

FACULTY NAME: DR. PADMAJA K
SUBJECT TITLE AND CODE: PARALLEL COMPUTING – BCS702
SEMESTER AND SCHEME: VII SEMESTER

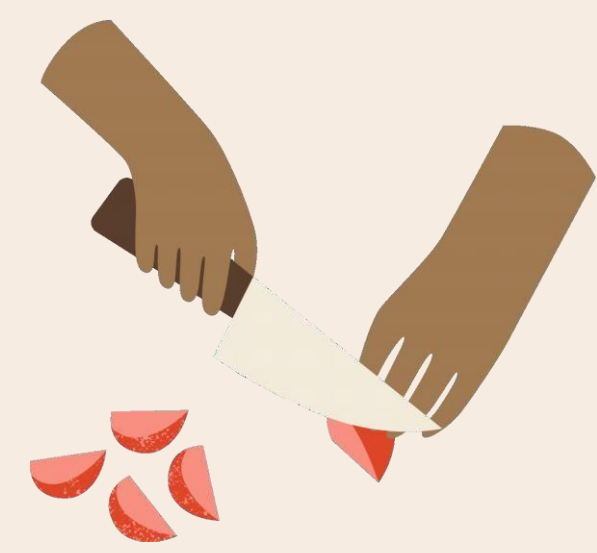
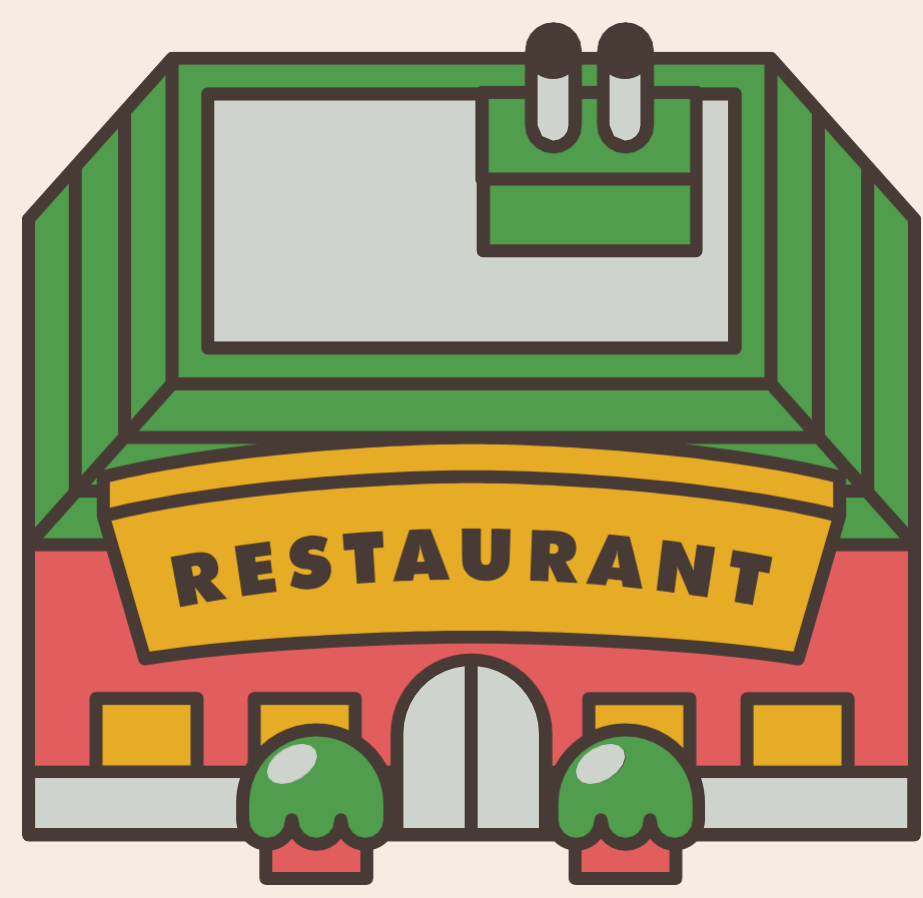
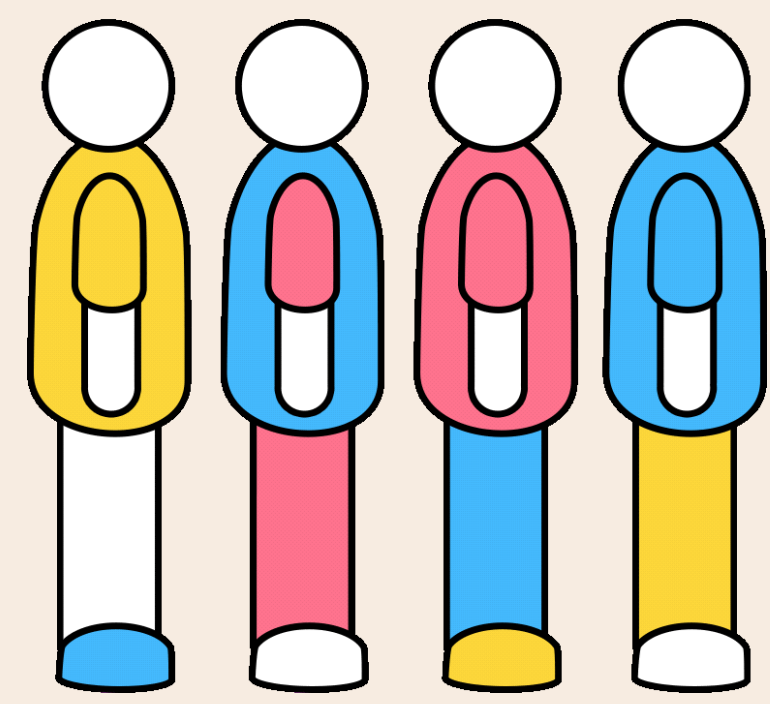
(ACADEMIC YEAR 2026-27)

PARALLEL COMPUTING		Semester	VII
Course Code	BCS702	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	Total Marks	100
Credits	04	Exam Hours	03
Examination nature (SEE)	Theory/Practical		
Course outcomes (Course Skill Set): At the end of the course, the student will be able to: ● Explain the need for parallel programming ● Demonstrate parallelism in MIMD system. ● Apply MPI library to parallelize the code to solve the given problem. ● Apply OpenMP pragma and directives to parallelize the code to solve the given problem ● Design a CUDA program for the given problem.			

