

## SIMD Systems (Single Instruction, Multiple Data)

- **Definition:** One control unit, multiple datapaths.
    - A single instruction is broadcast, and each datapath applies it to its data item (or stays idle).
  - **Example:** Vector addition ( $x[i] += y[i]$ ).
    - If system has  $n$  datapaths  $\rightarrow$  each datapath handles one element.
    - If system has  $m < n$  datapaths  $\rightarrow$  execute in chunks of size  $m$ .
  - **Issue:** Datapaths must do the **same instruction synchronously**.
    - If conditionals exist (e.g., if ( $y[i] > 0$ )), some datapaths will be idle  $\rightarrow$  performance loss.
  - **Best for:** Large array operations, **data parallelism**.
  - **Limitations:** Not good for irregular problems, limited flexibility.
  - **History:**
    - 1990s: Thinking Machines  $\rightarrow$  SIMD supercomputers.
    - Late 1990s: SIMD mainly in **vector processors**.
    - Now: SIMD in **GPUs** and CPUs.
- 

## Vector Processors

Operate on **vectors (arrays)** instead of scalars. Key features:

1. **Vector Registers**
  - Hold multiple operands (4–256 elements).
2. **Vectorized & Pipelined Units**
  - Same operation applied to all vector elements in parallel (SIMD-style).
3. **Vector Instructions**
  - Operate on entire vectors (e.g., one load-add-store for a block vs. element-wise in scalar CPUs).
4. **Interleaved Memory**
  - Multiple banks allow near-simultaneous access  $\rightarrow$  high memory bandwidth.
5. **Strided & Scatter/Gather Access**
  - Efficiently handle data with patterns (e.g., every 4th element) or irregular access with hardware support.

**Advantages:**

- Fast for many applications.
- Compilers can automatically vectorize loops.
- Efficient memory use (every loaded element is used).

**Limitations:**

- Not good with irregular data structures.
  - Limited scalability (vector length can't grow indefinitely).
  - Long-vector processors are expensive → custom-built.
  - Commodity CPUs/GPU provide SIMD but mostly for **short vectors**.
- 

**Graphics Processing Units (GPUs) – Key Points (10 Marks)**

**1. Graphics Pipeline & Shaders**

- GPUs use a graphics pipeline to convert objects (points, lines, triangles) into pixels.
- Programmable stages use **shader functions** (short C-like code), applied in parallel.

**2. SIMD Parallelism**

- Shader operations on nearby elements follow similar control flow.
- GPUs exploit **SIMD parallelism** with many datapaths (e.g., 128 per core).

**3. High Data Requirements**

- Single images may require **hundreds of MB** of data.
- GPUs use **hardware multithreading** to hide memory stalls, maintaining many suspended threads per core.

**4. Performance Characteristics**

- Excellent for large problems with massive data.
- Relatively poor performance on **small problems** due to overhead of keeping many threads busy.

**5. Architecture**

- Not pure SIMD → GPUs support **both SIMD and MIMD**.
- Dozens of cores, each capable of running independent instruction streams.

**6. Memory**

- Can use **shared or distributed memory**.
- Large GPU systems often combine both, with communication over networks if needed.

**7. General-Purpose Use**

- GPUs are widely used in **high-performance computing (HPC)**.
  - Programming models (like CUDA, OpenCL) allow users to exploit GPU power.
- 

## MIMD Systems (Multiple Instruction, Multiple Data)

### 1. Definition

- Support **multiple instruction streams** operating on **multiple data streams**.
- Consist of **independent processors/cores**, each with its **own control unit and datapath**.

### 2. Asynchronous Operation

- Processors run at their **own pace** (no global clock needed).
- Even if executing the same program, they may be at **different points** in the code.

### 3. Types of MIMD Systems

- **Shared-Memory Systems**
  - Processors connected to a **common memory** via interconnection network.
  - Communication is **implicit** through shared data structures.
- **Distributed-Memory Systems**
  - Each processor has its **own private memory**.
  - Communication is **explicit** via message passing or remote memory access.

### 4. Comparison with SIMD

- SIMD → one instruction, many data (synchronous).
  - MIMD → many instructions, many data (asynchronous, flexible).
- 

## Shared-memory Systems

- **Definition:** Multiple cores/processors share access to the same main memory.
- **Multicore processors:**
  - Each core usually has **private L1 cache**.
  - Higher-level caches may be **shared**.
- **Types:**

### 1. UMA (Uniform Memory Access)

- All processors access memory with **equal latency**.
- **Easier to program**, but memory may become a bottleneck.

