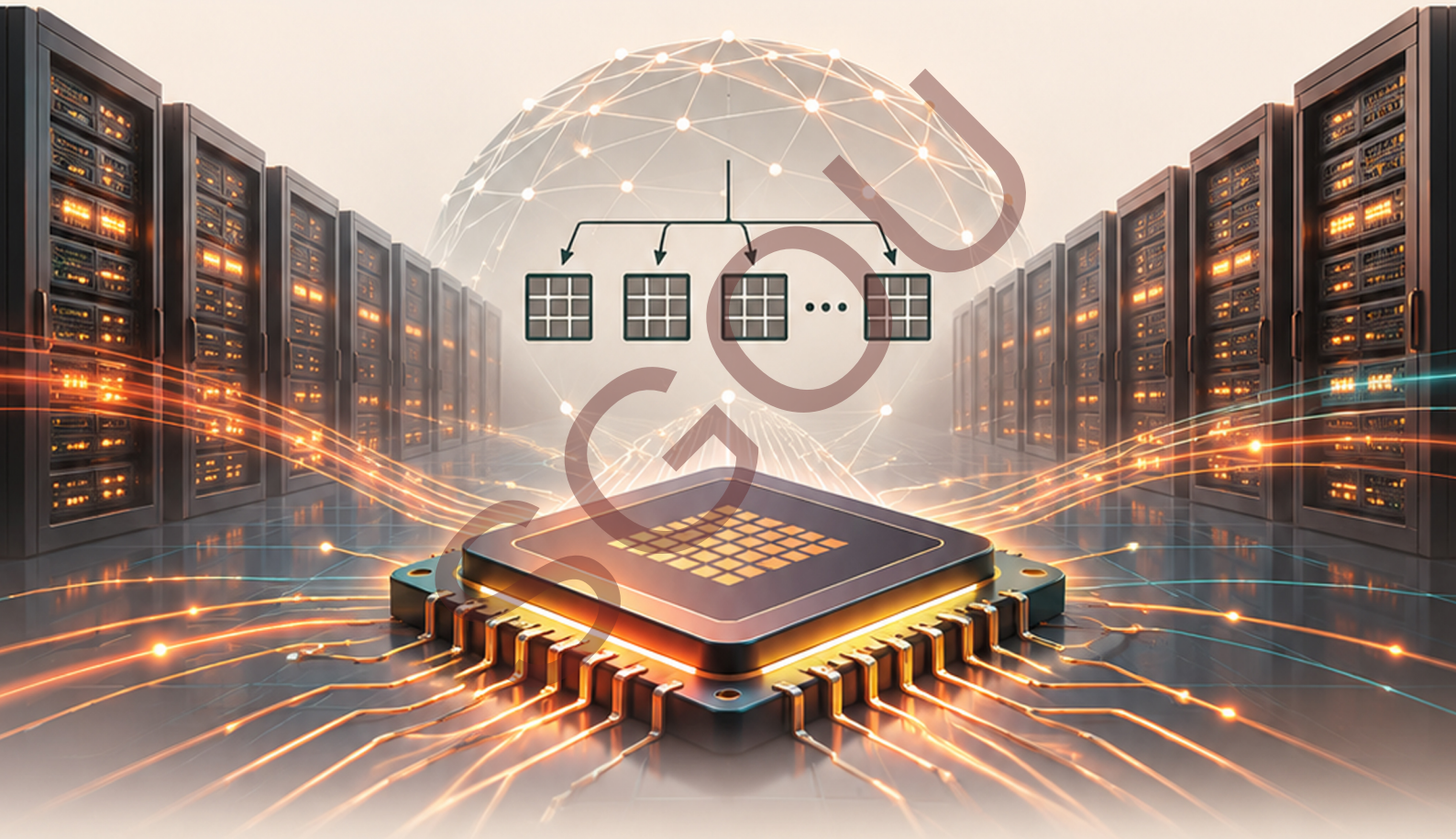
SGOU
Higher Education for All

HIGH PERFORMANCE COMPUTING

COURSE CODE: B21CA05DE

Bachelor of Computer Applications
Discipline Specific Elective Course
Self Learning Material



Parallel Computing | Distributed Systems | Large-Scale Data Processing | Performance & Scalability

SREENARAYANAGURU OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU OPEN UNIVERSITY

SGOU

Vision

To increase access of potential learners of all categories to higher education, research and training, and ensure equity through delivery of high quality processes and outcomes fostering inclusive educational empowerment for social advancement.

Mission

To be benchmarked as a model for conservation and dissemination of knowledge and skill on blended and virtual mode in education, training and research for normal, continuing, and adult learners.

Pathway

Access and Quality define Equity.

High Performance Computing

Course Code: B21CA05DE

Semester - VI

Discipline Specific Elective Course
Undergraduate Programme
Bachelor of Computer Applications
Self Learning Material

(With Model Question Paper Sets)

SGOU

SREENARAYANAGURU
OPEN UNIVERSITY

SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

SREENARAYANAGURU
OPEN UNIVERSITY

HIGH PERFORMANCE COMPUTING

Course Code: B21CA05DE
Semester- VI
Discipline Specific Elective Course
Bachelor of Computer Applications

Academic Committee

Dr. M.V. Judy
Dr. Aji S.
Dr. Vishnukumar S.
Rajesh R.
Dr. Rafidha Rehiman K.A.
P.M. Ameera Mol
Dr. Ajitha R.S.
Dr. Bindu Lal T.S.
Dr. Sreeja S.

Development of the Content

A.A. Aswathy
Pankaj Kumar G.
Shameem Anzar

Review and Edit

Dr. Mohammed Siyad B.
Dr. Tomsy Paul
Anju J.
Dr. Arun Kumar M.N.

Proofreading

Dr. Mohammed Siyad B.
Dr. Tomsy Paul
Anju J.
Dr. Arun Kumar M.N.

Scrutiny

Subi Priya Laxmi S.B.N.
Dr. Kanitha D.K.
Greeshma P.P.
Shamin S.
Aswathy V.S.
Sreerekha V.K.

Cover Design

Jobin J.

Co-ordination

Director (IC), MDDC :
Dr. Sajeevkumar G.
Coordinator, Development:
Dr. Anfal M.
Coordinator, Distribution:
Dr. Abdul Razak Kunnathodi

Scan this QR Code for reading the SLM on a digital device.

Edition
November 2026

Copyright
© Sreenarayanaguru Open University

ISBN 978-81-903304-7-3

9 788190 330473

All rights reserved. No part of this work may be reproduced in any form, by mimeograph or any other means, without permission in writing from Sreenarayanaguru Open University. Printed and published on behalf of Sreenarayanaguru Open University by Registrar, SGOU, Kollam.

www.sgou.ac.in

Visit and Subscribe our Social Media Platforms

SGOU

MESSAGE FROM VICE CHANCELLOR

Dear

With immense joy and excitement, I extend my heartfelt greetings to all of you and warmly welcome you to Sreenarayanaguru Open University.

Established in September 2020 as a state-driven initiative, Sreenarayanaguru Open University is dedicated to advancing higher education through open and distance learning. Our vision is guided by the principle of "access and quality define equity," laying the foundation for a celebration of excellence in education. I am delighted to share that we are steadfast in our commitment to uphold the highest standards and refrain from compromising on the quality of education we offer. The university draws its inspiration from the legacy of Sreenarayana Guru, a revered figure in the Indian renaissance movement. His name serves as a constant reminder for us to prioritize quality in all our academic endeavors.

Sreenarayanaguru Open University operates within the practical framework of the widely recognized "blended format." Acknowledging the constraints faced by distance learners in accessing traditional classroom settings, we have curated a pedagogical approach centered on three main components: Self Learning Material, Classroom Counselling, and Virtual Modes. This comprehensive blend is poised to deliver dynamic learning and teaching experiences, maximizing engagement and effectiveness. Our unwavering commitment to quality ensures excellence across all aspects of our educational initiatives.

The University aims to offer you an engaging and stimulating educational environment that fosters active learning. The SLM is designed to offer a comprehensive and cohesive learning experience, fostering a deep interest in the study of technological advancements in IT. Careful consideration has been given to ensure a logical progression of topics, facilitating a clear understanding of the discipline's evolution. The curriculum is thoughtfully crafted to provide ample opportunities for students to navigate through the current trends in information technology. Furthermore, this course is designed to provide essential insights into computer hardware, software classification, and foundational HTML concepts crucial for web development.

We assure you that the university student support services will closely stay with you for the redressal of your grievances during your studentship. Feel free to write to us about anything that seems relevant regarding the academic programme.

Wish you the best.



Regards,
Dr. Jagathy Raj V.P.

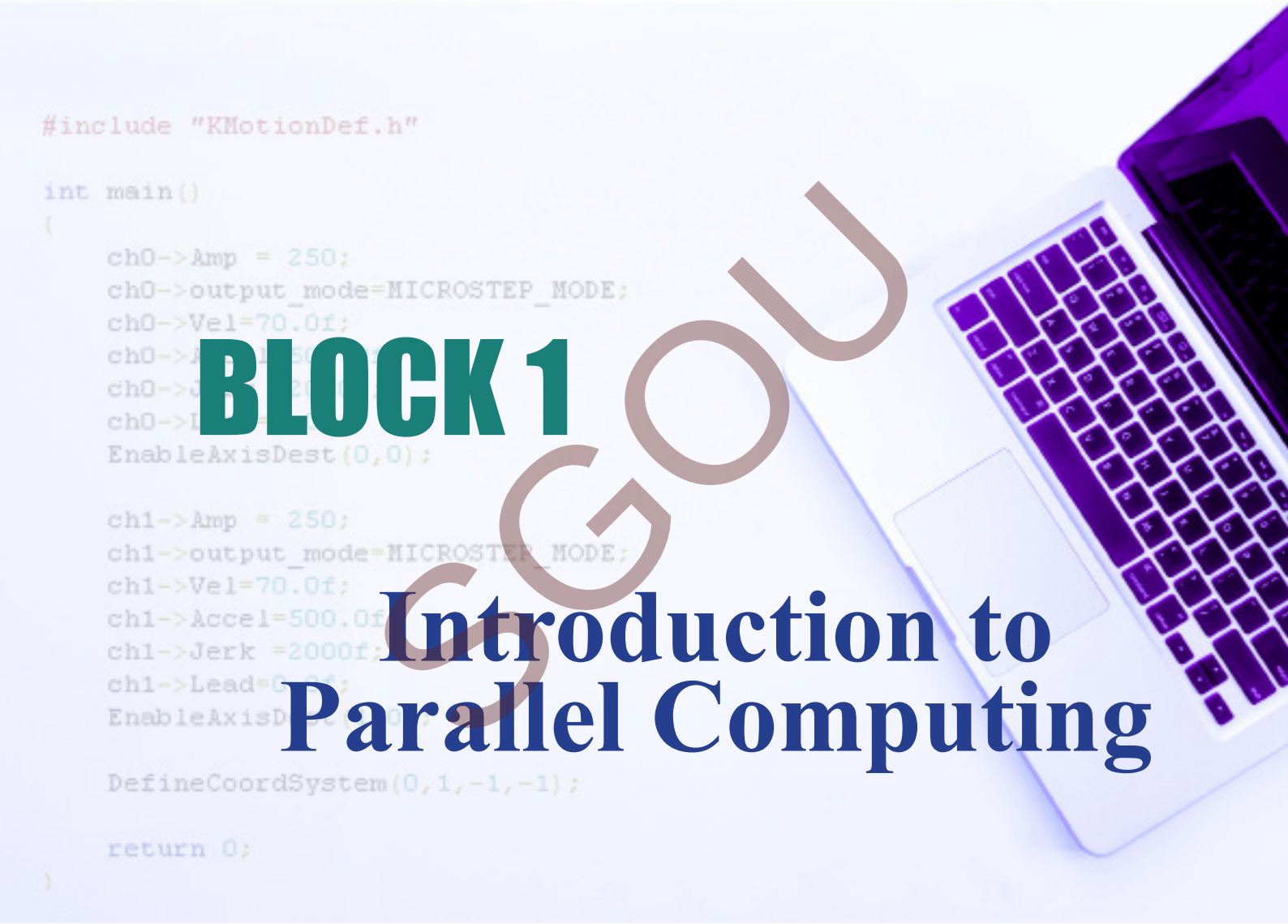
01-11-2026

SGOU

Contents

BLOCK	**1**	**Introduction to Parallel Computing**	**1**
UNIT	1	High Performance and Parallel Computing Basics	2
UNIT	2	Parallel Computer Structures	14
UNIT	3	Architectural Classification Schemes	38
UNIT	4	Parallel Processing Applications	55
BLOCK	**2**	**Pipelining and Vector Processing**	**68**
UNIT	1	Principles of Pipeline Processors	69
UNIT	2	Instruction and Arithmetic Pipelines	85
UNIT	3	Principles of Designing Pipeline Processors	96
UNIT	4	Principles of Vector Processing	107
BLOCK	**3**	**Array Processors**	**118**
UNIT	1	SIMD Array Processors	119
UNIT	2	SIMD Interconnection Networks	129
UNIT	3	Parallel Algorithms for Array Processors	140
UNIT	4	Associative Array Processing	152
BLOCK	**4**	**Computer Networks and Internet**	**163**
UNIT	1	Functional Structures	164
UNIT	2	Interconnection Networks	183
UNIT	3	Parallel Memory Organizations	203
UNIT	4	Multiprocessor Operating Systems	221
BLOCK	**5**	**Data Flow Computers, Performance Analysis and Examples**	**241**
UNIT	1	Data-Driven Computing and Languages	242
UNIT	2	Data-Flow Computer Architectures	256
UNIT	3	Performance Analysis in Multiprocessor and Data-Flow Architectures	276
UNIT	4	Case Studies of Multiprocessor and Data-Flow Systems	296
BLOCK	**6**	**GPU Computing**	**314**
UNIT	1	Heterogeneous Computing and Modern GPU Architectures	315
UNIT	2	Fundamentals of GPU Computing	328
UNIT	3	CUDA Programming and Data Parallelism	343
UNIT	4	Kernel Functions and Threading	355
Model Question Paper Sets			**370**

SGOU



BLOCK 1

Introduction to Parallel Computing

SGOU



High Performance and Parallel Computing Basics

Learning Outcomes

After completion of this unit, learner will be able to:

- define High Performance Computing (HPC) and explain its significance
- describe the concept of parallel processing and distinguish it from serial processing
- identify the forms of parallelism in uniprocessor systems
- analyze instruction-level parallelism including pipelining and superscalar execution
- evaluate the benefits and limitations of parallelism at different levels

Prerequisite

For many years, computers followed a sequential (serial) execution model, where instructions were processed one after another. A typical instruction pipeline divides instruction execution into stages such as Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). The exact number and organization of stages depend on the processor design. In a non-pipelined processor, an instruction completes its processing before the next instruction begins; in a pipelined processor, multiple instructions can occupy different stages simultaneously. While simple and easy to implement, this approach often leaves parts of the processor idle, limiting the overall performance.

As computing applications such as weather forecasting, artificial intelligence, scientific simulations, and big data analytics became increasingly complex, improving the speed of a single processor was no longer sufficient. This led to the development of parallel computing, where multiple instructions or tasks are executed simultaneously using multiple processing units or cores. The transition from sequential to parallel computing represents one of the most significant advancements in modern computer architecture.

Imagine assembling a piece of furniture using an instruction manual.

- In serial execution, one person completes Step 1, then Step 2, and finally Step 3, one after another.

SGOU

- In parallel execution, one person assembles the legs while another builds the drawers, and a third prepares the tabletop. Since different tasks are performed simultaneously, the furniture is completed much faster.

The main challenge in parallel computing is coordinating multiple tasks efficiently so that they work together without conflicts or unnecessary waiting.

Keywords

High Performance Computing (HPC), Parallel Processing, Instruction-Level Parallelism (ILP), Pipelining, Superscalar Architecture, Throughput, Latency, Speedup

Discussion

1.1.1 What is High Performance Computing?

High Performance Computing (HPC) refers to the use of powerful computing systems, including supercomputers, clusters, and parallel processing techniques, to solve computationally demanding problems efficiently. HPC combines the processing power of multiple processors or computing nodes to execute large and complex tasks efficiently.

Modern HPC systems achieve performance measured in petaflops (10^{15} floating-point operations per second), while the world's fastest supercomputers have reached the exaflop (10^{18} operations per second) scale.

Why is HPC Important?

Many real-world problems involve enormous amounts of computation that cannot be completed within a reasonable time using a single processor. HPC makes it possible to solve such problems quickly by distributing the workload across many processors.

Examples include:

- Weather forecasting and climate modelling
- Artificial Intelligence and Deep Learning
- Scientific simulations
- Drug discovery and genome sequencing
- Financial risk analysis
- Space exploration

For example, a 7-day global weather forecast generated on a standard laptop may require several weeks of computation, making the prediction useless. An HPC system can complete the same computation within a few hours, enabling timely and accurate forecasting.

SGOU

Definition

High Performance Computing (HPC) is the use of multiple computing resources working together to solve complex computational problems significantly faster than a single computer.

1.1.2 Defining Parallel Processing

Parallel processing is the simultaneous execution of multiple tasks or parts of a problem using two or more processing elements. By dividing a large problem into smaller independent sub-problems, the overall execution time can be significantly reduced.

Processors coordinate their execution through shared memory or message passing, depending on the system architecture.

Definition

Parallel Processing is the technique of decomposing a problem into smaller tasks that execute concurrently on multiple processing elements to improve performance.

Serial Computing vs Parallel Computing

Table 1.1.1 Comparison Between Serial Computing and Parallel Computing

Serial Computing	Parallel Computing
Single processor	Multiple processors
One task at a time	Multiple tasks simultaneously
Longer execution time	Reduced execution time
Simpler implementation	Requires coordination and synchronization

Example: Sum of N Numbers

Suppose N numbers are to be added.

- **Serial execution:** One processor performs $N - 1$ additions sequentially.
- **Parallel execution:** Using P processors, each processor computes the sum of N/P numbers, after which the partial sums are combined.

Approximate computational cost:

$$\text{Parallel Time} \approx N/P + \log_2(P)$$

Solved Example 1

Find the speedup obtained when adding 10,000 numbers using 4 processors.

Serial Execution

- Number of additions = 9,999

SGOU

Parallel Execution

- Each processor performs 2,499 additions
- Combining four partial sums requires 3 additions
- Critical path = 2,502 additions

Therefore,

Speedup = 9,999 / 2,502 ≈ 4.0

This represents near-ideal speedup, indicating efficient utilization of the processors.

1.1.3 Levels of Parallelism

Table 1.1.2 Levels of Parallelism in Computing

Level	Description	Example
Bit-Level Parallelism (BLP)	Increasing processor word size allows more bits to be processed per instruction.	64-bit processors
Instruction-Level Parallelism (ILP)	Multiple instructions execute simultaneously within a single processor.	Pipelining, Superscalar
Data-Level Parallelism (DLP)	The same operation is applied to multiple data elements simultaneously.	SIMD, Vector Processors
Thread-Level Parallelism (TLP)	Multiple threads execute concurrently on multiple processor cores.	Multi-core CPUs, Hyper-Threading
Task-Level Parallelism	Independent tasks execute simultaneously on different processors or machines.	Distributed Computing, MapReduce

1.1.4 Parallelism in Uniprocessor Systems

Although a uniprocessor contains only a single processing core, it can still exploit parallelism by overlapping operations within the processor. This improves processor utilization and increases throughput without requiring multiple processors.

Common techniques include:

- Instruction-Level Parallelism (ILP)
- Instruction Pipelining
- Superscalar Execution

SGOU

- Simultaneous Multithreading (SMT)
- Overlapping CPU and I/O operations

Instruction Pipelining

Instruction pipelining divides instruction execution into a sequence of stages so that multiple instructions can be processed simultaneously at different stages.

A typical RISC processor consists of five stages:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EX)
4. Memory Access (MEM)
5. Write Back (WB)

In a non-pipelined processor, each instruction completes all five stages before the next instruction begins, requiring approximately five clock cycles per instruction.

In a pipelined processor, once the pipeline is full, one instruction completes during every clock cycle, greatly increasing throughput while keeping the latency of an individual instruction approximately the same.

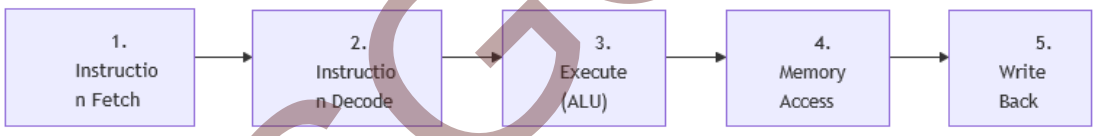


Fig. 1.1.1 Instruction pipeline

Pipeline Execution:

Table 1.1.3 Instruction pipeline execution

Cycle	IF	ID	EX	MEM	WB
1	I1	—	—	—	—
2	I2	I1	—	—	—
3	I3	I2	I1	—	—
4	I4	I3	I2	I1	—
5	I5	I4	I3	I2	I1

Pipeline Hazards

A pipeline hazard is a condition in a pipelined processor that prevents the next instruction from executing in its designated clock cycle, causing delays (called stalls) or incorrect execution if not handled properly.

SGOU

Three major hazards reduce pipeline efficiency.

1. Structural Hazard

Occurs when two or more instructions require the same hardware resource simultaneously.

2. Data Hazard

Occurs when an instruction depends on the result of a previous instruction that has not yet completed.

Example:

LOAD R1, [100]

ADD R2, R1, R3

The ADD instruction cannot proceed until the value of R1 becomes available.

3. Control Hazard

Occurs due to branch or jump instructions that alter the normal flow of execution.

Common solutions include:

- Data Forwarding (Bypassing)
- Pipeline Stalling
- Branch Prediction

Superscalar Architecture

A superscalar processor issues and executes multiple instructions during a single clock cycle by using multiple functional units. Unlike simple pipelining, which overlaps instruction execution, superscalar processors improve performance by increasing the number of instructions that begin execution in each clock cycle.

Key Characteristics

- Multiple instruction issue (typically 2–8 instructions per cycle)
- Multiple functional units
- Dynamic instruction scheduling
- Improved instruction-level parallelism

Superscalar processors issue multiple instructions per clock cycle using multiple functional units:

SGOU

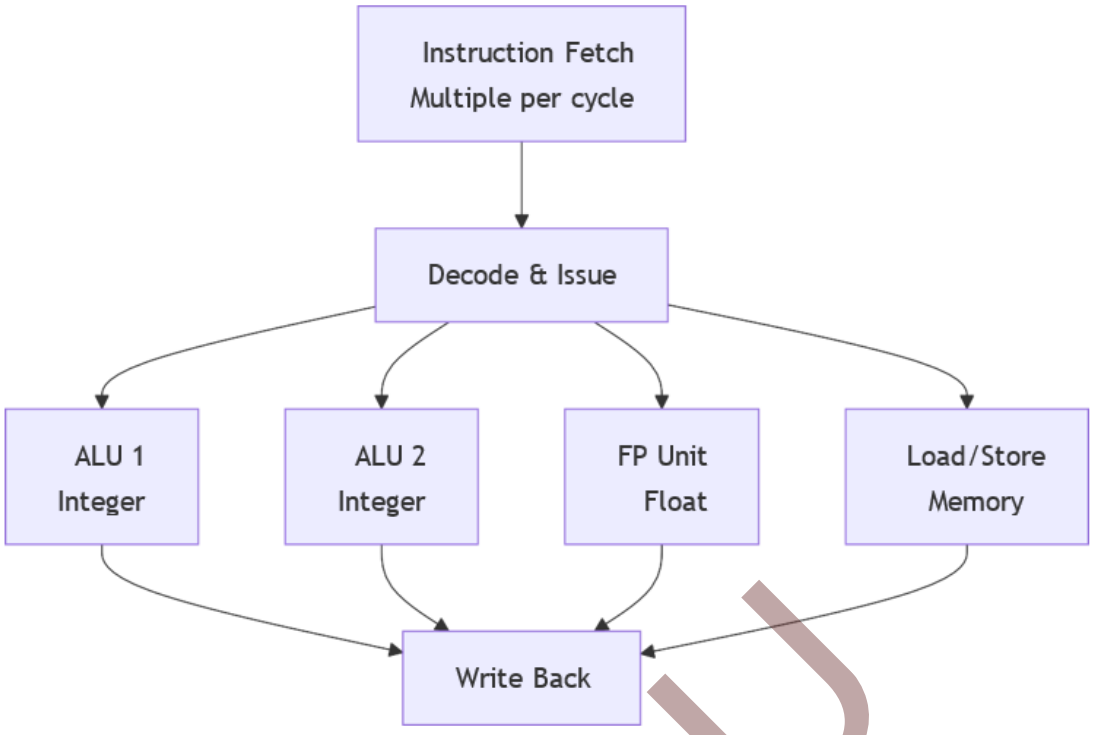


Fig. 1.1.2 Simplified architecture of a superscalar processor

Multiple instructions are decoded and dispatched simultaneously to different functional units, allowing parallel execution within a single processor core.

1.1.5 Simultaneous Multithreading (SMT)

Simultaneous Multithreading (SMT) is a processor design technique that allows a single CPU core to execute instructions from multiple threads simultaneously. Instead of leaving execution units idle when one thread is waiting for data (for example, due to a cache miss or memory access), the processor schedules instructions from another ready thread. This improves processor utilization and overall throughput.

Intel's implementation of SMT is known as Hyper-Threading Technology (HT), where a single physical core appears as two logical processors to the operating system. This enables the processor to keep its execution units busy by switching between threads whenever one thread stalls.

SMT does not create additional physical cores. Instead, it improves the utilization of existing hardware resources by allowing multiple threads to share the same execution units.

Example:

Suppose a processor has 4 physical cores with 2-way SMT (Hyper-Threading).

- Physical cores = 4
- Logical processors = 8

SGOU

- Up to 8 hardware threads can be supported by the 8 logical processors, although the threads share execution resources within the 4 physical cores.

Although SMT improves throughput, the performance gain is typically 15–30%, not 100%, because the logical threads share the same execution resources.

1.1.6 Amdahl's Law

Amdahl's Law states that the maximum performance improvement achieved through parallel computing is limited by the portion of a program that must be executed sequentially. Even if an unlimited number of processors are available, the serial portion of the program becomes the bottleneck.

If **f** is the fraction of the program that is serial and **P** is the number of processors, the maximum speedup is given by

$$S = \frac{1}{f + \frac{1-f}{P}}$$

where:

- f = Fraction of the program that is serial
- 1−f = Fraction that can be parallelized
- P = Number of processors

As the number of processors approaches infinity (P→∞),

$$S_{max} = \frac{1}{f}$$

Thus, if 10% of a program is serial (f=0.1), the maximum possible speedup is 1/0.1=10

Even with an infinite number of processors, the program cannot execute more than 10 times faster.

Solved Example 2

A program is 95% parallelizable and is executed using 8 processors. Determine the speedup.

Given

- Parallel fraction = 95%
- Serial fraction, f=0.05
- Number of processors, P=8

SGOU

Solution

$$S = \frac{1}{0.05 + \frac{0.95}{8}} = 5.93$$

Therefore, the program achieves a speedup of approximately 5.93×.

If an infinite number of processors were available,

Maximum Speedup=1/0.05=20

Hence, the maximum theoretical speedup is 20×.

1.1.7 Gustafson's Law

While Amdahl's Law assumes a fixed problem size, Gustafson's Law assumes that the problem size increases as more processors become available. Instead of completing the same task faster, additional processors are used to solve larger and more complex problems within approximately the same execution time.

The scaled speedup is given by

$$S = P - f(P - 1)$$

where:

- S = Scaled speedup
- P = Number of processors
- f = Fraction of the program that remains serial

As the problem size grows with the number of processors, the achievable speedup approaches P, making Gustafson's Law more suitable for large-scale scientific and engineering applications.

SGOU

Recap

- HPC uses aggregated resources to solve complex problems.
- Parallel processing divides problems into concurrent sub-tasks.
- Parallelism levels: bit-level, ILP, DLP, TLP, task-level.
- Uniprocessor techniques: pipelining, superscalar, SMT.
- Pipeline hazards: structural, data, control. Managed via forwarding, prediction, stalling.
- Amdahl's Law: Speedup bound by serial fraction.
- Gustafson's Law: Optimistic scaling with problem size growth.

Objective Type Questions

1. What is the term for using multiple computing resources to solve problems faster than a single processor?
2. Which law limits parallel speedup by the serial fraction?
3. How many stages does the classic instruction pipeline have?
4. What type of hazard occurs when an instruction depends on an incomplete previous result?
5. Which architecture issues multiple instructions per clock cycle?
6. What is the maximum speedup if 90% is parallelized with infinite processors?
7. Which technique executes instructions from multiple threads on one core?
8. What does SMT stand for?
9. After how many cycles does a 5-stage pipeline deliver one result per cycle (no hazards)?
10. What type of parallelism applies the same operation to multiple data elements?