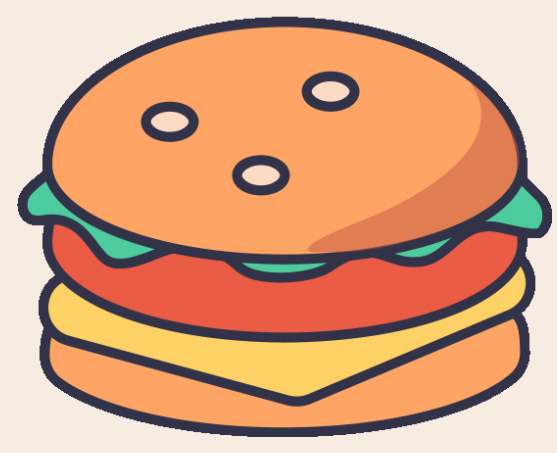


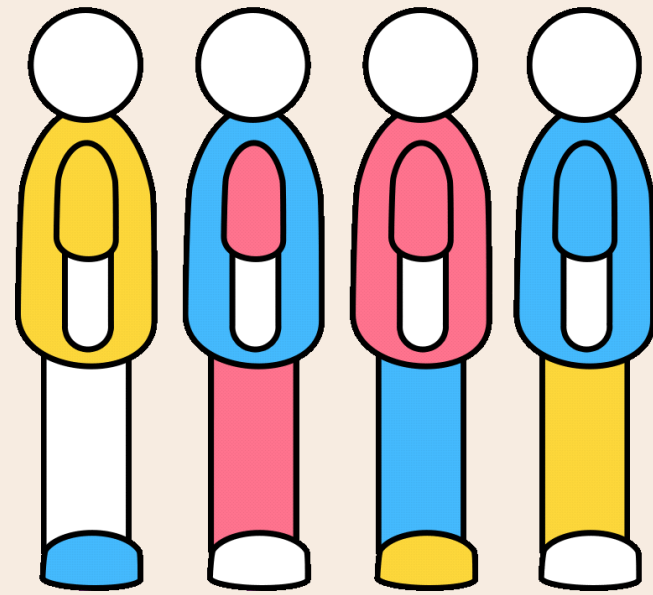
MODULE-1

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.

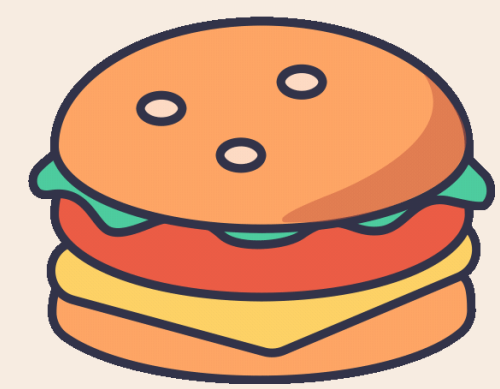
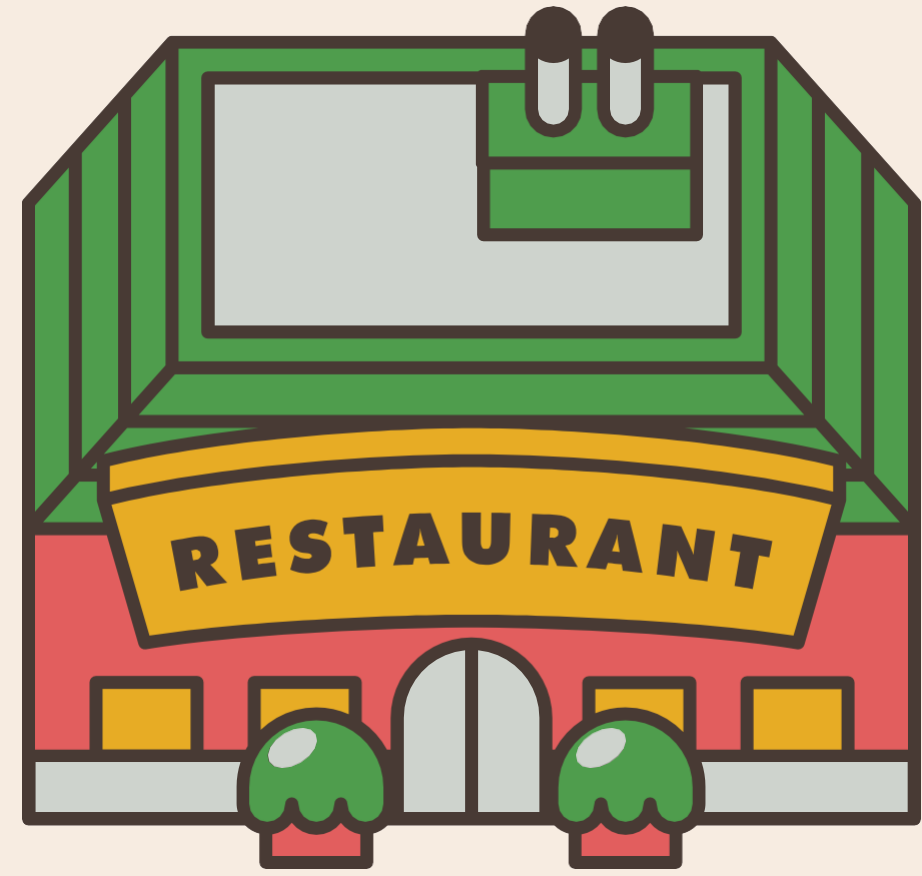


Pizza
Burger
Pasta
Cut
fruits





Pizza
Burger
Pasta
Cut
fruits



“Have you ever noticed your phone running multiple apps at once?”

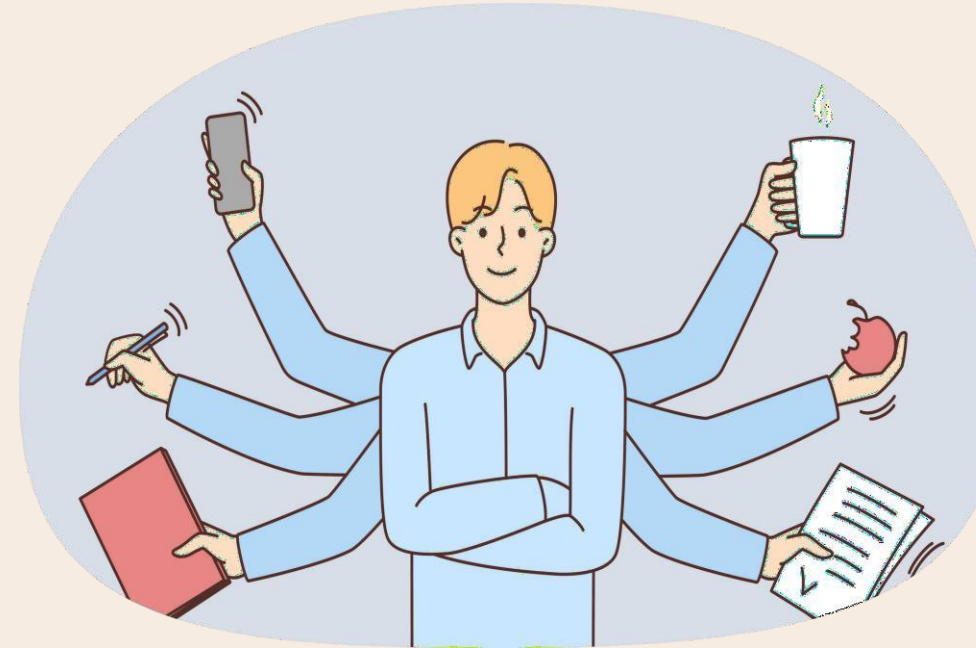
Video streaming (Netflix downloading + playing simultaneously)