### 2. NUMA (Non-Uniform Memory Access)

- Each processor has **direct access** to part of memory; other parts accessed via interconnect.
  - **Faster local access**, scalable, supports **larger memory**, but harder to program.
- 

### Distributed-memory Systems

- **Definition:** Each processor has its **own private memory**; processors communicate via a **network**.
  - **Clusters** (most common):
    - Built from commodity systems (e.g., PCs) + commodity interconnect (e.g., Ethernet).
    - Each node is typically a **shared-memory multicore system** → called **hybrid systems**.
  - **Grid Systems:**
    - Large geographically distributed systems combined into one.
    - Typically **heterogeneous** (different hardware types).
- 

### 2.3.4 Interconnection Networks

#### Importance

- Performance of parallel systems (both shared-memory & distributed-memory) depends heavily on **interconnect speed**.
  - Even fast processors & memory will suffer if the interconnect is slow.
- 

#### 1. Shared-Memory Interconnects

- **Bus (older)**
  - Collection of shared wires.
  - **Advantages:** Low cost, flexible (easy to add devices).
  - **Disadvantages:** Shared medium → contention increases with more processors → poor scalability.
- **Switched Interconnects (modern)**
  - Use switches to route data between cores and memory.
  - **Crossbar:**
    - Each processor has a dedicated path to memory (if not contending for same module).
    - **High speed**, supports simultaneous communication.
    - **Disadvantage:** Expensive (switches + links).

---

## 2. Distributed-Memory Interconnects

- Divided into **Direct** and **Indirect** networks.

### Direct Interconnects

- Switches directly connected to processor–memory pairs.
- **Examples:**
  - **Ring:** Simple, better than bus, but limited connectivity (bisection width = 2).
  - **2D Toroidal Mesh:** More links ( $2p$  vs  $p$ ), better connectivity than ring, but more expensive.
  - **Fully Connected Network:** Ideal (bisection width =  $p^2/4$ ), but impractical (too many links).
  - **Hypercube:**
    - $d$ -dimensional cube,  $p = 2^d$  nodes.
    - Each node connected to  $d$  others.
    - **Bisection width =  $p/2$ :** high connectivity, but switch complexity grows with  $\log_2(p)$ .
- **Measures:**
  - **Bisection width** = number of links needed to split network into two equal halves.
  - **Bisection bandwidth** = total data transfer rate across those links.

### Indirect Interconnects

- Switches not directly connected to processors.
- **Examples:**
  - **Crossbar:** High connectivity, simultaneous communication possible, but very expensive ( $O(p^2)$  switches).
  - **Omega Network:**
    - Built with  $2 \times 2$  crossbar switches.
    - Cheaper than full crossbar ( $O(p \log p)$  switches).
    - Some communications block others (less flexible).

---

## Distributed-Memory Interconnects

Distributed-memory systems rely on interconnection networks for processor-to-processor communication. They are classified into **direct** and **indirect** networks.

---

### 1. Direct Interconnects

### **Definition:**

Each switch is directly connected to a processor-memory pair, and switches are connected to each other.

### **Examples**

#### **1. Ring**

- Each switch connects to 2 neighbors.
- Links =  $p$  (number of processors).
- **Bisection width = 2** (worst-case).
- **Pros:** Simple, better than bus, allows multiple simultaneous communications.
- **Cons:** Limited connectivity, possible blocking.

#### **2. 2D Toroidal Mesh**

- Each switch connects to 4 neighbors (+ processor).
- Links =  $2p$  (vs  $p$  in ring).
- **Bisection width =  $2\sqrt{p}$ .**
- **Pros:** More simultaneous communication patterns.
- **Cons:** More expensive, higher switch complexity (5 ports).

#### **3. Fully Connected Network**

- Each node connects directly to all others.
- Links =  $(p^2/2 - p/2)$ .
- **Bisection width =  $p^2/4$  (ideal).**
- **Pros:** Maximum connectivity.
- **Cons:** Impractical for large  $p$  (too costly).

#### **4. Hypercube**

- $d$ -dimensional cube with  $p = 2^d$  nodes.
- Each node connects to  $d$  other nodes.
- **Bisection width =  $p/2$ .**
- Switch degree =  $\log_2(p) + 1$  (processor +  $d$  links).
- **Pros:** Very high connectivity, scalable.
- **Cons:** More complex and costly than mesh.

---

### **Measures of Connectivity**

- **Bisection Width** = min number of links to cut the network into equal halves.

- **Bisection Bandwidth** = (bisection width  $\times$  link bandwidth).

**Example:**

- Ring with bandwidth = 1 Gbps  $\rightarrow$  bisection bandwidth = 2 Gbps.
  - Mesh with  $\sqrt{p}$  rows  $\rightarrow$  bisection bandwidth =  $2\sqrt{p} \times$  bandwidth.
- 

## 2. Indirect Interconnects

**Definition:**

Switches are arranged in a switching network. Processors connect to the network but not directly to every other processor.

