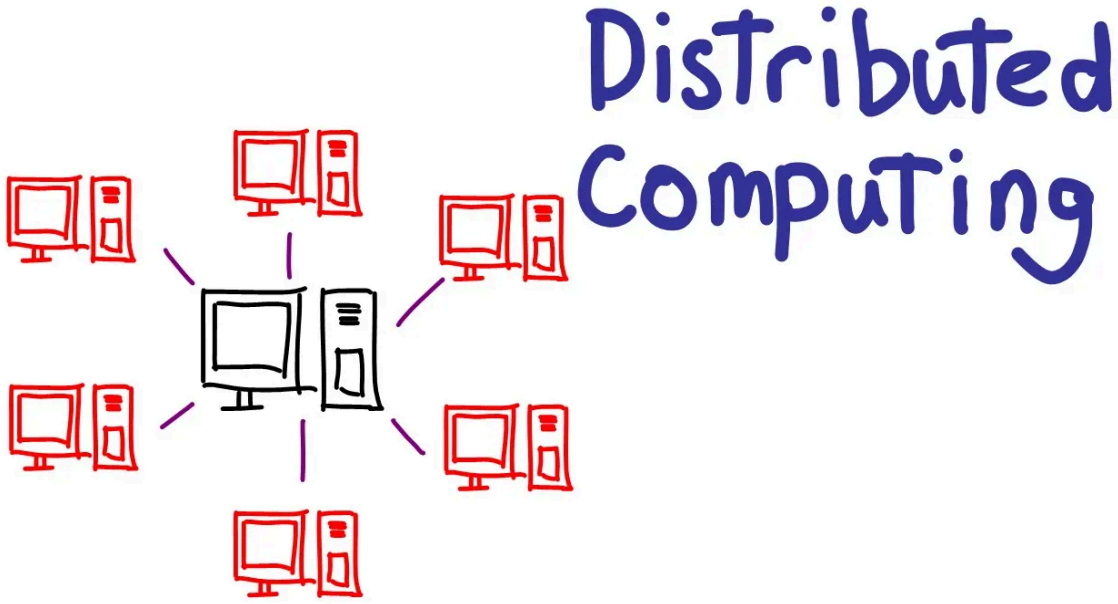
DISTRIBUTED COMPUTING

Unit-1

DISTRIBUTED COMPUTING:

Distributed computing is a way of using multiple computers (connected through a network) to work together on a single task. Each computer handles a part of the work, and together they solve the problem faster or more efficiently than a single computer.

Distributed computing is a computing paradigm that involves multiple computers working together to solve a single problem. The idea is to break down a large computational task into smaller, more manageable pieces that can be processed in parallel on multiple machines.



Why Distributed Computing?

1. **Performance & Speed (Parallelism):**
   Large problems can be divided into smaller tasks and solved simultaneously by multiple machines, which makes computations much faster than on a single system.

2. **Scalability:**
   As demand grows, more computers (nodes) can be added to the system to handle bigger

workloads.

3. **Resource Sharing:**
   Distributed systems allow sharing of resources (storage, computing power, data) across multiple machines, making efficient use of available infrastructure.

4. **Reliability & Fault Tolerance:**
   If one computer fails, others can continue working, ensuring the system keeps running. This increases reliability compared to a single machine.

5. **Cost-Effectiveness:**
   Instead of buying one supercomputer, organizations can use a network of inexpensive computers to achieve similar performance.

6. **Geographical Distribution:**
   Some applications (like Google Search, online banking, or video streaming) require data and services to be available across different regions. Distributed computing supports this global accessibility.

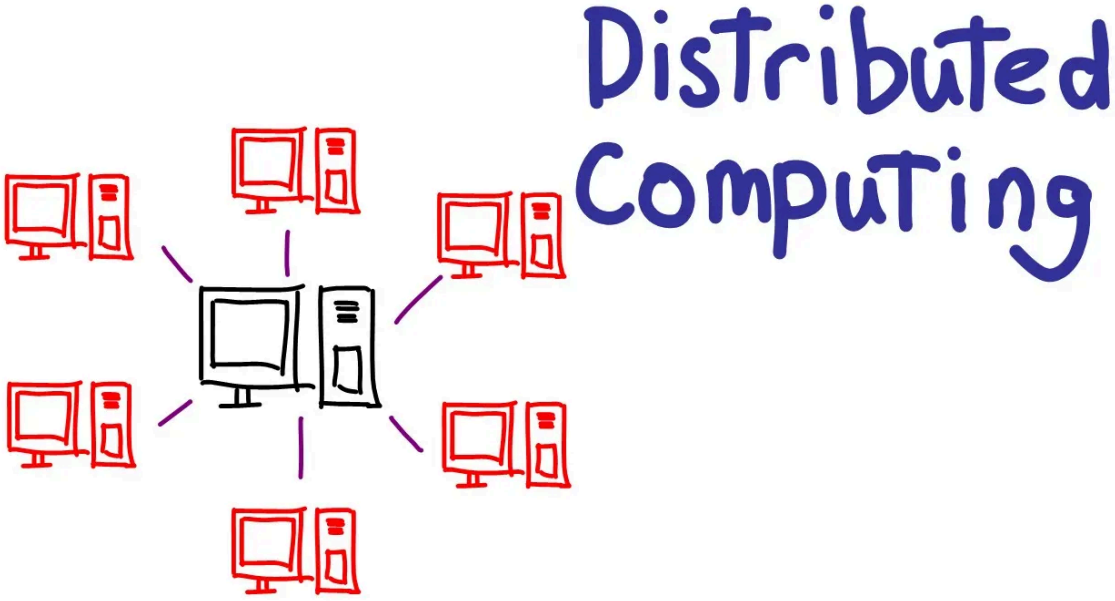
Distributed Computing

- **Meaning:** Doing one large task by dividing it among multiple computers so they work together in parallel.

- **Focus:** Computation (solving problems, processing data).

- **Goal:** Speed, efficiency, and handling big tasks.

✅ **Example:** Google Search → the query is split across thousands of servers, each does some computing, and results are merged.



2. Distributed System

- **Meaning:** A network of independent computers that appear to the user as a **single system**.
- **Focus:** System design (coordination, communication, resource sharing).
- **Goal:** Reliability, fault tolerance, and seamless service.

✅ **Example:** Netflix → when you stream a movie, the servers across different regions deliver the video smoothly as if it's one system, even though many computers are involved.

Example:
Think of a group project: instead of one student writing the entire report, each student writes one section. At the end, they combine all sections into one complete report.
Similarly, in distributed computing, different computers (students) work on different parts of a problem and then combine their results.