Gaming consoles with multiple players running on separate

AGENDA



- 1 Introduction to Parallel Computing
- 2 Why It's Needed
- 3 History and Motivation
- 4 Real-World Applications
- 5 Basic Concepts



“Parallel Computing means performing many tasks simultaneously”

INCREASING DATA

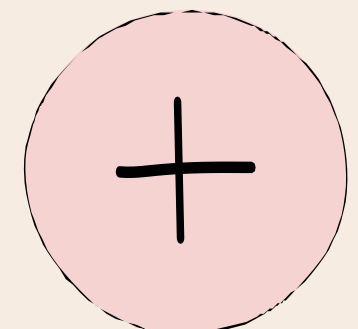
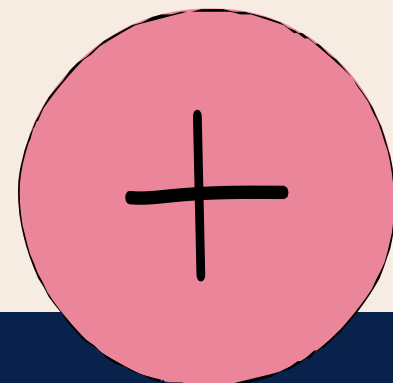
**CPU CLOCK SPEED
REACHED LIMITS**

HISTORY



REAL WORLD EXAMPLES

- Weather forecasting
- Space Exploration
- Online Platforms
- AI training chatgpt using GPUs
- autonomous vehicles



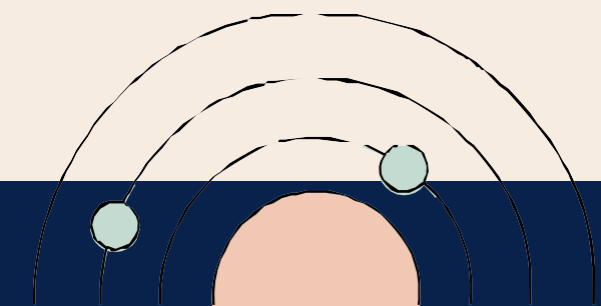


- how multiple apps can run simultaneously on a computer or phone and what happens internally

MULTITASKING

CPU CORE

OS SCHEDULER



HOW TASKS ARE ALLOCATED TO THE CORES OF THE CPU?

A task can be: A Process
A thread



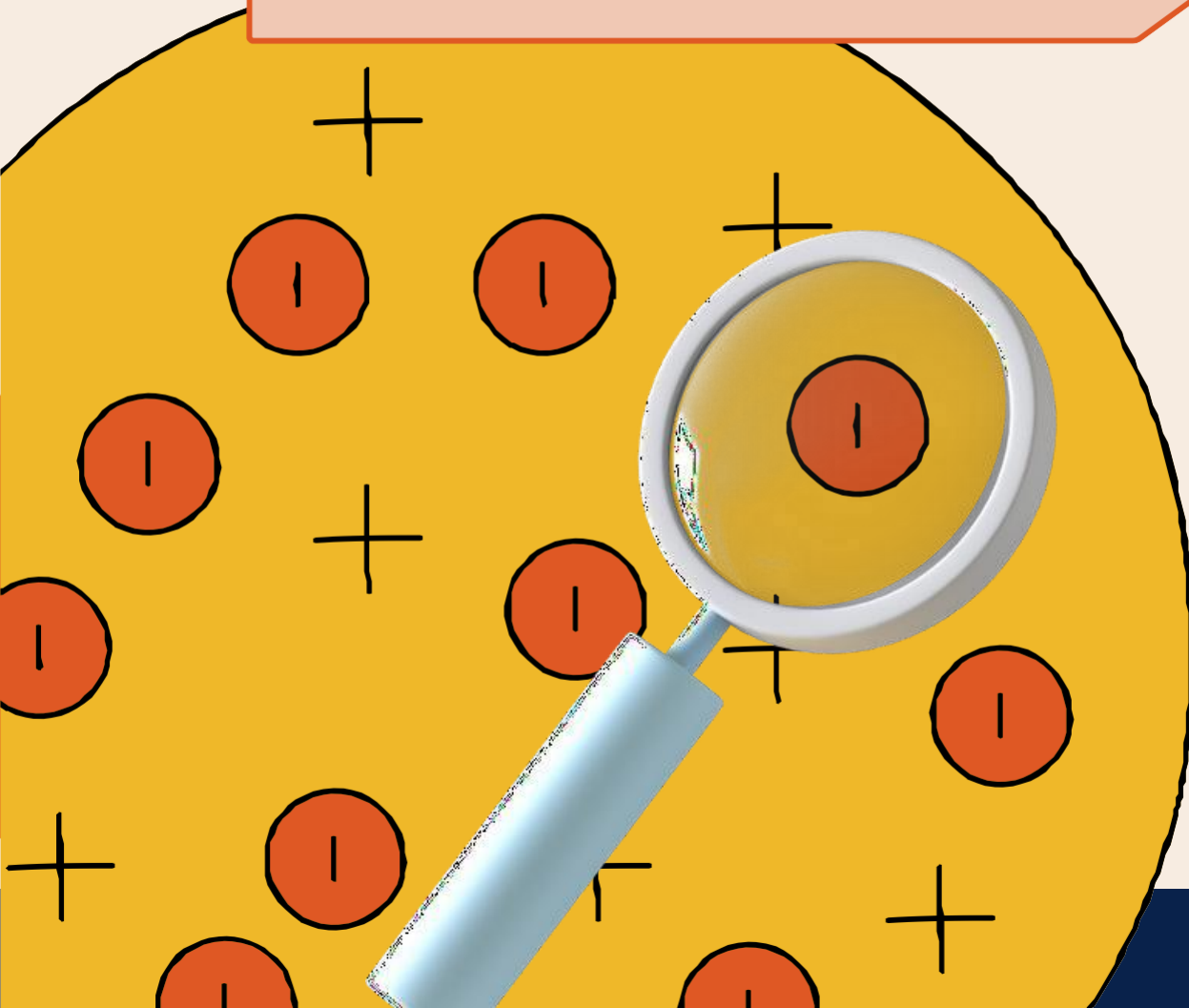
- 1 **TASK CREATED:**
- 2 **READY QUEUE:**
- 3 **CORE AVAILABILITY:**
- 4 **CONTEXT SWITCHING:**
- 5 **EXECUTION**
- 6 **MIGRATION**

