MODULE-1

Introduction to parallel programming, Parallel hardware and parallel software – Classifications of parallel computers, SIMD systems, MIMD systems, Interconnection networks, Cache coherence, Shared-memory vs. distributed-memory, Coordinating the processes/threads, Shared-memory, Distributed-memory.

What is Parallel Programming?

Parallel programming is a computing paradigm where many operations are carried out simultaneously to solve a problem. It divides a task into multiple subtasks and processes them concurrently using multiple processors or cores.

For example, when adding elements of a large array, instead of using one processor to add each element sequentially, the array is split into parts, and each processor adds its part at the same time. This results in faster computation.

Parallel programming is a **method of computation** where many calculations or processes are carried out simultaneously. This is achieved by **dividing a task** into subtasks and executing them concurrently on **multiple processors or cores**.

Example: Imagine calculating the sum of a large array. Instead of one processor adding all elements sequentially, the array is divided into chunks and processed by different cores in parallel.

Why Do We Need Parallel Programming?

Historically, increasing the clock speed of processors made programs run faster. However, this approach reached a limit due to physical constraints such as power consumption and heat dissipation.

Today, instead of increasing clock speed, manufacturers add more cores to processors. To fully utilize these cores, we must write programs that run tasks in parallel.

Parallel programming allows for better performance and efficiency in tasks like simulations, data analysis, image processing, and real-time applications.

In earlier days, computers became faster by increasing **clock speed** (GHz). But now, this has reached a limit due to:

- **Power consumption**
- **Heat dissipation**
- **Physical limits of silicon chips**

To continue improving performance, we now add **multiple cores (multi-core CPUs)** or use **clusters of machines** to solve bigger and faster problems.

Real-life Uses:

- Weather forecasting
- Image and video processing
- Machine Learning and AI

- Scientific simulations

Types of Parallelism

Data Parallelism: Focuses on performing the same operation on different pieces of data simultaneously. Example: Applying a filter to every pixel in an image.

Task Parallelism: Involves executing different tasks or functions at the same time. Example: A computer playing music while downloading a file and running a virus scan.

a. Data Parallelism

- The same task is performed on different parts of the data.
- Useful when the **same operation** is to be applied to a **large dataset**.

Example: Adding two arrays element by element.

```
for (i = 0; i < n; i++)
  C[i] = A[i] + B[i];
```

Each C[i] can be computed independently in parallel.

b. Task Parallelism

- Different tasks or functions are executed at the same time, possibly on different data.

Example: In a web browser:

- One thread renders a page,
- Another downloads data from the server.

Both are parallel but perform **different tasks**.

1.1 Parallel Hardware

In modern computing, hardware that enables **parallel execution** plays a significant role in enhancing performance. Although **multiple issue** and **pipelining** allow multiple operations within a processor to be executed in parallel, these mechanisms are not directly observable or controllable by programmers. For the purpose of parallel programming, we define *parallel hardware* as that which is **visible to the programmer** and whose capabilities can be **exploited or must be adapted to** through source code changes.

1.1.1 Classifications of Parallel Computers

Flynn's Taxonomy and Memory-Based Classification

Parallel computers can be classified using two distinct approaches. One of the most commonly referred schemes is **Flynn's taxonomy**, which categorizes systems based on the number of **instruction streams** and **data streams** they handle simultaneously.

A traditional system based on the **von Neumann architecture** is called a **SISD** system—**Single Instruction Stream, Single Data Stream**—since it processes one instruction and one data item at a time. In contrast, parallel systems can either follow the **SIMD** model (**Single Instruction, Multiple Data**) or the **MIMD** model (**Multiple Instruction, Multiple Data**).

SIMD systems apply the same instruction to multiple data items concurrently, whereas MIMD systems allow independent instruction streams operating on different data.

Another classification scheme focuses on how memory is accessed by processing units. In a **shared memory system**, all cores access a common memory space and synchronize via shared variables. In a **distributed memory system**, each core operates with its **private memory**, and coordination happens through **explicit communication**, often over a network.

1.1.2 SIMD Systems

SIMD (Single Instruction, Multiple Data) architectures apply a **single instruction** simultaneously to **multiple data items**, making them highly effective for **data-parallel operations**. Internally, a SIMD system has a **single control unit** that broadcasts instructions to several **datapaths** or **processing elements**. Each datapath processes a portion of the data, either executing the instruction or remaining **idle** if there's no matching data.

Consider the task of **vector addition**, where two arrays x and y of length n need to be added element-wise:

```
for (i = 0; i < n; i++)
   x[i] += y[i];
```

If there are n datapaths, all elements can be processed in one cycle. If there are fewer datapaths, say m, the operation is performed in **blocks of m elements**. With m = 4 and n = 15, elements are added in four separate stages: 0–3, 4–7, 8–11, and 12–14. In the final group, one datapath remains idle, illustrating **underutilization** due to unequal division.

Challenges arise when conditional logic is involved. For example:

```
for (i = 0; i < n; i++)
   if (y[i] > 0.0)
      x[i] += y[i];
```

Here, **branching logic** forces some datapaths to **idle** while others execute, reducing efficiency. Also, traditional SIMD systems operate **synchronously**, meaning all datapaths must **wait** for the next instruction to be broadcast. They do not store instructions, so they cannot **defer execution**.

Nevertheless, SIMD is highly effective for processing **large, uniform datasets**, such as image pixels or signal samples. It excels when the same instruction must be applied to many data points, making it useful for **graphics**, **matrix operations**, and **scientific computing**. Over time, SIMD systems have evolved. Initially, companies like **Thinking Machines** pioneered their use in **supercomputing**. Later, their prominence declined, leaving **vector processors** as the most notable SIMD representatives. Today, **GPUs** and **desktop CPUs** often integrate SIMD features to accelerate multimedia and numeric processing.

Vector Processors

Vector processors are computing systems designed to perform operations on **entire arrays or vectors of data** simultaneously. In contrast, traditional **CPUs** process one data element at a time, known as **scalar processing**. Vector processors are especially effective in applications that involve performing the same operation repeatedly across large datasets.

A central component of a vector processor is the **vector register**, which holds multiple data elements and allows simultaneous operations on all elements. The **vector length** is fixed by the architecture and typically ranges from **4 to 256** elements, each being **64 bits**.

Example snippet:

```
for (i = 0; i < n; i++)
   x[i] += y[i];
```

In a vector processor, this loop can be executed with **one vector load, one vector add, and one vector store per block** of vector_length elements.