Real time example:

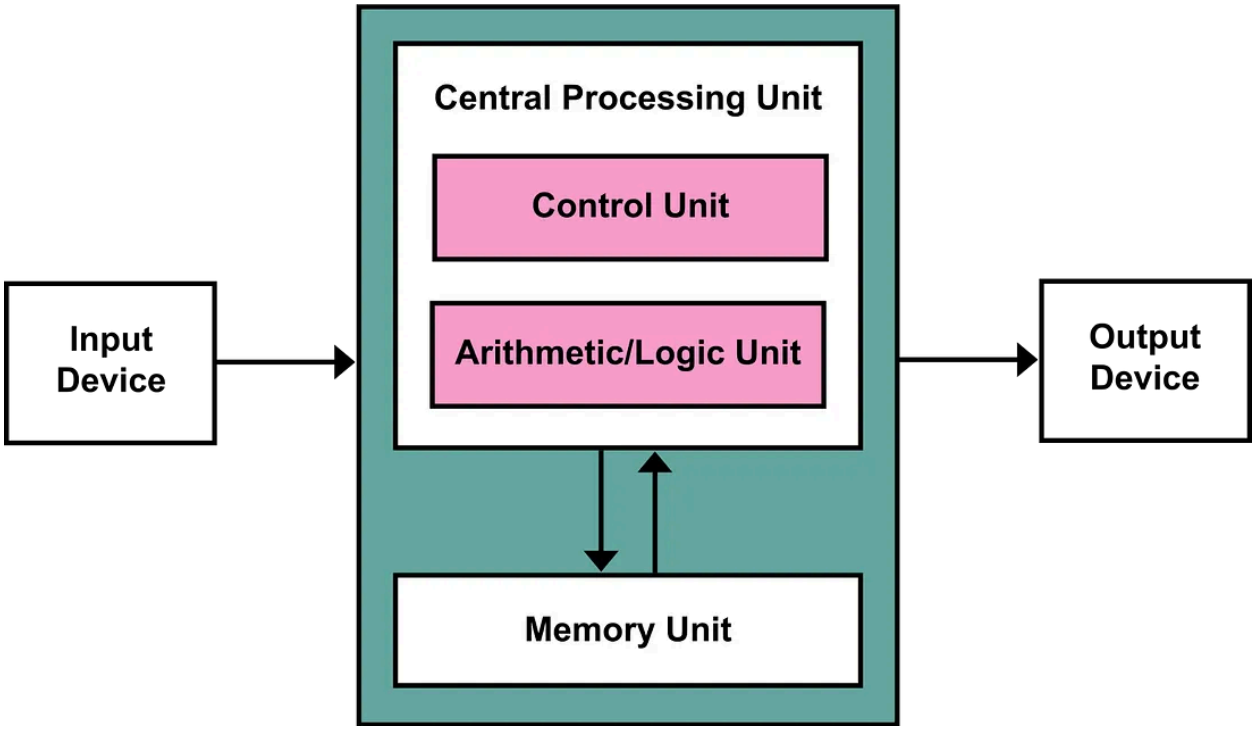
Google Search as Distributed Computing

When you type something into Google (say *“best restaurants in Bengaluru”*), it looks like one simple action to you. But behind the scenes, thousands of computers are working **together in parallel** to give you results in a fraction of a second.

Processors

A **processor** (also called **CPU – Central Processing Unit**) is the **brain of the computer**.
It carries out instructions (like calculations, comparisons, and data movement) so that software and applications can run.

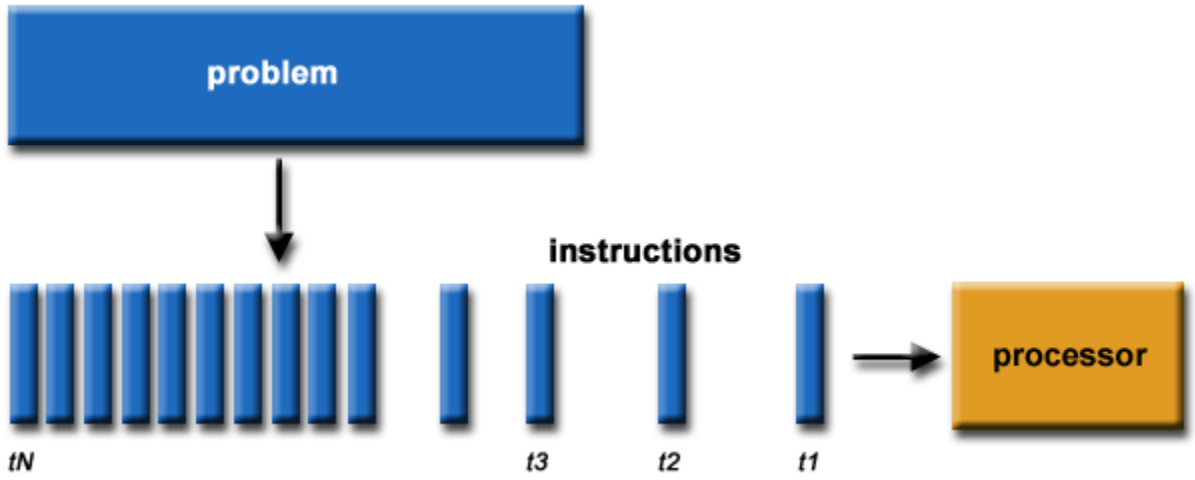
Imagine the processor aka CPU as the conductor of an orchestra. It guides all the different parts of the computer, making sure they work together like a musical performance.



Serial Computing

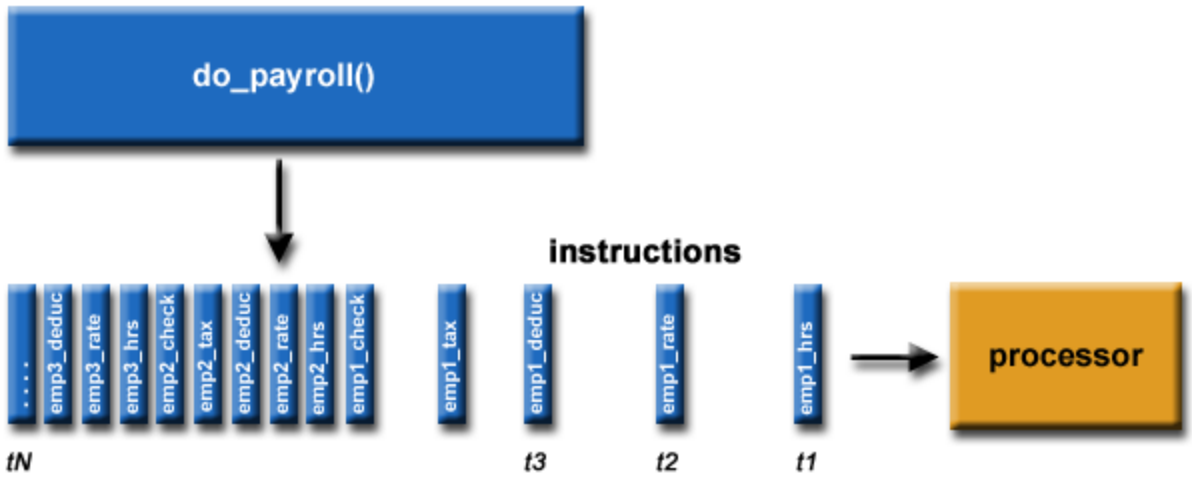
Traditionally, software has been written for *serial* computation:

- **A problem is broken into a discrete series of instructions**
- **Instructions are executed sequentially one after another**
- **Executed on a single processor**
- **Only one instruction may execute at any moment in time**



https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

For example:



- **Serial Computing in C**

(One core, executes tasks one by one)

```
#include <stdio.h>

int main() {
   int salaries[5] = {20000, 25000, 30000, 22000, 27000};
   int net[5];

   // Serial calculation
   for (int i = 0; i < 5; i++) {
      net[i] = salaries[i] + 5000 - 2000; // basic + allowance - deduction
      printf("Employee %d Salary = %d\n", i+1, net[i]);
   }

   return 0;
}
```

Output:

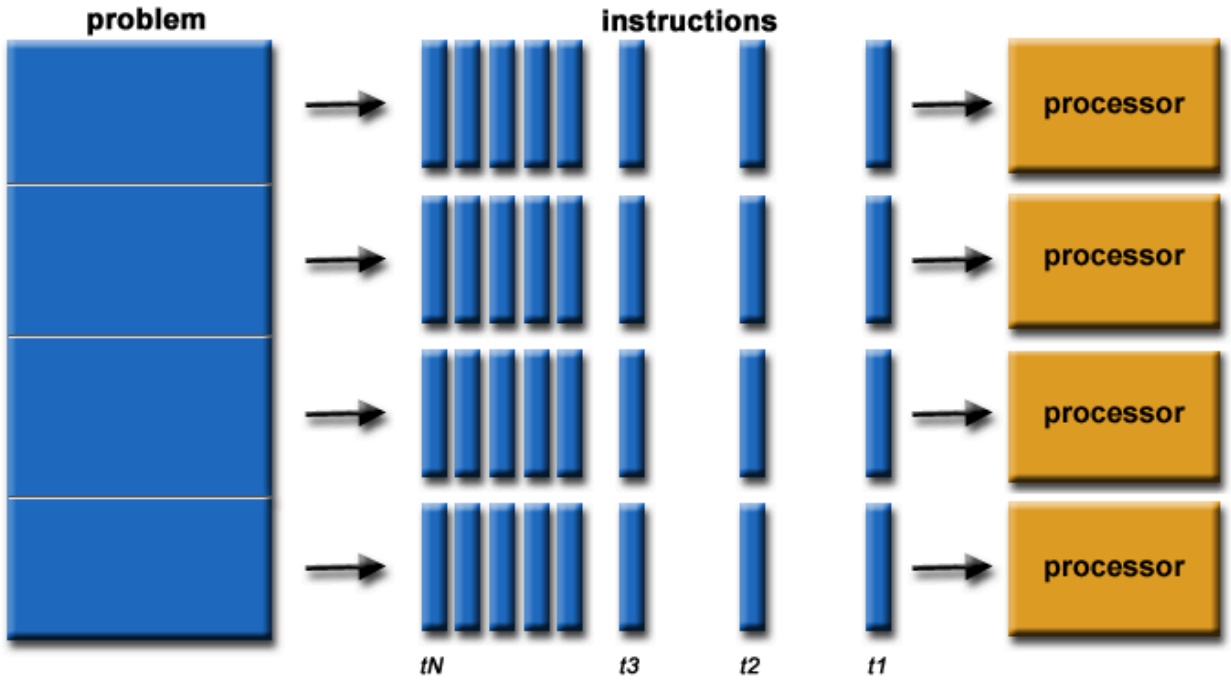
Employee 1 Salary = 23000
Employee 2 Salary = 28000
Employee 3 Salary = 33000

Employee 4 Salary = 25000
Employee 5 Salary = 30000

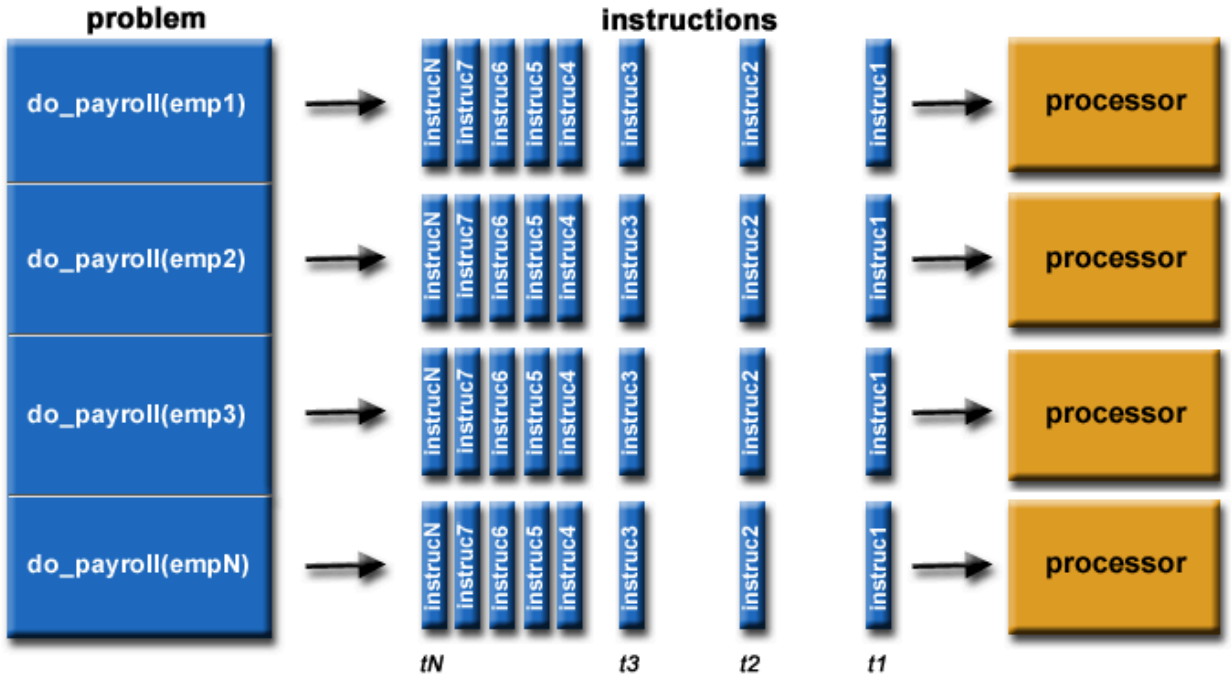
Parallel Computing

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**



For example:



- The computational problem should be able to:
  - Be broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Be solved in less time with multiple compute resources than with a single compute resource.
- The compute resources are typically:
  - A single computer with multiple processors/cores
  - An arbitrary number of such computers connected by a network

🔹 Parallel Computing in C (using OpenMP)

(Work divided across multiple cores)

General OpenMP Syntax in C

```
#pragma omp directive [clauses]
{
   // parallel code here
}
```

`#pragma omp` → tells the compiler this is an OpenMP directive.

`directive` → tells what kind of parallelism (e.g., `parallel`, `for`, `sections`).

`[clauses]` → optional settings (like number of threads).

#pragma omp parallel for

Breakdown:

- `#pragma` → special keyword in C/C++ used for compiler directives.
- `omp` → tells compiler this is an OpenMP instruction.
- `parallel` → says: *create multiple threads*.
- `for` → says: *split the iterations of the next `for` loop among those threads*.

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

int main() {
   int salaries[5] = {20000, 25000, 30000, 22000, 27000};
   int net[5];

   // Parallel calculation using OpenMP
   #pragma omp parallel for
```

```
    for (int i = 0; i < 50000000; i++)
    {
        net[i] = salaries[i] + 5000 - 2000;
        printf("Thread %d calculated Employee %d Salary = %d\n", omp_get_thread_num(), i+1, net[i]);
    }

    return 0;
}
```

OUTPUT:

Thread 0 calculated Employee 1 Salary = 23000

Thread 2 calculated Employee 3 Salary = 33000

Thread 1 calculated Employee 2 Salary = 28000

Thread 3 calculated Employee 4 Salary = 25000

Thread 0 calculated Employee 5 Salary = 30000

Compile Rule (always):