Answers

1. Parallel processing.
2. Amdahl's Law.
3. Five.
4. Data hazard.
5. Superscalar.
6. 10x.
7. SMT.
8. Simultaneous Multi-Threading.
9. After pipeline fills (cycle 5).
10. Data-level parallelism.

SGOU

Assignments

1. Explain latency vs. throughput using pipelining.
2. Program: 100s serial, 80% parallel on 4 processors. Calculate speedup.
3. Describe three pipeline hazards with examples.
4. How does superscalar differ from simple pipelining?
5. Compare Intel Hyper-Threading (SMT) with AMD's SMT implementations. Discuss how SMT improves processor resource utilization and explain its limitations compared with adding physical CPU cores.
6. Identify dependencies and min cycles on dual-issue superscalar:

```
I1: ADD R1, R2, R3
I2: SUB R4, R1, R5
I3: MUL R6, R4, R7
I4: LOAD R8, [R9]
I5: ADD R8, R8, R10
```

7. Explain how instruction pipelining improves processor throughput. Illustrate the execution of five instructions using a five-stage pipeline and compare it with non-pipelined execution.
8. A program consists of 20% serial code and 80% parallelizable code. Calculate the theoretical speedup when the program is executed using 4, 8, and 16 processors using Amdahl's Law. Comment on the effect of increasing the number of processors.
9. Explain the three major types of pipeline hazards—structural, data, and control hazards. Give one suitable instruction-level example for each hazard and discuss one technique used to reduce its effect.
10. A weather forecasting application requires a large number of mathematical computations to be performed on independent data sets. Explain how data-level parallelism and task-level parallelism could be used to improve the performance of the application.

SGOU

Reference

1. Hennessy, J. L., Patterson, D. A., & Kozyrakis, C. (2025). *Computer architecture: a quantitative approach*. Morgan kaufmann.

2. Culler, D. E., Singh, J. P., & Gupta, A. (1998). Parallel computer architecture: A hardware/software approach. Morgan Kaufmann Publishers.

3. Amdahl, G. M. (1967, April). Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (pp. 483-485).

4. Gustafson, J. L. (1988). Reevaluating Amdahl's law. *Communications of the ACM*, *31*(5), 532-534.

Suggested Reading

1. Kumar, V., Grama, A., Gupta, A., & Karypis, G. (1994). *Introduction to parallel computing* (Vol. 110). Redwood City, CA: Benjamin/Cummings.

2. Thomas. Rauber, & Gudula. Rünger. *Parallel programming: for multicore and cluster systems*. Springer-Verlag.

3. Pacheco, P. (2011). *An introduction to parallel programming*. Elsevier.

4. TOP500. *TOP500 Supercomputer Sites*. https://www.top500.org/

SGOU



Parallel Computer Structures

Learning Outcomes

After completion of this unit, learner will be able to:

- explain architecture and operation of pipeline computers
- describe design principles of array processors (SIMD)
- classify multiprocessor systems by memory organization and interconnection
- compare shared-memory and distributed-memory architectures
- analyze performance trade-offs of different parallel structures

Prerequisite

Building on Unit 1, which introduced instruction-level parallelism and pipelining, this unit explores how multiple processing elements are organized to achieve higher levels of parallelism. It examines three important parallel computer structures: pipeline computers, array processors, and multiprocessor systems.

Imagine an automobile factory:

- **Pipeline Computer:** A car moves through several assembly stations. Each station performs a different task (installing the engine, fitting doors, attaching wheels). Several cars are processed simultaneously, each at a different stage of the pipeline.
- **Array Processor:** One supervisor gives the same instruction to 100 workers, such as "tighten all wheel bolts." Every worker performs the identical operation on a different car at the same time.
- **Multiprocessor System:** Ten independent teams each assemble an entire car. Every team works independently on its own car, although they may share common resources such as tools or storage.

SGOU

Keywords

Pipeline Computer, Arithmetic Pipeline, Array Processor, Multiprocessor System, Shared Memory, Distributed Memory, Interconnection Network, NUMA

Discussion

1.2.1 Pipeline Computers

A pipeline computer is a computer architecture that improves performance by dividing the execution of instructions or arithmetic operations into a sequence of stages. Each stage performs a specific part of the operation, and different stages operate simultaneously on different instructions.

Instead of waiting for one instruction to complete before starting the next, several instructions are processed concurrently, with each instruction occupying a different stage of the pipeline. This overlapping of operations significantly increases the throughput of the processor.

A pipeline computer achieves parallelism primarily by overlapping different stages of instruction or arithmetic execution, whereas a multiprocessor system uses multiple processing units to execute work concurrently. The improvement in performance comes from overlapping execution rather than executing instructions on multiple processors.

1.2.1.1 Working Principle

A pipeline works similarly to an industrial assembly line.

Imagine a car manufacturing plant where different stations perform different tasks:

- Station 1 installs the engine.
- Station 2 installs the doors.
- Station 3 installs the wheels.
- Station 4 paints the car.
- Station 5 performs quality inspection.

Instead of completing one car before starting another, several cars move through the assembly line simultaneously. At any given moment, each station works on a different car.

Similarly, in a pipeline computer, different instructions occupy different execution stages simultaneously.

After the pipeline becomes full, one instruction completes during every clock cycle, thereby improving processor throughput.

SGOU

1.2.1.2 Basic Pipeline Structure

A typical instruction pipeline consists of five stages. The stages are shown in figure 1.2.1

Table 1.2.1 Stages of Instruction Execution in the Processor Pipeline

Stage	Function
Instruction Fetch (IF)	Fetches the instruction from memory
Instruction Decode (ID)	Decodes the instruction and identifies operands
Execute (EX)	Performs arithmetic or logical operations
Memory Access (MEM)	Reads or writes data from memory if required
Write Back (WB)	Stores the result in the destination register

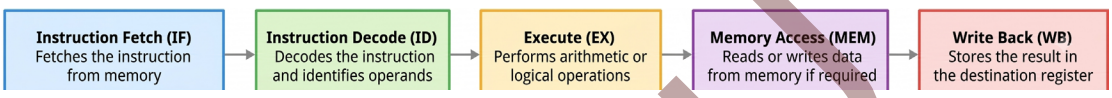


Fig. 1.2.1 Five-stage instruction pipeline

1.2.1.3 Pipeline Registers

Each pipeline stage is separated by a pipeline register.

These registers temporarily store intermediate results before passing them to the next stage.

Pipeline registers allow different stages to work independently during every clock cycle.

IF → Register → ID → Register → EX → Register → MEM → Register → WB

Pipeline Execution

Consider five instructions:I1, I2, I3, I4, I5

Their execution in a five-stage pipeline is shown below.

Table 1.2.2 Five - Stage Pipeline Execution Cycle

Clock Cycle	IF	ID	EX	MEM	WB
1	I1				
2	I2	I1			
3	I3	I2	I1		
4	I4	I3	I2	I1	
5	I5	I4	I3	I2	I1

After the fifth cycle, one instruction completes every clock cycle until all instructions finish.

SGOU

Pipeline Performance

Pipeline performance is evaluated using two important measures.

Latency

Latency is the time required for a single instruction to travel through all stages of the pipeline.

For a pipeline with five stages requiring one clock cycle each,

Latency = 5 clock cycles

Pipelining primarily improves throughput; the latency of an individual instruction is not necessarily reduced and may remain approximately equal to the total time required to pass through all pipeline stages.

Throughput

Throughput is the number of instructions completed per unit time.

After the pipeline is filled,

Throughput = One instruction per clock cycle

Thus, pipelining greatly increases throughput even though the latency of individual instructions remains almost unchanged.

Pipeline Depth

The number of stages in a pipeline is called the pipeline depth. A deeper pipeline divides instruction execution into more stages.

Advantages include:

- Higher clock frequency
- Smaller amount of work per stage
- Increased instruction throughput

However, deeper pipelines also increase:

- Pipeline overhead
- Branch penalties
- Pipeline stalls

For example, Intel's Pentium 4 NetBurst architecture used pipelines containing between 20 and 31 stages to achieve high clock frequencies. Although this improved clock speed, incorrect branch predictions caused larger performance penalties because more partially executed instructions had to be discarded.

SGOU

Arithmetic Pipeline

While an instruction pipeline processes CPU instructions, an arithmetic pipeline is designed specifically for arithmetic operations such as floating-point addition, multiplication, and division.

Instead of performing one floating-point operation at a time, different stages of the arithmetic operation are executed concurrently.

For example, floating-point addition may consist of the following stages:

1. Compare the exponents.
2. Align the mantissas.
3. Add the mantissas.
4. Normalize the result.
5. Round the final value.

Once the pipeline is full, a new pair of operands enters the arithmetic pipeline every clock cycle, allowing continuous production of results.

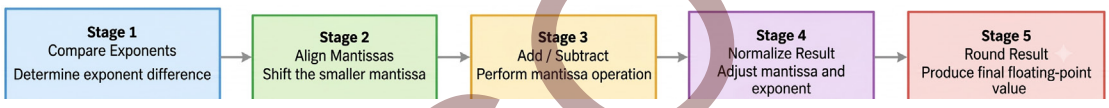


Fig. 1.2.2 Arithmetic pipeline for floating-point addition

Solved Example 1

A floating-point adder consists of 4 pipeline stages, each requiring 2 ns.

Determine:

1. Pipeline latency
2. Throughput

Solution

Latency = Number of stages × Stage time

$= 4 \times 2 \text{ ns} \quad = 8 \text{ ns}$

After the pipeline becomes full, One result is produced every 2 ns.

Therefore, Throughput = $1 / (2 \times 10^{-9})$

$= 5 \times 10^{8}$ additions/second

= 500 million additions per second

SGOU

1.2.1.5 Advantages of Pipeline Computers

- Increases processor throughput.
- Improves utilization of processor hardware.
- Allows multiple instructions to be processed simultaneously.
- Reduces idle time of functional units.
- Widely used in modern CPUs and digital signal processors.

1.2.1.6 Limitations of Pipeline Computers

- Performance decreases because of pipeline hazards.
- Branch instructions may require flushing the pipeline.
- Unequal stage delays reduce efficiency.
- Additional hardware is required for pipeline registers and control logic.

1.2.1.7 Applications

Pipeline computers are widely used in:

- General-purpose microprocessors
- Digital Signal Processors (DSPs)
- Graphics Processing Units (GPUs)
- Multimedia processors
- Embedded systems
- Scientific computing

1.2.2 Array Processors

A pipeline computer improves performance by overlapping different stages of instruction execution. An array processor, in contrast, achieves parallelism by using multiple processing elements to perform operations on different data items simultaneously.

Array processors are particularly effective when the same operation must be performed repeatedly on a large collection of data, such as elements of a matrix, pixels in an image, or values in a scientific computation.

Definition

Array Processor: An array processor is a parallel computer architecture containing multiple processing elements (PEs) that operate simultaneously on different data elements, typically under the control of a common instruction stream.

SGOU

Most classical array processors follow the SIMD (Single Instruction, Multiple Data) model.

In SIMD:

- A single control unit generates an instruction.
- The instruction is broadcast to multiple processing elements.
- Each processing element operates on a different data item.
- The processing elements generally execute the instruction simultaneously.

1.2.2.1 Basic Architecture of an Array Processor

A typical array processor consists of:

1. Control Unit (CU)
2. Multiple Processing Elements (PEs)
3. Memory
4. Interconnection Network

The control unit fetches and decodes instructions and broadcasts the instructions to the processing elements. Each PE receives the instruction and applies it to its own data.

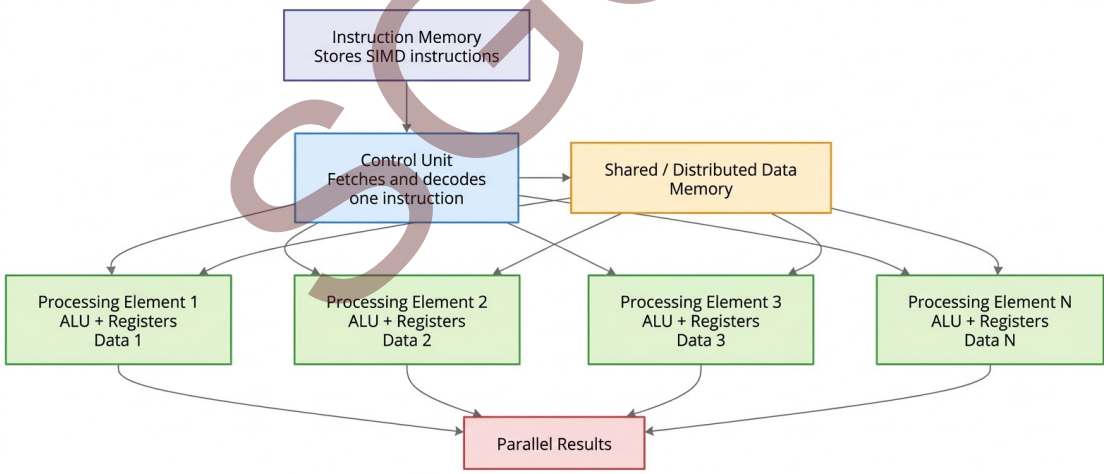


Fig. 1.2.3 Basic architecture of a SIMD array processor

Processing Elements

A Processing Element (PE) is an individual computing unit within an array processor.

Each PE generally contains:

- Arithmetic and Logic Unit (ALU)
- Registers

SGOU

- Local storage or access to memory
- Control and communication circuitry

The number of processing elements depends on the architecture.

For example, an array processor with 64 PEs can potentially perform the same operation on 64 data elements simultaneously.

1.2.2.2 Working of an Array Processor

Consider two arrays:

A = [10, 20, 30, 40]

B = [1, 2, 3, 4]

Suppose we want to calculate:

C[i] = A[i] + B[i]

Sequential Execution

A conventional processor may perform:

C[0] = A[0] + B[0]

C[1] = A[1] + B[1]

C[2] = A[2] + B[2]

C[3] = A[3] + B[3]

The additions are performed one after another.

Array Processor

With four processing elements:

PE0: C[0] = A[0] + B[0]

PE1: C[1] = A[1] + B[1]

PE2: C[2] = A[2] + B[2]

PE3: C[3] = A[3] + B[3]

All four additions can be performed simultaneously.

SGOU

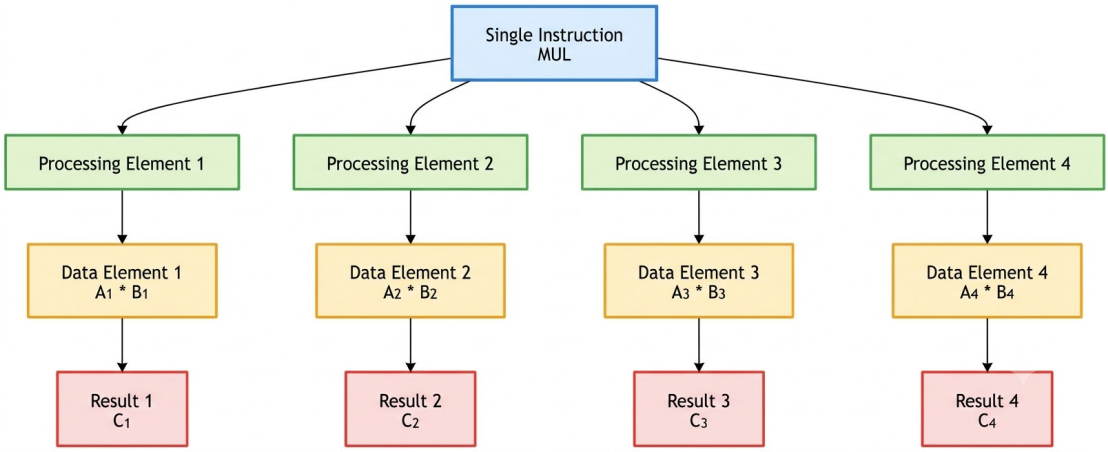


Fig. 1.2.4 A single instruction operating on multiple data elements in an array processor

SIMD Execution

The fundamental principle of an array processor can be represented as:

One Instruction→Multiple Data Elements

This is the meaning of SIMD — Single Instruction, Multiple Data.

For example, an ADD instruction can simultaneously operate on:

A0+B0, A1+B1, A2+B2, A3+B3

rather than performing the four additions sequentially.

Masking in Array Processors

Not every operation can always be performed on every processing element.

A mask allows selected processing elements to participate in an operation while others remain inactive.

For example:

PE0	PE1	PE2	PE3	PE4	PE5
ON	ON	OFF	ON	OFF	ON

If an instruction is issued, only the active processing elements execute it.

Masking is useful for conditional operations, but excessive masking can reduce processor utilization because some PEs remain idle.

1.2.2.3 Array Processor vs Vector Processor

Array processors and vector processors are closely related because both exploit data-level parallelism, but their organization differs.

SGOU

Table 1.2.3 Comparison Between Array Processors and Vector Processors

Feature	Array Processor	Vector Processor
Basic unit	Multiple processing elements	Vector functional units
Execution	Multiple PEs operate in parallel	Vector instruction operates on vector data
Architecture	SIMD	Vector processing
Example operation	Same instruction across many PEs	Vector addition
Best suited for	Highly parallel data	Long sequences of numerical data

Both architectures are particularly useful for numerical and scientific applications.

1.2.2.4 Advantages of Array Processors

1. **High Data-Level Parallelism:** Multiple data elements can be processed simultaneously.
2. **High Throughput:** Large numbers of similar operations can be completed efficiently.
3. **Efficient for Regular Computations:** They perform particularly well when the same operation is repeatedly applied to large datasets.
4. **Suitable for Matrix Operations:** Matrix addition, multiplication, and other numerical operations can benefit greatly from SIMD execution.
5. **Useful for Image Processing:** Image processing frequently applies the same operation to thousands or millions of pixels.

1.2.2.5 Limitations of Array Processors

1. **Poor Performance for Irregular Computations:** Applications in which different data elements require substantially different operations are less suitable.
2. **Branching Reduces Efficiency:** Conditional statements can cause some processing elements to remain idle.
3. **Memory Bottlenecks:** The processing elements may need to access large amounts of data. If memory cannot supply data quickly enough, the PEs remain idle.
4. **Limited Flexibility:** Array processors are most effective when the computation has a regular and predictable structure.

SGOU

5. **Masked Processing:** When only some PEs are active, hardware resources are not fully utilized.

1.2.2.6 Applications

Array processors are particularly useful in applications involving large amounts of data on which similar operations are performed. Examples include Matrix operations, Image processing, Digital signal processing, Scientific simulations, Weather and climate modelling, Computer graphics, Machine learning, Numerical computations.

1.2.2.7 Modern Connection: GPUs

The principles of array processing are highly relevant to modern Graphics Processing Units (GPUs).

GPUs contain a large number of arithmetic processing resources and are designed to execute operations on large amounts of data in parallel. This makes them particularly effective for Image processing, Matrix operations, Deep learning, Scientific computing, Computer graphics.

Modern GPUs use SIMT (Single Instruction, Multiple Threads) rather than being identical to classical SIMD array processors, but they share the fundamental idea of exploiting massive data-level parallelism.

Solved Example 2: Array Processor Speedup

An array contains 1,024 elements. The same operation must be performed on every element. Assume an array processor has 64 processing elements, and each PE processes one element per cycle.

Determine the approximate number of cycles required.

Solution

Number of data elements: N=1024

Number of processing elements: P=64

Each PE processes one element per cycle. Therefore,

Number of cycles=N/P =1024/64 =16

Therefore, approximately 16 cycles are required, assuming ideal conditions and ignoring instruction, memory, and synchronization overhead.

A sequential processor would require approximately 1,024 processing cycles for the same operation.

The ideal computational speedup is therefore: S=1024/16=64

Thus, the theoretical speedup is 64×, equal to the number of processing elements.

SGOU

Note: Actual speedup is generally lower because of memory access time, instruction overhead, synchronization, and non-uniform workloads.

Array Processor and Flynn's Taxonomy

Array processors are primarily associated with SIMD architecture in Flynn's taxonomy.

Table 1.2.4 Characteristics of an Array Processor in Flynn's Classification (SIMD)

Flynn Category	Array Processor
Instruction streams	Single
Data streams	Multiple
Classification	SIMD
Parallelism	Data-level parallelism

1.2.3 Multiprocessor Systems

Introduction

An array processor performs the same operation on multiple data elements simultaneously. A multiprocessor system, in contrast, contains multiple processors that can execute different instructions and work on different tasks concurrently.

Multiprocessor systems are therefore an important form of processor-level parallelism and are widely used in servers, workstations, high-performance computing systems, and modern multi-core computers.

Definition

Multiprocessor System: A computer system containing two or more processors that cooperate to execute programs concurrently. The processors may share memory and other system resources or may communicate through an interconnection network.

Each processor can execute instructions independently. Depending on the architecture, processors may:

- access a common memory,
- have local/private memory,
- communicate through an interconnection network, or
- use a combination of these approaches.

Basic Multiprocessor Architecture

A simple shared-memory multiprocessor system can be represented as follows.

SGOU

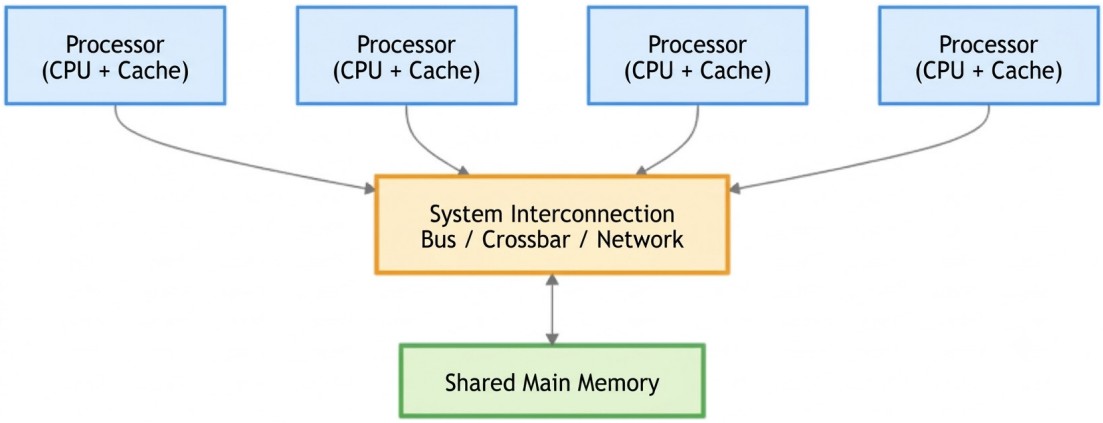


Fig. 1.2.5 Basic organization of a shared-memory multiprocessor system

The interconnection network provides communication between processors and memory.

Why Multiprocessor Systems?

A single processor has limitations in terms of performance and the amount of work it can complete in a given time.

Multiple processors allow different parts of a program to execute concurrently.

For example, consider a program containing four independent tasks:

Task A ⟶ Processor 1

Task B ⟶ Processor 2

Task C ⟶ Processor 3

Task D ⟶ Processor 4

The tasks can execute simultaneously when sufficient processor resources are available.

This can reduce the overall execution time and increase system throughput.

1.2.3.1 Characteristics of Multiprocessor Systems

Important characteristics include:

1. **Multiple Processing Units**

A multiprocessor system contains two or more processors or processing cores.

2. **Concurrent Execution**

Different processors can execute different instructions simultaneously.

3. **Communication**

Processors must communicate and exchange data to coordinate their execution.

SGOU

4. Resource Sharing

Processors may share:

- Main memory
- Storage
- Input/output devices
- Interconnection networks

5. Synchronization

When multiple processors access shared data, synchronization mechanisms are required to prevent conflicts.

1.2.3.2 Types of Multiprocessor Systems

Multiprocessor systems can be classified according to their memory organization.

The two important categories are:

1. Shared-memory multiprocessors
2. Distributed-memory systems

1. Shared-Memory Multiprocessors

In a shared-memory system, all processors can access a common address space.

A processor can access shared data using normal load and store operations.

Advantages

- Easier programming model.
- Processors can directly share data.
- Communication can be performed through shared variables.
- Suitable for multi-core and multi-processor systems.

Limitations

- Memory bandwidth can become a bottleneck.
- Synchronization is required.
- Cache coherence becomes important.
- Scalability may be limited as the number of processors increases.

SGOU

UMA and SMP

Uniform Memory Access (UMA)

In a UMA system, all processors have approximately the same access time to shared memory.

If processor P1 accesses memory location X and processor P4 accesses memory location Y, the memory access time is approximately the same, assuming the same conditions.

UMA systems are relatively simple to design and are commonly associated with smaller shared-memory multiprocessors.

Symmetric Multiprocessing (SMP)

Symmetric Multiprocessing (SMP) is a shared-memory architecture in which multiple processors are treated equally by the operating system.

In an SMP system:

- All processors are generally identical or equivalent.
- Each processor can execute operating-system tasks.
- All processors have access to shared memory.
- Work can be scheduled dynamically among processors.

Example

A computer with eight equivalent CPU cores running a common operating system can use an SMP-style organization.

Key Idea

SMP = Multiple equivalent processors + shared memory + equal processor access to system resources.

NUMA

As the number of processors increases, a single shared-memory design can become difficult to scale.

NUMA (Non-Uniform Memory Access) addresses this problem by dividing memory into regions associated with different processors or processor groups.

A processor can access:

- **Local memory** — relatively faster.
- **Remote memory** — relatively slower because it belongs to another processor or node.

SGOU

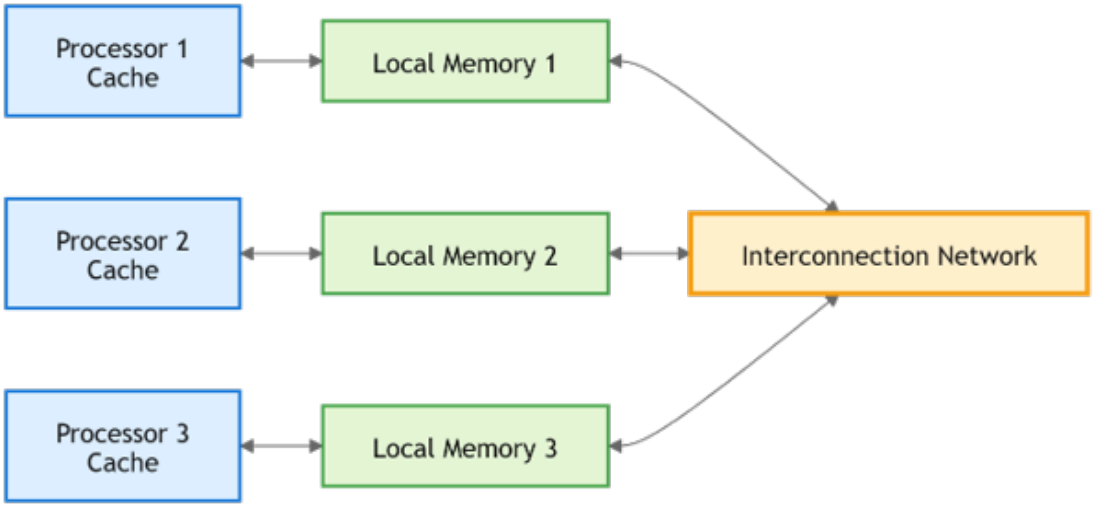


Fig. 1.2.6 Simplified NUMA organization

Advantages of NUMA

- Better scalability than a centralized shared-memory design.
- Local memory can provide faster access.
- Memory capacity can increase with additional nodes.

Limitation

Programs should preferably access local memory. Excessive remote-memory access can reduce performance.

Shared Memory vs Distributed Memory

A distributed-memory system gives each processor or computing node its own private memory.

Table 1.2.5 Comparison Between Shared-Memory and Distributed-Memory Architectures

Feature	Shared Memory	Distributed Memory
Address space	Common	Separate
Memory	Shared	Private/local
Communication	Shared variables	Message passing
Programming	Relatively easier	More complex
Synchronization	Important	Explicit communication required
Scalability	Generally limited	Generally high
Example	Multi-core server	HPC cluster

SGOU

In distributed-memory systems, processors communicate using a network and a message-passing system such as MPI (Message Passing Interface).

Interconnection Networks

Processors and memory must be connected using an interconnection network.

The interconnection network determines how efficiently processors can communicate with memory and with one another.

Common interconnection structures include:

Bus

All processors share a common communication bus.