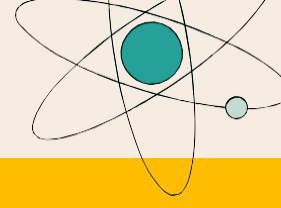
BASIC CONCEPTS

Data parallelism

TYPES

Task parallelism





PARALLEL ARCHITECTURES

- Shared Memory
- Distributed Memory

Why We Need to Write Parallel Programs

1. The problem with old (serial) programs

- Most existing programs were written for **single-core systems**.
- On a multicore system, you can run **multiple instances** of the same program (e.g., run 4 games at once), but that's **not useful** — users want one program to run **faster and better**, not more copies.
- Therefore: To use multiple cores effectively, programs must be **parallelized**.

2. Automatic conversion isn't enough

- Researchers have tried to create compilers that **translate serial code to parallel code**.
- Success has been **limited** because:
 - Translating each step independently into parallel code often leads to **inefficiency**.
 - Sometimes the **best parallel solution requires a completely new algorithm**, not just a step-by-step parallelization of the serial one.
 - Example: Matrix multiplication — turning it into parallel dot-products may be inefficient compared to designing a new parallel matrix multiplication algorithm.

Serial code(one core): Summation

```
sum = 0;
```

```
for (i = 0; i < n; i++) {
```

```
    x = ComputeNextValue(...);
```

```
    sum += x;
```

```
}
```

The second method: Pairwise (tree-style) reduction

- Instead of all cores sending to the master, we **combine results in stages**:
 - **Stage 1**: Pair the cores:
 - Core 0 + Core 1, Core 2 + Core 3, Core 4 + Core 5, Core 6 + Core 7.
 - **Stage 2**: Pair the winners (even-numbered cores now hold results):
 - Core 0 + Core 2, Core 4 + Core 6.
 - **Stage 3**: Final combination:
 - Core 0 + Core 4.

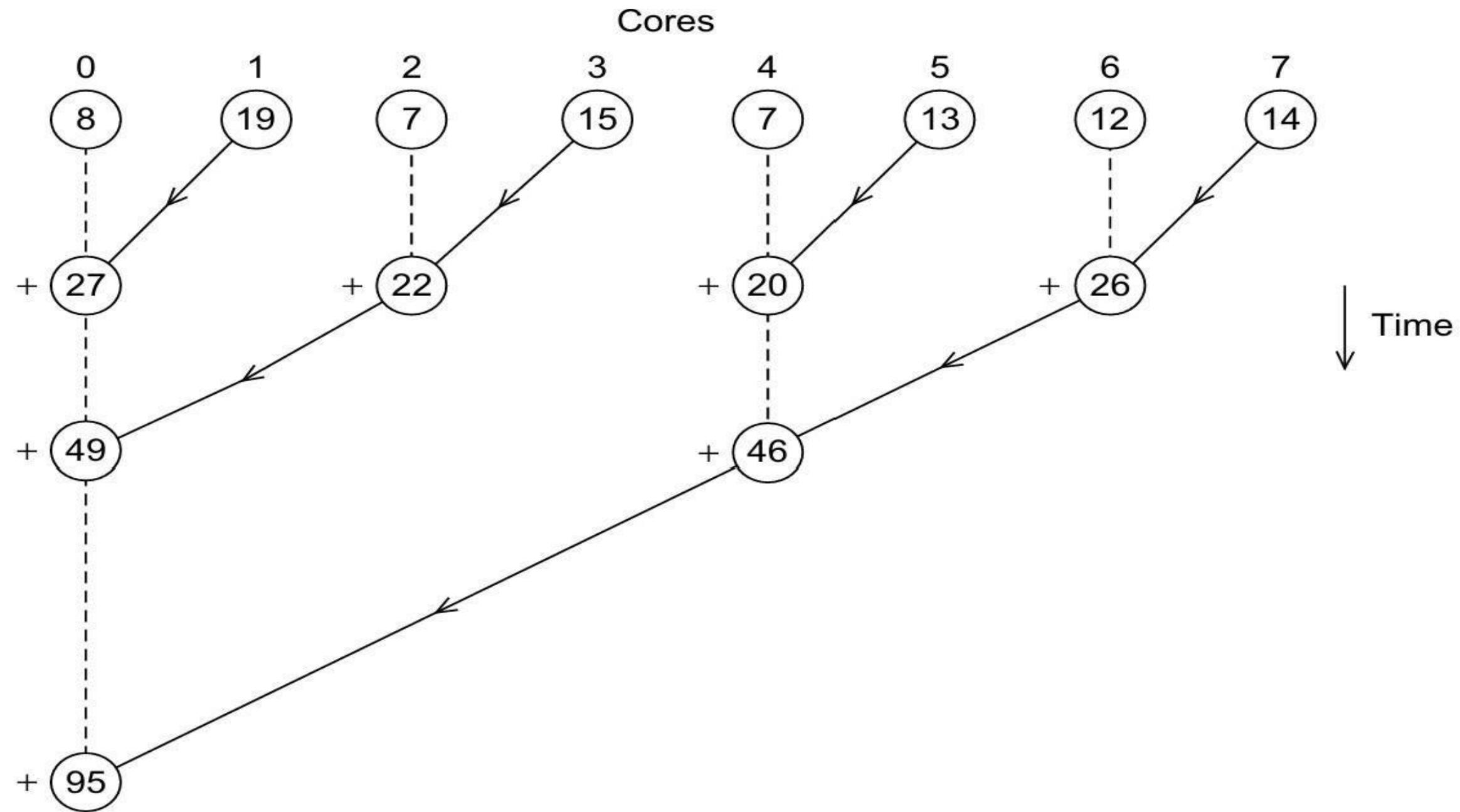


FIGURE 1.1

Multiple cores forming a global sum

Comparing the two global sum methods

- Method 1 (naïve / centralized):
 - Master adds up results from all cores.
 - Needs $p - 1$ operations (e.g., 999 adds for 1000 cores).
- Method 2 (tree reduction):
 - Results combined in pairs over stages.
 - Needs $\log_2(p)$ operations (e.g., only 10 adds for 1000 cores).
 - Much more efficient, especially as p grows.

how we actually write parallel programs and the main challenges

Two Main Approaches to Parallelism

1. Task Parallelism

- Different cores do *different tasks*.
- Example: In grading exams, one person grades only Question 1 (Shakespeare), another grades Question 2 (Milton), and so on.
- Each is doing a *different job*, so the instructions differ.