```
gcc -fopenmp file.c -o file


./file
```

What is a Core?

- **A core is the basic computing unit inside a CPU (Central Processing Unit).**
- **It contains the components needed to fetch, decode, and execute instructions (like an ALU, control unit, registers, etc.).**
- **Each core can handle its own sequence of instructions independently.**

Evolution

- **Early CPUs (single-core): Only *one core* → could execute only one instruction stream at a time.**
- **Modern CPUs (multi-core): A chip now contains multiple cores. Each core works like a small processor.**
  Example: A quad-core CPU has 4 cores → it can run 4 tasks in parallel.

Core vs Processor

- **People often say "processor" to mean the CPU chip.**
- **But inside one processor, you may have 1, 2, 4, 8, 16, or more cores.**
- **Example:**
  - **Intel i5 (4 cores, 8 threads)**
  - **AMD Ryzen 9 (16 cores, 32 threads)**
  - **Apple M2 Ultra (24 cores)**

🔹 What is a Thread (in CPU context)?

- **A thread is the smallest unit of execution scheduled by the CPU.**
- **Each core can run at least one thread at a time.**
- **With technologies like Hyper-Threading (Intel) or Simultaneous Multithreading (SMT – AMD), a single core can handle 2 threads simultaneously.**

✅ **Takeaway for class:**

- **Cores = actual hardware that executes instructions.**
- **Threads = virtual execution lanes (more threads help keep cores busy, but not as powerful as adding real cores).**

Why Threads ≠ Cores

1. Cores are real workers

- **A core is an actual hardware unit (with its own ALU, registers, pipelines).**
- **It can do real independent work.**

2. Threads are virtual workers

- **A thread is not a new core; it's just a way to use idle parts of a core more efficiently.**
- **When one thread is waiting (e.g., for data from memory), the core can keep busy by working on another thread.**
- **But both threads share the same core resources.**

Example (Payroll Case)

- **Single-core: Salaries calculated one after another (serial).**
- **Quad-core: Employee list split into 4 groups → 4 cores calculate salaries simultaneously → much faster.**

Quick Table (Core vs Processor Examples)

Processor (CPU Chip)	Cores	Threads	Typical Device
Intel Core i5	4–6 cores	8–12	Laptops/Desktops
Intel Core i7	6–16 cores	12–24	High-end laptops, desktops
AMD Ryzen 9	Up to 16 cores	Up to 32	Gaming PCs, Workstations
Apple MacBook Air M1/M2/M3	8 cores	8 threads	MacBook Air laptops
Apple M2 Ultra	24 cores	24 threads	Mac Studio (Pro systems)

What is Cache?

- **A small, very fast memory located inside or close to the CPU.**
- **Stores copies of frequently used data/instructions from main memory (RAM).**
- **Purpose → reduce the time the CPU waits for data (bridging the CPU–memory speed gap).**

🔹 Levels of Cache

Modern processors have multiple layers:

1. **L1 Cache (Level 1)**
    - **Smallest (e.g., 32KB–128KB per core).**
    - **Fastest, directly inside CPU core.**
    - **Split into L1d (data) and L1i (instructions).**
2. **L2 Cache (Level 2)**
    - **Bigger (256KB–2MB).**
    - **Slower than L1 but still much faster than RAM.**
    - **Usually private to each core.**
3. **L3 Cache (Level 3)**
    - **Large (4MB–64MB).**
    - **Shared among multiple cores.**
    - **Helps communication between cores.**
4. **L4 Cache (rare, in high-end CPUs like Intel Xeon / Apple M-series)**
    - **Even larger, shared, sometimes external.**

Introduction to parallel programming

➢ *Parallel programming is a way of writing computer programs so that **many tasks are executed at the same time** (in parallel), instead of one after another.*
*It uses **multiple processors/cores** working together to finish a task faster.*

➢ **Parallel programming = *solving a problem faster by splitting it into smaller tasks and running them at the same time on multiple processors.***

➢ ***Parallel programming is a computational technique that divides a large problem into smaller, independent sub-tasks that are executed simultaneously by multiple processors or computing resources.***

➢ ***The primary goal is to significantly reduce the time required to solve complex problems by leveraging parallel hardware to perform computations concurrently, rather than sequentially. Key models include shared memory, where processors access a common memory space, and message passing, where processors communicate by sending and receiving messages.***

Example

Imagine you need to **wash 100 dishes**:

- **Normal programming (sequential):** One person washes all 100 dishes one by one.
- **Parallel programming:** 4 people wash dishes at the same time. Each person does 25 dishes. The work gets done much faster.

Real-life Example in Computers

- Video processing: When editing a video, different parts of the video can be processed simultaneously using parallel programming.
- Weather forecasting: Large datasets are split and processed in parallel by supercomputers to give accurate forecasts quickly.

Parallel computing:

https://www.youtube.com/watch?v=EAQ1HE4o52w

https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial

Q)Compare the time complexity of summing n numbers sequentially vs. parallelly on p processors

I. Sequential Sum of n numbers

```
// sequential_sum.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int n, i;
    long long sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Initialize array with values 1..n
    for (i = 0; i < n; i++) {
        arr[i] =   ;
    }

    clock_t start = clock();
    for (i = 0; i < n; i++) {
        sum += arr[i];
    }
    clock_t end = clock();

    printf("\nSequential Sum = %lld", sum);
    printf("\nTime Taken = %f seconds\n",
           (double)(end - start) / CLOCKS_PER_SEC);

    free(arr);
```

```
    return 0;
}
```

Compilation :

gcc sequential_sum.c -o sequential_sum

./sequential_sum

Ii. Parallel Sum of n numbers (OpenMP)

```
// parallel_sum.c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main() {
    int n, i;
    long long sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Initialize array with values 1..n
    for (i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

    double start = omp_get_wtime();
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < n; i++) {
        sum += arr[i];
    }
    double end = omp_get_wtime();

    printf("\nParallel Sum = %lld", sum);
    printf("\nTime Taken = %f seconds\n", end - start);

    free(arr);
```

```
    return 0;
}
```

Compilation & Run:

```
gcc parallel_sum.c -fopenmp -o parallel_sum
./parallel_sum
```

```
#pragma omp parallel for reduction(+:sum)
```

Meaning:

1. `#pragma omp parallel for`
    - Tells the compiler to run the `for` loop in parallel using multiple threads (OpenMP will split iterations across threads).
2. `reduction(+:sum)`
    - Ensures that every thread keeps its own private copy of the variable `sum`.
    - At the end of the loop, OpenMP will combine (reduce) all these private sums into a single final value using the `+` operator.

🔹 Why is `reduction` Needed?

If you wrote:

#pragma omp parallel for

```
for (i = 0; i < n; i++) {
   sum += arr[i];   // ❌ Problematic
}
```

- All threads try to update the same `sum` at the same time.
- This causes a race condition (unpredictable wrong results).

With `reduction(+:sum)`, OpenMP handles it safely:

- Each thread computes a partial sum of its chunk.
- After the loop, OpenMP adds all partial sums together into the final `sum`.

🔹 Example

Suppose:

arr = [1, 2, 3, 4, 5, 6, 7, 8]
n = 8
p = 4 threads

Each thread's private sum:

- Thread 1 → adds `[1,2]` → 3
- Thread 2 → adds `[3,4]` → 7
- Thread 3 → adds `[5,6]` → 11
- Thread 4 → adds `[7,8]` → 15

OpenMP reduces them:

sum = 3 + 7 + 11 + 15 = 36

`#pragma omp parallel for` → **parallelizes the loop.**

`reduction(+:sum)` → **avoids race condition by giving each thread its own sum, then merging safely.**

Q. Estimate the value of pi using:

Parallelize the code by removing loop carried dependency and record both serial and parallel execution times.

- **Idea: Monte Carlo π Estimation**
  - Generate random points inside a square of side length 1.
  - Count how many points fall inside the unit quarter circle.
  - Ratio gives an estimate of π.

    $\pi \approx 4 \times$ points inside circle / total points

- Serial Code (C with OpenMP disabled)

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