Characteristics of Vector Processors:

- **Vector Registers**: Capable of holding a complete vector of operands; support simultaneous operations on all stored elements.
- **Vectorized and Pipelined Functional Units**: Perform the same operation (e.g., addition) on all elements of a vector or between pairs of elements from two vectors in a **SIMD** fashion.
- **Vector Instructions**: Operate on entire vectors, not just scalars. One instruction can handle an entire block of elements, improving performance by reducing instruction count.
- **Interleaved Memory**: Uses multiple **independent memory banks** to allow fast loading and storing of vector elements by minimizing memory access delay.
- **Strided Memory Access & Scatter/Gather**:
    - **Strided Access** allows accessing elements at regular intervals (e.g., every 4th item).
    - **Scatter/Gather** enables reading or writing elements at irregular memory locations using dedicated hardware support.

Vector processors offer high performance and ease of use for regular data patterns. **Vectorizing compilers** can automatically detect loops that can be transformed into vector operations and provide feedback on non-vectorizable code. These systems utilize **high memory bandwidth**, ensuring efficient data use without unnecessary memory fetches.

However, they are less suitable for **irregular data structures**. Additionally, there's a practical limit to how much vector processors can **scale** by increasing the vector length. Modern architectures overcome this by **increasing the number of vector units**, not the length.

Commodity systems today offer support for **short-vector operations**, but **long-vector processors** are **custom-built and expensive**, limiting their widespread deployment.

Graphics Processing Units (GPUs)

GPUs (Graphics Processing Units) were originally designed to accelerate **real-time graphics**. They process surfaces modeled as **points**, **lines**, and **triangles** and convert them into **pixels** via a **graphics pipeline**. Some stages in this pipeline are **programmable** through short routines called **shader functions**, usually written in a subset of **C**.

Shader functions are inherently **parallel** and apply uniformly to thousands of elements like **vertices** or **fragments**. Since similar elements tend to follow the same logic path, **SIMD parallelism** is used extensively. Each **GPU core** may contain **dozens of datapaths**, enabling massively parallel execution.

A single image may involve **hundreds of megabytes of data**, so GPUs are built for **high-throughput memory access**. They avoid delays using **hardware multithreading**, which allows the state of **hundreds of threads** to be stored and quickly swapped. Thread count per core depends on the **resources consumed** by each shader, such as the number of **registers** required.

While GPUs perform exceptionally well on **large workloads**, they struggle with **small tasks** due to overhead and underutilized resources. Importantly, GPUs are not pure SIMD machines. Although their internal datapaths operate in SIMD fashion, **multiple instruction streams** can run on a single GPU, making them resemble **hybrid systems** with both **SIMD and MIMD characteristics**.

GPUs may use **shared memory**, **distributed memory**, or a combination of both. For example, multiple cores can access a **common memory block**, while other cores use a **different memory region**. Inter-core communication may involve **networked connections**. However, in typical programming models, GPUs are treated as **shared-memory systems**.

Over time, GPUs have become popular beyond graphics. Their architecture supports **high-performance computing (HPC)** tasks, including **machine learning**, **scientific simulations**, and **data analysis**.

Example:
If a shader function operates on 1000 pixels, and each GPU core can process 128 elements simultaneously, then only ~8 instruction steps are needed instead of 1000.

1.1.3 MIMD Systems

- **MIMD (Multiple Instruction, Multiple Data)** systems are a class of parallel computers where multiple processors execute different instructions on different data sets **simultaneously**.
- These systems are composed of **independent processors**, each with its **own control unit** and **local memory/data path**.

CUDA, **OpenCL**, and **SYCL** have been developed to harness their computing potential.

Key Characteristics:

1. **Asynchronous Execution:**
   - Processors operate independently without needing to synchronize their clocks.
   - Each processor may execute at its own pace, making them *asynchronous* systems.
2. **No Global Clock or Lockstep Execution:**

- Unlike SIMD systems, there is no need for a global clock or for processors to execute in lockstep (simultaneously on the same instruction).

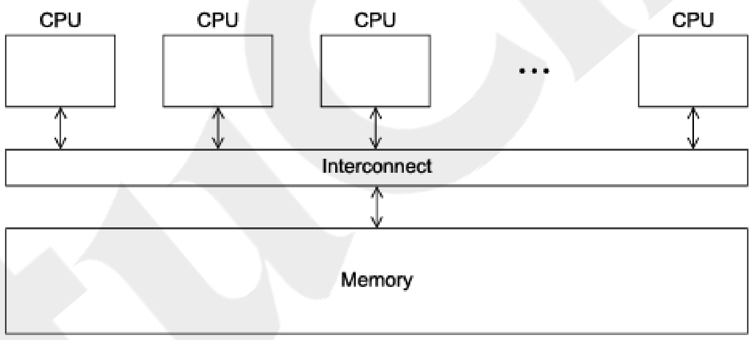
3. **Independent Control Units:**
   - Each processor has its own control unit, meaning they can run **different programs independently**.
4. **Scalability:**
   - MIMD systems scale well with increasing processors, making them suitable for large-scale computation tasks.

Types of MIMD Architectures:

MIMD systems are categorized based on **memory sharing**:

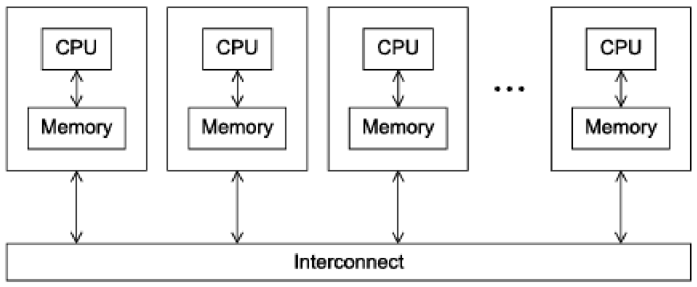
1. Shared-Memory MIMD Systems:

- All processors share a **common memory space**.
- Processors communicate via **shared memory**.
- Synchronization mechanisms (like semaphores or locks) are needed to prevent conflicts.
- Example: Multicore processors, symmetric multiprocessing systems.



2. Distributed-Memory MIMD Systems:

- Each processor has its own **local memory**.
- Processors communicate by passing **messages** over an **interconnection network**.
- Offers high scalability, suitable for cluster and grid computing.



Advantages of MIMD Systems:

- Supports a wide range of parallel applications (scientific computing, simulations, etc.).
- Flexible in executing various types of programs.
- Suitable for both **time-shared systems** and **real-time systems**.

Applications of MIMD Systems:

- Weather forecasting
- Financial modeling
- Large-scale simulations
- Machine learning and data mining

Comparison with SIMD:

Feature	**MIMD**	**SIMD**
Instruction	Multiple, independent	Single, same for all units
Data	Multiple sets	Multiple sets
Synchronization	Not required	Required
Use Case	General-purpose computation	Data-parallel applications

Shared-Memory Systems

Shared-memory systems are computer systems where **multiple processors (CPUs or cores)** access and share a **common main memory**. This model allows **inter-process communication via shared variables**, avoiding the need for explicit message passing.