Advantages:

- Simple
- Low cost

Limitation:

- Scalability is poor because many processors compete for the same bus.

Crossbar

A crossbar provides multiple possible connections between processors and memory modules. A crossbar can provide high communication bandwidth but requires significant hardware.

1.2.3.3 Multistage Interconnection Network

A multistage network uses multiple switching stages to connect processors and memory.

Examples include:

- Omega network
- Butterfly network

These networks provide a compromise between the cost of a bus and the connectivity of a crossbar.

Cache Coherence

In shared-memory multiprocessors, each processor may have its own cache.

This creates an important problem.

SGOU

Suppose:

Initially:

Memory X = 0

P1 cache → X = 0

P2 cache → X = 0

If P1 changes X:

P1 writes X = 5

P2 may still have the old value:

P2 cache → X = 0

This inconsistency must be prevented.

Definition

Cache coherence is the mechanism that ensures that different processor caches maintain a consistent view of shared memory.

Common cache-coherence approaches include:

- Snooping protocols
- Directory-based protocols

MESI is a commonly used cache-coherence protocol with four states:

- M — Modified
- E — Exclusive
- S — Shared
- I — Invalid

Synchronization

When multiple processors access shared data simultaneously, synchronization is required.

Consider:

Counter = 100

P1 reads Counter → 100

P2 reads Counter → 100

P1 increments → 101

P2 increments → 101

SGOU

The expected result after two increments is: 102

but the actual result can become: 101

This is a race condition.

Synchronization mechanisms such as:

- Locks
- Semaphores
- Barriers
- Atomic operations

help coordinate processor access to shared data.

1.2.3.4 Advantages of Multiprocessor Systems

1. **Increased Performance :** Multiple processors can execute independent tasks concurrently.
2. **Higher Throughput :** More tasks can be completed during a given period.
3. **Scalability :** Additional processors can be added to increase computational capacity, subject to architectural and software limitations.
4. **Reliability :** Some systems can continue operating even if one processor fails, depending on system design.
5. **Resource Sharing :** Processors can share memory, storage, and I/O resources.

1.2.3.5 Limitations

1. **Synchronization Overhead :** Processors must coordinate their execution.
2. **Communication Overhead :** Communication between processors consumes time and system resources.
3. **Memory Bottlenecks :** Multiple processors may compete for access to memory.
4. **Coherence :** Maintaining consistent cached copies of shared data adds hardware and performance overhead.
5. **Software Complexity :** Programs must be designed to exploit parallelism effectively.

SGOU

6. **Limited Speedup :** Adding more processors does not necessarily produce proportional performance improvement because of serial portions, communication, synchronization, and load imbalance.

This connects directly to Amdahl's Law introduced in Unit 1.

1.2.3.6 Applications

Multiprocessor systems are widely used in:

- Personal computers
- Workstations
- Database servers
- Web servers
- Cloud computing
- Scientific computing
- High-performance computing
- Artificial intelligence
- Simulation and modelling
- Large-scale data processing

1.2.3.7 Comparison of Multiprocessor Organizations

Table 1.2.6 Comparison of UMA/SMP, NUMA, and Distributed Memory

Feature	UMA/SMP	NUMA	Distributed Memory
Memory organization	Shared	Shared, distributed across nodes	Private per node
Memory access	Uniform	Non-uniform	Local
Communication	Shared memory	Shared memory	Message passing
Scalability	Moderate	High	Very high
Programming difficulty	Lower	Moderate	Higher
Typical application	Multi-core/multi-CPU systems	Large shared-memory servers	HPC clusters

Connection with Flynn's Taxonomy

Multiprocessor systems are generally associated with MIMD (Multiple Instruction, Multiple Data) architecture.

SGOU

Different processors can execute different instructions on different data.

Multiprocessor → MIMD → Task/Thread-Level Parallelism

This distinguishes a multiprocessor system from an array processor:

Table 1.2.7 Comparison Between Array Processors and Multiprocessors

Array Processor	Multiprocessor
SIMD	MIMD
Same instruction	Different instructions possible
Different data	Different data
Data-level parallelism	Task/thread-level parallelism
Lockstep execution often used	Independent execution

A multiprocessor system uses two or more processors to execute tasks concurrently. Shared-memory systems allow processors to communicate through a common address space, while distributed-memory systems provide private memory and rely on explicit communication.

Recap

- Array processors exploit data-level parallelism.
- Classical array processors generally follow the SIMD model.
- A control unit broadcasts instructions to multiple processing elements.
- Each PE operates on different data.
- Masking allows selected PEs to participate in an operation.
- Array processors are highly effective for regular numerical and data-parallel applications.
- Irregular computations and branching can reduce PE utilization.
- Modern GPUs apply related principles of massive data-parallel processing.
- Multiprocessor systems contain multiple processors or processing cores.
- Processors can execute tasks concurrently.
- Shared-memory systems provide a common address space.
- UMA provides approximately uniform memory access.
- NUMA provides different access times for local and remote memory.

SGOU

- SMP uses multiple equivalent processors sharing memory.
- Distributed-memory systems provide private memory to processors or nodes.
- Interconnection networks enable communication between processors and memory.
- Cache coherence maintains consistency among processor caches.
- Synchronization prevents conflicts during concurrent execution.
- Multiprocessor systems are generally associated with MIMD in Flynn's taxonomy.
- Actual speedup is limited by serial execution, communication, synchronization, and other overheads.

Objective Type Questions

1. What computer organization divides instruction execution into overlapping stages?
2. What historical system is regarded as a pioneering example of large-scale data-parallel computing?
3. Which memory architecture provides approximately equal memory access time for all processors?
4. What does NUMA stand for?
5. Which cache-coherence protocol uses the Modified, Exclusive, Shared, and Invalid states?
6. Which programming model is commonly used for communication in distributed-memory systems?
7. Which interconnection network provides multiple possible connections between processors and memory modules?
8. Which component broadcasts instructions to the processing elements in a classical array processor?
9. What problem occurs when multiple processors access shared data without proper synchronization?
10. What is the primary limitation of a bus-based multiprocessor interconnection?

SGOU

Answers

1. Pipeline computer
2. ILLIAC IV
3. UMA
4. Non-Uniform Memory Access
5. MESI
6. MPI
7. Crossbar
8. Control Unit
9. Race condition
10. Bus contention

Assignments

1. Why doesn't deeper pipeline give proportionally higher performance?
2. A SIMD array processor contains 128 processing elements operating at 1 GHz, with each processing element capable of performing one floating-point operation per cycle. What is the theoretical peak performance? Why might the actual performance be lower?
3. MESI advantages over simple write-invalidate?
4. Explain the working of a five-stage instruction pipeline and discuss how pipelining improves throughput without proportionally reducing the latency of an individual instruction.
5. A floating-point arithmetic pipeline contains 5 stages, with each stage requiring 3 ns. Calculate the pipeline latency and the maximum theoretical throughput after the pipeline becomes full.
6. Compare an array processor with a vector processor in terms of processing elements, instruction execution, data organization, and suitable applications.
7. Consider an image-processing application in which the same brightness adjustment must be applied to one million pixels. Explain how an array processor or SIMD architecture can improve performance.

SGOU

8. A program has 20% serial execution and 80% parallel execution. Calculate the theoretical speedup using 2, 4, and 8 processors using Amdahl's Law. Discuss why the speedup does not increase proportionally with the number of processors.

9. Compare UMA, NUMA, and distributed-memory architectures in terms of memory organization, memory access, communication, scalability, and programming complexity.

10. Explain the cache-coherence problem in a shared-memory multiprocessor system using a suitable example. Describe how protocols such as MESI help maintain a consistent view of shared data.

Reference

1. Hennessy, J. L., Patterson, D. A., & Kozyrakis, C. (2025). *Computer architecture: a quantitative approach*. Morgan kaufmann.

2. Culler, D. E., Singh, J. P., & Gupta, A. (1998). Parallel computer architecture: A hardware/software approach. Morgan Kaufmann Publishers.

3. Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE transactions on computers*, *100*(9), 948-960.

4. Amdahl, G. M. (1967). Validity of the single processor approach to achieving large-scale computing capabilities. *AFIPS Conference Proceedings, 30*, 483–485.

Suggested Reading

1. Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to Parallel Computing* (2nd ed.). Pearson Education.

2. Rauber, T., & Rünger, G. (2013). *Parallel programming* (pp. 169-226). Berlin, Germany:: springer.

3. Pacheco, P. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann.

4. Hwang, K., & Briggs, F. A. (1990). *Computer architecture and parallel processing*. McGraw-Hill, Inc.

SGOU



Architectural Classification Schemes

Learning Outcomes

After completion of this unit, learner will be able to:

- explain Flynn's taxonomy and its four categories
- illustrate each category with diagrams and examples
- dcmpare characteristics, strengths, and limitations of each
- evaluate where existing systems fit within Flynn's taxonomy
- explore alternative schemes (Feng's, Handler's)

Prerequisite

Flynn’s taxonomy classifies computer architectures based on two dimensions: the number of instruction streams and the number of data streams processed simultaneously. Similar to biological taxonomy, it provides a systematic way to categorize and compare computer architectures. It consists of four categories: SISD (Single Instruction, Single Data), SIMD (Single Instruction, Multiple Data), MISD (Multiple Instruction, Single Data), and MIMD (Multiple Instruction, Multiple Data).

A classroom analogy helps illustrate these categories. SISD is like one teacher teaching one student, while SIMD is like a teacher instructing 30 students to solve the same problem using different data. MIMD is similar to 30 students independently working on different problems with different instructions. In contrast, MISD resembles multiple teachers giving different instructions to the same student, a configuration that is rare in general-purpose computing.

Keywords

Instruction Stream, Data Stream, SISD, SIMD, MISD, MIMD, SPMD

SGOU

Discussion

1.3.1 Flynn's Taxonomy

Proposed by Michael J. Flynn, Flynn's taxonomy classifies computer architectures according to the number of instruction streams and data streams they can handle concurrently (Fig 1.3.1). Based on these two dimensions, architectures are classified into four categories:

- **SISD:** Single Instruction, Single Data
- **SIMD:** Single Instruction, Multiple Data
- **MISD:** Multiple Instruction, Single Data
- **MIMD:** Multiple Instruction, Multiple Data

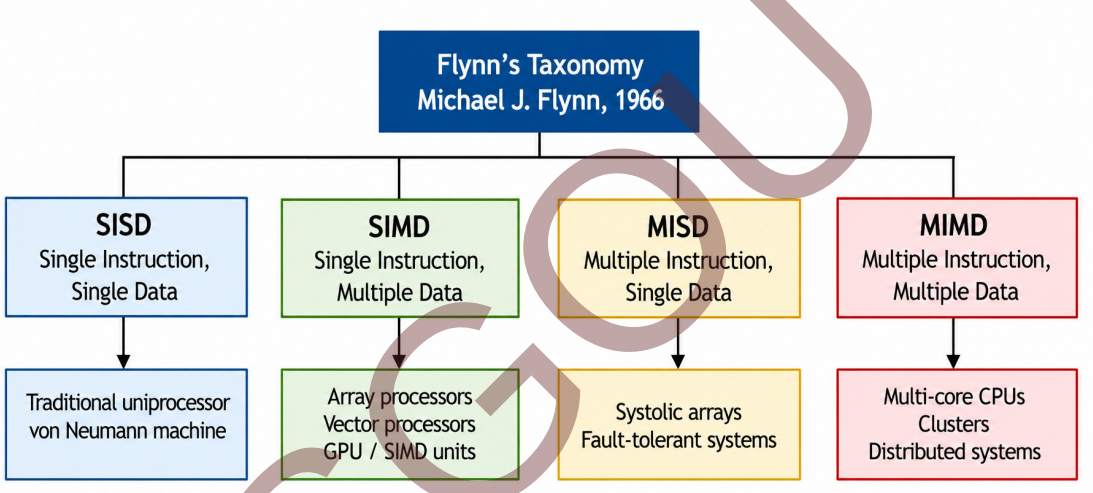


Fig. 1.3.1 Flynn's taxonomy of computer architectures

1.3.1.1 SISD

SISD stands for Single Instruction, Single Data. It represents a system in which a single instruction stream operates on a single data stream at a time (Fig 1.3.2).

A traditional von Neumann computer is the classical example of SISD. Instructions are fetched and executed sequentially, with each instruction operating on the required data.

Characteristics

- One instruction stream.
- One data stream.
- Traditionally associated with sequential execution.
- A single processing unit executes the instruction stream.
- Suitable for sequential algorithms and general-purpose computation.

SGOU

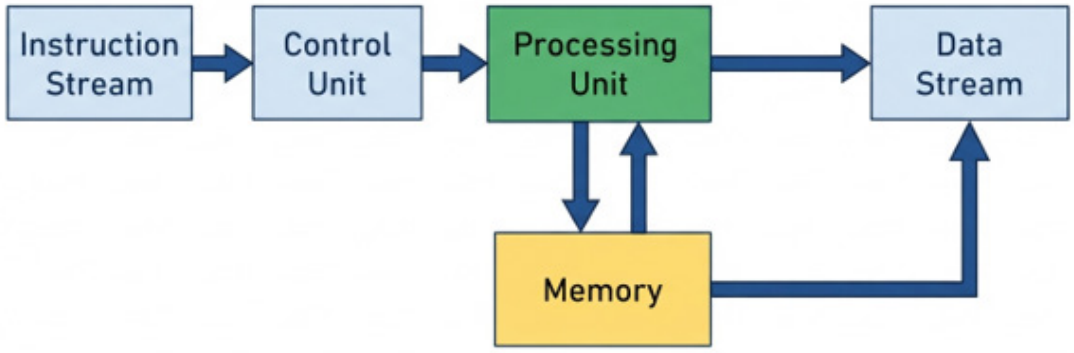


Fig. 1.3.2 Basic architecture of a Single Instruction, Single Data (SISD) system

For example, consider the following operations:

C1 = A1 + B1

C2 = A2 + B2

C3 = A3 + B3

C4 = A4 + B4

In a basic SISD system, these operations are performed sequentially.

Modern Processors and SISD

Modern processors are more complex than the classical SISD model. A single CPU core may use pipelining, superscalar execution, out-of-order execution, and SIMD extensions to exploit different forms of parallelism.

Therefore, it is more accurate to state:

Classical single-processor systems are commonly used as examples of SISD. Modern processors may incorporate multiple forms of parallelism, so their classification depends on the architectural level being considered.

A superscalar processor can issue and execute multiple instructions concurrently, while SIMD extensions explicitly provide multiple-data parallelism.

i.e., SISD = One instruction stream + One data stream.

A traditional sequential Von Neumann computer is the standard example.

1.3.1.2 SIMD

SIMD stands for Single Instruction, Multiple Data. In a SIMD architecture, a single instruction stream controls multiple processing elements or execution lanes, allowing the same operation to be performed simultaneously on multiple data elements (Fig 1.3.3).

SIMD is particularly useful when the same computation must be repeatedly applied to a large collection of independent data items. It is therefore an important form of data-level parallelism.

SGOU

Basic Organization

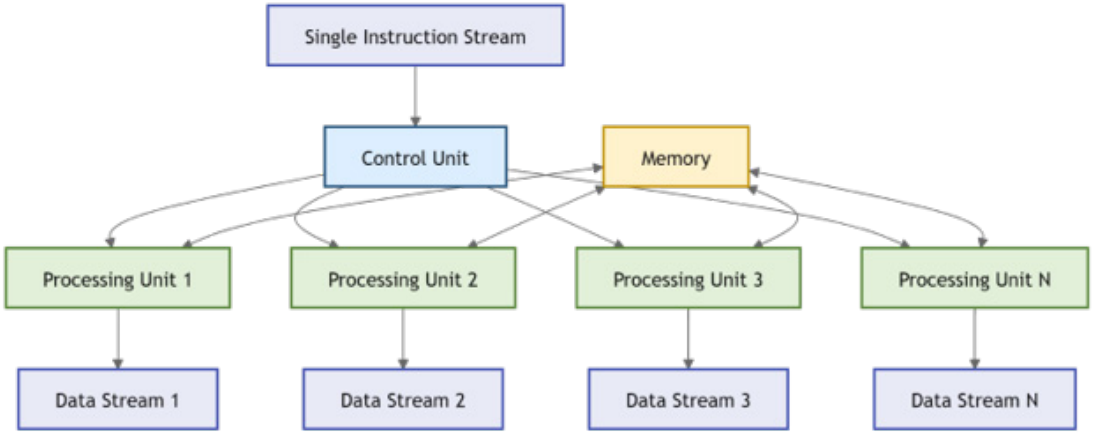


Fig. 1.3.3 Basic architecture of a Single Instruction, Multiple Data (SIMD) system

For example, consider the vector addition:

C[i]=A[i]+B[i]

For four data elements:

A = [10, 20, 30, 40]

B = [1, 2, 3, 4]

A SIMD processor can perform the additions conceptually as:

PE0: 10 + 1 = 11

PE1: 20 + 2 = 22

PE2: 30 + 3 = 33

PE3: 40 + 4 = 44

Thus,

C = [11, 22, 33, 44]

The same ADD operation is applied to multiple data elements.

Characteristics

- One instruction stream.
- Multiple data elements.
- Same operation is applied to multiple data values.
- Exploits data-level parallelism.
- Particularly effective for regular, independent computations.

SGOU

- Reduces the instruction-fetch and decode overhead compared with processing every data element separately.

Common Applications

SIMD is particularly useful in:

- Matrix and vector operations
- Image and video processing
- Digital signal processing
- Scientific and numerical computing
- Multimedia applications
- Machine learning and data analytics

Modern Examples

Modern CPUs provide SIMD instruction extensions such as Intel AVX/AVX-512 and ARM NEON/SVE. These allow a single instruction to operate on multiple numerical values packed into a vector register.

For example, a 256-bit SIMD register can hold:

256/32=8

32-bit floating-point values.

Therefore, a 256-bit SIMD addition can process eight 32-bit floating-point elements in one vector instruction, assuming the operation and data alignment are suitable.

Table 1.3.1 compares scalar processing, which handles one data element per instruction, with SIMD processing, which applies the same instruction to multiple data elements simultaneously.

Table 1.3.1 SIMD vs Scalar Processing

Feature	Scalar Processing	SIMD Processing
Data elements per instruction	One	Multiple
Instruction	One operation on one data item	Same operation on multiple data items
Parallelism	Limited	Data-level
Suitable for	Irregular operations	Regular, repetitive operations
Example	C[i] = A[i] + B[i] one element at a time	Multiple C[i] additions simultaneously

SGOU

Limitation

SIMD is most effective when the same operation can be applied to multiple data elements. Conditional branches or different operations for different elements can reduce SIMD utilization.

For example, if some elements require addition while others require multiplication, executing them using a single SIMD instruction becomes less efficient.

i.e., SIMD = Single Instruction + Multiple Data.

The defining characteristic of SIMD is that one instruction operates on multiple data elements simultaneously.

Connection to Array Processors

The array processor discussed in Unit 2 is a classical example of SIMD architecture. It uses multiple processing elements that receive the same instruction and operate on different data.

Thus,

Array Processor→ SIMD→ Data-Level Parallelism

Modern GPUs use the related SIMT (Single Instruction, Multiple Threads) execution model rather than being simply classified as classical SIMD machines. SIMT will be discussed separately when considering modern parallel architectures.

1.3.1.3 MISD

MISD stands for Multiple Instruction, Single Data. In this category, multiple instruction streams operate on the same data stream.

MISD is the least commonly encountered category in Flynn's taxonomy. Unlike SISD, SIMD, and MIMD, there are few widely accepted examples of pure MISD architectures in practical computer systems (Fig 1.3.4).

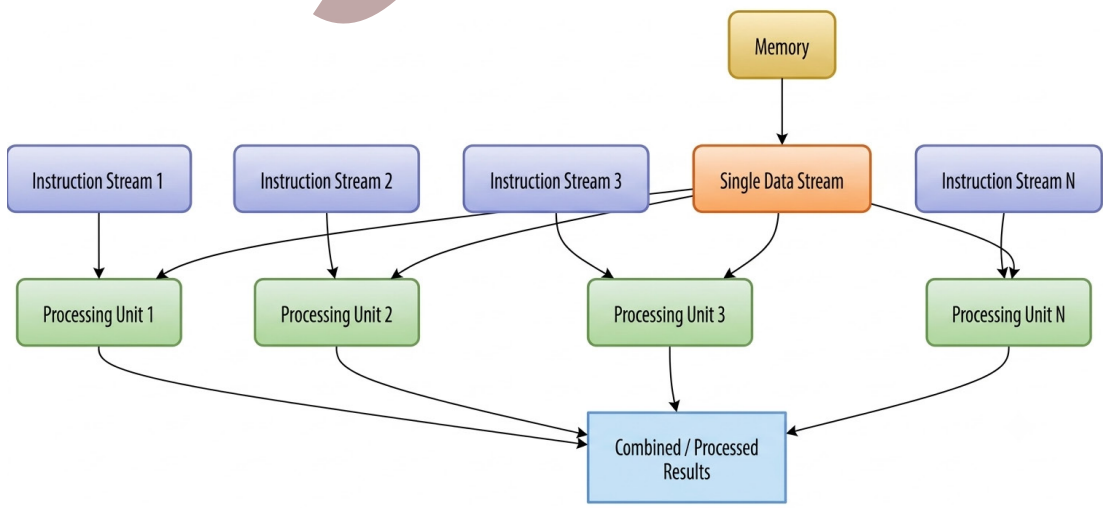


Fig. 1.3.4 Basic architecture of a Multiple Instruction, Single Data (MISD) system

SGOU

For example, consider the same input data X being processed by three different operations:

Instruction 1: X → Filtering

Instruction 2: X → Transformation

Instruction 3: X → Statistical Analysis

The same data stream is supplied to multiple processing paths, with each path performing a different operation.

This illustrates the basic MISD concept:

Multiple Instructions → Single Data Stream

Characteristics

- Multiple instruction streams.
- A single data stream.
- Different processing units may perform different operations on the same input data.
- Useful conceptually for applying multiple transformations or analyses to common data.
- Pure MISD implementations are rare.

Possible MISD-Like Applications

MISD has been associated with specialized applications such as:

- Fault-tolerant computing, where multiple processing units independently process the same input and their results are compared.
- Redundant computation, where different algorithms or implementations operate on the same input to improve reliability.
- Certain stream-processing or pipeline configurations may exhibit MISD-like characteristics, although they do not necessarily constitute pure MISD architectures.

Many architectures sometimes described as MISD do not fit the category perfectly. Therefore, MISD is best presented as a conceptual category of Flynn's taxonomy, rather than associating it with one definitive modern processor.

Why is MISD Rare?

Most practical parallel applications benefit from either:

- applying the same operation to many data elements → SIMD, or
- applying different operations to different data → MIMD.

SGOU

Performing multiple different instruction streams on exactly the same data stream is much less common.

Therefore, MISD is theoretically valid in Flynn's taxonomy but has relatively few clear, widely accepted practical implementations. Table 1.3.2 compares MISD and SIMD architectures in terms of instruction streams, data streams, processing methods, and typical applications.

Table 1.3.2 Comparison with SIMD

Feature	SIMD	MISD
Instruction streams	One	Multiple
Data streams	Multiple	One
Operation	Same operation	Different operations
Example	Add many array elements	Apply different processing paths to same data
Practical use	Very common	Rare

i.e., MISD = Multiple Instructions + Single Data.

It is the least common Flynn category and is primarily useful as a conceptual classification for systems involving multiple processing operations on a common data stream.

1.3.1.4 MIMD

MIMD stands for Multiple Instruction, Multiple Data. In a MIMD architecture, multiple processing elements can execute different instruction streams on different data sets simultaneously.

MIMD is the most flexible and widely used category for general-purpose parallel computing. Modern multi-core processors, multiprocessor systems, and distributed-memory clusters are commonly associated with MIMD architectures (Fig 1.3.5).

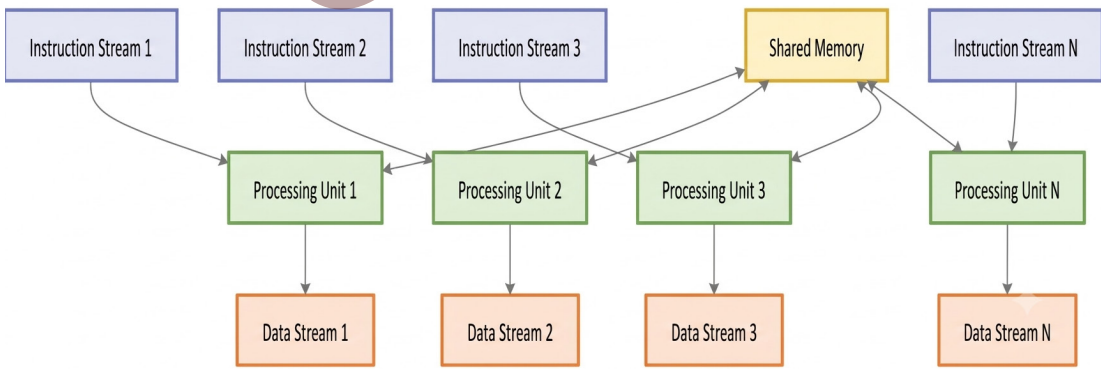


Fig. 1.3.5 Basic architecture of a Multiple Instruction, Multiple Data (MIMD) system

SGOU

Working Principle

Consider three independent tasks:

PE1: Sort a data set

PE2: Perform a matrix multiplication

PE3: Process an image

Each processing element can execute its own instructions and operate on its own data. Unlike SIMD, the processing elements do not have to execute the same instruction at the same time.

Characteristics

- Multiple instruction streams.
- Multiple data streams.
- Processing elements can execute independently.
- Supports task-level and thread-level parallelism.
- Processors may communicate and synchronize when required.
- Suitable for both regular and irregular computations.
- More flexible than SIMD architectures.

Shared-Memory MIMD

In a shared-memory MIMD system, multiple processors share a common memory space. Each processor can execute a different instruction stream while accessing shared data.

Examples include:

- Multi-core processors
- Symmetric multiprocessor systems
- Shared-memory servers

This connects directly to the shared-memory multiprocessor systems discussed in Unit 2.

Distributed-Memory MIMD

In a distributed-memory system, each processor or computing node has its own local memory. Processors communicate by exchanging messages over the network.

SGOU

Examples include:

- HPC clusters
- Beowulf clusters
- Distributed-memory supercomputers

MPI (Message Passing Interface) is commonly used for programming distributed-memory parallel applications.

MIMD and SPMD

SPMD (Single Program, Multiple Data) is a programming model commonly used on MIMD systems. In SPMD, multiple processors execute the same program but each processor may operate on different data and may follow different execution paths.

For example, four processors may execute the same program:

```
// Illustrative SPMD pseudocode
    for (int i = start; i < end; i++) {
       C[i] = A[i] + B[i];
    }
```

Each processor executes the same program, but start and end identify its assigned portion of the array. Therefore, each processor works on different data. In an actual parallel program, these ranges must be assigned so that the processors cover the required elements without unintended overlap.

Although the processors start with the same program, conditional statements, different data, synchronization, and communication can cause them to execute different instructions at different times.

Important Distinction: MIMD is an architectural classification, whereas SPMD is a programming model.

Advantages of MIMD

1. **High Flexibility:** Different processors can perform different operations simultaneously.
2. **Suitable for Complex Applications:** MIMD can handle applications containing different and irregular tasks.
3. **Task-Level Parallelism:** Independent tasks can be assigned to different processors.
4. **Scalability:** MIMD architectures can range from a few processor cores to large-scale HPC clusters.

SGOU

5. **Supports Different Memory Organizations:** MIMD systems can use

- Shared memory
- Distributed memory
- Hybrid memory architectures

Limitations of MIMD

1. **Programming Complexity:** Parallel programs require careful management of
   - Synchronization
   - Communication
   - Data dependencies
   - Load balancing
2. **Communication Overhead:** Processors may need to exchange data, which introduces additional execution time.
3. **Synchronization Overhead:** Processors may have to wait for one another at synchronization points.
4. **Load Imbalance:** If one processor receives significantly more work than others, some processors may remain idle.
5. **Limited Speedup:** The performance improvement is limited by serial portions of the program and parallel overheads, as described by Amdahl's Law.

Table 1.3.3 compares MIMD and SIMD architectures in terms of instruction streams, data streams, processing methods, and typical applications.

Table 1.3.3 MIMD vs SIMD

Feature	SIMD	MIMD
Instruction streams	One	Multiple
Data streams	Multiple	Multiple
Instructions	Same	Can be different
Data	Different	Different
Parallelism	Data-level	Task/thread-level
Control	Usually coordinated	Independent
Flexibility	Lower	Higher
Typical examples	SIMD units, array processors	Multi-core CPUs, clusters

SGOU

Solved Example 1

A parallel application contains four independent tasks:

- Task A requires 5 seconds.
- Task B requires 3 seconds.
- Task C requires 7 seconds.
- Task D requires 4 seconds.