```c
#include <time.h>

int main() {
    long num_points = 100000000;  // 100 million
    long inside_circle = 0;
    double x, y;

    clock_t start = clock();
    for (long i = 0; i < num_points; i++) {
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;
        if (x*x + y*y <= 1.0)
            inside_circle++;
    }
    double pi_estimate = 4.0 * inside_circle / num_points;
    clock_t end = clock();

    printf("Serial Pi Estimate: %f\n", pi_estimate);
    printf("Serial Time: %f seconds\n", (double)(end - start) / CLOCKS_PER_SEC);
    return 0;
}
```

- **Parallel Code (C with OpenMP)**

We **remove loop-carried dependency** by giving each thread a private counter and then reducing (`reduction(+:inside_circle)`):

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

int main() {
    long num_points = 100000000;  // 100 million
    long inside_circle = 0;
    double x, y;
```

```
    double start = omp_get_wtime();

    #pragma omp parallel for private(x, y) reduction(+:inside_circle)
    for (long i = 0; i < num_points; i++) {
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;
        if (x*x + y*y <= 1.0)
            inside_circle++;
    }

    double pi_estimate = 4.0 * inside_circle / num_points;
    double end = omp_get_wtime();

    printf("Parallel Pi Estimate: %f\n", pi_estimate);
    printf("Parallel Time: %f seconds\n", end - start);
    return 0;
}
```

#pragma omp parallel for private(x, y) reduction(+:inside_circle)

is an **OpenMP directive** that tells the compiler how to parallelize the loop. Let's break it down:

- **`#pragma omp parallel for`**
  - Splits the iterations of the **for loop** among available threads.
  - Each thread executes a chunk of iterations **independently**.

- **`private(x, y)`**

- Ensures that **each thread gets its own copy** of the variables `x` and `y`.
- Without `private`, threads might overwrite each other's values of `x` and `y` → **race condition**.
- After the loop ends, those private copies are discarded.

🔹 **`reduction(+:inside_circle)`**

- The variable `inside_circle` is **shared logically**, but each thread keeps a **private local counter** during execution.
- At the end of the loop, OpenMP **combines (reduces)** all private copies into one shared variable using the `+` operator.
- This removes the **loop-carried dependency** (since `inside_circle++` is normally sequential).

🔹 **Example Flow**

Suppose 4 threads run 1,000 iterations each:

- Thread 1 finds `inside_circle = 785`
- Thread 2 finds `inside_circle = 790`

- Thread 3 finds `inside_circle = 796`
- Thread 4 finds `inside_circle = 788`

At the end:

inside_circle=785+790+796+788=3159

✅ In short:

- `private(x, y)` → avoids variable conflicts.
- `reduction(+:inside_circle)` → safely sums up results from all threads.
- `parallel for` → distributes iterations to threads.

Q)Write a program to sort an array of n elements using both sequential and parallel merge
sort (using Section). Record the difference in execution time.

Sequential Merge Sort

```
 #include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Merge function using i=low, j=mid+1, and temp[]
void merge(int arr[], int low, int mid, int high) {
    int i = low;
    int j = mid + 1;
    int k = low;
```

```
    int temp[high + 1];  // temporary array

    while (i <= mid && j <= high) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }

    while (i <= mid) temp[k++] = arr[i++];
    while (j <= high) temp[k++] = arr[j++];

    for (i = low; i <= high; i++) {
        arr[i] = temp[i];
    }
}

// Recursive Merge Sort
void mergeSort(int arr[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];
    srand(time(NULL));
    for (int i = 0; i < n; i++) arr[i] = rand() % 100000;

    clock_t start = clock();
    mergeSort(arr, 0, n - 1);
    clock_t end = clock();

    double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Time taken for Sequential Merge Sort: %f seconds\n", time_taken);

    // Print sorted array (optional, for checking small inputs)
    // for (int i = 0; i < n; i++) printf("%d ", arr[i]);
```

```
    // printf("\n");

    return 0;
}
```

Parallel Version (with OpenMP `sections`)

What is `#pragma omp section`?

- In **OpenMP**, `#pragma omp sections` tells the compiler that the following block of code will contain **independent tasks** (sections) that can run in parallel.

- Each `#pragma omp section` defines one of those tasks.

- All sections inside the same `#pragma omp sections` region will be executed by different threads.

```
#pragma omp parallel sections
{
    #pragma omp section
    {
        // This part runs in one thread
        parallelMergeSort(arr, low, mid);
    }

    #pragma omp section
    {
        // This part runs in another thread
        parallelMergeSort(arr, mid + 1, high);
    }
}
```

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

// Merge using Levithin method
void merge(int arr[], int low, int mid, int high) {
    int i = low;
    int j = mid + 1;
    int k = low;
    int temp[high + 1];

    while (i <= mid && j <= high) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }

    while (i <= mid) temp[k++] = arr[i++];
    while (j <= high) temp[k++] = arr[j++];

    for (i = low; i <= high; i++) {
        arr[i] = temp[i];
    }
}

// Parallel Merge Sort
void parallelMergeSort(int arr[], int low, int high, int depth) {
    if (low < high) {
        int mid = (low + high) / 2;

        if (depth < 3) {  // limit parallel recursion depth
            #pragma omp parallel sections
            {
                #pragma omp section
                parallelMergeSort(arr, low, mid, depth + 1);

                #pragma omp section
                parallelMergeSort(arr, mid + 1, high, depth + 1);
            }
```

```
        } else {
            // fallback to sequential
            parallelMergeSort(arr, low, mid, depth + 1);
            parallelMergeSort(arr, mid + 1, high, depth + 1);
        }

        merge(arr, low, mid, high);
    }
}

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];
    srand(time(NULL));
    for (int i = 0; i < n; i++) arr[i] = rand() % 100000;

    double start = omp_get_wtime();
    parallelMergeSort(arr, 0, n - 1, 0);
    double end = omp_get_wtime();

    printf("Time taken for Parallel Merge Sort: %f seconds\n", end - start);

    return 0;
}
```

Alternative program:Parallel Merge Sort (without depth)

he `depth` parameter in my parallel version was just a **control mechanism** to avoid creating *too many threads* at every recursion level (otherwise, for large `n`, you'll end up spawning thousands of threads, which is slower than sequential).

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

```
// Merge function (Levithin style: i=low, j=mid+1, temp[])
void merge(int arr[], int low, int mid, int high) {
    int i = low;
    int j = mid + 1;
    int k = low;
    int temp[high + 1];

    while (i <= mid && j <= high) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }

    while (i <= mid) temp[k++] = arr[i++];
    while (j <= high) temp[k++] = arr[j++];

    for (i = low; i <= high; i++) {
        arr[i] = temp[i];
    }
}

// Parallel Merge Sort (no depth control)
void parallelMergeSort(int arr[], int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;

        #pragma omp parallel sections
        {
            #pragma omp section
            parallelMergeSort(arr, low, mid);

            #pragma omp section
            parallelMergeSort(arr, mid + 1, high);
        }

        merge(arr, low, mid, high);
    }
}
```

```
int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];
    srand(time(NULL));
    for (int i = 0; i < n; i++) arr[i] = rand() % 100000;

    double start = omp_get_wtime();
    parallelMergeSort(arr, 0, n - 1);
    double end = omp_get_wtime();

    printf("Parallel Merge Sort Time: %f seconds\n", end - start);

    return 0;
}
```

Sequential
gcc sequential_mergesort.c -o seq
./seq

Parallel (with OpenMP)
gcc -fopenmp parallel_mergesort.c -o par
./par

- `parallel for` → splits iterations of a **loop** among threads.
- `parallel sections` → splits **independent code blocks** among threads (not loops).

Q) i. Sequential Fibonacci (recursive)

ii. Parallel Fibonacci (OpenMP tasks)

I. Sequential Fibonacci

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Sequential Fibonacci (recursive)
long long fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}

int main() {
    int n;
    printf("Enter n (Fibonacci terms): ");
    scanf("%d", &n);

    clock_t start = clock();
    for (int i = 0; i < n; i++) {
        printf("%lld ", fib(i));
    }
    clock_t end = clock();

    double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
    printf("\nSequential Fibonacci Time: %f seconds\n", time_taken);

    return 0;
}
```

ii. Parallel Fibonacci (OpenMP tasks)