Multicore Processor Basics:

- A **multicore processor** includes **multiple CPUs (cores)** on a **single chip**.
- Each core usually has its **own private Level 1 cache**.
- **L2/L3 caches** may be shared between cores.
- These systems form the basis of modern shared-memory systems.

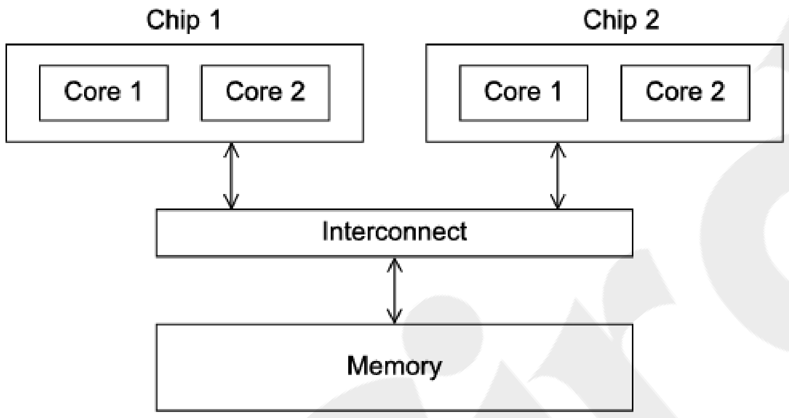
Memory Interconnection:

- In systems with multiple multicore processors:
  - The **interconnect** can link all processors **directly to the main memory**.
  - Or, each processor can connect to a **specific memory block**, with access to other blocks via the interconnect.

Types of Shared-Memory Architectures:

1. UMA – Uniform Memory Access

- All cores access memory with **equal latency** and **bandwidth**.
- Common memory is shared via a **single interconnect**.
- Easier to **program and manage**. **Example:**
- As shown in **Figure 2.5**, all cores on multiple chips are connected to **one memory block**.



Advantages:

- Simplified programming model.
- Memory access is consistent across processors.

Limitations:

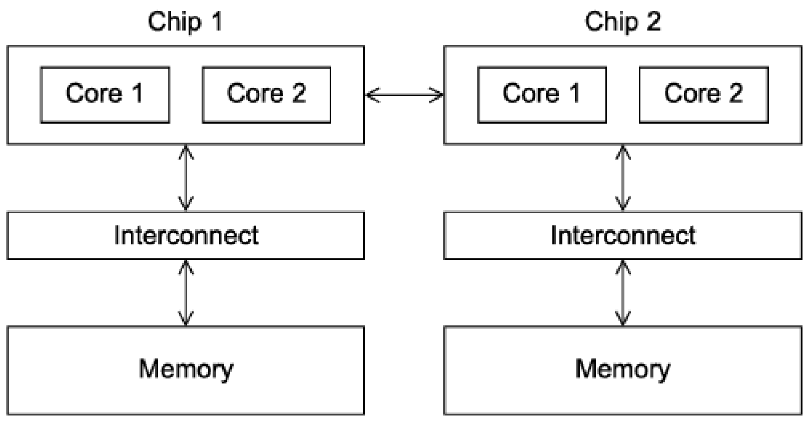
- Can become a **bottleneck** as the number of cores increases.
- Limited scalability.

2. NUMA – Non-Uniform Memory Access

- Each processor/core has its **own local memory**.
- A core can access local memory faster than remote memory.
- Access to other memory blocks goes through the **interconnect**.

Example:

- In **Figure 2.6**, each chip has its **own memory**, connected locally.



Advantages:

- **Faster access** to local memory.
- More **scalable** than UMA.
- Potential to use **larger memory spaces**.

Comparison: UMA vs NUMA

Feature	**UMA**	**NUMA**
Memory Access Time	Uniform (same for all cores)	Varies (local vs remote)
Programming Simplicity	Easier	More complex (locality aware)
Scalability	Limited	High
Performance Bottleneck	Possible at interconnect	Reduced due to local memory

Distributed-Memory Systems

Distributed-memory systems are computer architectures in which **each processor or node** has its **own local memory**, and processors **communicate by passing messages** over a network.

Cluster:

- A **cluster** is the most common form of distributed-memory system.
- It is made up of multiple **commodity systems** (like standard PCs).
- These systems are connected via a **commodity interconnection network**, such as **Ethernet**.
- Each individual computer in the cluster is known as a **node.**

Nodes:

- Nodes are the **computational units** in the system.
- In modern systems, each node is often a **shared-memory system** (e.g., multicore processor).

- When clusters consist of shared-memory nodes, the overall system is known as a **hybrid system**.

Hybrid Systems:

- These systems combine the **distributed-memory architecture at the cluster level** with **shared-memory systems within each node**.
- This structure enhances performance and scalability.

Grid Computing:

- The **grid** infrastructure connects **geographically distributed computers** into a single distributed-memory system.
- Grids are typically **heterogeneous**, meaning that:
    - The nodes may be built from different types of hardware.
    - Software and operating systems may also vary.

Characteristics of Distributed-Memory Systems:

Feature	**Description**
Memory Access	Each processor accesses only its **local memory**.
Communication	Achieved through **message passing** between nodes.
Scalability	Highly scalable, suitable for large-scale problems.
Cost	Relatively **low-cost** using commodity hardware.
Programming Complexity	Requires explicit **message-passing programming** (e.g., using MPI).

Advantages:

- Excellent for **parallel processing at scale**.
- Nodes can be added easily to increase computational power.
- **Fault tolerance** is improved due to distributed nature.

Disadvantages:

- **Programming is complex** due to manual handling of communication.
- **Latency** and **bandwidth** limitations can affect performance.
- Synchronization and data consistency across nodes must be managed carefully.

1.1.4 Interconnection Networks

The **interconnect** plays a decisive role in the performance of both **distributed-** and **shared-memory** systems: even if the processors and memory have virtually unlimited performance, a **slow interconnect** will seriously degrade the overall performance of all but the simplest parallel program.

Although some of the interconnects have a great deal in common, there are enough differences to make it worthwhile to treat **interconnects for shared-memory and distributed-memory separately**.

Shared-Memory Interconnects

If the two cores attempt to simultaneously access the same memory module. For example, Figure 2.7(c) shows the configuration of the switches if:

- **P1** writes to **M4**
- **P2** reads from **M3**
- **P3** reads from **M1**
- **P4** writes to **M2**

Crossbars allow simultaneous communication among different devices, so they are much **faster than buses**. However, the cost of the switches and links is relatively **high**. A small **bus-based** system will be much **less expensive** than a crossbar-based system of the same size.