2. Data Parallelism

Different cores do the *same task* on *different pieces of data*.

Example: Split the 100 exam papers into 5 piles of 20. Each TA grades *all questions* on their pile. Same instructions, but applied to different data.

Von Neumann Architecture (Classical Computer Design)

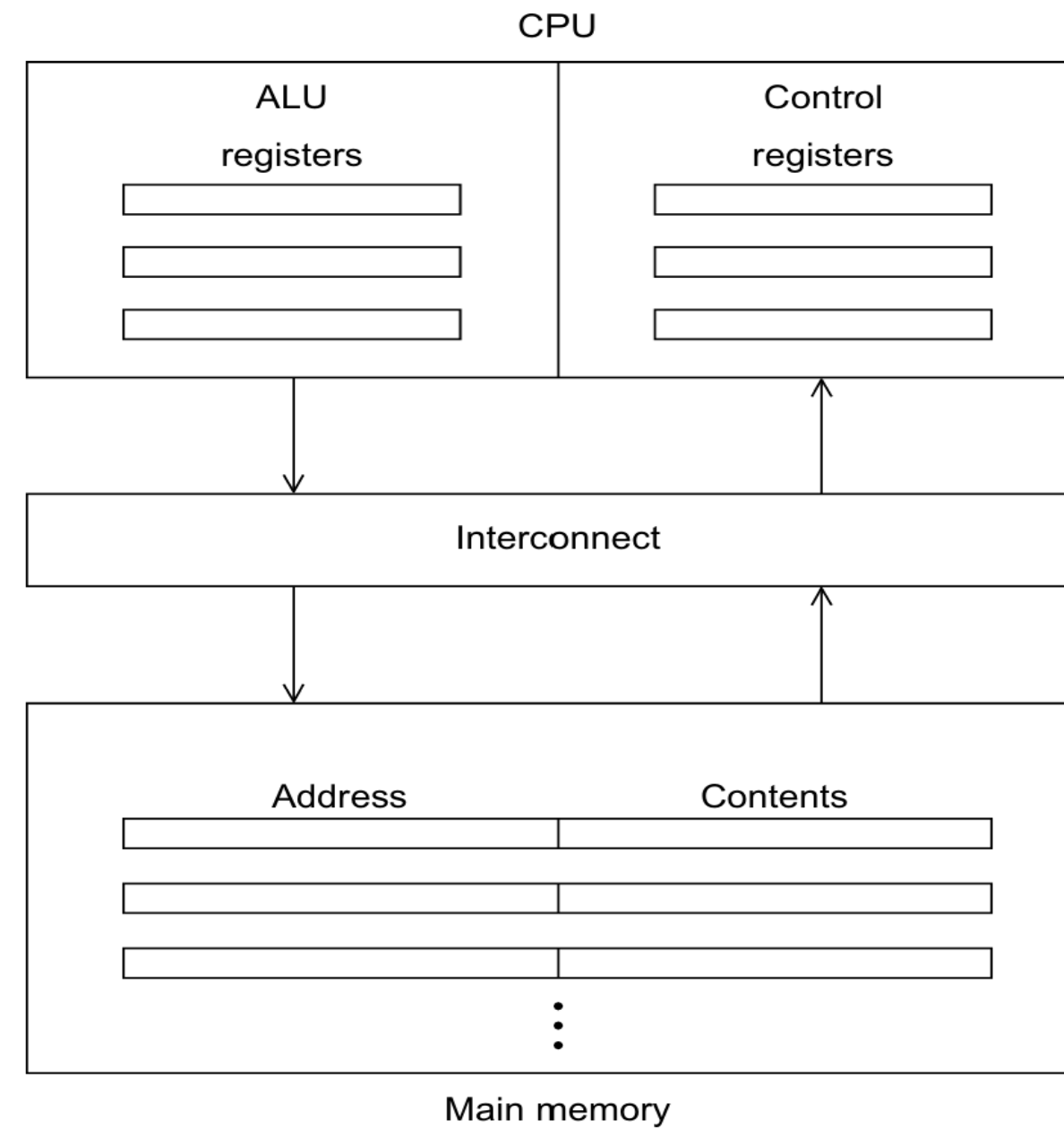


FIGURE 2.1

The von Neumann architecture

Von Neumann Bottleneck

- Problem: The CPU is much faster than the memory access speed.
- CPU may execute 100+ instructions in the time it takes to fetch one piece of data from memory.
- The bus/interconnect limits how quickly data & instructions travel.

Analogy:

- CPU is like factory making products.
- Memory is warehouse storing raw materials (data) and finished products (results).
- Road (bus) is the transport system between them.
- If the road is too narrow (limited bandwidth), the factory workers sit idle because raw materials arrive too slowly.

Modifications to the von Neumann model

The **von Neumann bottleneck** means the CPU is very fast, but memory (RAM) is much slower. Since the CPU often has to wait for memory, overall performance suffers.

To fix this, computer engineers added **caching, virtual memory, and parallelism**. This part is about **caching**.

What is caching?

Think of it like this:

- CPU = **factory**
- Main memory = **warehouse**
- Road between them = **slow, two-lane road**

The CPU constantly needs raw materials (data & instructions) from memory. If every time it has to go to the warehouse far away, it wastes time.

Solution is to Build a small storeroom (cache) right next to the CPU.
Cache stores a small amount of data that the CPU is very likely to need soon. It's much faster to access than main memory.

Cache mappings

1. Fully Associative
2. Direct Mapped
3. N-way Set Associative

Locality:

- **Spatial locality:** if you use $A[0][0]$, you'll probably use $A[0][1]$, $A[0][2]$ soon.
- **Temporal locality:** if you use a value once, you might use it again later.

```
for (i = 0; i < MAX; i++)  
    for (j = 0; j < MAX; j++)  
         $y[i] += A[i][j] * x[j];$ 
```

- Access pattern: row by row $\rightarrow$ contiguous memory.
- Example (MAX=4):
 - Access order: $A[0][0]$, $A[0][1]$, $A[0][2]$, $A[0][3]$ (all in **same cache line** $\rightarrow$ only **1 miss**).
 - Next row: $A[1][0] \dots A[1][3]$ $\rightarrow$ again, just **1 miss**.

virtual memory

Virtual memory gives:

Illusion of large memory (even bigger than RAM, because disk is used as backup).