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

```
// Parallel Fibonacci using tasks
long long fib_par(int n) {
    if (n <= 1) return n;

    long long x, y;

    #pragma omp task shared(x)
    x = fib_par(n - 1);

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

    #pragma omp taskwait
    return x + y;
}

int main() {
    int n;
    printf("Enter n (Fibonacci terms): ");
    scanf("%d", &n);

    double start = omp_get_wtime();

    #pragma omp parallel
    {
        #pragma omp single
        {
            for (int i = 0; i < n; i++) {
                printf("%lld ", fib_par(i));
            }
        }
    }

    double end = omp_get_wtime();
    printf("\nParallel Fibonacci Time: %f seconds\n", end - start);

    return 0;
}
```

Compilation & Run

Sequential

gcc sequential_fib.c -o seq
./seq

Parallel
gcc -fopenmp parallel_fib.c -o par
./par

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

I. Serial Prime Numbers

```
// serial_prime.c
#include <stdio.h>
#include <math.h>
#include <omp.h>

// Function to check if a number is prime
int isPrime(int num) {
    if (num <= 1) return 0;
    if (num == 2) return 1;
    if (num % 2 == 0) return 0;
    for (int i = 3; i <= sqrt(num); i += 2) {
        if (num % i == 0) return 0;
    }
    return 1;
}

int main() {
    int n;
    double start, end;

    printf("Enter the value of n: ");
    scanf("%d", &n);

    start = omp_get_wtime();

    printf("Prime numbers (Serial):\n");
    for (int i = 2; i <= n; i++) {
        if (isPrime(i)) {
```

```
            printf("%d ", i);
        }
    }

    end = omp_get_wtime();
    printf("\nExecution Time (Serial) = %f seconds\n", end - start);

    return 0;
}
```

ii. Parallel Prime Numbers

```
// parallel_prime.c
#include <stdio.h>
#include <math.h>
#include <omp.h>

// Function to check if a number is prime
int isPrime(int num) {
    if (num <= 1) return 0;
    if (num == 2) return 1;
    if (num % 2 == 0) return 0;
    for (int i = 3; i <= sqrt(num); i += 2) {
        if (num % i == 0) return 0;
    }
    return 1;
}

int main() {
    int n;
    double start, end;

    printf("Enter the value of n: ");
    scanf("%d", &n);

    start = omp_get_wtime();

    printf("Prime numbers (Parallel):\n");

    #pragma omp parallel for
    for (int i = 2; i <= n; i++) {
        if (isPrime(i)) {
```

```
            // Print safely from multiple threads
            #pragma omp critical
            {
                printf("%d ", i);
            }
        }
    }

    end = omp_get_wtime();
    printf("\nExecution Time (Parallel) = %f seconds\n", end - start);

    return 0;
}
```

Compile serial version
gcc serial_prime.c -o serial_prime -fopenmp
./serial_prime

Compile parallel version
gcc parallel_prime.c -o parallel_prime -fopenmp
./parallel_prime

What is `fork()` in C?

- `fork()` is a **system call** in **Unix/Linux** that is used to create a **new process**.
- The new process created is called the **child process**, and the original process is the **parent process**.
- After `fork()`, **both parent and child continue execution from the next line of code after fork().**

```c
#include <stdio.h>
#include <unistd.h>   // needed for fork()

int main() {
    printf("Best Team \n");
    fork();
    fork();
    printf(" RCB \n");

    return 0;
}
```

- **Return Values of `fork()`**
  - `fork()` returns **different values** in the parent and child:
    - **Parent process:** returns the **PID (Process ID)** of the child.
    - **Child process:** returns `0`.
    - If `fork()` fails → returns `-1`.

```c
#include <stdio.h>

#include <unistd.h>
```

```
int main() {
    int pid;

    pid = fork();

    if (pid > 0) {
        // Parent process
        printf("I am the Parent, my child's PID is %d\n", pid);
    }
    else if (pid == 0) {
        // Child process
        printf("I am the Child, my PID is %d\n", getpid());
    }
    else {
        // Error
        printf("Fork failed!\n");
    }

    return 0;
}
```

🔹 `fork()`

- **Definition:** Creates a new child process, which is a **duplicate of the parent**.
- **Memory:**
    - The child gets a **copy of the parent's address space**.
    - Copy-on-write is usually used → actual duplication happens only when either process modifies memory.
- **Execution:**
    - Both parent and child **run independently** after the fork.
- **Return Value:**
    - Parent gets **child PID**,
    - Child gets **0**.
- **Performance:** Slightly slower because of memory duplication (though modern OS uses copy-on-write).
- **Use Case:** General-purpose process creation.

🔹 `vfork()`

- **Definition:** Special version of fork, optimized for performance when child immediately calls `exec()` or `_exit()`.
- **Memory:**
    - The child **shares the same address space** as the parent until it calls `exec()` or `_exit()`.
    - No copy of parent's memory is made.
- **Execution:**
    - Parent is **suspended** until child either calls `exec()` or `_exit()`.
    - Child runs **first**.
- **Performance:** Faster than `fork()` since it avoids copying address space.
- **Use Case:** When the child process just wants to replace itself with another program using `exec()`.

Flynn's Taxonomy classifications of computer architectures

1. SISD
2. SIMD
3. MISD
4. MIMD

1. SISD (Single Instruction, Single Data)

Definition:
A single processor executes one instruction at a time on one data item at a time.

Example:

- **Traditional uniprocessor systems (like early personal computers).**
- **Classic Von Neumann architecture.**

2. SIMD (Single Instruction, Multiple Data)

Definition:
A single instruction operates simultaneously on multiple data elements. This is common in systems designed for parallel data processing.

Used in:

- **Image processing**
- **Matrix operations**
- **Vector processors and GPUs**

Example Systems:

- **GPUs (NVIDIA, AMD)**
- **Intel MMX/SSE, ARM NEON, Cray vector processors**

3. MISD (Multiple Instruction, Single Data)

Definition:
Multiple instructions operate on the same data simultaneously.
This is a rare and mostly theoretical model.

Used in:

- **Some fault-tolerant systems (where multiple processors perform different operations on the same input for error checking).**

Example Systems:
Few practical implementations — sometimes used conceptually in redundant computation (e.g., spacecraft systems).

4. MIMD (Multiple Instruction, Multiple Data)

Definition:
Multiple processors execute different instructions on different data independently.
This is the most common architecture for modern multiprocessor and multicore systems.

Used in:

- **Supercomputers**
- **Cloud servers**
- **Multi-core CPUs**

Example Systems:

- **Intel i7, AMD Ryzen (multi-core CPUs)**
- **Distributed systems, clusters**

📊 **Summary Table**

Type	Instructions	Data	Example Use	Example System

SISD	**Single**	**Single**	**Basic sequential processing**	**Intel 8085**
SIMD	**Single**	**Multiple**	**Parallel data ops (images, vectors)**	**GPU, Cray**
MISD	**Multiple**	**Single**	**Fault-tolerant redundancy**	**Spacecraft systems**
MIMD	**Multiple**	**Multiple**	**General multiprocessor**	**Modern multi-core CPUs**

UNIT-3:Distributed Memory Programming

In distributed memory systems, each processor has its own private memory. There is no shared memory among processors. Therefore, to communicate data between processors, message passing is used.

For distributed programming, the most widely used library is:

MPI – Message Passing Interface

MPI provides standard functions that allow processes to communicate by sending and receiving messages.