Assume four processors are available and each task is assigned to a different processor.

Solution:

Sequential Execution

If executed sequentially:

$T_{serial}=5+3+7+4=19$ seconds

MIMD Execution

The four tasks execute concurrently.

The completion time is determined by the longest-running task:

$T_{parallel} = \max(5,3,7,4)$

$T_{parallel} = 7$ seconds

Therefore, the idealized speedup is:

$S=T_{serial} / T_{parallel} =19/7\approx2.71$

Thus, the system achieves an idealized speedup of approximately **2.71**×.

Note: Actual performance may be lower because of communication, synchronization, processor scheduling, and other overheads.

Applications of MIMD

MIMD architectures are widely used in:

- Multi-core personal computers
- Web and database servers
- Cloud computing
- High-performance computing
- Scientific simulations
- Artificial intelligence

SGOU

- Big-data processing
- Distributed applications
- Parallel search and optimization

MIMD in Flynn's Taxonomy

MIMD represents the most general category in Flynn's original four-way classification.

MIMD → Multiple Instructions + Multiple Data

It can be summarized as:

Multiple Instruction Streams

i.e., MIMD = Multiple Instructions + Multiple Data.

MIMD systems allow different processing elements to execute different instruction streams on different data. This flexibility makes MIMD the dominant classification for general-purpose multiprocessor and distributed parallel computing systems.

Think About It

A four-core processor is running four different applications simultaneously. Each core may execute different instructions and operate on different data.

Which category of Flynn's taxonomy best describes this system?

Answer: MIMD — Multiple Instruction, Multiple Data.

1.3.2 Comparison of SISD, SIMD, MISD and MIMD

Table 1.3.4 compares SISD, SIMD, MISD, and MIMD architectures based on the number of instruction and data streams, and typical examples.

Table 1.3.4 Comparison of SISD, SIMD, MISD and MIMD

Feature	SISD	SIMD	MISD	MIMD
Instr Streams	1	1	Multiple	Multiple
Data Streams	1	Multiple	1	Multiple
Examples	Single CPU	SIMD units, vector processors, AVX	fault-tolerant	Multi-core, cluster

1.3.3 Limitations of Flynn's Taxonomy

- **Coarse Classification:** Flynn's taxonomy classifies architectures based on instruction and data streams but does not describe memory organization, communication mechanisms, or synchronization in detail.

SGOU

- **Hybrid Architectures:** Modern GPUs use the SIMT (Single Instruction, Multiple Threads) execution model, in which groups of threads execute the same instruction when possible. Different Streaming Multiprocessors (SMs) can execute different thread blocks concurrently, making modern GPUs difficult to classify strictly under a single Flynn category.

- **Alternative Classifications:** Other models provide more detailed descriptions of parallel architectures. Feng's taxonomy considers bit-level and word-level parallelism, while Handler's taxonomy uses a three-dimensional model based on the number of processors (P), memory units (M), and control units (C).

Flynn's Taxonomy: Historical Perspective

Flynn's 1966 paper introduced four categories of computer architectures, including MISD for completeness. Practical systems exhibiting MISD-like characteristics, such as certain systolic arrays, became more prominent in later decades, although pure MISD implementations remain rare.

Flynn's Taxonomy in Your Laptop

A modern laptop can exhibit different forms of parallelism at different abstraction levels. It may execute a program sequentially (SISD-like behavior), use SIMD instructions such as AVX or NEON to process multiple data elements simultaneously, and employ MIMD parallelism when multiple CPU cores execute different instruction streams. Therefore, Flynn's taxonomy depends on the level of the system being considered.

Recap

- Flynn's taxonomy classifies computer architectures into four categories: SISD, SIMD, MISD, and MIMD.

- SISD (Single Instruction, Single Data) represents traditional sequential computing, such as a conventional von Neumann computer.

- SIMD (Single Instruction, Multiple Data) processes multiple data elements using the same instruction, as seen in array processors, vector processors, and AVX instruction extensions.

- MISD (Multiple Instruction, Single Data) applies multiple instruction streams to a single data stream; pure implementations are rare, although some fault-tolerant and systolic-array-like systems exhibit similar characteristics.

- MIMD (Multiple Instruction, Multiple Data) supports independent instruction and data streams and is widely used in multicore processors and computer clusters.

SGOU

- MIMD architectures are commonly classified into shared-memory and distributed-memory systems.
- SPMD (Single Program, Multiple Data) is a parallel programming model in which multiple processors execute the same program on different data; it is not a hardware classification in Flynn's taxonomy.
- Limitations of Flynn's taxonomy include its inability to describe memory organization and its limited representation of modern hybrid architectures.

Objective Type Questions

1. Who proposed Flynn's taxonomy of computer architectures?
2. Which category represents the traditional sequential von Neumann computer?
3. What does SIMD stand for?
4. Which category is the least commonly implemented in practical computer systems?
5. Which execution model is commonly associated with GPU warp-level parallelism?
6. What does SPMD stand for?
7. What are the two forms of parallelism represented by Feng's classification?
8. Which category of Flynn's taxonomy generally describes a multi-core processor at the system level?
9. What is the primary advantage of SIMD processing?
10. Which type of system is commonly associated with MISD-like processing?

SGOU

Answers

1. Michael J. Flynn.
2. SISD.
3. Single Instruction, Multiple Data.
4. MISD.

5. SIMT.
6. Single Program, Multiple Data.
7. Word parallelism and bit parallelism.
8. MIMD.
9. Data-level parallelism.
10. Fault-tolerant or redundant processing systems.

Assignments

1. Classify the following systems according to Flynn's taxonomy and justify your classification: a traditional single-core processor, a SIMD processor, a multi-core processor, and a distributed-memory cluster.
2. Compare SISD, SIMD, MISD, and MIMD in terms of instruction streams, data streams, parallelism, flexibility, and typical applications.
3. Consider an image-processing application in which the same operation must be performed on millions of pixels. Which Flynn category is most suitable? Explain your answer.
4. An array contains 1,024 independent data elements, and a SIMD architecture has 64 processing elements. Assuming one operation per element per cycle and ideal conditions, calculate the number of cycles required and the theoretical speedup over sequential execution.
5. Explain why MISD is considered the least commonly implemented category of Flynn's taxonomy. Give examples of applications or systems that exhibit MISD-like characteristics.
6. Distinguish between MIMD as an architectural classification and SPMD as a programming model. Explain why SPMD is commonly implemented on MIMD systems.
7. Discuss the limitations of Flynn's taxonomy when applied to modern hybrid architectures such as GPUs and multi-core systems. Explain why a modern system may exhibit characteristics of more than one category.

SGOU

Reference

1. Culler, D. E., Singh, J. P., & Gupta, A. (1998). *Parallel computer architecture: A hardware/software approach*. Morgan Kaufmann Publishers.

2. Duncan, R. (1990). A survey of parallel computer architectures. *Computer, 23*(2), 5–16.

3. Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers, 21*(9), 948–960.

4. Hennessy, J. L., Patterson, D. A., & Kozyrakis, C. (2025). *Computer architecture: A quantitative approach*. Morgan Kaufmann.

Suggested Reading

1. Feng, T. Y. (1972, August). Some characteristics of associative/parallel processing. In *Proceedings of the 1972 Sagamore Computing Conference* (pp. 5–16).

2. Hwu, W. M., & Kirk, D. (2009). *Programming massively parallel processors* (Special ed.).

SGOU



Parallel Processing Applications

Learning Outcomes

After completion of this unit, learner will be able to:

- identify key application domains for parallel computing
- analyze how parallel techniques are applied in science, AI, and big data
- explain computational characteristics for parallelization
- evaluate case studies of real-world parallel applications
- assess future trends in parallel computing

Prerequisite

Before studying this unit, learners should be familiar with the basic concepts of parallel processing, parallel computer structures, and architectural classifications introduced in the earlier units. The previous units explained how parallel computers are organized and how parallelism is classified. This unit moves from architecture to application by asking a practical question: What kinds of real-world problems benefit from parallel computing, and why?

Weather forecasting, artificial intelligence, molecular simulation, financial modelling, computer graphics, and big-data analytics involve enormous amounts of computation. Many of these problems would take impractical amounts of time on a single processor. Parallel computing makes it possible to divide such workloads among many processing elements and obtain results within a useful time.

This unit therefore connects architectural concepts with real-world applications and helps learners understand why a particular form of parallelism is suitable for a particular problem.

Keywords

Data Parallelism, Task Parallelism, Embarrassingly Parallel, Domain Decomposition, Functional Decomposition, Monte Carlo Simulation, Load Balancing, Strong and Weak Scaling

SGOU

Discussion

1.4.1 Application Domains

Parallel applications differ in the way their computations can be divided among processors. Some problems can be divided into almost completely independent tasks, while others require processors to communicate frequently. Understanding these characteristics helps in selecting an appropriate parallelization strategy.

Table 1.4.1 Parallel Computing Applications, Decomposition Strategies, and Resources

Domain	Examples	Parallelization Pattern	Typical Resources
Weather and Climate	Global climate models, hurricane prediction	3D domain decomposition	HPC clusters and supercomputers
CFD	Aerodynamics, ocean currents	Mesh-based solvers, sparse linear algebra	HPC clusters
Molecular Dynamics	Protein folding, drug design	N-body calculations, Particle interactions, spatial decomposition	CPU/GPU clusters
AI/ML	Deep neural networks, natural language processing	Matrix multiplication, data parallelism	GPU clusters
Bioinformatics	Genome sequencing	String matching, Hidden Markov Models (HMMs)	CPU/GPU clusters
Financial Computing	Risk analysis, option pricing	Monte Carlo simulation	CPU/GPU clusters
Computer Graphics	CGI, gaming	Ray tracing	GPU systems
Big Data Analytics	Search, recommendation systems	MapReduce	Distributed clusters

Data parallelism applies the same operation to multiple data elements at the same time. For example, an image-processing application can apply the same operation to many pixels simultaneously. Data parallelism is widely used in scientific computing, image processing, and machine learning.

Task parallelism divides an application into different tasks that can execute concurrently. The tasks may perform different operations, and each task can be assigned to a different processing unit. Task parallelism is useful when the tasks have sufficient independence.

SGOU

Embarrassingly parallel problems consist of independent tasks that require little or no communication between processors. Monte Carlo simulations are a common example because independent random trials can be distributed among processors and their results combined later.

Domain decomposition divides a large problem or data domain into smaller regions and assigns these regions to different processors. Weather and climate modeling is an important example. The atmosphere can be represented as a three-dimensional grid, with different processors responsible for different regions.

Functional decomposition divides a problem according to the functions or tasks that must be performed. For example, in a multimedia application, one processor may handle video processing while another handles audio processing. This approach is useful when different functions can proceed concurrently.

Table 1.4.2 Comparison of Parallelization Strategies

Strategy	Basic Idea	Communication	Example
Embarrassingly Parallel	Divide independent tasks among processors	Very low	Monte Carlo simulation
Domain Decomposition	Divide the problem or data space into regions	Usually required between related regions	Weather simulation
Functional Decomposition	Divide the application into different functions or tasks	Depends on task dependencies	Multimedia processing
Data Parallelism	Apply the same operation to different data elements	Usually low to moderate	Image processing
Task Parallelism	Execute different tasks concurrently	Depends on task dependencies	Independent application tasks

SGOU

1.4.1.1 Weather and Climate Modeling

Weather and climate modeling are important applications of parallel computing. They are used to understand atmospheric conditions, predict weather, and study long-term changes in the Earth's climate.

Weather modeling focuses mainly on short-term atmospheric conditions such as temperature, rainfall, wind speed, air pressure, and humidity. Weather forecasting systems use mathematical equations to represent the behavior of the atmosphere. These equations require a large number of calculations, especially when forecasts cover large geographical areas.

Climate modeling studies atmospheric and environmental conditions over much longer periods, often covering decades or centuries. Climate models are used to study phenomena such as global warming, changes in rainfall patterns, melting of ice, and rising sea levels.

The Earth's atmosphere is divided into a large number of small regions or grid cells. Mathematical calculations are performed for each cell to estimate atmospheric conditions. Since there may be millions of grid cells and many calculations must be repeated over time, the computational workload is very large.

Parallel computing helps by dividing these calculations among many processors. Different processors can work on different regions of the atmosphere at the same time. This significantly reduces the time required to produce forecasts and run climate simulations.

Weather and climate models are commonly executed on high-performance computing (HPC) systems and supercomputers. Increasing the number of processors and using more detailed grid resolutions can improve the accuracy and complexity of simulations, although it also increases computational requirements.

Applications include:

- Short-term weather forecasting.
- Prediction of severe storms and cyclones.
- Study of global temperature changes.
- Prediction of rainfall and drought conditions.
- Analysis of sea-level rise and melting ice.
- Planning for climate change and natural disasters.

Thus, weather and climate modeling is a good example of how parallel computing can solve large and complex scientific problems efficiently. By performing many calculations simultaneously, parallel computers enable faster and more detailed atmospheric and climate simulations.

1.4.1.2 AI and Machine Learning

Artificial Intelligence (AI) is a field of computing that enables computers to perform tasks that normally require human intelligence. These tasks include recognizing images, understanding language, making decisions, and solving problems.

Machine Learning (ML) is a major part of AI. It allows computers to learn patterns from data and make predictions or decisions without being explicitly programmed for every situation.

AI and machine learning often require processing very large amounts of data. Training a machine learning model involves performing many mathematical operations repeatedly. For example, neural networks may contain millions or even billions of parameters that must be processed during training.

Parallel computing makes AI and machine learning faster by dividing these calculations among multiple processing units. Graphics Processing Units (GPUs) are widely used

SGOU

because they can perform many similar mathematical operations simultaneously. CPUs and other specialized processors can also be used for parallel AI workloads.

For example, when training an image-recognition model, thousands of images can be processed in batches. Different parts of the computation can be performed at the same time, reducing the overall training time.

Applications of AI and machine learning include:

1. Image and face recognition.
2. Speech recognition and language processing.
3. Recommendation systems.
4. Autonomous vehicles.
5. Medical image analysis.
6. Fraud detection.
7. Generative AI and chatbots.

Parallel computing is therefore an important technology for modern AI and machine learning. It allows large datasets and complex models to be processed efficiently and helps reduce the time required for model training and prediction.

1.4.1.3 Big Data Analytics

Big Data Analytics is the process of examining very large amounts of data to identify useful patterns, relationships, and information. The data may come from sources such as social media, websites, sensors, business transactions, and mobile devices.

Big data applications often involve processing millions or billions of records. Performing these operations on a single computer can take a long time. Parallel computing helps by dividing the data and processing different portions simultaneously on multiple processors or computers.

For example, an online shopping company may analyze millions of customer transactions to identify popular products and understand customer preferences. Different processors can analyze different groups of transactions at the same time, allowing the results to be obtained much faster.

Parallel processing is also useful for operations such as data searching, sorting, filtering, classification, and statistical analysis. Distributed computing systems can divide large datasets across multiple computers and combine their results after processing.

Applications of Big Data Analytics include:

1. Customer behavior analysis.
2. Business and sales forecasting.

SGOU

3. Fraud and unusual-activity detection.
4. Social media analysis.
5. Healthcare data analysis.
6. Traffic and transportation analysis.
7. Scientific and research data processing.

Big Data Analytics demonstrates the importance of parallel computing in handling large-scale data. By processing data simultaneously across multiple processors or computers, organizations can analyze large datasets more quickly and make better decisions.

1.4.1.4 Molecular Dynamics

Molecular Dynamics (MD) is a computer simulation technique used to study the movement and interaction of atoms and molecules. It is widely used in physics, chemistry, biology, and materials science.

In a molecular dynamics simulation, a system is represented by many atoms. The computer calculates the forces acting on these atoms and uses them to determine how their positions and velocities change over small time intervals. These calculations are repeated many times to study the behavior of the system.

Molecular dynamics can involve thousands or millions of atoms, making the simulations computationally demanding. Parallel computing helps by dividing the calculations among multiple processors. Different processors can calculate the interactions or movements of different groups of atoms at the same time.

For example, molecular dynamics can be used to study how a protein changes its shape, how a drug interacts with a target molecule, or how a material behaves under different conditions.

Applications of Molecular Dynamics include:

1. Studying protein and molecular structures.
2. Drug discovery and molecular interactions.
3. Designing new materials.
4. Studying chemical reactions.
5. Understanding the behavior of liquids and gases.
6. Simulating biological systems.
7. Studying properties of nanomaterials.

Parallel computing allows molecular dynamics simulations to handle larger systems and run for longer periods in less time. This makes it possible to study complex molecular behavior that would be difficult to analyze using a single processor.

SGOU

Thus, Molecular Dynamics is an important scientific application of parallel computing, where large numbers of calculations are performed repeatedly and simultaneously.

1.4.1.5 Computer Graphics

Computer Graphics is the field of computing concerned with creating, displaying, and processing images and visual content using computers. It is used in applications such as games, movies, simulations, engineering, and virtual reality.

Creating a detailed image requires many calculations. For example, each pixel in an image may require calculations for color, lighting, shadows, texture, and the position of objects. When images contain millions of pixels, processing them sequentially can take a significant amount of time.

Parallel computing improves computer graphics by allowing many pixels or objects to be processed simultaneously. Graphics Processing Units (GPUs) are specifically designed to perform large numbers of similar operations in parallel.

For example, in a 3D computer game, different processors or GPU cores can simultaneously calculate the appearance of different pixels and objects in a scene. This allows images and animations to be generated quickly.

Applications of Computer Graphics include:

1. Video games and interactive applications.
2. Movies and animation.
3. 3D modeling and visualization.
4. Virtual and augmented reality.
5. Computer-aided design (CAD).
6. Scientific and medical visualization.
7. Image processing and rendering.

Parallel computing is therefore essential for modern computer graphics. It enables complex images, animations, and 3D environments to be rendered quickly and with high visual quality.

1.4.1.6 Financial Computing

Financial Computing is the use of computers and computational methods to solve problems related to banking, finance, investment, and financial markets. It involves processing large amounts of financial data and performing complex calculations.

Financial applications often require many calculations to be performed repeatedly. For example, financial institutions may need to analyze market data, calculate risks, evaluate investments, and predict possible outcomes. These tasks can become computationally expensive when large datasets and many possible scenarios are involved.

SGOU

Parallel computing helps by dividing financial calculations among multiple processors. Different processors can analyze different datasets or perform different simulation scenarios at the same time. This reduces processing time and allows financial decisions to be made more quickly.

For example, risk analysis may involve running thousands of possible market scenarios to estimate potential losses. These simulations can be executed simultaneously using multiple processors.

Applications of Financial Computing include:

1. Risk analysis and management.
2. Stock market analysis.
3. Investment and portfolio optimization.
4. Fraud detection.
5. Financial forecasting.
6. Algorithmic trading.
7. Monte Carlo simulations.

Parallel computing is particularly useful when a financial problem requires a large number of independent calculations. By performing these calculations simultaneously, financial systems can process data faster and respond more efficiently to changing market conditions.

Thus, Financial Computing is an important application of parallel computing, particularly in areas that require large-scale data processing, simulation, and optimization.

1.4.2 Scaling and Load Balancing

Strong scaling measures how the execution time of a fixed-size problem changes as more processors are added. Ideally, adding processors reduces execution time.

For example, if the same weather simulation is executed using 4, 8, and 16 processors, the change in execution time can be used to evaluate strong scaling.

Weak scaling measures how well a system performs when the problem size increases along with the number of processors. Ideally, if the workload assigned to each processor remains approximately constant, execution time should remain relatively stable.

Load balancing is the process of distributing computational work so that processors have approximately equal workloads. Poor load balancing can cause some processors to finish early and remain idle while others continue working.

SGOU

1.4.3 Challenges

1. **Amdahl's bottleneck:** Serial portions limit scalability.
2. **Communication overhead:** Data movement dominates.
3. **Load imbalance:** Uneven work distribution.
4. **Debugging:** Race conditions, deadlocks — hard to reproduce.
5. **Portability:** Architecture-specific optimizations.

Solved Example 1:

P1=10s, P2=10s, P3=15s, P4=5s.

Speedup = 40/15 = 2.67x.

Efficiency = 2.67/4 = 66.7% (load imbalance).

The execution time is determined by the processor with the largest workload (15 s), so the other processors remain idle for part of the computation. This illustrates the effect of load imbalance on parallel efficiency.

Human Genome Project

Celera Genomics used commodity PC clusters vs public supercomputer — demonstrated commodity parallel computing.

CUDA (2006)

Helped make GPU computing accessible for general-purpose computation. By providing a programming platform for GPU-based computation, CUDA contributed significantly to the growth of GPU-accelerated applications, including deep learning.

SGOU

Recap

- Parallel applications: Parallel computing is used in weather and climate modeling, computational fluid dynamics, artificial intelligence, big-data analytics, molecular dynamics, bioinformatics, computer graphics, and financial computing.
- Data parallelism: The same operation can be applied concurrently to multiple data elements, making it important in applications such as matrix computation, image processing, and machine learning.
- Task parallelism: Different tasks can execute concurrently when they have sufficient independence, allowing applications to exploit multiple processing units.

- Embarrassingly parallel problems: Independent tasks requiring little or no communication are particularly suitable for parallel execution and can often achieve good scalability.
- Domain decomposition: Large spatial or numerical domains can be divided into smaller regions and assigned to different processors. Weather and climate simulations are important examples.
- Functional decomposition: An application can be divided into different functions or tasks that execute concurrently when their dependencies permit.
- Monte Carlo simulation: Repeated random trials can be distributed among processors, making Monte Carlo methods suitable for applications such as financial risk analysis and numerical estimation.
- Load balancing: Effective parallel execution requires computational work to be distributed appropriately so that processors do not remain idle while others are overloaded.
- Scalability: Strong scaling evaluates performance for a fixed problem size as more processors are added, whereas weak scaling evaluates performance when the problem size increases with the number of processors.
- Application challenges: Communication overhead, synchronization, load imbalance, memory access, and scalability can limit the performance of parallel applications.

Objective Type Questions

1. What is the technique of partitioning a spatial domain among multiple processors called?
2. Which framework follows the Map, Shuffle/Sort, and Reduce processing stages?
3. Which type of simulation commonly uses random sampling and is naturally parallel?
4. What is the technique of splitting a training dataset across multiple GPUs called?
5. What problem occurs when work is distributed unevenly among processors, causing some processors to remain idle?
6. Which volunteer computing project is widely known for large-scale protein-folding simulations?

SGOU

7. What does NWP stand for?

8. What type of scaling keeps the problem size per processor approximately constant as processors are added?

9. Which NVIDIA platform, introduced in 2006, enabled general-purpose GPU computing using CUDA?

10. What communication operation is commonly required between neighboring subdomains in a domain-decomposed weather simulation?

Answers

1. Domain decomposition.

2. MapReduce.

3. Monte Carlo simulation.

4. Data parallelism.

5. Load imbalance.

6. Folding@home.

7. Numerical Weather Prediction.

8. Weak scaling.

9. CUDA.

10. Halo exchange.

SGOU

Assignments

1. Case Study: Select one parallel-computing application from weather forecasting, computational fluid dynamics, artificial intelligence, bioinformatics, finance, graphics, or big-data analytics. Describe the problem, parallelization strategy, computational resources, and benefits obtained.

2. Problem Solving: A parallel application takes 40 seconds to execute sequentially and 15 seconds when executed on four processors. Calculate the speedup and parallel efficiency. Explain why the efficiency is less than 100%.

3. Programming: Implement a parallel Monte Carlo algorithm for estimating π using threads. Execute the program using 1, 2, 4, and 8 threads and compare execution time, speedup, and efficiency.

4. Application Analysis: Explain how domain decomposition can be used in weather and climate modeling. Describe how neighbouring processors exchange boundary information during a simulation.

5. Comparative Analysis: Compare data parallelism, model parallelism, and pipeline parallelism in deep-learning workloads. Explain when each approach is useful.

6. Analytical Thinking: Explain why MapReduce is suitable for large-scale data analytics. Describe the roles of the Map, Shuffle/Sort, and Reduce stages using a suitable example.

7. Problem Solving: A workload contains 80% parallelizable computation and 20% serial computation. Calculate the theoretical speedup using 2, 4, 8, and 16 processors using Amdahl's Law. Discuss the scalability limit.

Reference

1. Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to Parallel Computing* (2nd ed.). Addison-Wesley.

2. Pacheco, P. (2011). *An introduction to parallel programming*. Elsevier.

3. Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107-113.

4. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436-444.

5. Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE*, *96*(5), 879-899.

Suggested Reading

1. Karniadakis, G. E., & Kirby, I. I. RM 2003. Parallel Scientific Computing in C++ and MPI.

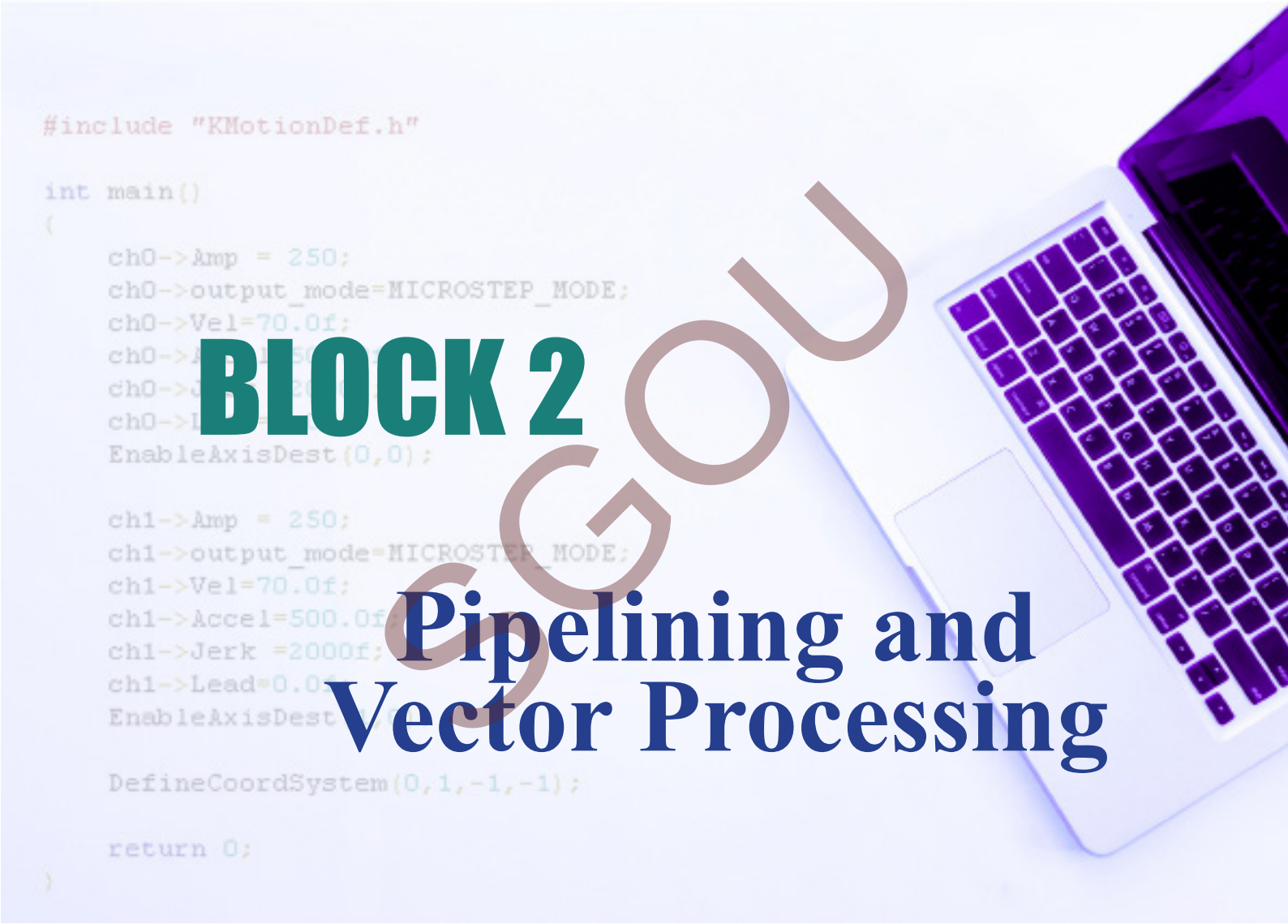
2. Mattson, T. G., Sanders, B., & Massingill, B. (2004). *Patterns for parallel programming*. Pearson Education.

