Task Parallel, Shared/Distributed Memory, Synchronization

Task Parallelism

---

1. Definition

Task Parallelism is a type of parallelism where **different tasks or functions of a program execute concurrently** on **different processors or threads**, rather than the same operation on multiple data elements.

- Each task may operate on **the same or different data**.
- Focuses on **functional decomposition** rather than **data decomposition**.

---

2. Key Concepts

Concept	Description
Task	A unit of computation (function, thread, or process)
Concurrency	Tasks run simultaneously, possibly on different cores or nodes
Communication	Tasks may need to share data via shared memory or message passing
Synchronization	Required if tasks depend on each other or share resources
Contrast with Data Parallelism	Data parallelism applies the same operation to multiple data elements; task parallelism executes **different operations**

---

3. Illustrative Example

Scenario: Web server handling multiple requests

- Task 1 → Authenticate user
- Task 2 → Retrieve data from database
- Task 3 → Log request
- Task 4 → Send response

Implementation:

- Each task runs on a separate thread or processor.
- Tasks that do not depend on each other can execute **simultaneously**, improving throughput.

Code Snippet (Pseudo-OpenMP Example):

Code Snippet (Pseudo-OpenMP Example):

c Copy code

```c
#pragma omp parallel sections
{
    #pragma omp section
    authenticate_user();

    #pragma omp section
    retrieve_data();

    #pragma omp section
    log_request();

    #pragma omp section
    send_response();
}
```

- Each section executes **concurrently** on a separate thread.

---

4. Behavior in Parallel and Distributed Systems

Feature	Parallel Systems	Distributed Systems
Memory Access	Shared memory among threads	Each node may have local memory; tasks communicate via messages
Synchronization	Mutex, semaphore, barriers	Message passing, distributed coordination
Granularity	Coarse-grained or fine-grained tasks	Coarse-grained tasks preferred to reduce communication overhead

Scalability	Limited by cores and shared memory bus	Scales across nodes; dependent on network latency
Use Cases	GUI applications, multithreaded programs	Distributed pipelines, workflow systems, cloud computing

---

5. Key Points

- **Task parallelism = functional parallelism**: different tasks on same/different data.
- Improves **throughput and resource utilization**.
- Requires careful **synchronization** if tasks share resources.

Shared Memory vs Distributed Memory

---

1. Definition

A. Shared Memory

- **Definition:** A memory model where **all processors/threads can access a single, global memory space**.
- **Key Idea:** Processors communicate **implicitly** by reading/writing shared variables.
- **Used In:** Multi-core CPUs, SMP (Symmetric Multiprocessing) systems.

B. Distributed Memory

- **Definition:** Each processor has its **own local memory**; processors communicate **explicitly** via message passing.
- **Key Idea:** No direct access to other processors' memory; all communication is **explicit**.
- **Used In:** Cluster computing, HPC clusters, grid computing.

---

2. Comparison Table

Feature	Shared Memory	Distributed Memory
Memory Access	Global shared memory accessible by all processors	Private local memory per processor
Communication	Implicit via memory	Explicit via messages (MPI, TCP/IP)

Synchronization	Required to avoid race conditions (mutex, semaphore)	Required to coordinate messages and task execution
Programming Complexity	Moderate; threads share data	High; explicit message passing needed
Scalability	Limited by memory bus and contention	Highly scalable across nodes
Fault Tolerance	Low; memory corruption affects all threads	Higher; node failure affects only local tasks
Example Systems	OpenMP, Pthreads, multi-core CPUs	MPI clusters, cloud nodes, distributed databases

---

3. Illustrative Example

Problem: Sum an array of 16 numbers using 4 processors

Shared Memory Approach (OpenMP)

- All threads can access the same array in memory
- Parallel sum with critical section or atomic operations

```c
#pragma omp parallel for reduction(+:sum)
for(int i=0; i<16; i++){
    sum += array[i];
}
```

- **Notes:** Threads share memory; synchronization is implicit via `reduction`.