Currently, the two most widely used interconnects on shared-memory systems are:

1. **Buses**
2. **Crossbars**

A **bus** is a collection of **parallel communication wires** together with some hardware that controls access to the bus. The key characteristic of a bus is that the communication wires are **shared** by the devices that are connected to it.

Buses have the virtue of **low cost and flexibility**: multiple devices can be connected to a bus with little additional cost. However, since the communication wires are shared, as the number of devices connected to the bus increases, the **likelihood of contention** increases, and the **expected performance decreases**.

Therefore, if we connect a large number of processors to a bus, we would expect that the processors would **frequently have to wait** for access to main memory. Thus, as the size of shared-memory systems increases, **buses are rapidly being replaced by switched interconnects**.

As the name suggests, **switched interconnects** use **switches** to control the routing of data among the connected devices.

A **crossbar** is illustrated in **Figure 2.7(a)**.

- The lines are **bidirectional communication links**,
- The **squares** are cores or memory modules,

- The **circles** are switches.

The individual switches can assume one of the two configurations shown in **Figure 2.7(b)**. With these switches and at least as many memory modules as processors, there will only be a **conflict** between two cores attempting to access memory **if they attempt to access the same module simultaneously**.

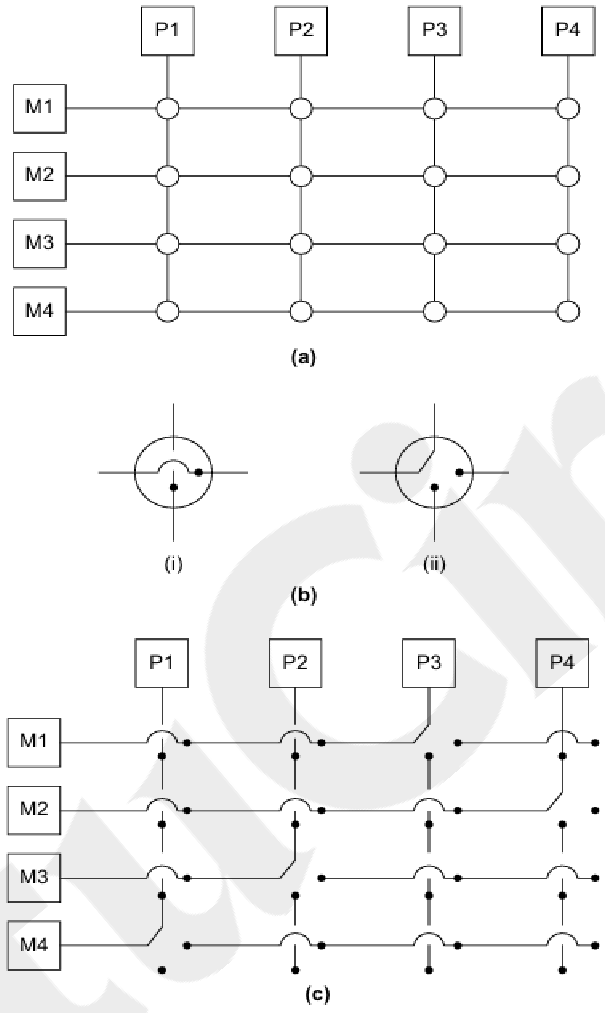


FIGURE 2.7
(a) A crossbar switch connecting four processors (P_i) and four memory modules (M_j); (b) configuration of internal switches in a crossbar; (c) simultaneous memory accesses by the processors

Distributed-memory interconnects

In **distributed-memory systems**, the communication between processor-memory pairs is handled by **interconnects**, which are mainly categorized into **direct interconnects** and **indirect interconnects**.

In a **direct interconnect**, each **switch** is directly linked to a **processor-memory pair**, and switches are also interconnected. Common examples include the **ring** and the **two-dimensional toroidal mesh**. A **ring topology** connects each node to two neighbors. It allows **multiple simultaneous communications** but can suffer from **communication bottlenecks** where some processors may have to wait. The number of **links** required in a ring is **2p**, where *p* is the number of processors. The **toroidal mesh**, in contrast, is more complex and connects each switch to **five links**, making it more expensive but allowing for a **greater number of simultaneous communication patterns**. A toroidal mesh requires **3p links** for *p* processors.

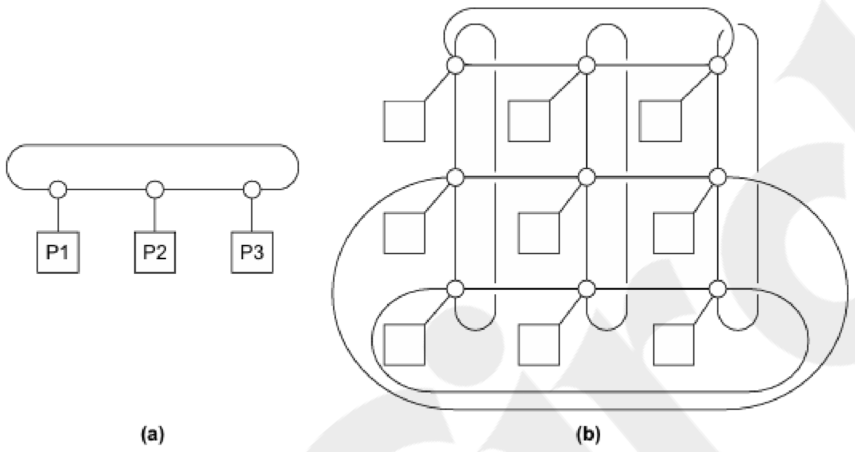


FIGURE 2.8

(a) A ring and (b) a toroidal mesh

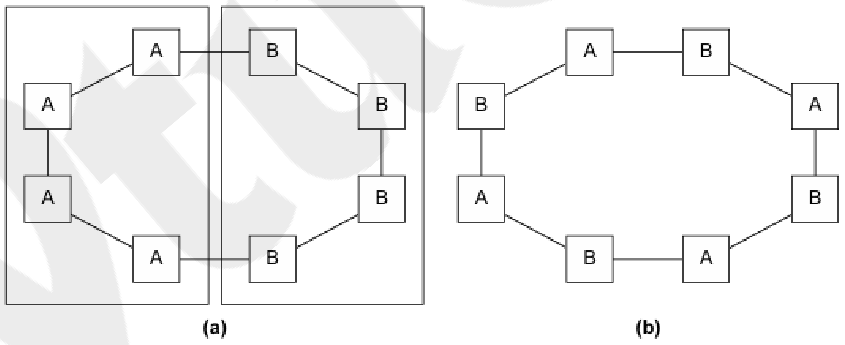


FIGURE 2.9

Two bisections of a ring: (a) only two communications can take place between the halves and (b) four simultaneous connections can take place