SGOU

3. Hwu, W. M., & Kirk, D. (2009). Programming massively parallel processors. *Special Edition*, *92*, 157.

4. Pacheco, P. (2011). *An Introduction to Parallel Programming*. Morgan Kaufmann.

SGOU



BLOCK 2

Pipelining and Vector Processing

SGOU



Principles of Pipeline Processors

Learning Outcomes

After completion of this unit, learner will be able to:

- explain the concept and working principle of pipelining in computer processors
- describe the basic stages of a processor pipeline and explain the role of pipeline registers
- analyze the advantages and limitations of pipelined execution in terms of throughput and execution time
- identify and analyze structural, data, and control hazards in a pipelined processor
- explain common techniques used to reduce the effects of pipeline hazards, including stalling, forwarding, and basic branch-handling techniques

Prerequisite

Before studying pipeline processors, learners should be familiar with the basic organization of a processor and the steps involved in executing an instruction. In a conventional processor, an instruction is fetched from memory, decoded, executed, and eventually produces a result. If each instruction has to complete all these steps before the next instruction begins, a significant amount of processor hardware may remain idle during parts of the instruction cycle.

Consider a simple example: while one instruction is being executed by the arithmetic unit, the instruction-fetch hardware may have nothing to do. Pipelining addresses this inefficiency by allowing different instructions to occupy different stages of execution at the same time.

Thus, instead of asking "How can one instruction be completed faster?", pipeline design asks an equally important question:

"How can the processor keep all of its functional stages busy?"

SGOU

This idea forms the foundation for understanding modern high-performance processors.

Illustrations for Capturing the Background

Think of baking cookies. In a sequential kitchen: you mix dough, shape cookies, bake one tray, cool it, then start the next tray. Each batch takes 30 minutes. In a pipelined kitchen: while the first tray bakes, you shape the second batch. While the first tray cools, the second bakes and you mix dough for the third. After the initial setup, a new tray finishes every 10 minutes instead of every 30. This temporal overlap is the essence of pipelining.

Keywords

Pipelining, Pipeline Stage, Pipeline Register, Pipeline Fill, Pipeline Stall, Pipeline Hazard, Pipeline Flush

Discussion

2.1.1 Introduction to Pipelining

A processor executes an instruction through a sequence of operations. In a simplified processor, these operations may include:

1. Fetching the instruction from memory.
2. Decoding the instruction and reading its operands.
3. Performing the required operation.
4. Accessing memory when necessary.
5. Writing the result back to a register.

If these operations are performed sequentially for every instruction, the processor must wait for the current instruction to complete before starting the next instruction.

Pipelining divides instruction execution into multiple stages so that several instructions can be processed simultaneously, with each instruction occupying a different stage.

The concept is similar to an industrial assembly line. In an assembly line, different workers perform different tasks on different products at the same time. One product may be undergoing painting while another is being assembled and another is being inspected.

Similarly, in a processor pipeline:

- one instruction may be fetched;
- another may be decoded;

SGOU

- another may be executing;
- another may be accessing memory; and
- another may be writing its result.

The instructions are therefore overlapped, even though each individual instruction still passes through the complete sequence of stages.

Key Idea

Pipelining improves instruction throughput by overlapping instruction execution; it does not necessarily reduce the latency of an individual instruction.

2.1.2 Basic Pipeline Stages

A commonly used simplified processor pipeline consists of five stages:

1. Instruction Fetch (IF)
2. Instruction Decode / Register Read (ID)
3. Execute (EX)
4. Memory Access (MEM)
5. Write Back (WB)

See fig. 2.1.1.

2.1.2.1 Instruction Fetch (IF)

During the Instruction Fetch stage, the processor obtains the instruction from instruction memory or the instruction cache.

The program counter (PC) identifies the address of the instruction to be fetched. After the instruction is fetched, the PC is normally updated to point to the next instruction.

2.1.2.2 Instruction Decode / Register Read (ID)

During the Instruction Decode stage, the processor determines what operation the instruction represents.

The processor may also:

- decode the opcode;
- identify source and destination registers;
- read operands from the register file;
- generate control signals.

For a simple RISC-style pipeline, register operands are normally read during this stage.

SGOU

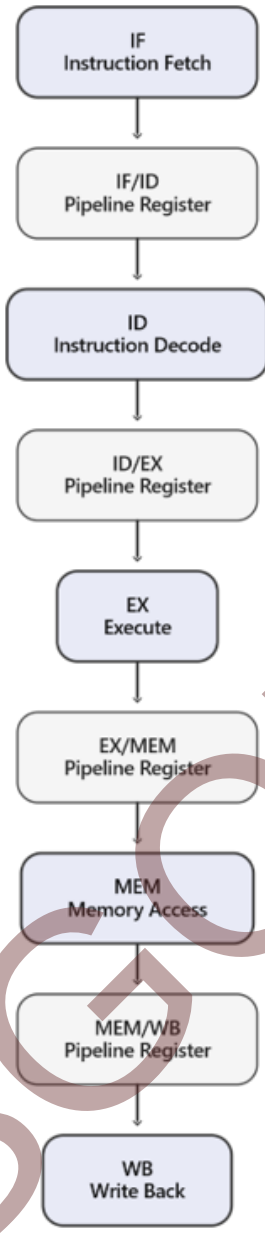


Fig. 2.1.1 Basic Five-Stage Instruction Pipeline

2.1.2.3 Execute (EX)

During the Execute stage, the required computational operation is performed.

Depending on the instruction, this stage may involve:

- arithmetic or logical operations;
- comparison;
- address calculation;
- branch-condition evaluation.

SGOU

For load and store instructions, the effective memory address is commonly calculated during this stage in a classic five-stage pipeline.

2.1.2.4 Memory Access (MEM)

The Memory Access stage is used when the instruction requires access to data memory.

For example:

- a **load** instruction obtains data from memory;
- a **store** instruction writes data to memory.

For arithmetic instructions that do not access data memory, this stage may simply pass the result through the pipeline.

2.1.2.5 Write Back (WB)

During the Write Back stage, the result produced by the instruction is written into the destination register when required.

For example, after an arithmetic instruction computes a result, the result may be written into a register during the WB stage.

2.1.3 Pipeline Registers

Pipeline stages operate concurrently, so intermediate information must be preserved as an instruction moves from one stage to the next.

Pipeline registers are storage elements placed between pipeline stages. They hold:

- intermediate data;
- instruction information;
- register values;
- control signals;
- computed addresses or results.

For example:

IF/ID → ID/EX → EX/MEM → MEM/WB

Each pipeline register transfers the required information to the next stage at the appropriate clock edge.

Why are pipeline registers necessary?

Without pipeline registers, the output of one stage could not be reliably separated from the processing of the next instruction. Pipeline registers therefore provide the boundaries needed for concurrent execution.

SGOU

2.1.4 Pipeline Timing

Consider four instructions:

- I1
- I2
- I3
- I4

in a five-stage pipeline.

The instructions can be scheduled as follows:

Table 2.1.1 Timing of Instructions in a Five-Stage Pipeline

Cycle	I1	I2	I3	I4
1	IF			
2	ID	IF		
3	EX	ID	IF	
4	MEM	EX	ID	IF
5	WB	MEM	EX	ID
6		WB	MEM	EX
7			WB	MEM
8				WB

The timing diagram illustrates how instructions overlap as they move through the pipeline. The first instruction must pass through all five stages before completing, while subsequent instructions enter the pipeline in successive clock cycles. Once the pipeline is filled, multiple instructions are being processed simultaneously.

2.1.5 Pipeline Fill and Throughput

When a pipeline starts, its stages become occupied gradually as instructions enter. This initial period is called pipeline filling. Once the pipeline is filled, different instructions occupy different stages simultaneously.

Under ideal conditions, a five-stage pipeline can complete one instruction per clock cycle after the initial filling period. In practice, hazards, stalls, and unequal stage delays may reduce throughput.

Detailed calculations of pipeline timing, speedup, and performance are discussed in Unit 2: Instruction and Arithmetic Pipelines.

2.1.6 Advantages of Pipelining

Pipelining provides several important advantages.

SGOU

Increased Throughput

The primary advantage is increased instruction throughput. Once the pipeline is full, multiple instructions are being processed simultaneously.

Better Hardware Utilization

Different functional units can operate concurrently. Instead of waiting for an entire instruction to finish, each stage can work on another instruction.

Higher Instruction Processing Rate

A well-designed pipeline can complete instructions at a much higher rate than a non-overlapped processor.

Support for High-Performance Processor Design

Pipelining is a fundamental technique used in modern processor architectures to achieve high instruction throughput.

2.1.7 Limitations of Pipelining

Pipelining is not free from limitations.

Pipeline Fill and Drain

At the beginning and end of execution, not all stages are active. Therefore, short instruction sequences may not achieve the ideal throughput.

Pipeline Imbalance

If one stage takes considerably longer than the others, the clock period must accommodate the slowest stage.

Pipeline Register Overhead

Registers between stages introduce storage and timing overhead.

Pipeline Hazards

Dependencies and resource conflicts may prevent instructions from progressing in every cycle.

Increased Design Complexity

As the number of stages increases, maintaining correct operation becomes more complicated.

2.1.8 Pipeline Hazards

A pipeline hazard is a condition that prevents an instruction from executing in the intended clock cycle.

SGOU

Hazards are generally classified into three categories:

1. Structural hazards
2. Data hazards
3. Control hazards

See fig. 2.1.2.

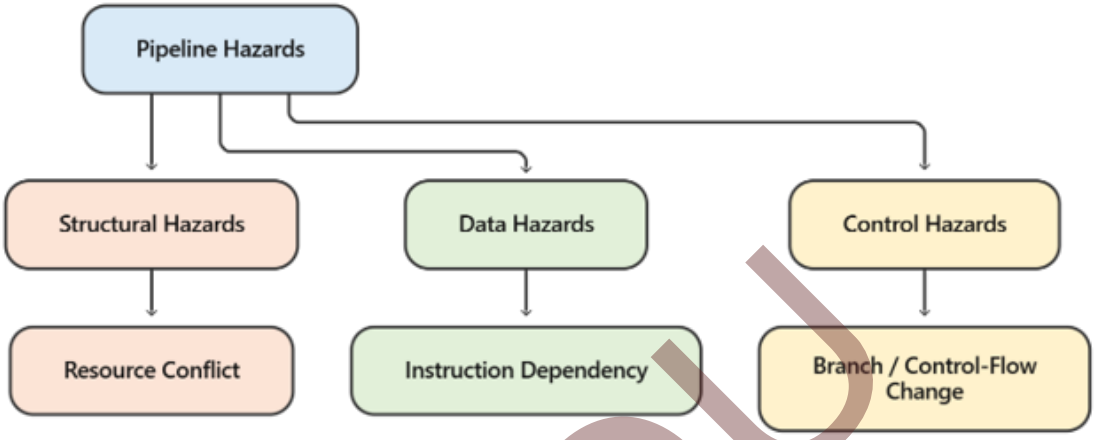


Fig. 2.1.2 Classification of Pipeline Hazards

2.1.9 Structural Hazards

A structural hazard occurs when two or more instructions require the same hardware resource during the same clock cycle, but the processor does not have sufficient resources to satisfy all requests simultaneously.

Example

Suppose a processor has a single memory that is used for both:

- instruction fetching; and
- data access.

Consider a situation where:

- one instruction needs to fetch the next instruction; and
- another instruction needs to access data memory.

If both operations occur in the same cycle and only one memory access is possible, a conflict occurs.

The processor must delay one of the operations. See fig.2.1.3.

SGOU

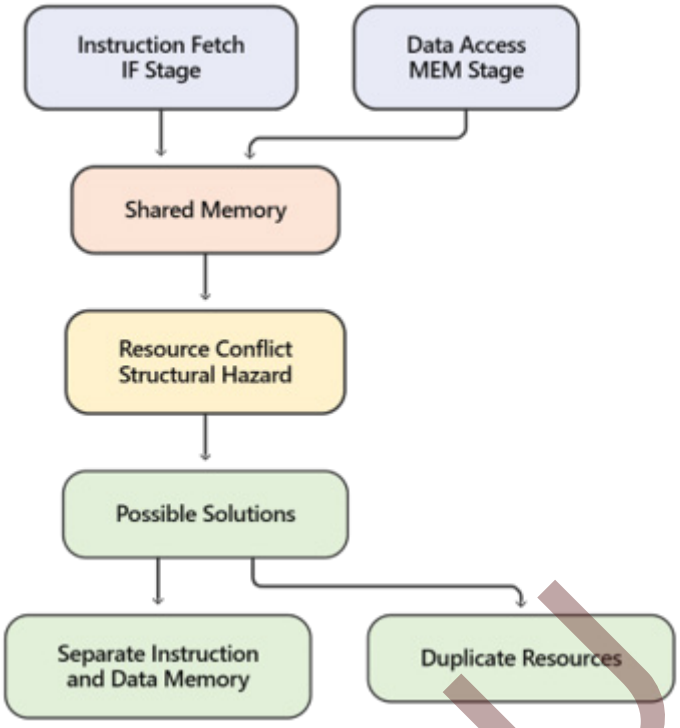


Fig. 2.1.3 Structural Hazard Caused by Resource Conflict

Methods of Reducing Structural Hazards

Structural hazards can be reduced by:

- providing separate instruction and data caches;
- duplicating frequently used hardware resources;
- providing multiple functional units;

2.1.10 Data Hazards

A data hazard occurs when an instruction depends on data produced or modified by another instruction that has not yet completed the required operation.

Consider:

I1: ADD R1, R2, R3

I2: SUB R4, R1, R5

Instruction I2 requires the value of R1 produced by I1.

In a pipeline, I2 may reach the Execute stage before I1 has written the result to the register file.

This creates a dependency.

SGOU

2.1.11 Types of Data Dependencies

Three classical types of data dependencies are:

Read After Write (RAW)

A later instruction reads a value after an earlier instruction writes it.

I1: ADD R1, R2, R3

I2: SUB R4, R1, R5

I2 must obtain the value produced by I1.

RAW is the most important data hazard in a simple in-order pipeline.

Write After Read (WAR)

A later instruction writes to a location before an earlier instruction has completed reading it.

WAR hazards are generally associated with processors that allow instructions to execute out of order. They do not normally occur in a simple in-order five-stage pipeline.

Write After Write (WAW)

Two instructions write to the same destination, and the later instruction must not write its result before the earlier instruction.

WAW hazards are also primarily associated with more advanced pipelines that permit out-of-order execution.

2.1.12 Data Hazard Mitigation

Several techniques can reduce the performance impact of data hazards.

2.1.12.1 Pipeline Stalling

A stall temporarily prevents an instruction from advancing until the required data becomes available. A stall introduces one or more idle cycles into the pipeline. These idle cycles are sometimes called bubbles. See fig. 2.1.4.

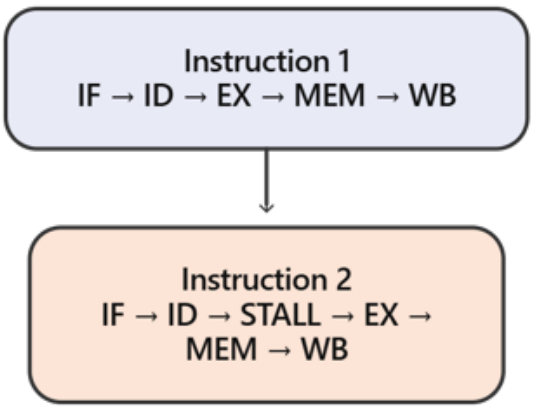


Fig. 2.1.4 Pipeline Stall

SGOU

2.1.12.2 Forwarding

Forwarding, also called bypassing, allows a result to be transferred directly from an intermediate pipeline stage to another stage that needs it, without waiting for the result to be written back to the register file.

For example:

I1: ADD R1, R2, R3

I2: SUB R4, R1, R5

Instead of waiting for I1 to reach WB and then reading R1 from the register file, the processor can forward the result of I1 to the appropriate input of the EX stage for I2.

Forwarding can therefore eliminate many RAW stalls.

2.1.13 Control Hazards

A control hazard occurs when the next instruction to be fetched depends on the outcome of a branch or other control-flow instruction.

Consider:

BEQ R1, R2, TARGET

ADD R3, R4, R5

Until the branch condition is evaluated, the processor may not know whether the next sequential instruction should be executed or whether control should transfer to TARGET.

The pipeline may therefore fetch instructions that later turn out to be incorrect.See fig. 2.1.5.

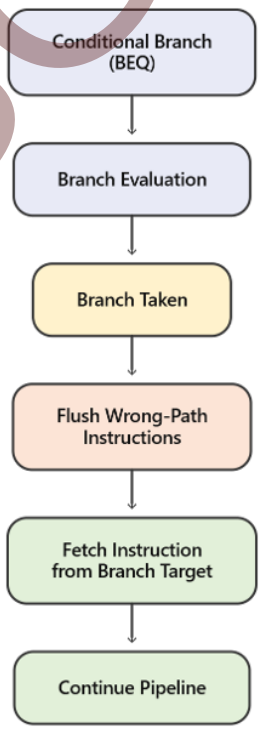


Fig. 2.1.5 Control Hazard and Pipeline Flush

SGOU

2.1.13.1 Effect of a Branch

Suppose the processor fetches instructions sequentially while a branch is being evaluated. If the branch is taken, instructions fetched after the branch may have to be discarded. This process is called a pipeline flush.

2.1.14 Basic Techniques for Handling Control Hazards

2.1.14.1 Stalling

The processor can temporarily stop fetching or advancing instructions until the branch outcome is known.

The disadvantage is the loss of pipeline cycles.

2.1.14.2 Static Branch Handling

A processor may use a simple predetermined rule, such as:

- predict branch not taken;
- predict branch taken.

If the prediction is wrong, the incorrectly fetched instructions are flushed.

2.1.14.3 Dynamic Branch Prediction

A processor can use information about previous branch behavior to make a prediction.

At this stage, it is sufficient to understand the basic principle:

The processor predicts the likely branch direction so that instruction fetching can continue without waiting for the branch decision.

2.1.15 Comparison of Pipeline Hazards

Table 2.1.2 Types of Pipeline Hazards

Hazard	Basic Cause	Example	Typical Solution
Structural	Insufficient hardware resource	Instruction fetch and data access require the same memory	Duplicate/separate resources
Data	Dependency between instructions	ADD R1, followed by instruction using R1	Forwarding, stalling
Control	Uncertainty caused by branch/control instruction	Conditional branch	Stalling, prediction, flushing

SGOU

Pipelining and Assembly Lines

An assembly line does not make an individual product pass through every manufacturing step instantaneously. Instead, it allows different products to occupy different stages at the same time. Processor pipelining follows the same principle: overlapping different instructions allows the processor to complete instructions at a higher rate.

Deep Pipelines

Some commercial processors have used much deeper pipelines than the simple five-stage model presented in this unit. Deeper pipelines can support higher clock frequencies by dividing instruction processing into smaller stages. However, deeper pipelines also increase the cost of pipeline stalls and branch mispredictions.

Recap

- Pipelining divides instruction execution into stages and allows multiple instructions to be processed concurrently.
- A typical five-stage pipeline consists of IF, ID, EX, MEM, and WB.
- Pipeline registers hold intermediate data and control information between stages.
- Pipelining primarily improves instruction throughput, rather than necessarily reducing the latency of an individual instruction.
- Under ideal conditions, a filled pipeline can approach a throughput of one completed instruction per clock cycle.
- Structural hazards occur when competing instructions require the same hardware resource.
- Data hazards occur because of dependencies between instructions.
- RAW is the principal data hazard in a simple in-order pipeline.
- WAR and WAW hazards are generally associated with more advanced pipelines that permit out-of-order execution.
- Forwarding can reduce many RAW hazards by transferring results directly between pipeline stages.
- Control hazards occur when the next instruction depends on the outcome of a branch.

SGOU

- Stalling and flushing help maintain correct execution in the presence of hazards.
- Basic branch prediction can reduce control-hazard penalties; advanced prediction techniques are considered in later pipeline-design topics.
- Effective pipeline operation depends on proper stage organization and minimizing stalls, resource conflicts, and control-flow disruptions.

Objective Type Questions

1. What is pipelining in a computer processor?
2. Which five stages are commonly used in a basic instruction pipeline?
3. What is the purpose of a pipeline register?
4. Which pipeline stage performs the main arithmetic or logical operation in a classic five-stage pipeline?
5. What type of hazard occurs when two instructions require the same hardware resource at the same time?
6. Which type of data hazard occurs when an instruction reads a value produced by an earlier instruction?
7. What technique transfers a result directly between pipeline stages to reduce data-hazard stalls?
8. What type of hazard is caused by a conditional branch?
9. What is a pipeline stall?
10. What happens to incorrectly fetched instructions when a branch prediction is found to be incorrect?

SGOU

Answers

1. Overlapped instruction execution.
2. IF, ID, EX, MEM, and WB.
3. To hold intermediate data and control information.
4. Execute (EX).

5. Structural hazard.
6. Read After Write (RAW).
7. Forwarding.
8. Control hazard.
9. A temporary delay in pipeline execution.
10. They are flushed.

Assignments

1. Why does pipelining improve instruction throughput?
2. How does a pipeline register help coordinate adjacent pipeline stages?
3. What is the difference between a pipeline stall and a pipeline flush?
4. Why is RAW the most important data hazard in a simple in-order five-stage pipeline?
5. How does forwarding reduce the performance impact of data hazards?
6. Why do branch instructions create control hazards?
7. Why can pipeline hazards prevent a processor from achieving its ideal instruction throughput?

SGOU

Reference

1. Hennessy, J. L., Patterson, D. A., & Kozyrakis, C. (2025). *Computer architecture: a quantitative approach*. Morgan kaufmann.
2. Patterson, D. A., & Hennessy, J. L. (2017). Computer organization and design RISC-V edition: The hardware/software interface. Morgan Kaufmann.
3. Stallings, W. (2016). *Computer organization and architecture: designing for performance*. Pearson Education India.
4. Hamacher, V. C. (2012). *Computer organization and embedded systems/ Carl Hamacher...[et al.]*. New York, NY: McGraw-Hill,.

Suggested Reading

1. Harris, S., & Harris, D. (2021). *Digital design and computer architecture, RISC-V edition*. Morgan Kaufmann.

2. Sima, D., & Kacsuk, P. (1997). *Advanced computer architectures*. Addison-Wesley Longman Publishing Co., Inc..

3. Shen, J. P., & Lipasti, M. H. (2013). Modern processor design: Fundamentals of superscalar processors. Waveland Press.

4. Null, L., & Lobur, J. (2018). Essentials of computer organization and architecture (5th ed.). Jones & Bartlett Learning.

SGOU



Instruction and Arithmetic Pipelines

Learning Outcomes

After completion of this unit, learner will be able to:

- trace the complete instruction pipeline stages from Fetch to Write-back with timing diagrams
- explain the concept of arithmetic pipelining and how complex operations are decomposed into pipeline stages
- calculate pipeline performance metrics: CPI, speedup, throughput, and efficiency
- design and analyze arithmetic pipeline examples, particularly floating-point addition
- analyze how pipeline stage delays, clock cycle time, and stalls affect pipeline performance

Prerequisite

The previous unit introduced the abstract concept of pipelining and hazard classification. This unit dives into the operational details: exactly how data and control flow through each stage, how timing diagrams capture pipeline behavior, and how arithmetic operations — much more complex than simple ALU operations — benefit from pipelining.

Illustrations for Capturing the Background

Picture a restaurant kitchen during a dinner rush:

- **Station 1 (Prep):** Ingredients are gathered and measured.
- **Station 2 (Cook):** Food is cooked on the stovetop or in the oven.
- **Station 3 (Plate):** The dish is arranged on the plate.
- **Station 4 (Garnish/Serve):** Final garnishes are added and the dish is sent to the table.

SGOU

A complex dish like a soufflé requires precise timing through each stage. The arithmetic pipeline works similarly — a complex operation like floating-point addition goes through Compare Exponents, Align Mantissas, Add Mantissas, and Normalize Result — each handled by dedicated hardware in sequence.

Keywords

Instruction Pipeline Diagram, Pipeline Timing, Arithmetic Pipeline, Latency, Initiation Rate, Floating-Point Unit (FPU), Mantissa Alignment, Normalization, Pipeline Efficiency

Discussion

2.2.1 Instruction Pipeline Stages

Let us trace the full journey of a MIPS-style instruction through the five pipeline stages with precise detail about what happens at each step.

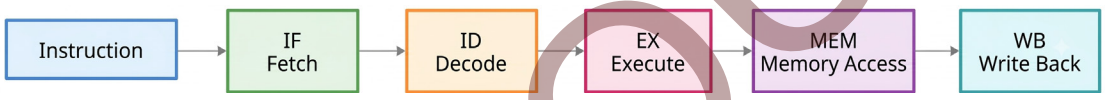


Fig. 2.2.1 Basic five-stage instruction pipeline showing the flow of an instruction from Fetch to Write-back

Stage 1 — Instruction Fetch (IF):

```
Input: PC (Program
Counter) Action: Memory[PC]
→ Instruction
            PC ← PC + 4 (prepare for next instruction)
Output: IF/ID Pipeline Register ← { Instruction, PC+4 }
```

The instruction memory is read at the address given by the PC. Simultaneously, the PC is incremented by 4 (each instruction is 4 bytes). The fetched 32-bit instruction and the incremented PC are stored in the IF/ID pipeline register for the next stage.

Stage 2 — Instruction Decode (ID):

```
Input: IF/ID Pipeline Register
       Action: Decode instruction fields (opcode, rs, rt,
      rd, immediate, funct) Read Register[rs] → Value A
             Read Register[rt] → Value B (or sign-
      extend immediate) Generate control signals
      for EX, MEM, WB stages
Output: ID/EX Pipeline Register ← { Control Signals, Value A, Value B,
                                                    sign-extended
```

SGOU

The 32-bit instruction is decoded into its fields. Source registers are read, and control signals are generated for the later EX, MEM, and WB stages. During the Decode stage, the instruction is interpreted and the required operands are obtained from the register file.

Stage 3 — Execute (EX):

```
Input: EX/MEM Pipeline Register
       Action: if (MemRead) Memory[ALU
      Result] → Read Data if (MemWrite)
      Memory[ALU Result] ← Value B
       if (Branch & branch_taken) PC ← branch
```

Stage 4 — Memory Access (MEM):

```
Input: ID/EX Pipeline Register
       Action: if (ALUSrc) operand2 = sign-extended immediate
      else operand2 = Value B ALU Result = Value A OP operand2
      (based on ALUOp)
             Calculate branch target: PC+4 + (sign-extended
      immediate << 2) Evaluate branch condition: Value A ==
      Value B
Output: EX/MEM Pipeline Register ← { ALU Result, Value B (for store),
```

Stage 5 — Write Back (WB):

```
 Input: MEM/WB Pipeline
Register Action: if
(RegWrite)
                  if (MemtoReg)
         Register[dest] ← Read Data else
            Register[dest] ← ALU Result
 Output: (None – instruction complete)
```

Pipeline Timing Diagram:

For a sequence of 5 independent arithmetic instructions, the timing diagram shows perfect overlap:

Table 2.2.1 Timing Diagram

Cycle	1	2	3	4	5	6	7	8	9
I1	IF	ID	EX	MEM	WB				
I2		IF	ID	EX	MEM	WB			
I3			IF	ID	EX	MEM	WB		
I4				IF	ID	EX	MEM	WB	
I5					IF	ID	EX	MEM	WB

SGOU

In a five-stage pipeline, the first instruction completes in cycle 5. The pipeline becomes fully occupied during cycle 4, assuming no stalls. After the pipeline is filled, one instruction can ideally complete per clock cycle. Once all instructions have entered, the pipeline gradually empties; this is called pipeline drain.

2.2.2 Arithmetic Pipeline Concept

While the instruction pipeline applies to all instructions, an arithmetic pipeline is a specialized pipeline within the execution unit that accelerates complex arithmetic operations — particularly floating-point operations. Arithmetic pipelines are essential because floating-point operations (addition, multiplication, division) take significantly longer than integer operations and would otherwise become the performance bottleneck.

Why Arithmetic Pipelines Are Necessary:

A single-cycle floating-point adder would require the clock period to accommodate all of:

1. Exponent comparison and subtraction
2. Mantissa alignment (shifting — potentially 24 or 53 bits)
3. Mantissa addition (carry-propagate adder)
4. Leading-zero detection
5. Mantissa normalization (shifting)
6. Rounding

This would make the clock period significantly longer than that required for simpler integer operations and could reduce the overall operating frequency of the processor.