**Examples**

### 1. Crossbar

- Every input connected to every output.
- Total switches =  $p^2$ .
- **Pros:** Any processor can communicate simultaneously (non-blocking).
- **Cons:** Expensive, impractical for large p.

### 2. Omega Network

- Built with  $2 \times 2$  switches in  $\log_2(p)$  stages.
  - Total switches =  $2p \log_2(p)$ .
  - **Pros:** Cheaper than crossbar, scalable.
  - **Cons:** Blocking possible (not all pairs can communicate simultaneously).
- 

## Cache Coherence in Shared-Memory Systems

### 1. The Problem

- In **shared-memory multiprocessors**, each core typically has a **private cache**.
  - If multiple cores cache the same variable, updates made by one core may **not be visible** to others immediately.
  - Example:
    - Core 0 writes  $x = 7$
    - Core 1 still reads the old cached value  $x = 2$ .
  - This leads to **inconsistent views of memory**, known as the **cache coherence problem**.
- 

### 2. Cache Coherence Approaches

#### (a) Snooping Protocol

- Based on **broadcasting updates** over a shared bus/interconnect.
  - When a core updates a cache line, it **sends a signal** on the bus.
  - Other cores “snoop” the bus and either:
    - **Invalidate** their copy (invalid state), or
    - **Update** their copy.
  - Works with both **write-through** and **write-back** caches.
  - **Pros:** Simple, fast for small systems.
  - **Cons:** Not scalable (broadcast overhead increases with number of cores).
- 

#### (b) Directory-Based Protocol

- A **directory** keeps track of which cores hold a copy of each cache line.
  - On an update, the directory **notifies only those cores** that have cached the variable.
  - Directory can be distributed (each core manages part of it).
  - **Pros:** Scales to large systems, avoids full broadcasts.
  - **Cons:** Requires extra storage and directory lookup overhead.
- 

### 3. False Sharing

- Cache works with **cache lines** (e.g., 64 bytes), not individual variables.
- If two cores update **different variables** that lie in the **same cache line**, the line keeps getting invalidated unnecessarily.
- Example:
  - Array  $y[8]$  of doubles (8 bytes each).
  - Stored in **1 cache line (64 bytes)**.
  - Core 0 updates  $y[0..3]$ , Core 1 updates  $y[4..7]$ .
  - Even though they touch different elements, the **entire line** is repeatedly invalidated.
- This is **false sharing** → correct results, but **very poor performance**.

**Fix:** Use **temporary private storage** or padding to ensure different cores access **different cache lines**.

---

### Shared-Memory vs. Distributed-Memory Systems

#### 1. Shared-Memory Systems

- All processors share a **common address space**, so data can be accessed directly by any processor.
- Programming is **simpler**: communication is implicit via shared variables.

- **Hardware Examples:** Multicore processors, UMA and NUMA architectures.
  - **Advantages:**
    - Easier to program, especially for beginners.
    - No need for explicit message passing.
  - **Limitations:**
    - **Scaling problem:** As more processors are added, contention for the bus or interconnect increases.
    - Large crossbars are **expensive** and impractical beyond a few processors.
- 

## 2. Distributed-Memory Systems

- Each processor has its **own private memory**.
  - Processors communicate **explicitly** via messages over an interconnection network.
  - **Hardware Examples:** Clusters, hypercube networks, toroidal meshes.
  - **Advantages:**
    - Easier to **scale to thousands of processors**.
    - Interconnect cost grows moderately compared to large shared-memory systems.
    - Well-suited for **large computations and massive data sets**.
  - **Limitations:**
    - Programming is more complex: requires **explicit message passing**.
    - Communication overhead can affect performance if not carefully managed.
- 

### 2.4.3 Shared-Memory Programming

#### Shared and Private Variables

- **Shared variables:** Accessible and modifiable by all threads.
  - **Private variables:** Accessible only by the thread that owns them.
  - **Communication among threads** is usually **implicit**, via shared variables.
- 

#### Dynamic vs. Static Threads

- **Dynamic threads:**
  - Master thread forks worker threads on demand.
  - Efficient use of resources: threads use resources only while running.
- **Static threads:**

- All threads are forked at the beginning and run until completion.
  - Can provide **better performance** if resources allow, as fork/join overhead is avoided.
  - Closer to distributed-memory paradigms, preserving a consistent mindset.
- 

## Nondeterminism

- Occurs in MIMD systems where threads execute asynchronously.
  - **Definition:** A computation is **nondeterministic** if the same input can produce different outputs due to thread execution order.
  - Example: Printing private variables from two threads may yield different output orders across runs.
- 

## Race Conditions

- Occur when multiple threads simultaneously access and modify a **shared resource**.
  - Example:
    - `my_val = Compute_val(my_rank);`
    - `x += my_val; // Shared variable x`
      - Threads may read, modify, and write x in overlapping sequences, producing **incorrect results**.
  - **Solution:** Ensure **atomic operations or mutual exclusion**.
- 