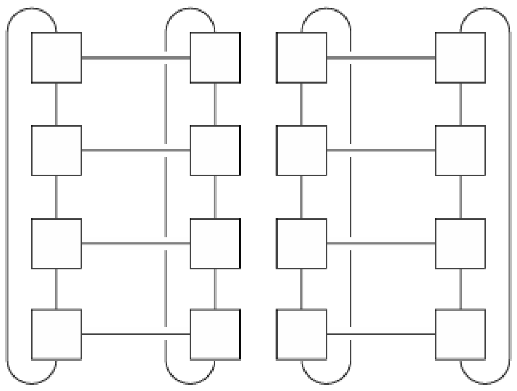


FIGURE 2.10

A bisection of a toroidal mesh

A key measure of communication capability is the **bisection width**, which refers to the **minimum number of links** that must be removed to divide the system into two equal halves. For a ring with 8 nodes, the **bisection width is 2**, while in a **square toroidal mesh** with $p = q^2$ (where q is even), the **bisection width is** $\mathbf{2\sqrt{p}}$.

Another important metric is the **bisection bandwidth**, which is the **sum of the bandwidths** of the links connecting two halves of the system. For example, if each link in a ring has a **bandwidth of 1 Gbps**, the total **bisection bandwidth** would be **2 Gbps**. This metric gives a good idea of the **data transfer capacity** across the network.

An ideal but impractical network is the **fully connected network**, where each switch is directly connected to every other switch. Its **bisection width** is $\mathbf{p^2 / 4}$, but it requires $\mathbf{p(p - 1)/2}$ **links**, making it unfeasible for large systems. It serves as a **theoretical best-case model**.

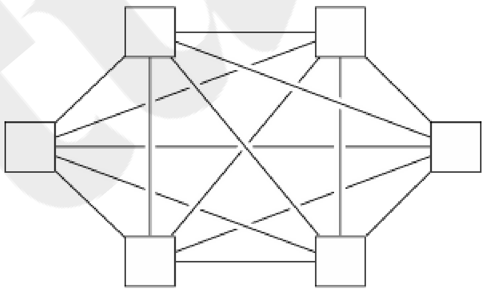


FIGURE 2.11

A fully connected network

The **hypercube** is another direct interconnect used in actual systems. It is built **inductively**—a 1D hypercube has 2 nodes, and each higher-dimensional hypercube is built by joining two lower-dimensional ones. A hypercube of dimension d has $\mathbf{p = 2^d}$ **nodes**, and each switch connects to **d other switches**. The **bisection width** is **p / 2**, offering higher connectivity than rings or meshes. However, the switches are more complex, requiring $\mathbf{\log_2(p)}$ connections, making them **more expensive** than mesh-based switches.

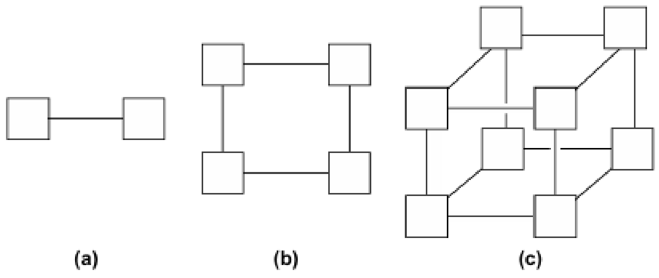


FIGURE 2.12

(a) One-, (b) two-, and (c) three-dimensional hypercubes

In contrast, **indirect interconnects** separate the switches from direct processor connections and route communications through a **switching network**. Two popular examples are the **crossbar** and the **omega network**. In a **crossbar**, all processors can communicate **simultaneously** with different destinations as long as there is no conflict. Its **bisection width is p**, and it offers **high flexibility** but is **very costly**, requiring **p^2 switches**.

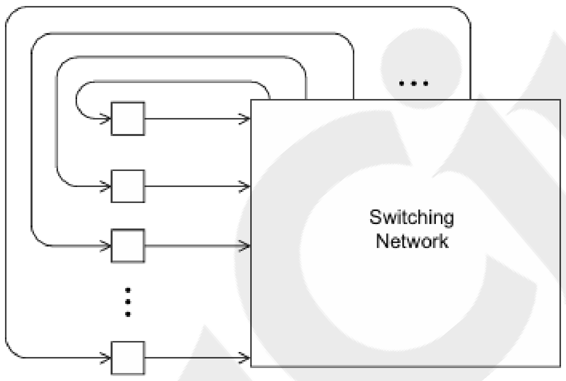


FIGURE 2.13

A generic indirect network

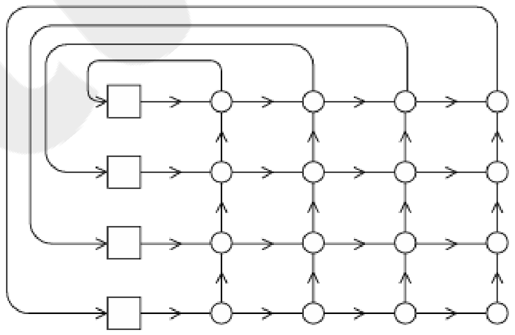


FIGURE 2.14

A crossbar interconnect for distributed-memory

The **omega network** is a more cost-efficient design using **2×2 crossbar switches** arranged in stages. It allows for some parallel communications but has **contention**—certain communications cannot occur at the same time. Its **bisection width is p / 2**, and it uses only **$p \times \log_2(p)$ switches**, making it significantly cheaper than a full crossbar.