2.2.3 Pipeline Timing and Performance

Key Performance Metrics:

1. Clock Cycle Time (T_clk):

```
T_clk = max(T_stage_i) + T_pipeline_register_overhead
```

Determined by the slowest pipeline stage plus the setup/hold time of pipeline registers.

2. Cycles Per Instruction (CPI):

```
CPI_ideal = 1 (in a perfect pipeline with no
```

In practice, CPI is slightly above 1 due to hazards.

SGOU

3. Execution Time:

```
Time = Instruction_Count × CPI × T_clk
```

In an ideal pipeline, CPI approaches 1 instruction per cycle after the pipeline is filled. Dividing instruction processing into stages can also allow a shorter clock period than an equivalent non-pipelined implementation.

4. Pipeline Speedup:

```
Speedup = T_nonpipelined / T_pipelined
            = (N × k × T_clk_old) / ((k + N - 1) × T_clk_new)
```

Where N = number of instructions, k = number of pipeline stages. For large N:

```
Speedup ≈ k × (T_clk_old / T_clk_new)
```

If each stage takes the same time as the original single-cycle machine, and we divide it into k balanced stages: T_clk_new ≈ T_clk_old/k, so Speedup ≈ k.

5. Pipeline Efficiency:

```
Efficiency = Actual_Speedup / Ideal_Speedup
                 = Useful_work_cycles / Total_cycles
```

In an ideal pipeline, efficiency = 1 (100%). Hazards reduce efficiency.

6. Throughput:

```
Throughput = 1 / (CPI × T_clk) instructions per second
```

Ideal throughput: 1 instruction per cycle (IPC = 1).

Solved Example 1: A non-pipelined processor has a clock period of 10 ns and CPI of 1. A 5-stage pipelined version has: stage times = [1.5, 1.2, 2.8, 2.0, 1.5] ns and pipeline register overhead = 0.2 ns. It has a stall rate of 0.15 stalls per instruction due to hazards. Calculate: (a) clock period, (b) CPI, (c) speedup for 1,000 instructions.

Solution:

a. T_clk(pipelined) = max(1.5, 1.2, 2.8, 2.0, 1.5) + 0.2 = 2.8 + 0.2 = 3.0 ns

b. CPI(pipelined) = 1 + 0.15 = 1.15

c. T_nonpipelined = 1,000 × 1 × 10 = 10,000 ns, T_pipelined = (k + N − 1 + N × stall_rate) × T_clk = (5 + 999 + 150) × 3.0 = 1,154 × 3.0 = 3,462 ns

Speedup = 10,000 / 3,462 ≈ 2.89× (vs. ideal 10/3 = 3.33× before accounting for stalls)

SGOU

2.2.4 Arithmetic Pipeline Example — Floating-Point Addition

Let us design and analyze a complete arithmetic pipeline for floating-point addition. We'll use the IEEE 754 single-precision format (32 bits: 1 sign, 8 exponent, 23 mantissa + hidden bit).

Floating-point addition is not a simple ALU operation — it involves several sequential steps that naturally map to pipeline stages.

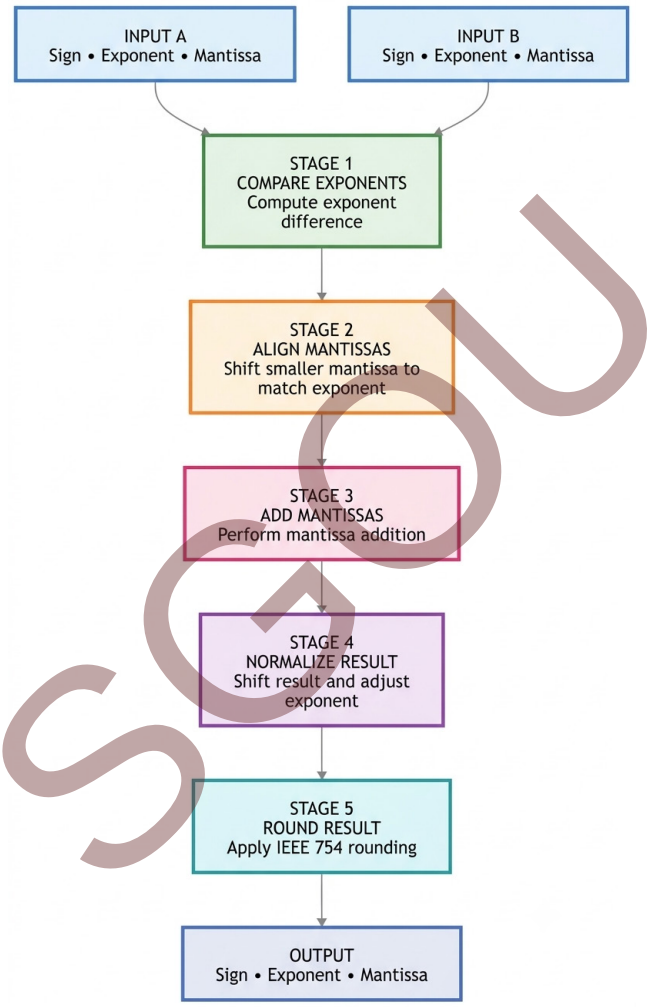


Fig. 2.2.2 Five-stage arithmetic pipeline for floating-point addition

Stage 1 — Compare Exponents:

- Inputs: Two IEEE 754 single-precision numbers A and B.
- Subtract exponents: $\Delta e = e_A - e_B$.
- Determine which mantissa needs alignment (the one with the smaller exponent).

SGOU

- Pass through: larger exponent, both mantissas (24-bit including hidden bit), signs.

Stage 2 — Align Mantissas:

- The mantissa corresponding to the smaller exponent is shifted right by $|\Delta e|$ positions.
- A barrel shifter performs this in one cycle (logarithmic or constant-time).
- Guard bits (G, R, S — Guard, Round, Sticky) are preserved for accurate rounding later.
- The shifted mantissa now has the same effective exponent as the larger one.

Stage 3 — Add/Subtract Mantissas:

- Based on the signs: if signs are the same, add mantissas; if different, subtract (the smaller from the larger).
- Produce a 25-bit intermediate result (possible carry-out from addition).
- Pass through: exponent, sign of result.

Stage 4 — Normalize Result:

- If there was a carry-out from addition, shift result right by 1 and increment exponent.
- If the result starts with zeros (after subtraction), shift left and decrement exponent until the leading bit is 1.

Stage 5 — Round Result:

- Apply the IEEE 754 rounding mode (default: round to nearest, ties to even).
- The result is adjusted if rounding produces an overflow, and the final sign, exponent, and mantissa are assembled.

Performance: With a 5-stage FP adder and a clock of 2 ns per stage:

- **Latency:** 5 × 2 = 10 ns per floating-point addition.
- **Throughput:** 1 result per 2 ns = 500 million floating-point additions/second after pipeline fill.
- A non-pipelined FP adder would either take 10 ns per operation (1 result per 10 ns = 100 MFLOPS) or require a 10 ns clock, slowing the entire processor.

SGOU

Recap

- The five-stage instruction pipeline (IF → ID → EX → MEM → WB) forms the baseline for understanding all pipelined processor designs.
- Pipeline timing diagrams clearly show the overlap of instructions and the impact of stalls.
- Arithmetic pipelines apply pipelining principles to complex arithmetic operations (especially floating-point), dividing them into sub-operations handled by specialized hardware.
- Key performance metrics include clock cycle time, CPI, speedup, throughput, and pipeline efficiency. Clock cycle time is determined by the slowest pipeline stage and register overhead, while stalls can increase CPI and reduce performance. The floating-point addition pipeline exemplifies arithmetic pipelining: (1) compare exponents, (2) align mantissas, (3) add/subtract mantissas, (4) normalize, (5) round.
- Pipeline stages should be balanced — ideally, each stage takes approximately the same time — to maximize throughput and minimize idle time.
- Pipeline registers between stages add overhead (setup/hold time) but are essential for decoupling stages and enabling concurrent operation.
- Pipelining trades latency (a single operation takes more cycles) for throughput (more operations complete per unit time).
- Modern processors may also use Fused Multiply-Add (FMA) units for efficient numerical computation.

SGOU

Objective Type Questions

1. In the 5-stage pipeline, which stage computes the branch target address?
2. What is the term for the cycles between when the first instruction enters the pipeline and when it completes?
3. What does the barrel shifter do in the alignment stage of a floating-point adder pipeline?
4. In a pipelined processor, what determines the clock cycle time?
5. A 6-stage pipelined processor has a clock period of 1.5 ns, while the equivalent non-pipelined processor requires 8 ns per instruction. Ignoring pipeline register overhead and assuming a large number of independent instructions, what is the approximate speedup?

6. What bits (G, R, S) are preserved during mantissa alignment to enable accurate rounding?

7. In pipeline performance, what is the ideal CPI for a perfectly balanced pipeline with no hazards?

8. In the floating-point addition pipeline, which stage uses a leading-zero detector?

9. What does FMA stand for in the context of floating-point operations?

10. Which hardware unit typically performs integer arithmetic and logical operations in the EX stage?

Answers

1. The EX (Execute) stage.

2. Pipeline latency.

3. Shifts the mantissa of the smaller operand right by the exponent difference.

4. The slowest pipeline stage (plus pipeline register overhead).

5. Approximately 5.33× (8/1.5).

6. Guard, Round, and Sticky bits.

7. CPI = 1.

8. The Normalize stage.

9. Fused Multiply-Add.

10. ALU (Arithmetic Logic Unit).

SGOU

Assignments

1. A pipelined FP adder has 4 stages with times: [3, 2, 4, 2] ns. Pipeline register overhead is 0.5 ns. Calculate the clock period, latency of a single addition, and steady-state throughput in MFLOPS.

2. Why is the rounding stage necessary in an arithmetic pipeline? What problems would arise without it?

3. A program has 1,000 instructions running on a 5-stage pipeline with a 0.10 stall rate per instruction and clock period of 2 ns. The same program on a non-pipelined processor takes 8 ns per instruction. Calculate and compare the execution times and speedup.

4. Pipeline Timing: Draw a timing diagram for six independent instructions executing in a five-stage instruction pipeline. Identify the pipeline fill period and the point at which steady-state throughput is reached.

5. Performance Calculation: A processor has a five-stage pipeline with a clock period of 2 ns. Assuming no stalls, calculate the time required to execute 100 independent instructions and determine the approximate throughput.

6. CPI and Stalls: A pipelined processor has an ideal CPI of 1 and experiences an average of 0.15 stall cycles per instruction. Determine the actual CPI and explain the effect of the stalls on performance.

7. Pipeline Speedup: Compare the execution time of a non-pipelined processor and a five-stage pipelined processor for a sufficiently large number of instructions. State the assumptions used in your calculation.

8. Arithmetic Pipeline: Explain how floating-point addition can be divided into multiple pipeline stages. Describe the function of each stage.

9. Floating-Point Addition: Trace the addition of two floating-point numbers through the stages of exponent comparison, mantissa alignment, addition/subtraction, normalization, and rounding using a suitable numerical example.

10. Pipeline Design Analysis: Explain why balanced pipeline stages are desirable and discuss what happens when one stage takes significantly longer than the others.

11. Research / Self-Directed Learning: Select a modern processor and investigate one of its arithmetic or floating-point execution pipelines. Briefly describe the operation being performed, the role of pipelining, and any publicly documented information about its throughput or latency.

Reference

1. Hennessy, J. L., Patterson, D. A., & Kozyrakis, C. (2025). *Computer architecture: A quantitative approach.* Morgan Kaufmann.

2. Patterson, D. A., & Hennessy, J. L. (2017). *Computer organization and design RISC-V edition: The hardware/software interface*. Morgan Kaufmann.

SGOU

3. Stallings, W. (2016). *Computer organization and architecture: Designing for performance*. Pearson Education India.

4. Harris, D. M., & Harris, S. L. (2021). *Digital design and computer architecture: RISC-V edition*. Morgan Kaufmann.

Suggested Reading

1. Koren, I. (2018). *Computer arithmetic algorithms*. CRC Press.

2. Institute of Electrical and Electronics Engineers. (2019). IEEE standard for floating-point arithmetic (IEEE Std 754-2019). https://doi.org/10.1109/IEEESTD.2019.8766229

3. Oberman, S. F., & Flynn, M. J. (1997). Design issues in division and other floating-point operations. *IEEE Transactions on computers*, *46*(2), 154-161.

SGOU



Principles of Designing Pipeline Processors

Learning Outcomes

After completion of this unit, learner will be able to:

- explain the concept of Instruction-Level Parallelism (ILP) and its role in pipeline processor design
- describe the fundamentals of branch prediction and evaluate different prediction schemes
- apply techniques for optimizing pipeline performance, including forwarding, loop unrolling, and scheduling
- identify and analyze the key design issues in pipelined processors: exception handling, precise interrupts, and pipeline depth trade-offs
- evaluate the interplay between hardware complexity and performance gains in advanced pipeline designs

Prerequisite

Imagine an airport security checkpoint where every passenger must pass through several stages, such as identity verification, baggage screening, and body scanning. If each passenger completes all the stages before the next passenger begins, the process may take considerable time. By allowing different passengers to move through different stages simultaneously, the airport can process more passengers in a given period. However, delays or problems at one stage can affect the overall flow.

A similar principle applies to pipelined processors, where instruction execution is divided into stages to improve performance. But how can a processor execute multiple independent instructions efficiently while handling branches, data dependencies, and exceptions? How can its design balance execution speed, hardware complexity, and power consumption? Exploring these challenges provides a foundation for understanding instruction-level parallelism, branch prediction, pipeline performance optimization, and the design issues involved in building efficient pipelined processors.

SGOU

Keywords

Instruction-Level Parallelism (ILP), Branch Prediction, Speculative Execution, Pipeline Depth, Pipeline Balancing, Loop Unrolling, Static Scheduling, Dynamic Scheduling

Discussion

2.3.1 Instruction-Level Parallelism (ILP)

Instruction-Level Parallelism is the foundation upon which advanced pipeline processors are built. ILP refers to the property of a program that allows multiple instructions to be executed simultaneously.

What Creates ILP?

ILP exists in programs naturally because many consecutive instructions are independent of each other:

```
I1: ADD R1, R2, R3        // Independent of I2 and I3
I2: SUB R4, R5, R6 I3:    // Independent of  I1 and I3
AND R7, R8, R9            // Independent of  I1 and I2
```

These three instructions can execute in parallel if the hardware supports it. These three instructions are independent and could execute concurrently if the processor has sufficient issue width and execution resources. Thus, this instruction group exposes three-way parallelism.

Limiting Factors

1. **Data Dependencies:** True dependencies (RAW) are inherent to the algorithm and cannot be eliminated. They form the fundamental limit on ILP.

2. **Control Dependencies:** Branches limit how far ahead we can look for independent instructions. The processor cannot know which instructions follow a branch until it resolves.

3. **Resource Constraints:** Even if ILP exists, the hardware may not have enough functional units, register ports, or issue width to exploit it.

4. **Name Dependencies:** WAR and WAW are artifacts of limited register names. Register renaming eliminates them, exposing more true ILP.

Measuring ILP:

The amount of exploitable ILP varies across programs. Programs with independent operations and regular computation generally provide more opportunities for instruction-level parallel execution, whereas data dependencies and frequent branches can limit available ILP.

SGOU

Techniques to Exploit ILP:

Table 2.3.1 Techniques for Exploiting Instruction-Level Parallelism (ILP)

Technique	Main Mechanism	Primary Benefit
Basic Pipelining	Overlaps instruction stages	Improves instruction throughput
Superscalar Execution	Issues multiple instructions per cycle	Exploits available ILP
VLIW	Compiler schedules independent operations	Reduces some hardware scheduling complexity
Out-of-Order Execution	Executes ready instructions before stalled ones	Hides latency and exploits ILP
Speculative Execution	Executes predicted instruction paths	Reduces control-dependency delays
Loop Unrolling	Replicates loop body	Reduces branch overhead and exposes ILP

2.3.2 Branch Prediction Basics

Branches occur frequently in many programs and determine which instructions should be fetched next. Without effective prediction, the pipeline may have to wait for the branch outcome before fetching from the correct path.

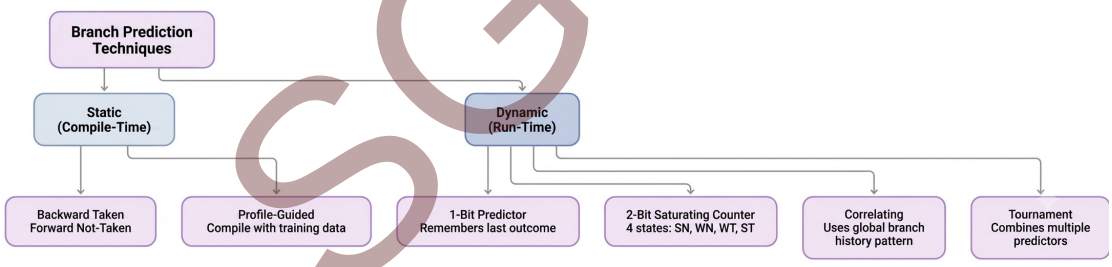


Fig. 2.3.1 Classification of branch prediction techniques

2-Bit Saturating Counter in Detail:

The 2-bit predictor maintains one of four states for each branch:

- **00:** Strongly Not-Taken
- **01:** Weakly Not-Taken
- **10:** Weakly Taken
- **11:** Strongly Taken

State Transitions:

- On a taken branch: move toward Strongly Taken (01→10, 10→11)

SGOU

- On a not-taken branch: move toward Strongly Not-Taken (10→01, 01→00)
- In Strong states: a single misprediction changes to the corresponding Weak state, not to the opposite prediction.

Solved Example 1: A loop iterates 5 times. Trace the 2-bit predictor state starting from Weakly Not-Taken (01).

Iteration	Actual	Prediction	State Before	State After	Correct?
1	Taken	Not-Taken	01 (WN)	10 (WT)	No
2	Taken	Taken	10 (WT)	11 (ST)	Yes
3	Taken	Taken	11 (ST)	11 (ST)	Yes
4	Taken	Taken	11 (ST)	11 (ST)	Yes
5 (exit)	Not-Taken	Taken	11 (ST)	10 (WT)	No

Accuracy: 3 correct out of 5 = 60%. The example illustrates that a 2-bit predictor does not immediately change its prediction after a single anomalous outcome because it uses two bits of state. This behavior can improve stability for branches with mostly consistent behavior.

Correlating Predictors:

Consider the code:

```
 if (a == 2) a = 0;
if (b == 2) b = 0;
 if (a != b) ... // This branch's outcome depends on the previous two branches
```

A simple 2-bit predictor treats each branch independently and misses the correlation. A (m,n) correlating predictor uses the behavior of the last m branches (m-bit global history) to select among 2^m n-bit predictors for each branch. This captures patterns like alternating taken/not-taken (the pattern 101010...), which simple predictors fail on.

Branch Target Buffer (BTB):

The BTB is a small, fast cache indexed by the PC (or low-order bits thereof) that stores:

- **Tag:** The full PC of the branch (to verify match).
- **Target Address:** The PC of the branch target (where to fetch if taken).
- **Prediction Bits:** The predictor state.

SGOU

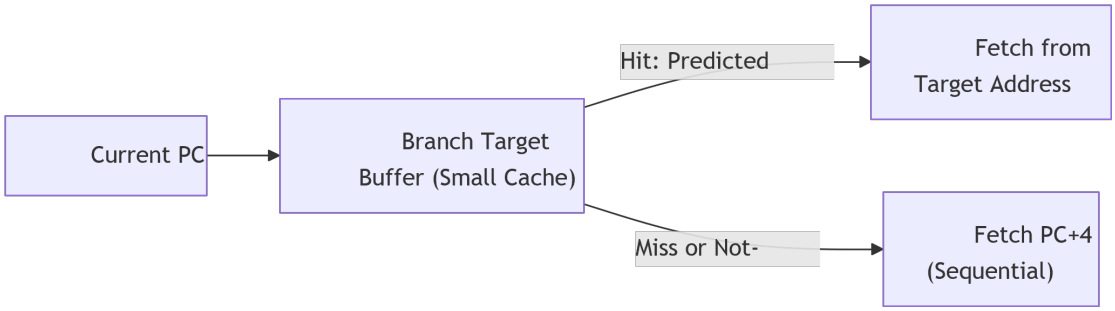


Fig. 2.3.2 Branch Target Buffer operation for predicted-taken and sequential instruction fetch

When the IF stage fetches an instruction, it simultaneously checks the BTB. If the PC hits and the predictor says "taken," the next fetch comes from the stored target address. When the BTB provides a matching branch target and the prediction indicates that the branch is taken, the processor can begin fetching from the predicted target without waiting for the branch to be fully resolved. This can substantially reduce the effective branch penalty.

2.3.3 Pipeline Performance Optimization

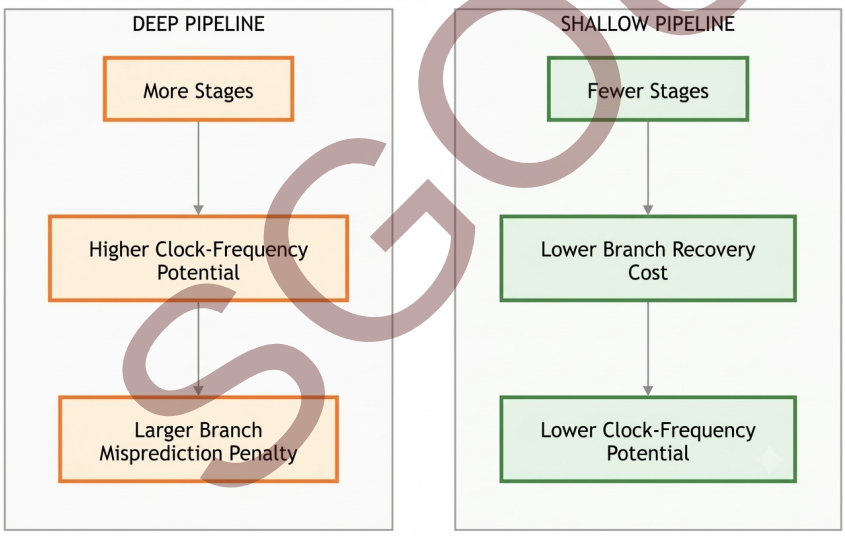


Fig. 2.3.3 Pipeline Depth and Branch Penalty

Beyond basic pipelining, several techniques optimize performance:

1. **Deep Pipelining (Superpipelining):** Splitting existing stages into more, smaller stages:
   - ♦ Pros: Higher clock frequency, higher peak throughput.
   - ♦ Cons: Increased branch misprediction penalty, more pipeline registers = more area and power, diminishing returns due to imbalance.
2. **Multiple Issue (Superscalar):** Multiple instructions may be issued and executed in the same clock cycle when sufficient independent instructions and hardware

resources are available. The issue width must be balanced against the amount of ILP available in the workload.

3. **Out-of-Order Execution:** Instructions execute as soon as their operands are available, not necessarily in program order:

   - Instructions after a stalled instruction can proceed if they are independent.
   - Requires complex hardware: reservation stations, reorder buffer (ROB), register renaming.
   - The Reorder Buffer ensures that results are committed in program order (for precise exceptions), even though they execute out of order.

4. **Loop Unrolling (Compiler Technique):**

```
Original:
    for (i = 0; i < 100; i++)
     sum += a[i];
```

```
Unrolled (4x):
    for (i = 0; i < 100; i += 4) {
        sum += a[i];
        sum += a[i+1];
        sum += a[i+2];
        sum += a[i+3];
    }
```

Benefits: reduces branch instructions by 4×, exposes more independent operations for the scheduler, and enables better register allocation.

5. **Software Pipelining (Compiler Technique):** Restructures loops so that iterations overlap (similar to hardware pipelining), initiated at a constant rate. Particularly effective for loops with inter-iteration dependencies.

Solved Example 2: A loop has 10 instructions per iteration, including 1 branch, and runs for 100 iterations. The branch is predicted correctly 98% of the time with a 2-cycle misprediction penalty. Unrolling the loop 5× reduces branches to 20. Compare total branch penalty cycles.

Solution:

- **Expected penalty** = 100 × 0.02 × 2 = 4 cycles per complete execution, on average.
- **Expected penalty** = 20 × 0.02 × 2 = 0.8 cycles per complete execution, on average.
- Additionally, the unrolled loop exposes more ILP (5× more operations without dependencies in each iteration), potentially improving IPC.

SGOU

2.3.4 Design Issues in Pipelined Processors

1. Exception Handling and Precise Interrupts:

An exception (page fault, divide by zero, illegal instruction) can occur in any pipeline stage. In a pipelined processor, by the time the exception is detected (e.g., in the MEM stage for a page fault), several subsequent instructions may already be partially executed.

Precise Interrupts require that:

- All instructions before the excepting instruction are completed.
- The excepting instruction and all instructions after it have no effect on the processor state.
- The PC saved is that of the excepting instruction.

Implementation approaches:

- **Pipeline Drain:** Force all instructions before the exception to complete, squash all others. Simple but slow.
- **Reorder Buffer (ROB):** Instructions are committed (their results written to architectural state) in program order. An exception in any instruction prevents it and all subsequent instructions from committing. This is the standard approach in out-of-order processors.
- **History Buffer:** Save old register values before overwriting. On exception, restore from the history buffer.

2. Pipeline Depth Trade-offs:

Table 2.3.2 Comparison of Shallow and Deep Pipeline Designs

Pipeline Depth	Clock Frequency	Branch Penalty	Hardware Complexity	General Characteristics
Shallower	Generally lower	Lower	Lower	Simpler design and lower control overhead
Deeper	Generally higher potential	Higher	Higher	Higher frequency potential but greater branch and recovery costs

The appropriate pipeline depth depends on the processor's target workload, clock-frequency goals, branch behavior, power budget, and implementation complexity. A deeper pipeline is not automatically better because the benefits of a shorter stage delay must be balanced against increased branch penalties, register overhead, power consumption, and control complexity.

SGOU

3. Power and Thermal Constraints:

Pipelining increases power consumption through:

- Pipeline registers (clocking thousands of flip-flops each cycle).
- Speculative execution (doing work that may be discarded).
- Higher clock frequencies (Dynamic power is approximately proportional to switching activity, clock frequency, and the square of supply voltage. Therefore, increasing frequency and voltage can significantly increase power consumption.).

Modern designs must balance pipeline depth against the "power wall." Techniques include clock gating (disabling clocks to idle stages), power gating (cutting power entirely), and dynamic voltage/frequency scaling (DVFS).

4. Interrupt and Context Switch Overhead:

On a context switch, processor state must be preserved and the pipeline may need to be redirected to the new process. Deeper pipelines can increase the cost associated with recovering and refilling the pipeline.

The Tomasulo Algorithm: A Landmark in Pipeline Design

Robert Tomasulo developed the Tomasulo algorithm at IBM in 1967 for the System/360 Model 91 floating-point unit. The algorithm introduced important concepts such as reservation stations, register renaming, and dynamic scheduling. These ideas became influential in the development of modern out-of-order processors.

Interesting Fact: The Pentium FDIV Bug

In 1994, a flaw in the floating-point division unit of some early Intel Pentium processors could produce incorrect results for a small set of division operations. The incident highlighted the importance of verification, testing, and reliable exception handling in processor design.

Recap

- ILP allows multiple independent instructions to execute concurrently.
- Data, control, and resource dependencies limit exploitable ILP.
- Branch prediction reduces delays caused by control dependencies.
- Static prediction uses compile-time rules; dynamic prediction uses run-time information.

SGOU

- A 2-bit predictor uses four states and is less sensitive to isolated branch behavior.
- A BTB stores predicted branch target information.
- Loop unrolling and software pipelining are compiler-based optimization techniques.
- Out-of-order execution uses hardware mechanisms to exploit available ILP.
- Precise exceptions require architectural state to correspond to a well-defined program point.
- Deeper pipelines can increase clock-frequency potential but also increase branch penalties, power, and design complexity.

Objective Type Questions

1. What property of a program allows multiple instructions to execute simultaneously within a single processor?
2. In a 2-bit branch predictor, how many states are there and what are their names?
3. What small fast cache stores branch target addresses to reduce branch penalty?
4. What hardware structure in out-of-order processors ensures precise exceptions by committing instructions in program order?
5. What compiler technique replicates loop bodies to reduce branch frequency and expose ILP?
6. In deep pipelines, what penalty increases proportionally with pipeline depth?
7. What is the name of the algorithm that introduced reservation stations and register renaming?
8. What type of scheduling does a compiler perform to minimize pipeline stalls by reordering instructions?
9. What metric expresses how closely the actual pipeline performance approaches the ideal pipeline performance?
10. What does DVFS stand for in the context of power management?

