CT-1 Solutions

By Madhurima Rawat

UNIT 1

1. Primary Purpose of High Performance Computing (HPC)

Answer: B. To solve large-scale, computationally intensive problems

2. Fastest Level of Memory Hierarchy

Answer: C. Cache memory

3. Multi-core Processors Primarily Enable

Answer: B. Parallel processing of tasks

4. Parallel Algorithm Description

Answer: B. Divides a problem into sub-tasks that can be executed concurrently

5. Significance of Parallel Programming and Parallel Algorithms in HPC

- **Need for HPC:** It enables solving complex and large-scale scientific, engineering, and commercial problems that are infeasible for traditional computers due to time and resource constraints.
- **Key Concepts:** Parallelism (data/task), concurrency, synchronization, and communication between processors.
- **Challenges:** Load balancing, minimizing communication overhead, avoiding race conditions, and managing synchronization.
- **Parallel Algorithms Improve Performance:** By executing tasks simultaneously on multiple processors, reducing total execution time compared to sequential algorithms.
- **Implications of Multi-core and Vector Computing:** Increased throughput in scientific simulations, real-time data processing, and applications like weather modeling and genomics.

6. Amdahl's Law Calculation

Problem: 90% parallelizable code, 10% sequential, 4 processors.

```
Speedup = 1 / [ (1 - P) + (P / N) ]
P = 0.9
N = 4
Sequential fraction = 0.1

Speedup = 1 / [0.1 + (0.9 / 4)] = 1 / (0.1 + 0.225) = 1 / 0.325 ≈ 3.08
```

Answer: Theoretical Maximum Speedup ≈ 3.08

7. Pseudocode for Parallel Algorithm

Problem: Parallel Sum of an Array of 10,000 Numbers

```
Initialize array[10000]
Divide array into 4 chunks (chunk_size = 2500)
For each processor P_i (i = 1 to 4):
  sum_i = sum(array[start_i : end_i])

Combine all sum_i into total_sum
Output total_sum
```

Task Division: Each processor handles 1/4th of the data ensuring balanced workload, reducing execution time by approximately 4x in ideal conditions.

UNIT 2

8. OpenMP is Primarily Used for

Answer: B. Shared-memory parallel programming

9. OpenMP Directive to Parallelize Loops

Answer: A. `#pragma omp parallel for`

10. Common Challenge with OpenMP

Answer: B. Data race conditions

11. MPI is Best Suited for

Answer: B. Distributed-memory systems

12. Fundamental Differences Between OpenMP and MPI

⚙️ Parameter	🟦 OpenMP	🟥 MPI
Memory Model 🧠	Shared Memory	Distributed Memory
Parallelism 🔗	Thread-based (Multithreading)	Process-based (Multiprocessing)
Communication 📨	Implicit (via shared variables)	Explicit (message passing)
Scalability 📈	Limited to single node/multicore systems	Scales across multiple nodes/computers
Ease of Use 👌	Easy (uses compiler directives)	Complex (requires explicit communication code)
Fault Tolerance ⚠️	Low (shared memory issues)	High (failure in one node doesn't crash all)
Use Case 💡	Multicore CPUs, shared memory systems	Clusters, supercomputers, distributed systems
Programming Language Support 🛠️	C, C++, Fortran (with compiler support)	C, C++, Fortran, Python (library-based)

13. OpenMP Parallel Sum Code Snippet 🧮

📝 **Code:**

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

int main() {
  int n = 10000;
int array[n], sum = 0;

// Initialize array
for (int i = 0; i < n; i++)
    array[i] = 1;

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

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

🧐 **Explanation:**

- ✅ **Shared Memory Model:** All threads share the same array.
- ✅ **Parallel For Loop:** Iterations split among threads automatically.
- ✅ **Reduction(+):** Safely adds each thread's partial sum into the final result.
- ✅ **Result:** Sum of all array elements printed at the end.

📋 **Pseudocode:**

```
Start
Initialize array of size 10000 with 1
Set sum = 0

Parallel For with reduction(+)
    sum = sum + array[i]
End Parallel

Print sum
End
```

14. How OpenMP Reduces Execution Time 🕒

Answer: OpenMP reduces execution time by dividing the loop iterations among multiple threads. It exploits multi-core architectures where each thread executes a portion of the workload simultaneously, significantly reducing the total execution time compared to sequential processing.

🚀 1. What is OpenMP?

OpenMP (Open Multi-Processing) is an API that supports **multi-threaded parallel processing** on shared memory architectures. It allows developers to write parallelized versions of code easily, especially loops and regions that process large data.

⚙️ 2. How Execution Time is Reduced

✅ Concept	🔍 Explanation
Parallelism	OpenMP divides a task (like a loop) into smaller sub-tasks. Each sub-task runs in parallel on different CPU cores.
Shared Memory Model	All threads share the same address space (memory), so data does not need to be transferred between processes.
Thread Management	OpenMP manages threads automatically, distributing workload efficiently among them.
Work Sharing Constructs	Constructs like `#pragma omp parallel for` allow multiple threads to work simultaneously, reducing the time it would take if only one thread were working.
Reduction Operations	Operations like `sum += array[i]` use reduction to combine results from multiple threads without conflict or errors.

⏱️ 3. Time Comparison: Sequential vs Parallel

🖥️ Sequential Processing	🧵 Parallel Processing (OpenMP)
Single core/thread does all the work.	Multiple cores/threads do parts of the work at the same time.
Long execution time for large tasks.	Reduced execution time because the workload is divided.
CPU utilization is low (1 core).	CPU utilization is high (many cores).

📈 4. OpenMP in Action

Imagine a loop that runs 1000 iterations sequentially.

- On a single core, it might take **10 seconds**.
- Using OpenMP on a quad-core CPU, the same loop can be divided like this:

```
Core 1 → 0 to 249
Core 2 → 250 to 499
Core 3 → 500 to 749
Core 4 → 750 to 999
```

✅ So, instead of 10 seconds, you get close to **2.5 seconds** total!
(This is simplified—actual speedup depends on overhead and efficiency.)

🏗️ 5. Process Flow Diagram (Text Version)

```
Sequential:
Main Thread → Task 1 → Task 2 → Task 3 → Task 4 → Done!

Parallel (OpenMP):
Thread 1 → Task 1          ↘
Thread 2 → Task 2           ↘
Thread 3 → Task 3             → Merge Results → Done!
Thread 4 → Task 4          ↗
```

🔨 6. Example Parallel Region

```
#pragma omp parallel
{
    int thread_id = omp_get_thread_num();
    printf("Thread %d is working!\n", thread_id);
}
```

Each thread runs the block of code inside the `parallel` region simultaneously, using its own CPU core.

📋 7. Key Takeaways

- ✅ OpenMP splits tasks over multiple CPU cores, doing work in parallel.
- ✅ Less waiting → faster execution.
- ✅ Best for tasks that can be broken into independent parts (like loops over arrays).

15. MPI Matrix Multiplication Execution Time

Computation Time: 50 seconds

Communication Overhead Per Process: 8 seconds

Total Overhead: 4 x 8 = 32 seconds

```
Total Execution Time = Computation Time + Communication Overhead
= 50 + 32 = 82 seconds
```

Answer: 82 seconds