Protection (one program cannot overwrite another's memory).

Flexibility (any program can use any free RAM block).

Memory is divided into **pages** (usually 4 KB–16 KB).

Disk also has **swap space** divided into same-sized **pages**.

A program uses **virtual addresses** → these get mapped to **physical addresses** in RAM.

Table 2.3 Pipelined Addition. Numbers in the Table Are Subscripts of Operands/Results

Time	Fetch	Compare	Shift	Add	Normalize	Round	Store
0	0						
1	1	0					
2	2	1	0				
3	3	2	1	0			
4	4	3	2	1	0		
5	5	4	3	2	1	0	
6	6	5	4	3	2	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
999	999	998	997	996	995	994	993
1000		999	998	997	996	995	994
1001			999	998	997	996	995
1002				999	998	997	996
1003					999	998	997
1004						999	998
1005							999

Distributed-memory interconnects

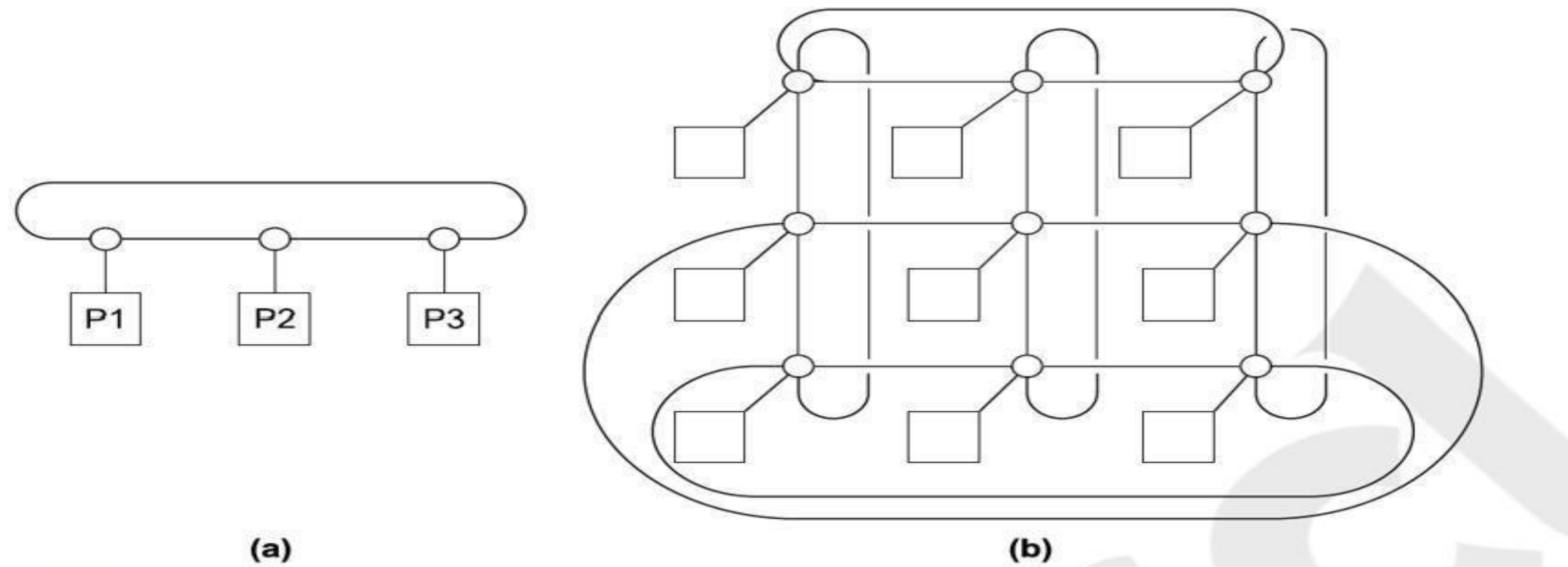


FIGURE 2.8

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

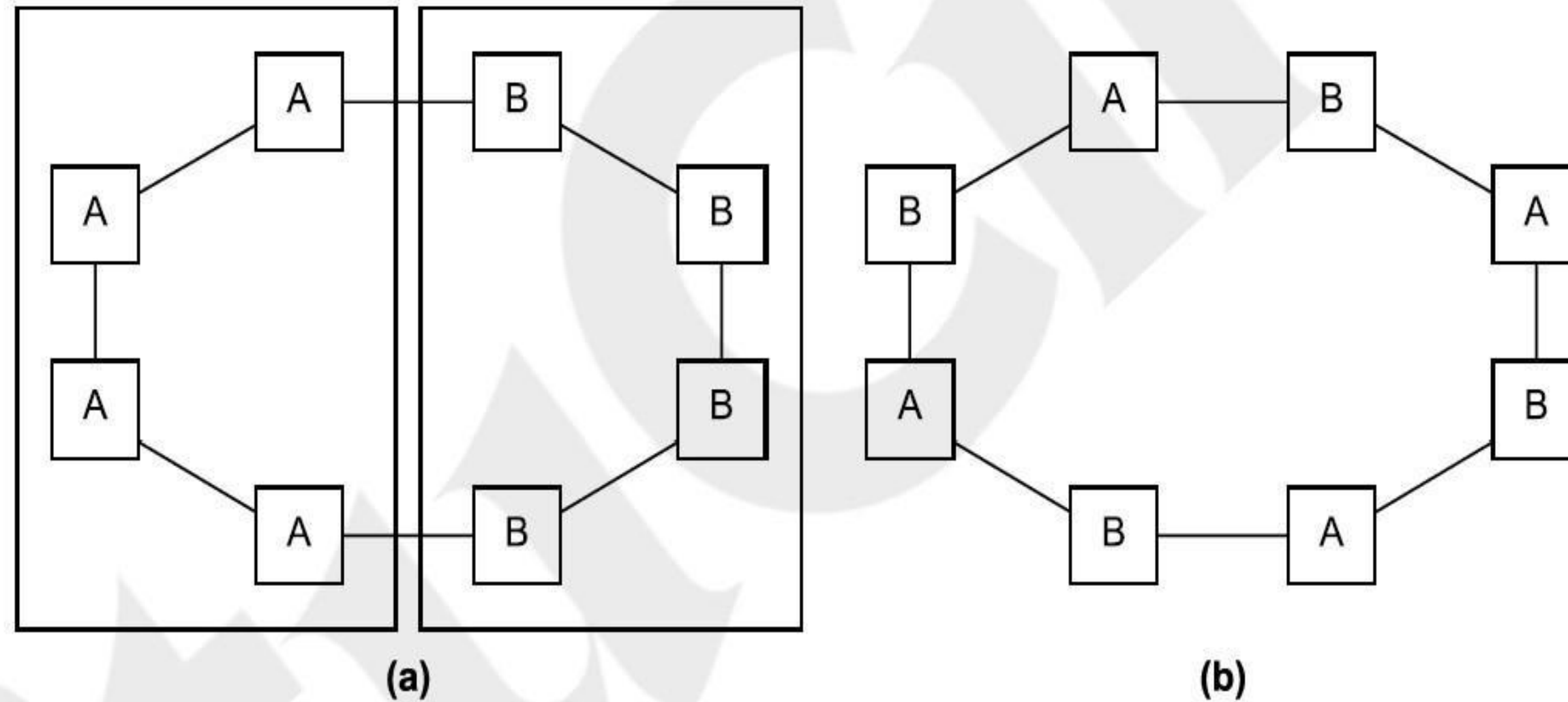


FIGURE 2.9

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

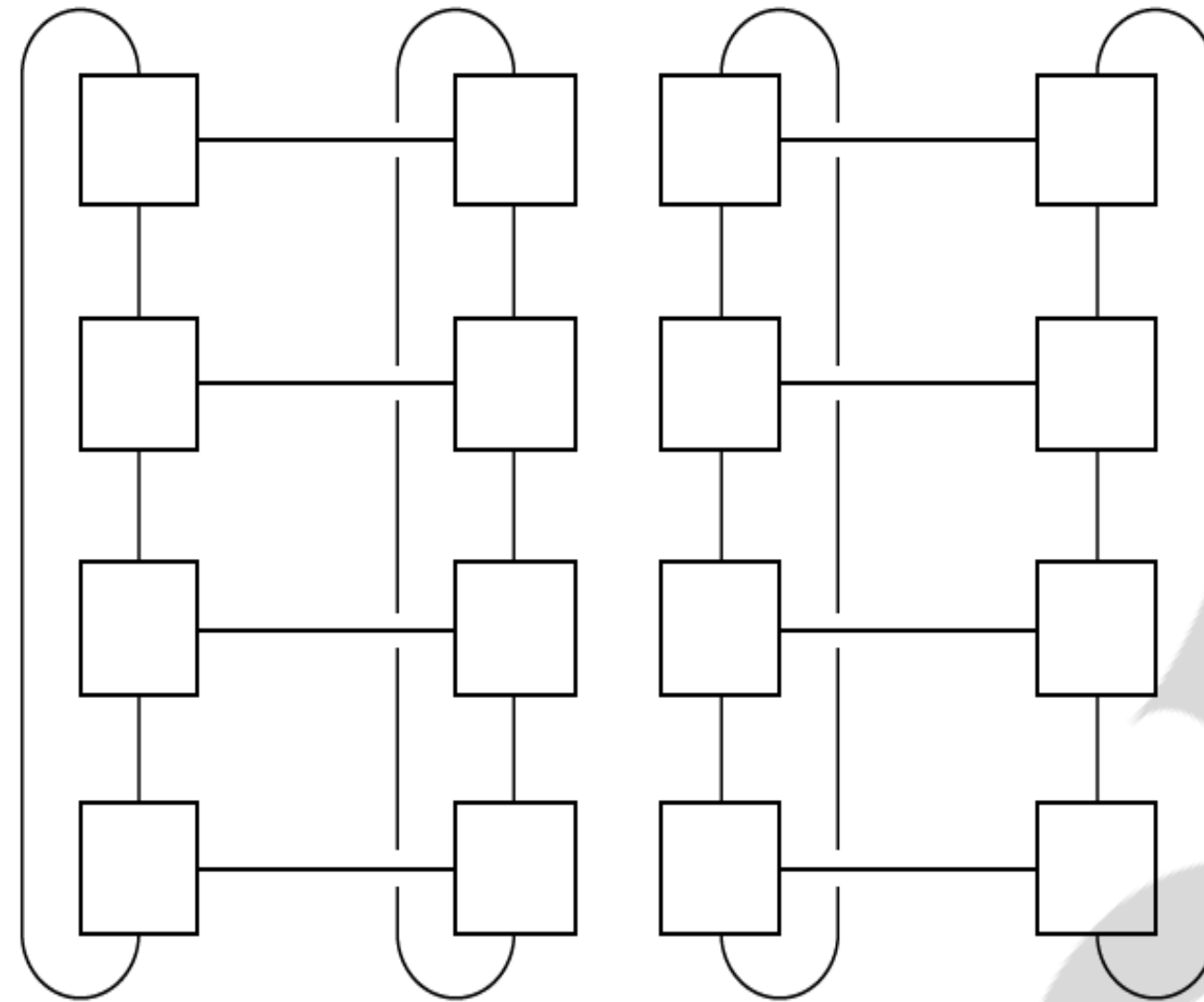


FIGURE 2.10

A bisection of a toroidal mesh.