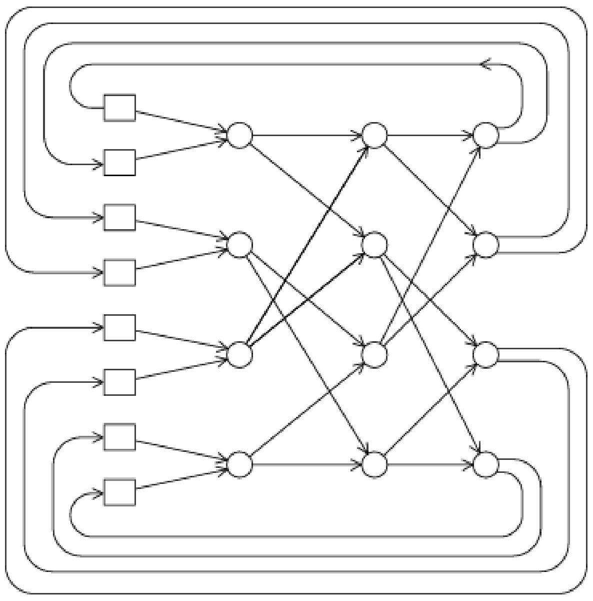


FIGURE 2.15

An omega network

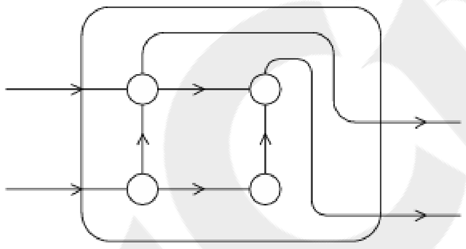


FIGURE 2.16

A switch in an omega network

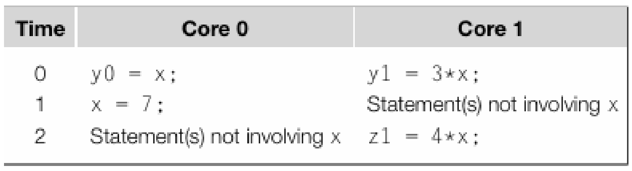
latency and bandwidth

When evaluating interconnect performance, two key terms are used: **latency** and **bandwidth.** **Latency** refers to the **time delay** from the start of transmission to the reception of the **first byte**, while **bandwidth** is the **rate of data transfer** once the transmission has started. The **total transmission time** for a message of size *n* bytes is given by: **Transmission Time = latency + (n / bandwidth).** It's important to note that in distributed-memory systems, latency may also include **message assembly and disassembly time**, such as adding headers, error correction data, and message size info.

1.3.5 Cache coherence

- In shared-memory systems, **CPU caches are hardware-managed**, and **programmers do not have direct control** over cache updates.

- This lack of control leads to important **issues in shared-memory systems**, especially when **multiple cores access and modify shared data**.



Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving x
2	Statement(s) not involving x	`z1 = 4*x;`

- A system with **two cores**, each having a **private data cache**.
- Variables:
    - x: Shared variable (initialized to 2)
    - y0: Private to Core 0
    - y1, z1: Private to Core 1
- When **both cores read shared data** (e.g., x), **no problem occurs**.
- Problem arises when **Core 0 writes to x** (e.g., x = 7) and then **Core 1 uses x** in a computation.

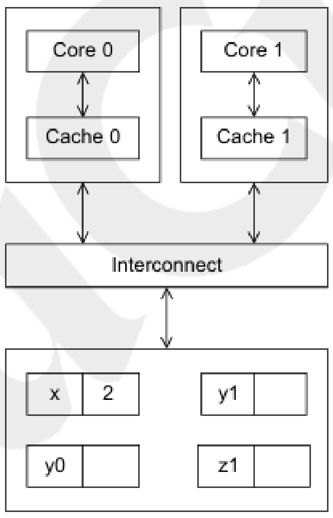


FIGURE 2.17

A shared-memory system with two cores and two caches

- Even though x = 7 is executed before using it in z1 = 4 * x + 8, **Core 1 may still use the old cached value of x = 2**.
- This leads to **incorrect value in z1**.
    - **Expected**: z1 = 4 * 7 + 8 = 36
    - **Actual (due to stale cache)**: z1 = 4 * 2 + 8 = 16

- At time 0, x resides in **Core 1's cache**.

- Even after Core 0 updates x, unless x is **evicted and reloaded** in Core 1's cache, Core 1 continues using the old value.
- This happens **regardless of write policy**:
    - **Write-through**: Updates go to **main memory**, but **not to other caches**.
    - **Write-back**: Update stays **only in Core 0's cache**, not visible to others.

Caches designed for single-processor systems do not ensure coherence when multiple processors cache the same variable.

- There's **no mechanism** to ensure that an update by one core is **seen by others**.
- This results in **unpredictable behavior** in shared-memory programs.

The **cache coherence problem** arises when **multiple caches hold the same variable**, but an update by one processor is **not reflected in others' caches**.

- This leads to **inconsistent views** of memory.
- Programs **cannot rely** on hardware caches to behave consistently across cores.

In shared-memory multiprocessor systems, each processor core has its own private cache. When multiple cores cache and access the same memory location, inconsistencies can arise due to updates not being reflected across all caches. This leads to the **cache coherence problem**.

To handle cache coherence, two main approaches are used: **Snooping cache coherence** and **Directory-based cache coherence**.

Snooping cache coherence is based on the principle used in bus-based systems. All cores share a common bus, and any communication on the bus can be observed by all cores. When Core 0 updates a shared variable x in its cache, it broadcasts the update on the bus. If Core 1 is observing the bus, it can detect the update and invalidate its own cached copy of the variable. The broadcast indicates that the cache line containing x has been updated, but not the value of x itself.

Snooping does not require the interconnect to be a bus, but it must support broadcast. It works with both write-through and write-back cache policies. In write-through, updates are immediately written to memory, which other cores can observe. In write-back, updates remain in the local cache until evicted, so additional messages are required to notify other cores.

Snooping cache coherence does not scale well to large systems because it requires broadcasting every time a variable is updated. As the number of cores increases, the communication overhead becomes a bottleneck.

Directory-based cache coherence is suitable for larger systems. These systems support a single address space, and a core can access a variable in another core's memory by direct reference. In this approach, a data structure called a **directory** is used to keep track of which cores have a copy of each cache line.

The directory is usually distributed, with each core or memory module maintaining the status of its local memory blocks. When a cache line is read by a core, the directory entry is updated to reflect that the core has a copy. When a write occurs, the directory is checked to identify all cores holding the copy, and only those are notified to invalidate or update their cache lines.

Directory-based coherence avoids global broadcasts and scales better in systems with many cores. However, it requires additional storage and maintenance for the directory structure.

False sharing

False sharing occurs when CPU caches operate on **cache lines** (not on individual variables), and **multiple cores update variables** that lie on the **same cache line**, even if they are **logically independent**.
This results in **unnecessary cache invalidations** and causes **performance degradation**.

Example – Sequential Code:

```
int i, j, m, n;
double y[m];

// Assign y[i] = 0.0
for (i = 0; i < m; i++)
for (j = 0; j < n; j++)
   y[i] += f(i, j);
```

Parallel Version (using multiple cores):

Assume core_count cores. Divide iterations among cores:

```
/ Private variables
int i, j, iter_count;
// Shared variables initialized by one core
int m, core_count;
double y[m];

iter_count = m / core_count;
Core 0 executes:
for (i = 0; i < iter_count; i++)
  for (j = 0; j < n; j++)
    y[i] += f(i, j);
Core 1 executes:
for (i = iter_count; i < 2 * iter_count; i++)
  for (j = 0; j < n; j++)
    y[i] += f(i, j);
```

Problem:

- Assume s = 2 (two cores), and double is 8 bytes.