What is MPI?

MPI (Message Passing Interface) is a communication standard used to write parallel programs for distributed memory systems such as clusters and multi-node supercomputers.

Features

- **Processes run independently and communicate through messages.**
- **Provides point-to-point and collective communication.**

- **Portable across many systems (Linux clusters, supercomputers, cloud, etc.)**
- **Works mainly in C, C++, Fortran.**

Commonly Used MPI Functions

Function	Purpose
`MPI_Init(&argc, &argv)`	Initializes MPI environment
`MPI_Finalize()`	Terminates MPI environment
`MPI_Comm_size(MPI_COMM_WORLD, &size)`	Returns number of processes
`MPI_Comm_rank(MPI_COMM_WORLD, &rank)`	Returns unique ID of each process (rank)
`MPI_Send(&data, count, type, dest, tag, comm)`	Sends data to another process
`MPI_Recv(&data, count, type, source, tag, comm, &status)`	Receives data from another process
`MPI_Barrier(MPI_COMM_WORLD)`	Synchronization point (all processes wait here)
`MPI_Bcast(&data, count, type, root, comm)`	Broadcast data from one process to all
`MPI_Scatter()`	Distributes different parts of data to multiple processes

`MPI_Gather()`	**Collects data from multiple processes to one process**
`MPI_Reduce()`	**Combines data from many processes (sum, max, min, etc.)**

Example MPI Program ©

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

int main(int argc, char** argv) {
   MPI_Init(&argc, &argv);

   int size, rank;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   printf("Hello from process %d of %d\n", rank, size);
   printf(“ Manchester United”);

   MPI_Finalize();
   return 0;
}
```

Compile and Run:
mpicc hello.c -o hello
mpirun -np 4 ./hello

OUTPUT:
Hello from process 0 out of 4 processes.

Hello from process 1 out of 4 processes.
Hello from process 2 out of 4 processes.
Hello from process 3 out of 4 processes.

Types of MPI Communication

1. Point-to-Point Communication

- **MPI_Send**
- **MPI_Recv**

Used for direct data transfer between two processes.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char** argv) {

   MPI_Init(&argc, &argv);

   int rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // Identify which process you are

   if(rank == 0) {
      int number = 77770778589;          // Data to send
      MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
      printf("Process 0 sent number %d to Process 1.\n", number);
   }
   else if(rank == 1) {
      int received_number;
      MPI_Recv(&received_number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      printf("Process 1 received number %d from Process 0.\n", received_number);
   }
   MPI_Finalize();
```

```
    return 0;
}
```

Run it:
mpicc p2p.c -o p2p
mpirun -np 2 ./p2p

OUTPUT:
Process 0 sent number **77770778589** to Process 1.
Process 1 received number **77770778589** from Process 0.

Step 1: Compilation
mpicc p2p.c -o p2p

- `mpicc` is like the normal `gcc` compiler but it understands MPI libraries.
- `p2p.c` is your program file.
- `-o p2p` means "create an output executable named `p2p`".

Step 2: Run the program with multiple processes
- `mpirun` (or sometimes `mpiexec`) is used to run MPI programs.
- `-np 2` means **run the program using 2 processes**.
- `./p2p` is the program we compiled.

❖ Collective Communication in MPI

Collective communication involves **all processes in the communicator** (usually `MPI_COMM_WORLD`). These routines do not require you to specify sender/receiver ranks like `MPI_Send` / `MPI_Recv`.

Common collective operations:

- `MPI_Bcast` (broadcast one value to all)
- `MPI_Scatter` (send portions of data to each process)
- `MPI_Gather` (collect data from all processes)
- `MPI_Reduce` (reduce values using sum, max, etc.)

Example: MPI_Bcast (Broadcast one value from rank 0 to all ranks)

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

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

    if(rank == 0) {
        data = 100;  // root has data
```

```
        printf("Root broadcasting data = %d\n", data);
    }

    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Process %d received data = %d\n", rank, data);

    MPI_Finalize();
    return 0;
}
```

mpicc bcast.c -o bcast
mpirun -np 8 ./bcast

Output:

Root broadcasting data = 100
Process 0 received data = 100
Process 1 received data = 100
Process 2 received data = 100
Process 3 received data = 100
Process 4 received data = 100
Process 5 received data = 100
Process 6 received data = 100
Process 7 received data = 100

MPI_Gather – Explanation

`MPI_Gather` is a collective communication operation that collects data from all processes in a communicator and delivers it to the root process.

syntax:

```
MPI_Gather(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype,
           root, comm);
```

Meaning of Parameters

- **sendbuf → data each process sends**
- **sendcount → number of elements each process sends**
- **sendtype → MPI datatype of send buffer**
- **recvbuf → array on root where all data is stored**
- **recvcount → number of elements received from each process**
- **recvtype → MPI datatype of recv buffer**
- **root → rank of the collecting process**
- **comm → communicator (usually `MPI_COMM_WORLD`)**

Example

Simple MPI Program using MPI_Gather

Program: Each process sends its rank, and root gathers all ranks

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

int main(int argc, char *argv[]) {
   int rank, size;
   int send_value, recv_values[10];   // assume max 10 processes

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

   send_value = rank*rank;   // Each process sends its rank value

   // Gather all send_values into recv_values array at root (process 0)
```

```
    MPI_Gather(&send_value, 1, MPI_INT,
               recv_values, 1, MPI_INT,
               0, MPI_COMM_WORLD);

    // Root process prints the collected data
    if (rank == 0) {
        printf("Data gathered at root:\n");
        for (int i = 0; i < size; i++) {
            printf("%d ", recv_values[i]);
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
```

UNIT-4

OpenMP Pragmas and Directives in Shared-Memory Programming

OpenMP uses compiler directives (pragmas) to create and manage threads in shared-memory systems. These directives allow programmers to parallelize loops, sections, and critical regions easily.

The most commonly used OpenMP directives are:

- **`#pragma omp parallel`**
- **`#pragma omp for`**
- **`#pragma omp critical`**

1. `#pragma omp parallel`

Purpose

Creates a team of threads.
Each thread executes the block of code inside the parallel region.

Syntax:

```
#pragma omp parallel [clauses]
{
   // parallel code
}
```

Example:

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

int main() {
   #pragma omp parallel
   {
      int id = omp_get_thread_num();
      printf("Hello from thread %d\n", id);
   }
   return 0;
}
```

OUTPUT:

Hello from thread 0

Hello from thread 1

Hello from thread 2

Hello from thread 3

Explanation

- **If 4 threads are created, the message prints 4 times, once from each thread.**
- **All threads execute the same block.**

2. OpenMP Program to Create 8 Threads

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

int main() {
```

```
    // Set number of threads
    omp_set_num_threads(8);

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        printf("Hello World from thread %d\n", tid);
    }

    return 0;
}
```

Output:

Hello World from thread 0
Hello World from thread 1
Hello World from thread 2
Hello World from thread 3
Hello World from thread 4
Hello World from thread 5
Hello World from thread 6
Hello World from thread 7

How to Compile and Run an OpenMP Program (GCC)

Step 1: Save the file

```
$ gedit      CR7.c
```

Step 2: Compile using GCC with OpenMP flag

```
gcc -fopenmp    CR7.c    -o      CR7
```

Explanation:

- `-fopenmp` → enables OpenMP support

Step 3: Run the program

```
./CR7
```

2. `#pragma omp for`

Purpose

Used inside a parallel region to divide loop iterations among threads automatically.