SGOU

Answers

1. Instruction-Level Parallelism (ILP).
2. Four: Strongly Not-Taken, Weakly Not-Taken, Weakly Taken, Strongly Taken.
3. Branch Target Buffer (BTB).
4. Reorder Buffer (ROB).
5. Loop unrolling.
6. Branch misprediction penalty.
7. Tomasulo's algorithm.
8. Static scheduling.
9. Pipeline efficiency.
10. Dynamic Voltage and Frequency Scaling.

Assignments

1. Explain Instruction-Level Parallelism (ILP) with suitable examples. Discuss the factors that limit the exploitation of ILP in pipelined processors.
2. A 2-bit saturating branch predictor starts in the Weakly Not-Taken (01) state. Trace its state transitions and determine the number of correct predictions and mispredictions for the following sequence of actual branch outcomes: Taken, Taken, Not-Taken, Taken, Not-Taken, Not-Taken. Assume that the prediction is Not-Taken in states 00 and 01, and Taken in states 10 and 11.
3. Explain how forwarding, loop unrolling, static scheduling, and out-of-order execution help improve pipeline performance. Discuss the benefits and limitations of these techniques.
4. Consider a loop that executes 100 iterations, with one branch instruction per iteration. If the branch prediction accuracy is 98% and the misprediction penalty is 2 cycles, calculate the expected total branch penalty before and after unrolling the loop by a factor of 5. Assume that unrolling reduces the number of branch instructions to 20 while maintaining the same prediction accuracy and misprediction penalty.

SGOU

5. What are precise interrupts in pipelined processors? Explain how a Reorder Buffer (ROB) helps maintain precise exceptions in out-of-order execution. Compare this approach with pipeline draining.

6. Compare shallow and deep pipelines in terms of clock frequency, branch misprediction penalty, hardware complexity, power consumption, and overall performance. Explain why increasing pipeline depth does not always improve processor performance.

7. Select a modern processor and identify one documented feature related to pipeline design, such as out-of-order execution, branch prediction, or deep pipelining. Describe the feature, explain its purpose, and discuss its potential impact on processor performance using reliable sources.

Reference

1. Hennessy, J. L., Patterson, D. A., & Kozyrakis, C. (2025). *Computer architecture: A quantitative approach*. Morgan Kaufmann.

2. Patterson, D. A., & Hennessy, J. L. (2017). *Computer organization and design RISC-V edition: The hardware/software interface*. Morgan Kaufmann.

3. Stallings, W. (2016). *Computer organization and architecture: Designing for performance*. Pearson Education India.

4. Harris, S., & Harris, D. (2021). *Digital design and computer architecture: RISC-V edition*. Morgan Kaufmann.

Suggested Reading

1. Tomasulo, R. M. (1967). An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, *11*(1), 25-33.

2. Smith, J. E. (1998, August). A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)* (pp. 202-215).

3. Yeh, T. Y., & Patt, Y. N. (1991, September). Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture* (pp. 51-61).

SGOU



Principles of Vector Processing

Learning Outcomes

After completion of this unit, learner will be able to:

- define vector processing and explain its relationship to the SIMD execution model
- compare scalar and vector processing, identifying the key architectural and performance differences
- describe the fundamental elements of a vector instruction set, including vector registers, vector length, and stride
- analyze how vectorization transforms scalar loops into efficient vector operations
- identify key application domains where vector processing provides substantial performance advantages

Prerequisite

Imagine checking 100 students' answer papers. Instead of checking each paper from beginning to end, you could check the same question across all the papers before moving to the next question. This approach can save time when the same operation must be repeated many times.

A similar idea is used in vector processing, where a processor applies a single instruction to multiple data elements. For example, when adding two large arrays of numbers, a vector processor can process several pairs of elements together rather than handling each pair separately. This approach is particularly useful in scientific calculations, image processing, and machine learning, where large amounts of data often require similar operations. Building on the concept of instruction pipelining, vector processing takes advantage of data-level parallelism to improve computational efficiency. Understanding this idea provides a foundation for exploring vector registers, vector instructions, vector length, and the applications of vector processing.

SGOU

Keywords

Vector Processor, SIMD (Single Instruction, Multiple Data), Vector Register, Maximum Vector Length (MVL), Vector Stride, Vector Chaining, Vector Mask, Strip Mining

Discussion

2.4.1 Concept of Vector Processing and SIMD

Vector processing is a computing paradigm where a single instruction (a vector instruction) operates on an entire array of data called a vector rather than on a single scalar element. This is a direct architectural implementation of the SIMD model: one instruction stream, multiple data elements.

A vector processor contains:

- Vector registers that can hold multiple elements, with the number of elements depending on the architecture and element size. For example, the Cray-1 used vector registers capable of holding 64 elements of 64 bits each.
- Vector functional units are typically deeply pipelined and, after startup latency, can often produce one result per clock cycle under ideal conditions.
- Vector load/store units that can fetch or store multiple elements from/to memory efficiently.

Why Vector Processing?

Vector processing addresses several inefficiencies of scalar (superscalar) processors:

1. **Instruction Fetch Bandwidth:** A scalar loop requires fetching the same instructions (load, add, store, branch) for each element. A vector instruction encodes the operation for many elements in a single instruction — dramatically reducing instruction fetch pressure.

2. **Data Dependencies:** In a scalar loop, each iteration has a control dependency (the branch at the end) and potential data dependencies. Vector instructions have no intra-instruction dependencies — the vector operation on element i is independent of the operation on element j (unless explicitly chained).

3. **Memory Access Patterns:** Vector processors are optimized for regular memory access patterns (contiguous, constant stride). The hardware can prefetch, pipeline, and interleave memory accesses much more efficiently than scalar processors chasing individual loads.

4. **Pipeline Efficiency:** A vector functional unit is a deep pipeline. Once started (after the initial latency), it produces one result per cycle. For an element-wise

SGOU

vector operation such as array addition, the individual element computations are independent. Therefore, the processor can exploit substantial data-level parallelism without executing a branch for every element.

The SIMD Connection:

SIMD is the broader architectural classification (Flynn's taxonomy). Vector processing is one important form of SIMD execution. Other SIMD implementations include fixed-width multimedia extensions in general-purpose processors. Other SIMD forms include:

- **Multimedia SIMD Extensions:** MMX, SSE, AVX, NEON operate on short vectors (2-16 elements) within general-purpose registers.
- **GPU SIMT:** Groups of threads execute instructions in a coordinated manner on different data elements. SIMT shares important characteristics with SIMD but uses a thread-based programming model.

Modern vector architectures such as RISC-V RVV can support vector-length-agnostic programming, whereas conventional fixed-width SIMD extensions use instructions tied to particular register widths.

2.4.2 Scalar vs. Vector Processing

Scalar Code Example (C):

```
for (i=0; i < 64; i++)
  C[i] = A[i] + B[i];
```

Scalar Assembly (RISC-V-like):

```
li      t0, 64          // Loop count
loop:
   fld  f0, 0(t1)       // Load A[i]
   fld  f1, 0(t2)       // Load B[i]
   fadd f2, f0, f1      // C[i] = A[i] + B[i]
   fsd  f2, 0(t3)       // Store C[i]
   addi t1, t1, 8       // Increment pointers
   addi t2, t2, 8
   addi t3, t3, 8
   addi t0, t0, -1      // Decrement counter
   bnez t0, loop          // Branch if not
```

This executes 9 × 64 = 576 instructions for 64 elements.

SGOU

Vector Assembly (equivalent):

```
vsetvl t0, 64       // Set vector length to 64
vload v0, A         // Load A[0:63] into v0
vload v1, B         // Load B[0:63] into v1
vadd v2, v0, v1     // v2 = v0 + v1 (all 64 elements)
vstore v2, C        // Store v2 to C[0:63]
```

This executes only 5 instructions for the same work.

Performance Comparison:

The instruction counts shown here are based on the simplified instruction sequences used in this example. Actual execution time depends on vector length, pipeline latency, memory bandwidth, instruction issue width, and the processor implementation.

Table 2.4.1 Performance Comparison of Scalar and Vector Processing

Aspect	Scalar	Vector
Instructions executed	576	5
Instruction fetches	576	5
Branches executed	64	0
Branch mispredictions	Possible	None
Pipeline hazards	Possible (RAW on loop counter)	None within vector op
Memory access pattern	Individual loads/stores	Burst/streaming access

Key observations:

- For this 64-element example, the vector code eliminates the per-element loop overhead, including counter updates, pointer updates, and branches.
- Vector loads/stores are more efficient — the memory system sees a streaming access pattern and can optimize accordingly.
- The vector ADD has no intra-vector dependencies; elements 0-63 are independent.

2.4.3 Vector Instruction Set Basics

A vector instruction set typically includes these categories:

Note: The instruction formats shown below are illustrative pseudo-instructions used to explain vector-processing concepts. Actual instruction names and syntax vary across vector architectures such as RISC-V RVV, ARM SVE/SVE2, and x86 SIMD.

SGOU

1. **Vector Load/Store:**

```
VLD Vd, [Rs + offset]     // Load vector from memory
VST [Rd + offset], Vs     // Store vector to memory
VLDST Vd, [Rs + Rstride] // Strided load (non-contiguous)
VLDX Vd, [Rs + Vidx]      // Indexed (gather) load
VSTX [Rd + Vidx], Vs      // Indexed (scatter) store
```

2. **Vector Arithmetic:**

```
VADD Vd, Vs1, Vs2        // Element-wise addition

VSUB Vd, Vs1, Vs2        // Element-wise subtraction
VMUL Vd, Vs1, Vs2        // Element-wise multiplication
VDIV Vd, Vs1, Vs2        // Element-wise division
VFMA Vd, Vs1, Vs2, Vs3   // Fused multiply-add: Vd = Vs1*Vs2 + Vs3
```

3. **Vector Logical and Comparison:**

```
VAND     Vd, Vs1, Vs2         // Bitwise AND
VOR      Vd, Vs1, Vs2         // Bitwise OR
VCMP     Vmask, Vs1, Vs2, cond // Set mask based on comparison
```

4. **Vector Reduction:**

```
VSUM Rd, Vs                  // Sum all elements of Vs → scalar Rd
VMAX Rd, Vs                  // Maximum element → scalar Rd
VDOT Rd, Vs1, Vs2            // Dot product → scalar Rd
```

5. **Vector Control:**

```
VSETVL Rt, N                 // Set vector length to N
VMASKED Vd, Vs, Vmask        // Execute only where mask bit is set
```

Vector Length and Strip Mining:

Hardware vector registers have a finite capacity (Maximum Vector Length, MVL). If the application requires processing more elements than MVL, strip mining divides the data into chunks that fit within the available vector length. The compiler may perform this transformation automatically:

SGOU

```
// Process array of N elements with
// MVL-sized chunks

for (i = 0; i < N; i += MVL)
{
      VL = min(MVL, N - i);
      vsetvl(VL);
      vload v0, A[i : i+VL-1];
      vload v1, B[i : i+VL-1];
      vadd v2, v0, v1;
      vstore v2, C[i : i+VL-1];
}
```

Vector Chaining:

Chaining is the vector equivalent of scalar forwarding. When a vector instruction produces a result and the next vector instruction consumes it, chaining allows the consumer to start as soon as the first element of the producer is available — element by element overlap, rather than waiting for the entire producer vector to complete.

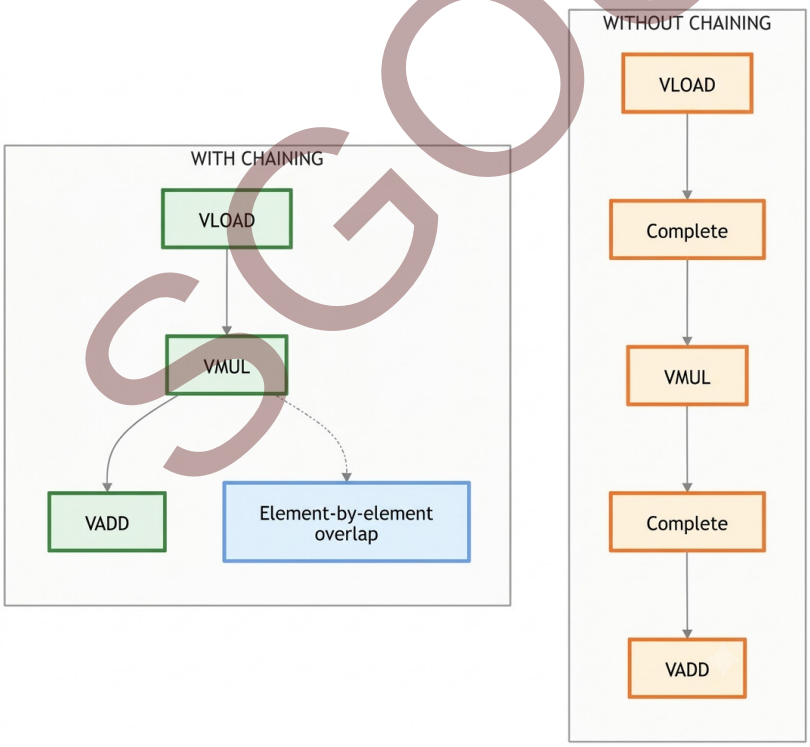


Fig. 2.4.1 Vector chaining and element-by-element overlap

Without chaining, each vector instruction must complete entirely before the next starts. With chaining, the MUL can begin on element 0 as soon as VLOAD produces it, and VADD can begin as soon as MUL produces its first result. The entire sequence behaves like a single deep pipeline.

SGOU

Solved Example 1: A vector processor has MVL = 64. A program needs to process 200 elements. How many strip-mined iterations are needed, and how many elements are processed in each iteration?

Solution:

- Iteration 1: VL = 64, elements 0-63 processed.
- Iteration 2: VL = 64, elements 64-127 processed.
- Iteration 3: VL = 64, elements 128-191 processed.
- Iteration 4: VL = min(64, 200-192) = 8, elements 192-199 processed.
- **Total:** 4 iterations.

2.4.4 Applications of Vector Processing

Vector processing excels in domains with regular, data-parallel computation:

1. Scientific Computing and Linear Algebra:

Matrix multiplication, LU decomposition, eigenvalue computation, and finite-difference methods — the core of computational science — are naturally vectorizable. Dense linear algebra libraries (BLAS, LAPACK) are highly optimized for vector architectures.

Example: Matrix multiply C = A × B involves dot products of rows of A with columns of B. On a vector machine, entire rows and columns are loaded into vector registers and processed with vector multiply-add instructions.

For illustration, a vector dot-product instruction or a sequence of vector multiply and reduction instructions can be used to compute the inner product.

```
// Scalar: O(n³) operations, each processed sequentially
// Vector: Each inner-product becomes a single
VDOT instruction for (i = 0; i < n; i++)
          for (j = 0; j < n; j++)
              C[i][j] = vdot(A_row[i], B_col[j]); // One vector instruction
```

2. Image and Signal Processing:

Convolution, filtering, FFT, DCT (used in JPEG/MPEG compression) — all operate on large arrays with the same operation applied to every element or window. A 2D convolution kernel applied to a 4K image becomes trivially vectorizable: load strips of the image, load the kernel, apply element-wise multiply-accumulate.

3. Machine Learning and Neural Networks:

The fundamental operation in neural networks, matrix multiplication, is dominated by dot products and multiply-accumulate. GPUs are highly parallel processors that

SGOU

are well suited to data-parallel operations such as matrix multiplication and multiply-accumulate computations. Their SIMT execution model shares important characteristics with SIMD but uses groups of threads as the basic programming abstraction. Modern CPU vector extensions (AVX-512, SVE) target the same operations.

4. Cryptography:

AES encryption processes 128-bit blocks with operations on all 16 bytes simultaneously — a perfect match for 128-bit SIMD vectors. Modern processors accelerate AES with dedicated SIMD instructions (AES-NI).

5. Physics Simulation and Gaming:

Game physics engines process thousands of objects with identical physics rules. Vectorized collision detection, particle systems, and rigid-body dynamics benefit significantly.

6. Database Operations:

Columnar databases store data in columns rather than rows. Scanning and filtering a column (e.g., "find all ages > 30") is trivially vectorizable — load a vector of ages, compare with 30, and produce a mask.

7. Bioinformatics:

DNA sequence alignment, protein structure prediction, and genome assembly involve comparing and processing massive strings and matrices. These operations map naturally to vector/SIMD execution.

Pioneers of Vector Computing

The Cray Legacy The Cray-1, introduced in the 1970s, was one of the landmark vector supercomputers. Its architecture included dedicated vector registers and pipelined vector functional units, allowing it to process large arrays efficiently. The system demonstrated the effectiveness of vector processing for scientific and engineering workloads.

RISC-V Vector Extension (RVV)

A Modern Renaissance After years of SIMD extensions dominating (SSE, AVX, NEON), the RISC-V Vector Extension (ratified in 2021) has revived interest in true vector processing. Unlike fixed-width SIMD, RVV is vector-length agnostic (VLA) — The RISC-V Vector Extension supports vector-length-agnostic programming, allowing software to operate across implementations with different vector capabilities. This eliminates the portability problems of width-specific SIMD code. RVV represents a return to the Cray-era philosophy that vector length should be a hardware implementation detail, not a programmer-visible constraint.

SGOU

Recap

- Vector processing applies one instruction to multiple data elements.
- It is an important implementation of the SIMD execution model.
- Scalar processing handles individual data elements, whereas vector processing handles multiple elements together.
- Vector registers, vector length, stride, masks, and vector instructions form the basic vector-processing model.
- Vectorization transforms suitable scalar loops into vector operations.
- Strip mining handles arrays larger than the available vector length.
- Vector chaining overlaps dependent vector operations element by element.
- Vector processing is particularly useful in scientific computing, image processing, machine learning, cryptography, databases, and bioinformatics.

Objective Type Questions

1. What architectural paradigm does vector processing implement, where one instruction operates on multiple data elements?
2. In vector processing, what is the register that holds multiple data elements called?
3. What term is used to describe the maximum number of elements that a vector architecture can process in one vector operation?
4. What technique handles arrays larger than the hardware vector register capacity by processing in chunks?
5. What is the forwarding technique between vector operations called, where results flow element-by-element?
6. What type of vector memory access uses a constant stride between consecutive elements?
7. In the scalar loop for array addition, what type of overhead does vectorization eliminate?
8. What bit mask selectively enables or disables operations on individual vector elements?

SGOU

9. Which RISC-V extension, ratified in 2021, supports vector-length-agnostic programming?

10. What is one major advantage of vector processing over scalar processing for large array computations?

Answers

1. SIMD (Single Instruction, Multiple Data).
2. Vector register.
3. Maximum Vector Length (MVL).
4. Strip mining.
5. Vector chaining.
6. Strided access (or constant-stride access).
7. Loop overhead (branch, counter decrement, pointer update).
8. Vector mask.
9. RISC-V (RVV — RISC-V Vector Extension).
10. Reduced instruction overhead.

Assignments

1. Define vector processing and explain its relationship with the SIMD execution model. Discuss how vector registers, vector functional units, and vector load/store units support vector processing.

2. Compare scalar and vector processing in terms of instruction count, instruction fetches, loop overhead, memory access patterns, and pipeline efficiency. Explain the advantages of vector processing for large array computations.

3. A vector processor has a Maximum Vector Length (MVL) of 64 elements and must process an array containing 250 elements. Determine the number of strip-mined iterations and the number of elements processed in each iteration.

SGOU

4. Explain vector chaining using a sequence of vector load, vector multiplication, and vector addition instructions. Illustrate how chaining allows successive vector operations to overlap element by element.

5. Explain the main categories of vector instructions, including vector load/store, arithmetic, logical and comparison, reduction, and control operations. Give suitable examples of each category.

6. Select any two application domains discussed in the unit, such as scientific computing, image processing, machine learning, cryptography, database operations, or bioinformatics. Explain how vector processing improves the execution of data-parallel operations in these domains.

Reference

1. Hennessy, J. L., Patterson, D. A., & Kozyrakis, C. (2025). *Computer architecture: a quantitative approach*. Morgan kaufmann.

2. Patterson, D. A., & Hennessy, J. L. (2017). *Computer organization and design RISC-V edition: The hardware/software interface*. Morgan Kaufmann.

3. Russell, R. M. (1978). The CRAY-1 computer system. *Communications of the ACM*, *21*(1), 63-72.

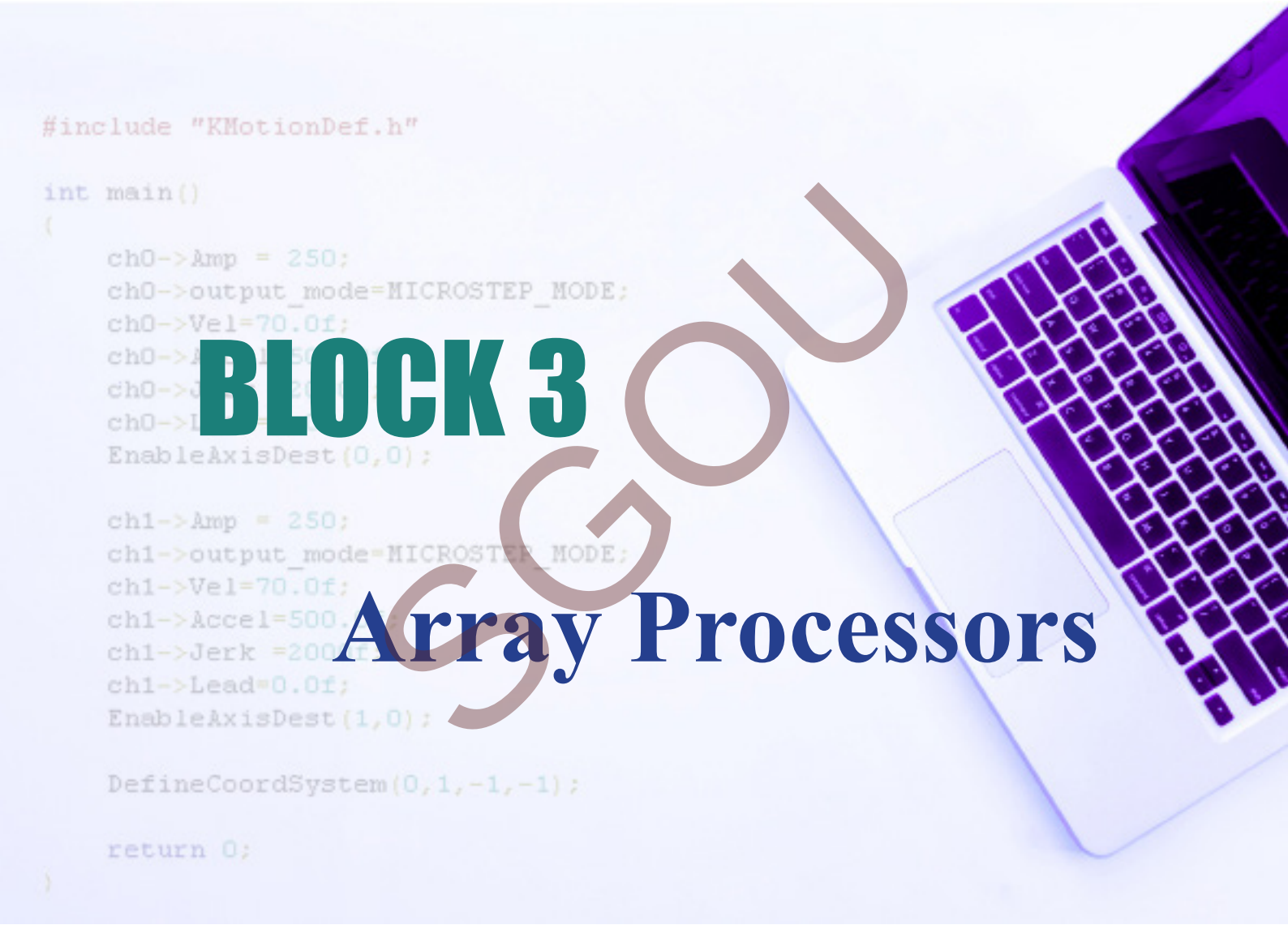
Suggested Reading

1. Asanović, K. (1998). Vector microprocessors (Technical Report No. UCB/CSD-98-1014) [Doctoral dissertation, University of California, Berkeley]. University of California, Berkeley. https://people.eecs.berkeley.edu/~krste/thesis.html

2. RISC-V V Extension Specification v1.0 (2021). https://github.com/riscv/riscv-v-spec

3. Stephens, N., Biles, S., Boettcher, M., Eapen, J., Eyole, M., Gabrielli, G., ... & Walker, P. (2017). The ARM scalable vector extension. *IEEE micro*, *37*(2), 26-39.

SGOU



BLOCK 3

Array Processors

SGOU



SIMD Array Processors

Learning Outcomes

After completion of this unit, learner will be able to:

- define the SIMD concept and contrast it with other architectural models
- describe the structure of an array processor, identifying the control unit and processing elements
- explain the operation of control units and processing elements, including masking
- identify key applications where SIMD array processors excel
- analyze the strengths and limitations of the SIMD model

Prerequisite

The SIMD model was introduced in Block I as one of Flynn's four architectural categories. This unit expands that concept into a concrete machine design — the array processor. Understanding the distinction between a single control unit (which fetches and decodes instructions) and many processing elements (which execute those instructions on local data) is fundamental.

Consider a classroom where the teacher says "everyone solve problem 7 from page 42." One instruction, many data (each student's own paper). The teacher is the control unit; each student is a processing element with their own notebook (local memory). If the teacher says "those with even roll numbers, raise your hand" — that's masking: only the selected students (PEs) execute the instruction, others stay idle. This is precisely how a SIMD array processor operates.

Keywords

Array Processor, Control Unit (CU), Processing Element (PE), Local Memory, Mask Bit, Lockstep Execution, Systolic Array

SGOU

Discussion

3.1.1 SIMD Concept

SIMD (Single Instruction, Multiple Data) is one of the four categories in Flynn's taxonomy. In a SIMD machine, a single instruction stream controls multiple processing elements, each operating on its own data stream. The key insight: many computational problems apply the same operation to many data items, and SIMD exploits this by fetching/decoding an instruction once, then broadcasting it to all PEs for parallel execution on different data. See fig. 3.1.1.

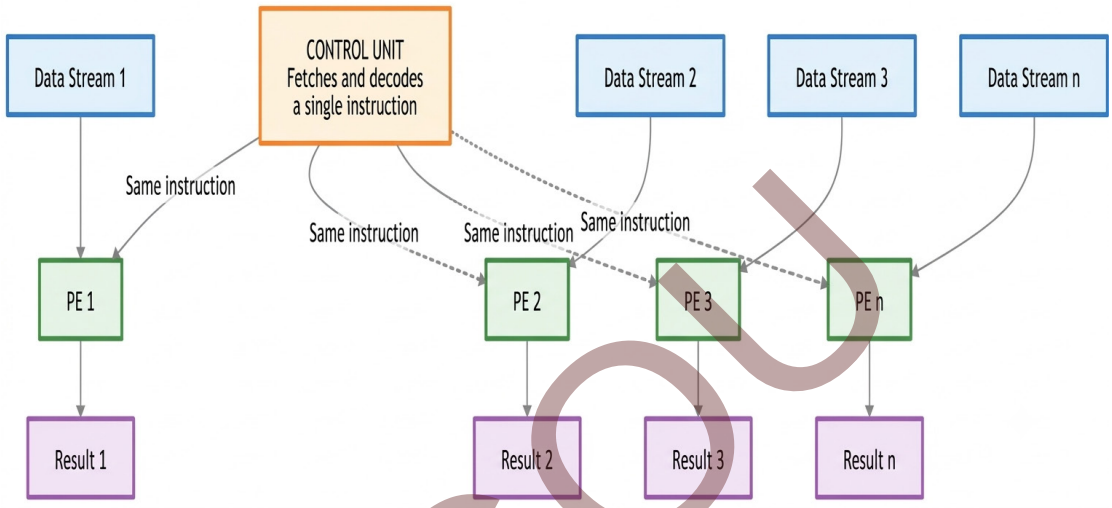


Fig. 3.1.1 Basic organization of a SIMD array processor

3.1.1.1 Distinction from MIMD

- **SIMD:** One instruction stream, many data streams. All PEs execute the same operation in lockstep. Simple, deterministic, efficient for regular data-parallel problems.
- **MIMD:** Many instruction streams, many data streams. Each processor runs its own program. More flexible, but with synchronization and communication overhead.

3.1.1.2 Distinction from Vector Processing

While both are SIMD in the broad sense, they differ in implementation:

- **Vector processors** process long vectors in deep pipelines, with a single vector instruction operating on up to 64-512 elements, flowing through a single (or few) pipelined functional units.
- **Array processors** (this block) have many spatially distributed, simple PEs each operating on its assigned data element(s) simultaneously. The parallelism is in space (many PEs) rather than time (pipeline stages).