## Critical Sections and Mutual Exclusion

- **Critical section:** A code block that must be executed by **only one thread at a time**.
  - **Mutex (Mutual Exclusion Lock):** Ensures that only one thread enters a critical section at a time.
    - `Lock(&add_my_val_lock);`
    - `x += my_val;`
    - `Unlock(&add_my_val_lock);`
  - **Busy-waiting:** Thread repeatedly checks a condition to enter a critical section. Simple but resource-wasteful.
  - **Semaphores and Monitors:** Higher-level constructs to manage **mutual exclusion and synchronization**.
- 

## Thread Safety

- **Thread-safe functions:** Safe to use in multithreaded programs; no shared data conflicts.
  - **Non-thread-safe functions:** May cause errors if multiple threads access shared data.
    - Example: C function strtok uses static variables and is **not thread safe**.
  - Programmers must identify and modify **non-thread-safe code** to avoid errors in multithreaded programs.
- 

## Nondeterminism and Race Conditions in Shared-Memory Programs

### Nondeterminism

- Occurs in **MIMD systems** where threads execute asynchronously.
- **Definition:** A program is nondeterministic if the same input can produce **different outputs** due to unpredictable thread execution order.
- Example: Two threads printing private variables my\_x (7 and 19) may output in any order:
  - Thread 0 > my\_x = 7
  - Thread 1 > my\_x = 19

or

Thread 1 > my\_x = 19

Thread 0 > my\_x = 7

- Nondeterminism becomes **problematic** when threads update **shared resources**, potentially causing incorrect results.
- 

### Race Condition

- Occurs when two or more threads **simultaneously access a shared resource**, and the result depends on the order of execution.
- Example: Adding thread-local values my\_val to a shared variable x:
  - my\_val = Compute\_val(my\_rank);
  - x += my\_val;
    - Interleaved execution can produce incorrect x due to concurrent reads and writes.

---

### Critical Sections and Atomicity

- **Critical section:** A code block that must be executed by only **one thread at a time** to ensure correctness.
- **Atomic operation:** Appears indivisible; no other thread can modify the memory location while the operation is in progress.
- **Mutex (Mutual Exclusion Lock):** Ensures exclusive access:

- `Lock(&add_my_val_lock);`
  - `x += my_val;`
  - `Unlock(&add_my_val_lock);`
    - Enforces **serialization** but introduces some performance overhead.
  - **Best practices:**
    - Minimize the number and size of critical sections to reduce serialization impact.
- 

### Alternatives to Mutexes

- **Busy-waiting:** Thread repeatedly checks a condition before entering a critical section.
  - `while (!lok_for_1); // Wait until allowed`
  - `x += my_val;`
    - Simple but **resource-wasteful**.
  - **Semaphores:** Similar to mutexes, often used for more flexible synchronization.
  - **Monitors:** Higher-level construct; ensures that **object methods are executed by only one thread at a time.**
- 

## Distributed-Memory Parallel Programming

### Key Characteristics

- Each core has **private memory**; cores **cannot directly access each other's memory**.
  - Programs are usually executed as **multiple processes**, not threads, because processes may run on **independent CPUs with independent OSes**.
  - Communication between processes is **explicit**.
- 

### Message-Passing

- Most common API for distributed-memory programming.
- **Core operations:** Send and Receive.
- Processes are identified by **ranks (0 to p-1)**.

### Example:

```
if (my_rank == 1)
    Send(message, 100, 0); // send to process 0
else if (my_rank == 0)
    Receive(message, 100, 1); // receive from process 1
```

- **SPMD model:** All processes run the same program but act differently based on rank.

- Variants of Send/Receive:
    - **Blocking send:** Waits until receive starts.
    - **Non-blocking send:** Returns after copying message.
  - **Collective operations:** Broadcast, reduction, gather, scatter, etc.
  - MPI (Message-Passing Interface) is the **most widely used API**.
  - Message-passing is low-level and requires **major rewriting of serial programs**.
- 

### One-Sided Communication (Remote Memory Access)

- Only **one process actively participates** in the communication.
  - Can **reduce synchronization overhead**, but **requires careful management** to avoid race conditions and ensure correctness.
- 

### Partitioned Global Address Space (PGAS)

- Provides a **shared-memory style programming model** on distributed-memory systems.
- Key features:
  - Private variables are local to a core.
  - Shared data distribution is controlled by the programmer.
- Ensures **predictable performance** by keeping memory access mostly local.
- Example: Vector addition where each process operates only on its local partition of arrays:

```
for (i = my_first_element; i <= my_last_element; i++)  
    x[i] += y[i]; // local memory access
```

---