- Cache line size = 64 bytes, and y[0] starts at beginning of a cache line.
- When two cores **simultaneously execute their sections**, they access different elements of y[], but these elements are **within the same cache line**.
- So **each core's cache gets invalidated**, and they must **fetch updated lines from memory**.

Effect:

- This creates **high memory traffic** and **poor performance** even though **no true data sharing** exists.
- Known as **false sharing**, this issue occurs **due to the way data is stored in memory**, not due to program logic.

1.1.6 Shared-Memory vs Distributed-Memory

Shared-Memory Systems:

- All processors **share a single memory space**.
- Threads or processes **communicate implicitly** by reading and writing to shared data structures.
- **Easier to program** due to implicit coordination.
- But difficult to **scale to large numbers of processors** due to hardware limitations:
  - **Bus-based interconnects** become congested as more processors are added.
  - **Crossbar switches**, while efficient, are **very expensive** and rare in large systems.

Distributed-Memory Systems:

- Each processor has its **own local memory**.
- Communication happens via **explicit message passing**.
- Requires more effort to program, but **hardware scales better**:
  - **Hypercube** and **toroidal mesh** interconnects are relatively **inexpensive**.
  - Can support **thousands of processors**.
- Well-suited for **problems involving large-scale data** or **high computation needs**.

Shared-Memory Systems	**Distributed-Memory Systems**
Processors **share a single memory space**.	Each processor has **its own local memory**.

Shared-Memory Systems	Distributed-Memory Systems
Communication is **implicit** using shared data structures.	Communication is **explicit** using **message passing**.
Easier to program and understand.	**More difficult to program**, requires managing communication.
Scalability is poor due to interconnect limits.	**Scales better**, suitable for large systems.
Bus-based interconnects become congested as processor count increases.	Uses **efficient and cheap interconnects** like **hypercube** and **toroidal mesh**.
Crossbars support more processors but are **very expensive**.	Supports **thousands of processors** cost-effectively.
Good for small to medium-scale parallel systems.	Suitable for **large-scale data or computational problems**.

1.2 Parallel Software

Parallel hardware is now widely available. Most desktops and servers today are built with **multicore processors**, allowing them to perform multiple tasks at once. However, **parallel software** has not advanced at the same pace. Apart from specialized systems such as **operating systems**, **database systems**, and **web servers**, most commonly used software still runs in a **sequential manner** and does not take full advantage of parallel hardware.

This mismatch is a concern. In the past, application performance improved steadily with better hardware and smarter compilers. Today, **performance gains depend on how well software uses parallelism**. To keep improving the speed and power of applications, **developers must learn to write software for shared- and distributed-memory architectures**.

Before diving into programming techniques, we must understand some basic terms. In **shared-memory programming**, a **single process creates multiple threads**. These threads work together and share memory, so we say **threads carry out tasks**. In **distributed-memory programming**, **multiple processes** are created, each with its **own memory**. These processes work independently and communicate by **sending messages**. In this book, whenever the topic applies to both models, we use the combined term **"processes/threads"**.

1.2.1 Caveats:

This section only discusses **software for MIMD (Multiple Instruction, Multiple Data) systems**. It does **not cover GPU programming**, because **GPUs use different programming interfaces (APIs)**. Also, the coverage in this book is **not exhaustive**; the goal is to provide a **basic idea of the issues**, not complete technical depth.

A major focus will be on the **SPMD (Single Program, Multiple Data)** model. In SPMD, all threads or processes **run the same program**, but they may perform different tasks depending on their **thread or process ID**. This is usually done using **conditional statements**, like:

```
if (I'm thread/process 0)
   do this;
else
   do that;
```

This style makes it easy to implement **data parallelism**, where each process works on a different part of the data. For example:

```
if (I'm thread/process 0)
   operate on the first half of the array;
else
   operate on the second half of the array;
```

Also, **SPMD programs can support task parallelism** by dividing different tasks among processes or threads. So, even though all threads run the same program, they may be doing **different work on different data** or **different work altogether**, depending on how the code is structured.

1.2.2 Coordinating the Processes/Threads

In some simple cases, achieving **parallel performance** is easy. For example, adding two arrays:

```
double x[n], y[n];
...
for (int i = 0; i < n; i++)
   x[i] += y[i];
```

To **parallelize** this, we divide the array elements among multiple **processes/threads**. If there are **p threads**, thread 0 handles elements 0 to n/p−1, thread 1 handles n/p to 2n/p−1, and so on.

The programmer must:

1. **Divide the work** so that:
   a. Each thread gets roughly **equal work** (called **load balancing**)
   b. **Communication is minimized**

In many programs, the amount of work is not known in advance. In such cases, dividing work equally and minimizing communication becomes more challenging.

The process of converting a serial program to a parallel one is called **parallelization**. If this is done by simply dividing the work, the program is called **embarrassingly parallel**. Despite the name, such solutions are efficient and useful.

However, most problems need more than just dividing work. We must also:

2. **Synchronize threads/processes**
3. **Enable communication** between them

In **distributed-memory systems**, communication often also ensures **synchronization**. In **shared-memory systems**, threads may **synchronize to communicate**.

1.2.3 Shared-Memory

In **shared-memory programs**, variables can be either **shared** or **private**.

- **Shared variables**: accessible by **all threads** (used for communication).
- **Private variables**: accessible by **only one thread**.

Communication between threads is usually **implicit** through shared variables—there's no need for explicit messages.

Dynamic and Static Threads

There are two main thread management approaches in shared-memory systems:

1. Dynamic Threads

- Often involves a **master thread** and multiple **worker threads**.
- The **master waits** for work (e.g., a network request).
- When work arrives, it **creates (forks)** a worker thread to handle it.
- Once the task is complete, the worker thread **terminates and joins** the master.
- This method uses **system resources efficiently**, as resources are used **only when the thread is active**.

2. Static Threads

- All threads are **created at the beginning** after setup.
- Threads continue to run **until all work is done**.
- Afterward, they **join the master thread**, which may do **cleanup** and then terminate.
- Less efficient in resource usage—**idle threads still hold resources** like memory and registers.
- However, **creating and terminating threads repeatedly** (as in the dynamic model) can be costly.
- If resources are available, the static model can offer **better performance**.