Syntax

```
#pragma omp for [clauses]
for (i = 0; i < n; i++) {
   // loop body
}
```

Example

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

int main() {
   int i;
   #pragma omp parallel
   {
      #pragma omp for
      for (i = 0; i < 8; i++) {
         printf("Iteration %d executed by thread %d\n",
               i, omp_get_thread_num());
```

```
    }
  }
  return 0;
}
```

OUTPUT:

```
Iteration 0 executed by thread 0
Iteration 2 executed by thread 1
Iteration 1 executed by thread 2
Iteration 3 executed by thread 3
Iteration 4 executed by thread 0
Iteration 5 executed by thread 1
Iteration 6 executed by thread 2
Iteration 7 executed by thread 3
```

Explanation

- **The `#pragma omp parallel` directive creates multiple threads.**
- **The `#pragma omp for` directive divides the loop iterations (0 to 7) among these threads.**
- **Each thread prints the iteration number and its thread ID.**

The number of threads depends on:

- **`OMP_NUM_THREADS` environment variable, OR**
- **System defaults (commonly the number of CPU cores).**

Q) Describe the purpose of the OpenMP #pragma omp for directive and explain how the private and reduction clauses help in parallelizing loops. Using these OpenMP constructs, write a program that estimates the value of π using numerical integration in parallel.

1. Purpose of `#pragma omp for`

The OpenMP `#pragma omp for` directive is used to parallelize loops by dividing the loop iterations among multiple threads.
Instead of all threads executing the whole loop, OpenMP automatically distributes different iterations to different threads, improving performance on shared-memory systems.

2. Purpose of `private` Clause

The `private` clause ensures that each thread gets its own local copy of a variable.
This prevents race conditions when each thread needs to compute values independently inside the loop.

Example:

```
private(x, y).
```

→ Each thread has a private `x`, so they do not overwrite each other.
→ Each thread has a private `y`, so they do not overwrite each other.

3. Purpose of `reduction` Clause

The `reduction` clause removes loop-carried dependencies for operations like addition, multiplication, min, max, etc.

Example:

```
reduction(+:count)
```

You are telling OpenMP:

- **Each thread gets its own private copy of the variable `count`.**
- **All copies start with the neutral value for `+`, which is 0.**
- **Each thread updates *its own* `count` (no conflicts = no race conditions).**
- **At the end of the parallel region, all private counts are added together into one final shared `count`.**

total count = count_thread1 + count_thread2 + ... + count_threadN

OpenMP Program to Calculate π Using Monte Carlo Method

📌 **Monte Carlo Idea**

Generate random points in a unit square [0,1]×[0,1].
Count how many fall inside the quarter circle:

$$x^2 + y^2 \leq 1$$

Then:

$$\pi \approx 4 \times \frac{\text{points inside circle}}{\text{total points}}$$

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

int main() {
   long long i, n = 10000000;  // Number of random points
   long long count = 0;
```

```
    double x, y, pi;

    // Parallel Monte Carlo simulation
    #pragma omp parallel for private(x, y) reduction(+:count)
    for (i = 0; i < n; i++) {
        x = (double)rand() / RAND_MAX;   // random x in [0,1]
        y = (double)rand() / RAND_MAX;   // random y in [0,1]

        if ((x * x + y * y) <= 1.0)
            count++;  // point inside circle
    }

    pi = 4.0 * (double)count / (double)n;

    printf("Estimated value of PI = %f\n", pi);

    return 0;
}
```

Q) how to compute the parallel sum of n numbers using OpenMP, and how `#pragma omp parallel for`, `private`, and `reduction` are used.

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

int main() {
    int n = 10;
    int a[10] = {1,2,3,4,5,6,7,8,9,10};
    int sum = 0;

    #pragma omp parallel for private(i) reduction(+:sum)
    for (int i = 0; i < n; i++) {
        sum += a[i];                    // each thread updates its local sum
```

```
    }

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

Meaning of Directives

1. `#pragma omp parallel for`

- Creates multiple threads.
- Divides loop iterations among threads.

2. `private(i)`

- Each thread gets its own copy of loop variable `i`.

3. `reduction(+:sum)`

- Each thread gets a temporary private `sum = 0`
- Local sums are added together at the end.

How it Works Internally

Thread 1 might compute:

sum1=a[0] + a[1] + a[2]

Thread 2:
sum2=a[3] + a[4] + a[5]

Thread 3:
sum3=a[6] + a[7]

Thread 4:
sum4=a[8] + a[9]

OpenMP internally performs:
final sum = sum1 + sum2 + sum3 + sum4

3. `#pragma omp critical`

Purpose

- **Ensures a block of code is executed by only one thread at a time.**
- **Used to prevent race conditions, especially when updating shared variables.**
- **The OpenMP `critical` directive is used to protect a section of code so that only one thread executes it at a time.**
- **It prevents race conditions, where multiple threads try to update shared data simultaneously.**

✔ When to use?

- **Updating shared counters**
- **Printing from multiple threads**
- **Writing to shared files**
- **Updating global variables**

Syntax

```
#pragma omp critical
{
```

```
    // code executed by only one thread at a time
}
```

Example

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

int main() {
    int sum = 0;

    #pragma omp parallel
    {
        int local = omp_get_thread_num();   // local value

        // Only ONE thread at a time can update 'sum'
        #pragma omp critical
        {
            sum += local;
            printf("Thread %d updated sum\n", local);
        }
    }

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

Expected Output (Example for 4 threads)

```
Thread 2 updated sum
Thread 0 updated sum
Thread 3 updated sum
Thread 1 updated sum
Final Sum = 6
```

Explanation

- **Each thread computes a local value.**
- **The `critical` section ensures that only one thread updates `sum` at a time.**
- **Prevents race conditions.**

🔍 Line-by-Line Explanation

1. `int sum = 0;`

A shared variable that all threads will update.

2. `#pragma omp parallel`

Creates a team of threads (default = number of CPU cores or OMP_NUM_THREADS).

Example: If system has 4 threads → threads: 0,1,2,3

3. `int local = omp_get_thread_num();`

Each thread gets its unique ID:

- **Thread 0 → local = 0**
- **Thread 1 → local = 1**
- **Thread 2 → local = 2**
- **Thread 3 → local = 3**

4. `#pragma omp critical`

This section is executed by one thread at a time to avoid race conditions.

Inside this section:

- **Each thread adds its ID to `sum`**
- **Prints which thread updated the sum**

5. Final output

After all threads finish, the main thread prints the total sum.

Why critical section is needed?

Without `critical`, threads would execute:

```
sum = sum + local;
```

simultaneously → race condition → wrong output.

`critical` ensures one-by-one update.

Summary Table

Directive	Purpose	Key Feature
`#pragma omp parallel`	Creates multiple threads	Runs same block on all threads
`#pragma omp for`	Splits loop iterations across threads	Automatic loop parallelization
`#pragma omp critical`	Protects shared data updates	Only one thread enters block at a time

Q)What problems can arise if critical sections are not used correctly in Open MP?

ANSWER

1. Problems When Critical Sections Are Not Used Correctly in OpenMP

If `#pragma omp critical` or other synchronization mechanisms are NOT used properly, several problems can occur in shared-memory parallel programs:

a) Race Conditions

Multiple threads may try to read and write shared variables at the same time, leading to unpredictable and incorrect results.

b) Data Corruption

Because threads update shared data simultaneously, the final value may be corrupted or inconsistent.

c) Non-deterministic Output

The program may produce different output each time it runs, making debugging extremely difficult.

d) Lost Updates

One thread's update to a shared variable may overwrite another thread's update, causing inaccurate computation.

e) Program Crashes or Deadlocks

If synchronization is misused, threads may wait forever or cause segmentation faults.

f) Incorrect Behavior in Accumulation Loops

Parallel summations without critical or reduction can produce wrong values (e.g., wrong total or missing elements).