapezoidal rule implementation. Also write Open MP parallel code for the following serial version of trapezoidal rule.

```
h = (b-a)/n;
approx = (f(a)+f(b))/2.0;
for(i=1;i<= n-1; i++){
    x_i = a + i*h;
    approx +=f(x_i);
}
approx. = h*approx;
```

(6 marks)
how loop iterations are mapped to threads- 3 marks
parallel version of trapezoidal rule- 3 marks

Suppose you want to integrate a function over $[a,b]$ $[a,b]$ using n n trapezoids and T T threads. You divide the interval into T T subintervals:
Each thread gets:
local_a = a + thread_id * (b - a) / T
local_b = local_a + (b - a) / T
local_n = n / T (assuming n is divisible by T)
Each thread then applies the trapezoidal rule over its own subinterval.

Open MP parallel code for the serial version of trapezoidal rule.

```
double a = 0.0, b = 1.0;
int n = 1000000;
double h = (b - a) / n;
double total_sum = 0.0;

#pragma omp parallel
{
   int thread_id = omp_get_thread_num();
   int num_threads = omp_get_num_threads();

   double local_a = a + thread_id * (b - a) / num_threads;
   double local_b = local_a + (b - a) / num_threads;
   int local_n = n / num_threads;
   double local_h = (local_b - local_a) / local_n;

   double local_sum = (f(local_a) + f(local_b)) / 2.0;
   for (int i = 1; i < local_n; i++) {
      double x = local_a + i * local_h;
      local_sum += f(x);
   }
   local_sum *= local_h;

   #pragma omp atomic
   total_sum += local_sum;
}
```

Following approach can also be given marks:
Open MP parallel code for the serial version of trapezoidal rule.

```
h = (b-a)/n;
approx = (f(a)+f(b))/2.0;
#pragma omp parallel for rduction(+:approx.)
for(i=1;i<= n-1; i++){
    x_i = a + i*h;
    approx +=f(x_i);
}
approx. = h*approx;
```

4a) Explain the purpose and syntax of MPI Scatter and MPI Gather. (6 marks)

purpose – 1 mark each
syntax – 2 marks each

MPI_Scatter sends chunks of an array to different processes.

```
int MPI_Scatter(
void *              send_buf_p     /*  in  */,
int                 send_count    /*  in  */,
MPI_Datatype  send_type      /*  in  */,
void*               recv_buf_p     /*  out  */,
int                 recv_count    /*  in  */,
MPI_Datatype  recv_type      /*  in  */,
int                 src_proc      /*  in  */,
MPI_Comm        comm         /*  in  */,

)
```

MPI_Gather takes elements from many processes and gathers them to one single process.

```
int MPI_Gather(
void *          send_buf_p    /* in */,
int             send_count    /* in */,
MPI_Datatype    send_type     /* in */,
void*           recv_buf_p    /* out */,
int             recv_count    /* in */,
MPI_Datatype    recv_type     /* in */,
int             dest_proc     /* in */,
MPI_Comm        comm          /* in */,
)
```

4b) Explain the purpose and syntax of MPI_Conn_rank and MPI_Comm_size (4 marks)

purpose – 1 mark each

syntax – 1 mark each

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 */);
```

5a) Write a MPI program to broadcast an integer value 50 from process 1 in a communicator with size of 5. (5 marks)

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char** argv)
{
        int rank, data = 0;
        MPI_Init(&argc, &argv);
        // Initialize the MPI environment
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        // Get the rank of the process

        if (rank == 1){
                data = 50;
        // Root process sets the data
                printf("Process %d is broadcasting data = %d\n", rank, data);
        }
        MPI_Bcast(&data, 1, MPI_INT, 1, MPI_COMM_WORLD); // Broadcast data from root to all
        printf("Process %d received data: %d\n", rank, data); // All processes print the data

        MPI_Finalize();

        return 0;
}
```

5b) Explain the need for derived data types in MPI communication. Describe the function and its arguments used to create a derived data type. (5 marks)

In distributed-memory systems, communication can be much more expensive than local computation. The cost of sending a fixed amount of data in multiple messages is usually much greater than the cost of sending a single message with the same amount of data.

Derived data types in MPI communication is an approach to consolidating data that might otherwise require multiple messages, hence a single message can be sent instead.

```
int MPI_Type_create_struct(
      int            count                       /* in   */,
      int            array_of_blocklengths[]     /* in   */,
      MPI_Aint       array_of_displacements[]    /* in   */,
      MPI_Datatype   array_of_types[]            /* in   */,
      MPI_Datatype*  new_type_p                  /* out  */);
```

count: the number of elements in the datatype. Each of the array arguments should have count elements
array_of_block_lengths: allows for the possibility that the individual data items might be arrays or subarrays
array_of_displacements specifies the displacements in bytes, from the start of the message
array_of_datatypes should store the MPI datatypes of the elements
new_type_p is the name of the derived data type

6a) Illustrate the approach used in Odd-Even Transposition Sort using a small list of 5 elements. (5 marks)

Start :5, 9, 4, 3, 1

Even phase: Compare-swap (5,9) and (4,3),
getting the list 5, 9, 3, 4, 1
Odd phase: Compare-swap (9, 3), (4,1)
getting the list 5, 3, 9, 1, 4
Even phase: Compare-swap (5,3), and (9,1)
getting the list 3, 5, 1, 9, 4
Odd phase: Compare-swap (5, 1), (9,4)
getting the list 3, 1, 5, 4, 9
Even phase: Compare-swap (3, 1), and (5, 4)
getting the list 1,3, 4, 5, 9
Sorted

6b) Explain how Odd-Even Transposition Sort approach can be parallelized and why bubble sort cannot be parallelized. (5 marks)

how Odd-Even Transposition Sort approach can be parallelized- 3 marks
why bubble sort cannot be parallelized- 2 marks

Odd-Even Transposition Sort operates in phases, alternating between:

- Odd phase: Compare and swap elements at indices (1,2), (3,4), (5,6)...
- Even phase: Compare and swap elements at indices (0,1), (2,3), (4,5)...

This ensures that adjacent elements are sorted over multiple passes. After n phases (for n elements), the array is guaranteed to be sorted.

The parallel algorithm- in simple:

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

Compute_partner

```
if (phase % 2 == 0)          /* Even phase */
   if (my_rank % 2 != 0)     /* Odd rank */
      partner = my_rank − 1;
   else                      /* Even rank */
      partner = my_rank + 1;


else                         /* Odd phase */
   if (my_rank % 2 != 0)     /* Odd rank */
      partner = my_rank + 1;
   else                      /* Even rank */
      partner = my_rank − 1;
if (partner == −1 || partner == comm_sz)
   partner = MPI_PROC_NULL;
```

Considering safety in MPI, implement the send and the receive with a single call to MPI_Sendrecv:

```
MPI_Sendrecv(my_keys, n/comm_sz, MPI_INT, partner, 0,
      recv_keys, n/comm_sz, MPI_INT, partner, 0, comm,
      MPI_Status_ignore);
```

Bubble sort repeatedly compares adjacent elements and swaps them if they're out of order. After each pass, the largest unsorted element "bubbles up" to its correct position. Each comparison depends on the result of the previous one. Swapping arr[j] and arr[j+1] affects the next comparison involving arr[j+1] and arr[j+2].