Nondeterminism

In any **MIMD system** where processors execute **asynchronously**, **nondeterminism** is likely. A computation is **nondeterministic** if the same input gives **different outputs**. This happens when **threads execute independently**, and their execution speeds vary from run to run.

Example: Two threads, one with rank 0 and another with rank 1, store:

- my_x = 7 (thread 0)
- my_x = 19 (thread 1)

Both execute:

printf("Thread %d > my_val = %d\n", my_rank, my_x);

Output can be:

mathematica

Thread 0 > my_val = 7

Thread 1 > my_val = 19

or:

mathematica

Thread 1 > my_val = 19

Thread 0 > my_val = 7

or even **interleaved output**. This is fine here since output is labeled, but **nondeterminism in shared-memory** programs can lead to **serious errors**.

Suppose both threads compute my_val and want to update a **shared variable x**, initially 0:

my_val = Compute_val(my_rank);

x += my_val;

The operation x += my_val is **not atomic**; it involves:

- **Load** x from memory
- **Add** my_val
- **Store** back to x

If two threads do this simultaneously, one update may **overwrite** the other. This is a **race condition**—result depends on which thread **finishes first**.

To prevent this, we use a **critical section**, where only one thread can execute at a time. This is ensured using a **mutex (mutual exclusion lock)**:

```
my_val = Compute_val(my_rank);
Lock(&addmyval_lock);
x += my_val;
Unlock(&addmyval_lock);
```

A **mutex** ensures **mutual exclusion**. When one thread locks the section, others wait. This avoids incorrect updates but **serializes** the critical section. Hence, critical sections should be **short and minimal**.

Alternatives to mutexes:

- **Busy-waiting**: a thread waits in a loop:

```
if (my_rank == 1)
  while (!ok_for_1); // Busy wait
x += my_val;
if (my_rank == 0)
  ok_for_1 = true;   // Allow thread 1
```

Simple to use but **wastes CPU time**.

- **Semaphores**: similar to mutexes, offer more flexibility in synchronization.
- **Monitors**: high-level objects; only one thread can call a method at a time.
- **Transactional memory**: treats critical sections like database transactions. If a thread can't complete, changes are **rolled back**.

Thread safety

In most cases, **serial functions** can be used in **parallel programs** without issues. But some functions, especially those using **static local variables**, can cause problems.

In C, **local variables** inside functions are stored on the **stack**, and since each thread has its own stack, these are **private** to each thread. But **static variables** persist across function calls and are **shared among threads**, which can lead to **unexpected behavior**.

Example: The C function strtok splits a string into **substrings** using a **static char* variable** to remember the position. If **two threads** call strtok on different strings at the same time, the shared static variable can be **overwritten**, causing **loss of data** or **wrong outputs**.

Such a function is **not thread-safe**. In **multithreaded programs**, using it can lead to **errors or unpredictable results**.

A function is **not thread-safe** if **multiple threads access shared data** without proper synchronization. So, although many serial functions are safe to use in parallel programs, **programmers must be careful** when using functions originally designed for **serial execution**.

1.2.4 Distributed-memory

In **distributed-memory systems**, each **core** or **processor** can directly access only its **own private memory**. Unlike shared-memory systems, cores do not have access to a single shared memory space. To facilitate communication between these isolated memories, various **APIs** are used, with the most widely used being **message-passing**. This approach enables data exchange between processes running on different nodes or cores.

Interestingly, distributed-memory APIs can even be implemented on **shared-memory hardware** by logically partitioning shared memory into private address spaces and handling communication via software tools or compilers.

Unlike shared-memory systems that use multiple threads, distributed-memory programs are generally implemented using **multiple processes.** These processes often run on **separate CPUs** under **independent operating systems**, and launching a single process that spawns threads across distributed systems is generally not feasible.

Message Passing

The **message-passing** approach revolves around the use of two core functions: Send() and Receive(). Each process is assigned a **rank** ranging from 0 to p - 1 (where p is the total number of processes). Communication between processes typically follows this model:

```
char message[100];
my_rank = Get_rank();

if (my_rank == 1) {
   sprintf(message, "Greetings from process 1");
   Send(message, MSG_CHAR, 100, 0);
}
else if (my_rank == 0) {
   Receive(message, MSG_CHAR, 100, 1);
   printf("Process 0 > Received: %s\n", message);
}
```

The function Get_rank() returns the process's unique rank. The behavior of Send() and Receive() can vary. In a **blocking send**, the sender waits until the matching Receive() is ready. In **non-blocking send**, data is copied into a buffer, and the sender resumes execution immediately.

Each process runs the **same program**, but performs different actions based on its **rank**—a model known as **SPMD (Single Program, Multiple Data)**. Variables like message[] exist separately in each process's private memory space.

Message-passing libraries often include **collective communication** functions, such as **broadcast**, where one process sends data to all others, and **reduction**, where individual results are combined (e.g., summing values).

The most widely used message-passing API is **MPI (Message Passing Interface)**, which we will study in more detail in the next module.

One-Sided Communication (Remote Memory Access)

Traditional message-passing requires both a sender and a receiver to explicitly participate. However, in **one-sided communication**, only one process performs an action—either **reading from** or **writing to** another process's memory.

This technique can reduce communication overhead and simplify synchronization. For instance, process P0 can copy data into P1's memory without requiring an explicit receive call by P1.

However, this convenience brings challenges. The writing process must know **when it is safe** to write, and the receiving process must know **when the memory has been updated**. These are usually handled by either:

- synchronizing processes before and after the operation, or
- using a **flag variable** that is polled by the receiving process.

Although one-sided communication can improve performance, improper use may introduce difficult-to-trace errors.

Partitioned Global Address Space (PGAS) Languages

While **shared-memory programming** is often simpler and more intuitive than message-passing, it is not directly suitable for distributed-memory hardware. PGAS languages aim to **bridge the gap** by providing a **shared-memory abstraction** for distributed-memory systems.

Simply treating all distributed memory as globally accessible may lead to **severe performance issues**, especially when memory access crosses node boundaries. To address this, PGAS languages:

- Allow variables to be marked as **private** (stored in local memory),
- Enable controlled distribution of **shared variables** across processes.

```
shared int n = ...;

shared double x[n], y[n];

private int i, my_first_element, my_last_element;


// Parallel vector addition

for (i = my_first_element; i <= my_last_element; i++)

   x[i] += y[i];
```

In the above example, the code assumes that each process works on a segment of the arrays x and y. If this memory is **properly allocated**—meaning local to the process accessing it—the performance is good. But if all of x is located in one process and all of y in another, performance deteriorates because of **remote memory access delays**.

PGAS languages provide a **compromise between shared- and distributed-memory programming**, offering high performance with simpler syntax, while allowing programmers control over data locality.