Distributed Memory Approach (MPI)

- Each processor gets a **local portion** of the array
- Partial sums computed locally
- Final sum sent to master processor via message passing

```c
MPI_Scatter(array, 4, MPI_INT, local_array, 4, MPI_INT, 0, MPI_COMM_WORLD);
local_sum = sum(local_array);
MPI_Reduce(&local_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

- **Notes:** Memory is local; communication required to combine results.

---

4. Behavior in Parallel and Distributed Systems

Feature	Shared Memory Systems	Distributed Memory Systems
Communication	Fast, via memory bus	Network-based; slower
Synchronization	Locks, barriers	Messages, handshakes
Programming Model	Thread-based (OpenMP, Pthreads)	Process-based (MPI)
Scalability	Limited (cores per machine)	Very scalable (clusters, cloud)
Fault Tolerance	Low	High, dynamic reassignment possible
Typical Use Cases	Multi-threaded applications, HPC within single machine	Large-scale simulations, grid/cloud computing

---

✅ **Key Points**

- **Shared Memory:** easier to program, low latency, limited scalability
- **Distributed Memory:** more complex, high latency, highly scalable
- **Hybrid Systems:** combine shared memory **within nodes** and distributed memory **across nodes** for optimal performance

Synchronization

---

1. Definition

Synchronization is the process of **coordinating the execution of parallel or distributed tasks** to ensure **correctness, data consistency, and proper sequencing**.

- Required when **multiple threads or processes access shared resources** or when tasks have **dependencies**.
- Prevents **race conditions, deadlocks, and inconsistent data**.

---

2. Key Concepts

Concept	Description
Race Condition	Occurs when two or more processes access shared data concurrently and the result depends on execution order
Critical Section	Code segment where shared resources are accessed; only one thread/process should execute it at a time
Atomic Operation	Operation that executes **completely or not at all**, preventing interruption
Barrier	Synchronization point where all threads/processes must reach before continuing
Deadlock	Situation where processes wait indefinitely for each other's resources

---

3. Types of Synchronization

A. In Parallel Systems (Shared Memory)

1. **Mutex / Locks**
   - Protect critical sections
   - Example: `pthread_mutex_lock()` in C
2. **Semaphore**
   - Counting mechanism to allow limited concurrent access
   - Example: controlling number of threads accessing a resource
3. **Barrier**
   - Threads wait until all threads reach the barrier
   - Example: OpenMP `#pragma omp barrier`
4. **Atomic Operations**
   - Ensure single-step execution for operations like increment/decrement
   - Example: `#pragma omp atomic`

B. In Distributed Systems (Distributed Memory)

1. **Message Passing Synchronization**
   - Use messages to signal readiness or completion
   - Example: `MPI_Barrier()`
2. **Logical Clocks / Lamport Timestamps**
   - Ensure proper **event ordering** across nodes
3. **Distributed Locks**
   - Control access to shared resources in a distributed environment
   - Example: Using Zookeeper or distributed semaphore

---

4. Illustrative Examples

Shared Memory Example (OpenMP)

```c
int sum = 0;
#pragma omp parallel for
for(int i=0; i<100; i++){
    #pragma omp atomic
    sum += i; // ensures thread-safe increment
}
```

Distributed Memory Example (MPI Barrier)

```c
// Each process computes local sum
local_sum = compute_local_sum();

// Synchronize all processes before reduction
MPI_Barrier(MPI_COMM_WORLD);

// Reduce local sums to total sum
MPI_Reduce(&local_sum, &total_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

5. Behavior in Parallel and Distributed Systems

Feature	Parallel Systems (Shared Memory)	Distributed Systems (Cluster/Nodes)
Memory Access	Shared memory; threads synchronize on critical sections	Local memory; synchronization via messages
Mechanisms	Mutex, semaphore, barrier, atomic	MPI_Barrier, message acknowledgment, distributed locks
Overhead	Low to moderate	Higher due to network latency
Scalability	Limited by memory bus contention	Scales well but communication overhead increases

Use Cases	Multithreaded computation, simulations	Distributed workflows, HPC clusters, cloud applications

---

✅ Key Points

- Synchronization **prevents data inconsistency** in parallel and distributed tasks.
- Shared memory uses **locks, barriers, atomic operations**, while distributed systems rely on **message passing and distributed coordination**.
- Proper synchronization **maximizes correctness** without unnecessarily reducing parallel efficiency.