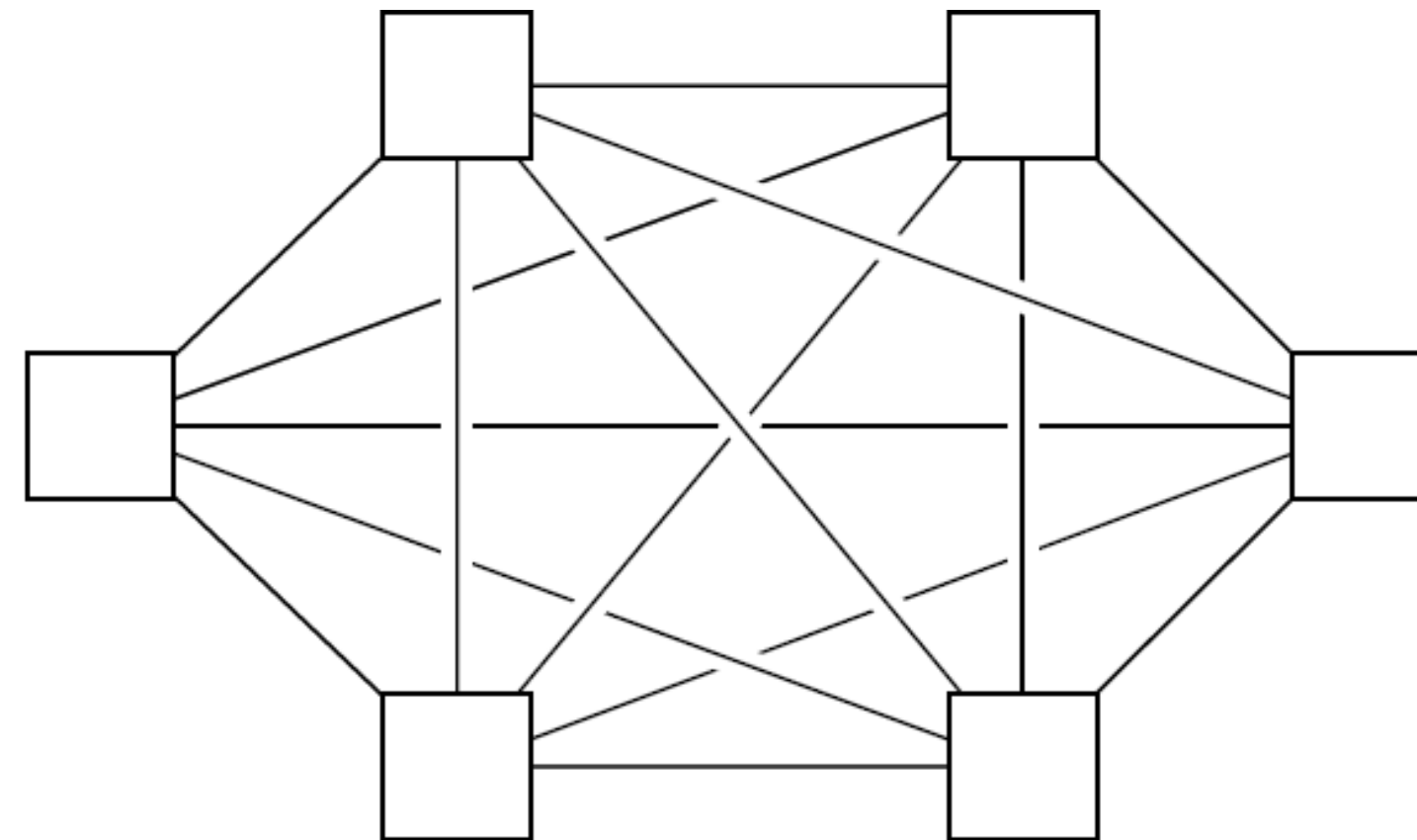


FIGURE 2.11

A fully connected network.

Cache coherence: CPU caches are managed by system hardware: programmers don't have direct control over them. This has several important consequences for shared-memory systems.

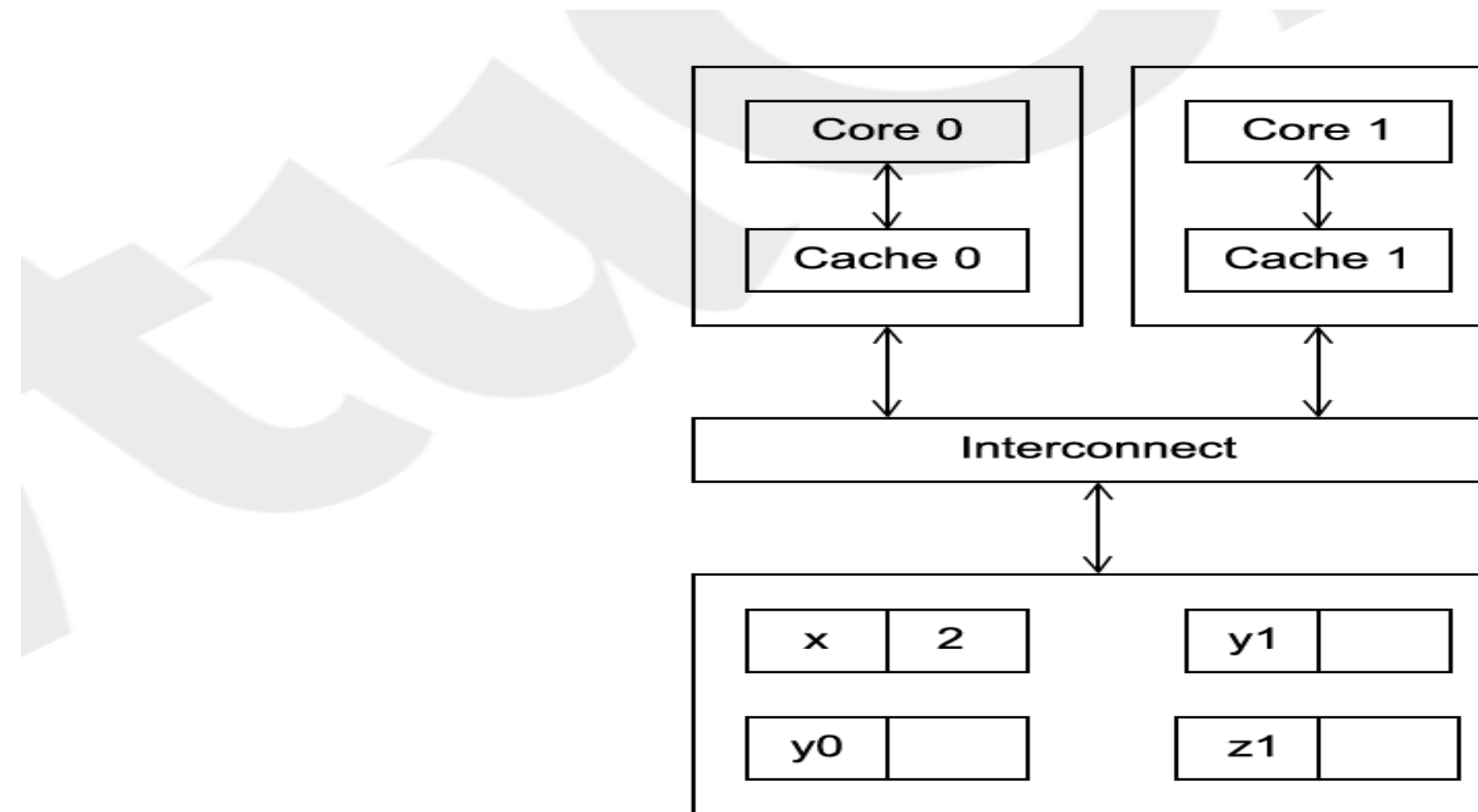


FIGURE 2.17

A shared-memory system with two cores and two caches.

Nondeterminism?

- In **MIMD systems** (Multiple Instruction, Multiple Data) where multiple processors (or threads) run at the same time, they usually don't stay perfectly in sync.
- This means **the same input** might produce **different outputs** depending on how the processors finish their tasks. This unpredictability is called **nondeterminism**.

Thank You