SGOU

3.1.2 Processor Structure

An array processor has a distinctive structure, consisting of two main components:

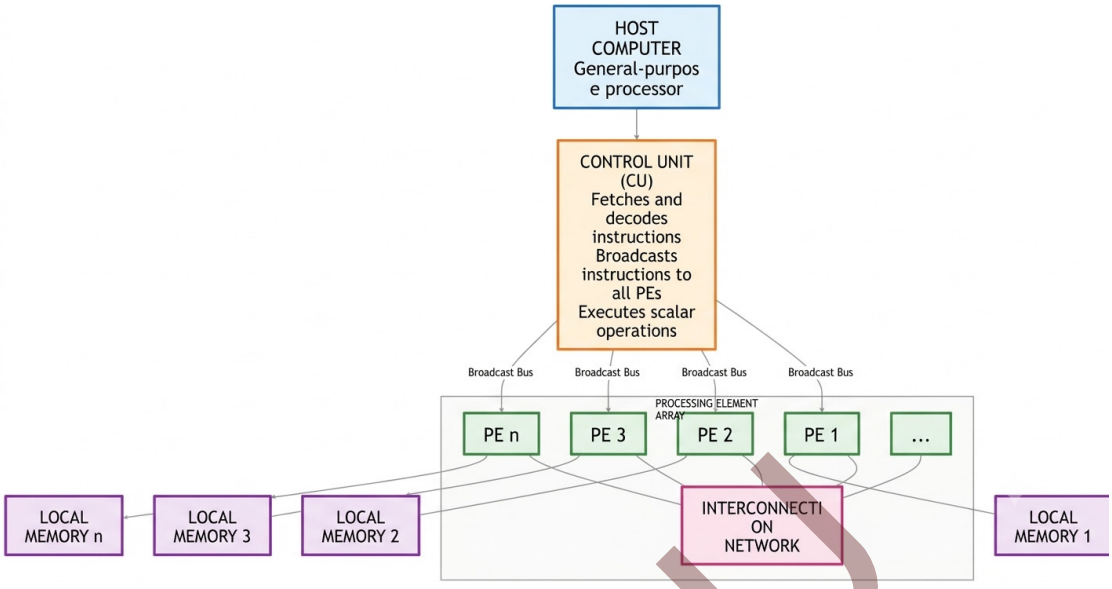


Fig. 3.1.2 Basic architecture of an array processor

Components:

1. **Host Computer:**
   - ♦ A general-purpose computer that loads programs and data into the array processor.
   - ♦ Handles I/O, operating system tasks, and any inherently scalar portions of the computation.
   - ♦ Transfers data to/from the array processor's local memories.
2. **Control Unit (CU):**
   - ♦ Fetches and decodes instructions from memory.
   - ♦ Broadcasts the decoded control signals to all PEs via a broadcast bus.
   - ♦ Executes scalar (non-parallel) instructions itself, since it has its own ALU.
   - ♦ Manages program flow: branches, loops, and subroutine calls. See fig. 3.1.2.
3. **Processing Elements (PEs):**
   - ♦ Each PE is a simple arithmetic-logic unit with local registers.
   - ♦ All active PEs execute the broadcast instruction simultaneously.
   - ♦ Each PE operates on data in its own local memory.

SGOU

4. **Local Memories:**

   - In the array-processor organization discussed here, each PE is associated with local memory containing its portion of the data. This distributed organization reduces contention for a single shared memory.

5. **Interconnection Network:**

   - Connects PEs, allowing data exchange (e.g., shift, rotate, neighbor communication).
   - Critical for algorithms requiring inter-PE communication (covered in Unit 2).

3.1.3 Control Unit and Processing Elements

3.1.3.1 The Control Unit (CU)

The CU is the brain of the array processor. It executes the instruction stream sequentially:

1. **Fetch:** Read the next instruction from the CU's instruction memory.
2. **Decode:** Determine the operation and which PEs should execute it.
3. **Broadcast:** Send the operation to all PEs (or a subset via masking).
4. **Execute (scalar):** If the instruction is a scalar operation (e.g., loop counter arithmetic), the CU executes it itself using its own ALU and registers.
5. **Control Flow:** Handle branches, loops, and subroutine calls. These execute only in the CU — the PEs are oblivious to program control flow.

3.1.3.2 Processing Elements (PEs)

Each PE consists of:

- An ALU (arithmetic-logic unit) for data operations.
- A set of local registers (accumulator, general-purpose).
- A mask (1 bit) that determines active/idle status.
- A connection to the local memory.
- A connection to the interconnection network (for PE-to-PE communication).

Lockstep Execution: All active PEs execute the same instruction in the same clock cycle. There is no concept of individual PEs executing different instructions — that would be MIMD. This lockstep synchronization is enforced by the hardware: every active PE receives the broadcast instruction simultaneously.

SGOU

Masking: For conditional execution (e.g., "if data[i] > 0, then add 1"), the CU cannot branch per-PE. Instead:

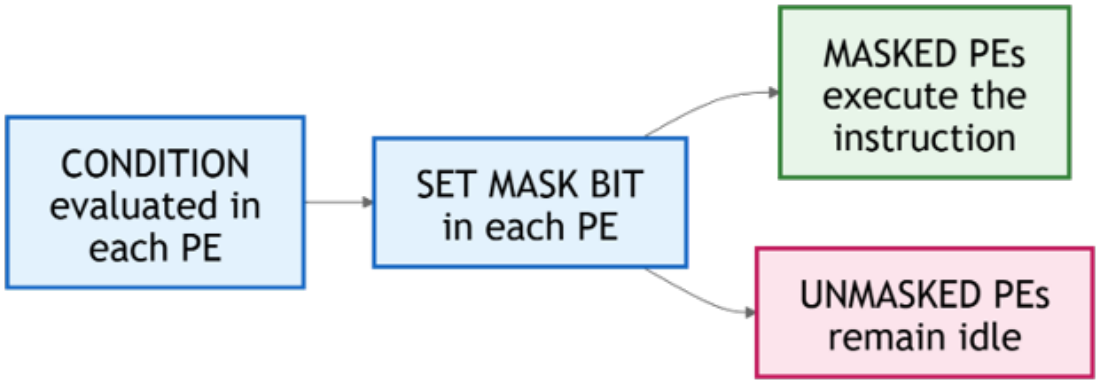


Fig. 3.1.3 Masking of processing elements in SIMD execution

1. The CU broadcasts a comparison instruction (e.g., "compare your data to 0").
2. Each PE performs the comparison locally and sets its mask bit (1 if condition true, 0 if false).
3. The CU broadcasts the conditional operation (e.g., "add 1").
4. Only PEs with mask = 1 execute; others idle.

This preserves the SIMD model (one instruction stream) while supporting data-dependent execution. However, when PEs are masked off, they waste cycles — a key limitation of SIMD for irregular/divergent computations. See fig. 3.1.3

Example — Vector Addition on an Array Processor:

Add two arrays A and B, each of 1,024 elements, on a 64-PE array processor:

1. Host loads A[0:15] into PE1's memory, A[16:31] into PE2's memory, etc. (each PE gets 16 elements of A and 16 of B).
2. CU broadcasts: "LOAD A element into accumulator."
   - ♦ Each PE loads its first A element (all 64 PEs simultaneously).
3. CU broadcasts: "ADD B element to accumulator."
   - ♦ Each PE adds its first B element.
4. CU broadcasts: "STORE accumulator to C."
   - ♦ Each PE stores its result.
5. Repeat steps 2-4 for the remaining 15 elements per PE.

   Total: 3 × 16 = 48 broadcast instructions for 1,024 additions.

SGOU

Solved Example 1.1: Compare the instruction count for adding two arrays of 1,024 elements on: (a) a scalar processor, (b) a 64-PE array processor.

Solution:

1. **Scalar:** For each element: LOAD A, ADD B, STORE C, plus loop overhead (increment, compare, branch) ≈ 6-7 instructions × 1,024 = ~6,500-7,000 instructions.

2. **64-PE array:** 1,024/64 = 16 elements per PE. Each element: LOAD, ADD, STORE = 3 instructions. Total = 3 × 16 = 48 instructions (plus loop overhead in CU, negligible).

- **Instruction reduction:** Using the lower estimate of 6,500 scalar instructions, the instruction-count reduction is approximately 135×.

3.1.4 Applications of SIMD Array Processors

SIMD array processors excel in applications with regular, data-parallel structure:

1. **Image Processing:** Every pixel of an image undergoes the same operation (filtering, contrast adjustment, edge detection). A 4K image (8 million pixels) maps perfectly to an array of PEs, each handling a region of pixels. Operations like convolution, thresholding, and morphological operations are inherently SIMD.

2. **Matrix Operations:** Matrix addition, subtraction, and multiplication involve element-wise and row/column operations that map naturally to SIMD. Dense linear algebra is a cornerstone of scientific computing and is highly amenable to array processing.

3. **Scientific Simulation (Regular Grids):** Finite-difference methods for solving PDEs (heat equation, fluid dynamics, electromagnetic fields) operate on regular grids where each grid point updates based on its neighbors. Array processors apply the same update rule to all grid points simultaneously.

4. **Signal Processing:** Digital filtering, FFT, and convolution operate on large arrays with uniform operations. SIMD accelerators are used in radar, sonar, and communication systems.

5. **Sorting and Searching (parallel algorithms):** SIMD algorithms exist for sorting, searching, and reduction operations (covered in Unit 3).

6. **Graphics and GPU Computing:** Modern GPUs are the most successful descendants of SIMD array processors. Modern GPUs use the SIMT (Single Instruction, Multiple Threads) execution model. Threads in a warp are commonly executed together under a shared instruction stream, making SIMT similar to SIMD in several respects, although the programming and execution models are not identical.

SGOU

3.1.5 Limitations of SIMD Array Processors

- **Control divergence:** If different PEs need to do different things (e.g., if-else with data-dependent conditions), masking wastes cycles.
- **Irregular data:** Sparse matrices, graph algorithms, and pointer-chasing applications map poorly to SIMD.
- **Communication-bound problems:** Algorithms requiring frequent, complex PE-to-PE communication may be limited by the interconnection network.

Historical: ILLIAC IV — An Early Large-Scale Array Processor

The ILLIAC IV, built at the University of Illinois in the early 1970s, was one of the earliest large-scale array processors and an important landmark in the development of SIMD computing. Designed by Daniel Slotnick's team, it featured 64 PEs arranged in an 8×8 grid, each with 2K 64-bit words of local memory. It achieved a peak performance of about 100 MFLOPS — unmatched for its era. The project was famously over-budget and delayed, but it pioneered the SIMD model and influenced the design of later machines like the Connection Machine. Its lessons — about the difficulty of mapping real problems to SIMD — still inform parallel computing today.

The Connection Machine CM-2 (1987)

Thinking Machines' CM-2 pushed SIMD to an extreme: up to 65,536 single-bit processing elements. Rather than being a liability, the tiny PEs (1-bit ALUs) could be combined to emulate wider operations. The CM-2's hypercube interconnection allowed flexible PE-to-PE communication. It was used for everything from VLSI simulation to artificial intelligence, and its iconic blinking-red-light cube became a symbol of 1980s supercomputing.

SGOU

Recap

- SIMD executes a single instruction stream on multiple data elements simultaneously, one control unit, many processing elements.
- Array processor structure: host computer, control unit (CU), an array of processing elements (PEs), local memories,
- and an interconnection network.
- The CU fetches, decodes, and broadcasts instructions; handles scalar operations and control flow.

- Each PE is a simple ALU with local registers and local memory; all active PEs execute in lockstep.
- Masking enables conditional execution: PEs set a mask bit based on a comparison, and only masked PEs execute the subsequent operation.
- SIMD dramatically reduces instruction fetch bandwidth — one instruction controls operations on many data elements.
- Applications: image processing, matrix operations, scientific simulation, signal processing, sorting/searching, and
- GPU computing.
- Limitations: control divergence (masking wastes cycles), irregular data, and communication-bound problems.
- ILLIAC IV and Connection Machine CM-2 are historical landmarks in SIMD array processor design.

Objective Type Questions

1. What does SIMD stand for in Flynn's taxonomy?
2. Which component of an array processor fetches and broadcasts instructions to all processing elements?
3. What is the term for all active PEs executing the same instruction simultaneously?
4. What control bit determines whether a processing element participates in a conditional SIMD operation?
5. What does each PE contain to perform arithmetic and logical operations?
6. In an array processor, what memory is attached to each individual PE?
7. How many processing elements did the ILLIAC IV feature?
8. What is the primary benefit of SIMD in terms of instruction bandwidth?
9. What happens to PEs that are masked off during a conditional operation?
10. Which modern computing device is the most widespread descendant of SIMD array processors?

SGOU

Answers

1. Single Instruction, Multiple Data.
2. The Control Unit (CU).
3. Lockstep execution.
4. The mask bit.
5. An ALU (Arithmetic-Logic Unit).
6. Local memory.
7. 64 processing elements.
8. One instruction controls operations on many data elements, reducing fetch bandwidth.
9. They remain idle (wasting a cycle).
10. The GPU (Graphics Processing Unit).

Assignments

1. Draw a detailed diagram of an array processor with 16 PEs arranged in a 4×4 configuration. Label the host computer, control unit, PEs, local memories, and interconnection network. Explain how vector addition would execute on this machine.
2. A SIMD array processor contains 64 PEs and must process 4,096 data elements. Determine the number of elements assigned to each PE. If each element requires three arithmetic instructions, explain how SIMD execution reduces the number of instruction broadcasts.
3. Compare the SIMD array-processor model with NVIDIA's SIMT GPU model. Explain the similarities and differences in instruction execution, thread/PE organization, and handling of control divergence.
4. Consider an array of 16 elements where only the elements satisfying $A[i] > 50$ should be increased by 10. Explain how masking can be used to implement this operation without separate control-flow instructions for each PE.
5. Explain why an interconnection network is required in an array processor. Compare nearest-neighbor communication with more flexible routing and discuss their effect on performance.

SGOU

6. Develop a SIMD-style algorithm for finding the maximum value in an array. Explain how comparisons and reductions can be performed across multiple processing elements.

7. Select one application such as image processing, matrix computation, scientific simulation, signal processing, or database searching. Explain why its computational pattern is suitable for SIMD array processing.

8. Research the Connection Machine CM-2. Describe its 1-bit processing elements, interconnection network, and method of combining multiple PEs to support wider computations. Discuss one application for which this architecture was suitable.

Reference

1. Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE transactions on computers*, *100*(9), 948-960.

2. Hwang, K., & Briggs, F. A. (1990). *Computer Architecture and Parallel Processing*. McGraw-Hill.

3. Duncan, R. (1990). A survey of parallel computer architectures. *Computer, 23*(2), 5–16.

4. Slotnick, D. L., Borck, J. B., & McReynolds, R. C. (1962). The SOLOMON computer. *Proceedings of the AFIPS Eastern Joint Computer Conference*, 22, 97–107.

Suggested Reading

1. Hwang, K., & Jotwani, N. (1993). Advanced computer architecture: parallelism, scalability, programmability (Vol. 199). New York: McGraw-Hill.

2. Hillis, W. D. (1985). *The Connection Machine*. MIT Press.

3. Kumar, V., Grama, A., Gupta, A., & Karypis, G. (1994). *Introduction to parallel computing* (Vol. 110). Redwood City, CA: Benjamin/Cummings.

SGOU



SIMD Interconnection Networks

Learning Outcomes

After completion of this unit, learner will be able to:

- classify interconnection networks used in SIMD systems
- distinguish between static and dynamic networks with concrete examples
- explain data routing mechanisms including routing functions, switching, and broadcasting
- evaluate performance metrics of interconnection networks: diameter, degree, bisection bandwidth, and latency
- select appropriate networks for specific SIMD communication patterns

Prerequisite

Imagine a SIMD system in which many processing elements work together to solve a problem by executing the same instruction on different data items. Although these processing elements perform computations simultaneously, they often need to exchange data, share results, and communicate with their neighbours to complete a task efficiently. How does data travel from one processing element to another? Can every processing element communicate directly with every other, or must the data pass through intermediate elements?

A basic understanding of SIMD architecture, processing elements (PEs), and the role of the control unit in coordinating parallel operations provides a foundation for exploring these questions. Familiarity with communication operations such as shifting, broadcasting, and reduction further helps explain how processing elements exchange data. This knowledge leads to an understanding of how interconnection networks organize communication, how routing techniques guide data through the network, and how network design influences the speed, cost, and overall performance of parallel computing systems.

SGOU

Keywords

Interconnection Network, Static Network, Dynamic Network, Degree, Diameter, Bisection Width, Data Routing, Routing Function

Discussion

3.2.1 Network Types

An interconnection network enables communication and data exchange between processing elements in a SIMD (Single Instruction, Multiple Data) architecture. Based on how connections are established between processing elements, interconnection networks are broadly classified into two types: static networks and dynamic networks.

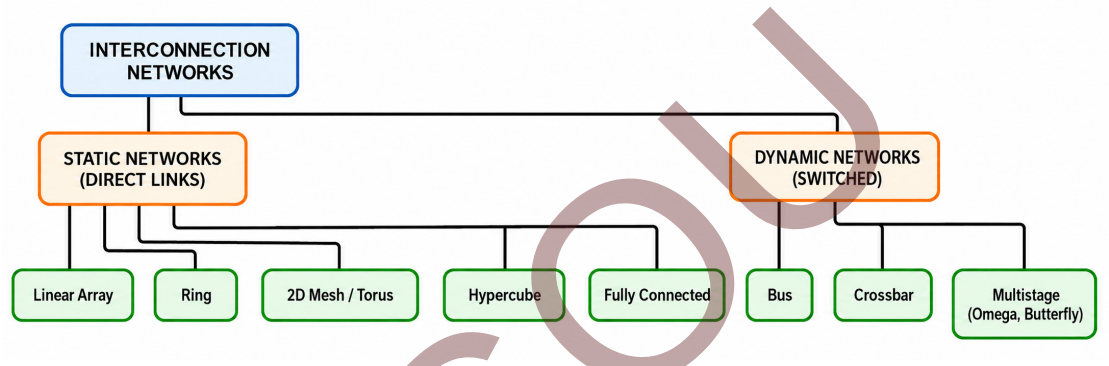


Fig. 3.2.1 Classification of interconnection networks

1. Static networks

Static networks have fixed, dedicated point-to-point links between nodes. The topology is fixed at design time. Communication between non-adjacent nodes requires routing through intermediate nodes (multi-hop).

2. Dynamic networks

Dynamic networks use Bus, Crossbar, Multistage switches that can be configured to establish temporary connections between nodes. The path between source and destination is set up dynamically for each communication.

3.2.2 Static vs. Dynamic Networks

3.2.2.1 Static Networks (Direct Interconnection)

Static networks have fixed links between nodes. The following table compares common static network topologies based on their structural characteristics and applications.

SGOU

Table 3.2.1 Comparison of Static Interconnection Network Topologies

Topology	Degree	Diameter	Bisection Width	Cost	Best For
Linear Array	Up to 2	N−1	1	Low	Simple, local communication
Ring	2	⌊N/2⌋	2	Low	Nearest-neighbor and circular communication
2D Mesh	Up to 4	2(√N−1)	√N	Medium	Nearest-neighbor communication, image and grid-based applications
2D Torus	4	√N	2√N	Medium	Nearest-neighbor communication with wrap-around
Hypercube	$\log_2 N$	$\log_2 N$	N/2	High	Many parallel algorithms and efficient data exchange
Fully Connected	N−1	1	$N^2/4$	Very High	Applications requiring direct communication between any pair of nodes

SGOU

3.2.2.2 Dynamic Networks / Indirect Network

Dynamic interconnection networks differ in their hardware requirements, communication latency, blocking characteristics, and scalability. The following table compares three common types such as bus, crossbar, and multistage interconnection networks, highlighting their key characteristics and typical applications.

Table 3.2.2 Comparison of Dynamic Interconnection Network Types

Network	Switch/ Hardware Cost	Latency	Blocking	Scalability	Typical Use
Bus	Very Low	Low (single shared medium)	High under contention	Poor	Small systems and simple shared communication

Crossbar	$O(N^2)$	Low (1 switching stage)	Non-blocking	Poor for large N due to high cost	High-performance systems requiring simultaneous connections
Multistage Interconnection Network (e.g., Omega)O(N log N)	O(N log N)	O(log N) switching stages	Blocking	Good	Scalable processor-memory and communication systems

Dynamic or indirect interconnection networks use switching elements or shared communication mechanisms to establish communication paths between processing elements. A bus provides a shared communication medium, whereas crossbar and multistage interconnection networks use explicit switching elements to establish communication paths.

Key trade-off:

- Static networks are cheaper and scale better, but non-adjacent communication requires multiple hops (higher latency).
- Dynamic networks can connect any source to any destination in fewer hops, but switches are expensive and may cause blocking/contention.

3.2.3 Data Routing

Data routing determines how a data item travels from a source PE to a destination PE through the network.

Routing Functions:

A routing function maps each (source, destination) pair to a path. Common routing functions:

1. **Dimension-Ordered Routing (for mesh/hypercube):** Route along dimensions one at a time, in a fixed order. For a 2D mesh: first route along X, then along Y (XY routing). Simple, deadlock-free, but may create congestion.
2. **Shortest-Path Routing:** Always route along a minimum-length path. Minimizes latency but requires global knowledge.
3. **Adaptive Routing:** Choose the path dynamically based on network congestion. Improves throughput but risks deadlock and requires more complex control.

SGOU

Routing in Static Networks:

In a static network, data must pass through intermediate nodes. Each intermediate PE receives the data and forwards it to the next node (store-and-forward or cut-through). This is analogous to how packets traverse a network.



Fig. 3.2.2 Example of routing a message in a 2D mesh network

Routing in Dynamic Networks:

Switches are configured to establish a path. In a crossbar, any input can be connected directly to any output through a single switching stage, and independent connections can be established simultaneously, making it a non-blocking network under the standard crossbar model. In a multistage network (e.g., Omega network), the path is determined by the destination address bits, routing stage-by-stage.

Common SIMD Communication Operations:

1. **Broadcasting (One-to-All):** The CU (or a PE) sends data to all PEs. Used for distributing constants, control signals, and scalar values.
2. **Permutation (One-to-One):** Each source PE sends data to exactly one destination PE. Used in matrix transpose, bit-reversal (FFT).
3. **Shift (Nearest-Neighbor):** Each PE sends to its immediate neighbor (left, right, up, down). Used in image filtering, stencil computations.
4. **Reduction (All-to-One):** All PEs contribute to a single result (sum, max, min). Implemented via tree-based routing.
5. **All-to-All:** Every PE communicates with every other PE. Used in matrix transpose, FFT.

Solved Example 1: In a 4×4 mesh (16 PEs), a data item needs to travel from PE at (0,0) to PE at (3,3) using XY routing. How many hops?

Solution

- XY routing: first move along X (east) from column 0 to column 3: 3 hops. Then move along Y (north) from row 0 to row 3: 3 hops.
- Total: 3 + 3 = 6 hops.
- This matches the diameter of a 4×4 mesh: $2(\sqrt{16} - 1) = 2(4-1) = 6$.

SGOU

3.2.4 Performance Considerations in Interconnects

Evaluating interconnection networks requires understanding several key metrics:

1. **Degree (Node Degree):** The number of links directly connected to a node. A higher degree means more wiring and higher cost, but also more direct communication paths. The degree is a measure of local communication capability and hardware cost.

2. **Diameter:** The maximum shortest-path distance between any two nodes. A smaller diameter means lower worst-case latency. The diameter grows with network size, but at different rates for different topologies:

    - Mesh: diameter grows as $\sqrt{N}$
    - Hypercube: diameter grows as log N
    - Fully connected: diameter is always 1

3. **Bisection Width:** The minimum number of links that must be removed to split the network into two equal halves. A larger bisection width generally indicates greater potential aggregate communication bandwidth when link bandwidths are equal.

4. **Latency:** The time for a message to travel from source to destination. Components:

    - **Startup latency:** Time to prepare the message at the source (fixed overhead).
    - **Per-hop latency:** Time to traverse each intermediate node/switch.
    - **Bandwidth component:** Message size / link bandwidth.

A simplified communication-time model can be written as:

$$T_{communication} = t_{startup} + (hops \times t_{hop}) + (message\ size / bandwidth)$$

5. **Throughput:** The rate at which the network can deliver messages. Limited by bisection bandwidth for collective operations, and by link bandwidth for individual transfers.

6. **Cost:** Number of links and switches. A fully connected network with N nodes has $N(N-1)/2$ links — $O(N^2)$ cost. A mesh has O(N) links. A hypercube has O(N log N) links.

7. **Fault Tolerance:** The ability to route around failed nodes/links. Networks with multiple paths between nodes (hypercube, torus) are more fault-tolerant than those with unique paths (linear array).

SGOU

Solved Example 2: Compare the diameter and degree of a hypercube and a 2D mesh, both with 64 nodes.

Solution

- **Hypercube (N=64):** Dimension = $\log_2(64) = 6$. Degree = 6. Diameter = 6.
- **2D Mesh (N=64):** 8×8 grid. Degree = 4 (interior nodes). Diameter = $2(8-1) = 14$.
- **Conclusion:** The hypercube has a much smaller diameter (6 vs 14) but a higher degree (6 vs 4). The hypercube provides faster worst-case communication at the cost of more links per node.

Solved Example 3: A 2D mesh and a 2D torus both have 256 nodes (16×16). What is the bisection width of each?

Solution

- **2D Mesh (16×16):** The minimum cut separates 8 columns from 8 columns, cutting 16 vertical links (one per row).

  Bisection width = 16.

- **2D Torus (16×16):** The torus has wrap-around links. Cutting the middle still cuts 16 links, but now there are 2 rings in each dimension, so the bisection must cut $2 \times 16 = 32$ links.

  Bisection width = 32.

- **Conclusion:** The torus has twice the bisection width of the mesh, providing greater potential aggregate bandwidth across the bisection when link bandwidths are equal.

Performance Trade-offs Summary:

The performance of an interconnection network depends on several factors, including connectivity, communication delay, bandwidth, hardware cost, and fault tolerance. The following table summarizes the major performance metrics and their corresponding design goals.

Table 3.2.3 Performance Metrics and Design Trade-offs in Interconnection Networks

Metric	Impact	Design Goal
Degree	Cost, local connectivity	Minimize for cost
Diameter	Worst-case latency	Minimize for speed
Bisection Bandwidth	Sustained throughput	Maximize for bandwidth
Cost (links/switches)	Hardware expense	Minimize
Fault Tolerance	Reliability	Maximize (multiple paths)

SGOU

The ideal network would have low degree (cheap), low diameter (fast), high bisection bandwidth (high throughput) but these are in tension. The hypercube achieves low diameter and high bandwidth at moderate degree, making it popular for SIMD systems. The mesh/torus offers lower degree at the cost of higher diameter, preferred when wiring cost dominates (e.g., VLSI layouts).

The Hypercube : A Powerful Idea

The hypercube (also called a binary n-cube) is an elegant topology: an n-dimensional hypercube has 2^n nodes, each connected to n neighbors. Node addresses differ by exactly one bit from each neighbor's address. This bit-addressing property makes routing trivial to send data, simply "flip" bits from source to destination address. The hypercube was the interconnection network of choice for many 1980s SIMD and MIMD machines, including the Intel iPSC, nCUBE, and the Connection Machine CM-2. Its balanced degree and logarithmic diameter made it an influential topology in the design and study of parallel interconnection networks.

The Omega Network : Routing by Bits

The Omega network is a multistage interconnection network that routes data based on the binary address of the destination. At each stage, a switch examines one bit of the destination address and routes accordingly. This "self-routing" property means no central controller is needed — each switch makes an independent local decision. The Omega network connects N inputs to N outputs with O(N log N) switches and O(log N) latency, making it a cost-effective alternative to the O(N^2) crossbar. However, it is a blocking network: certain permutations cannot be routed simultaneously without conflict.

SGOU

Recap

- Interconnection networks connect PEs in SIMD systems and static/direct (fixed links) or dynamic/indirect (dynamically established or shared paths).
- Static topologies: linear array, ring, 2D mesh, 2D torus, hypercube, fully connected. Each has a characteristic degree, diameter, and bisection width.
- Dynamic topologies: bus (simple, high contention), crossbar (non-blocking, O(N^2) cost), multistage (Omega/Butterfly, O(N log N) cost, blocking).
- Data routing: the path data takes from source to destination. Common functions: dimension-ordered (XY), shortest-path, adaptive.
- Common SIMD communication: broadcast (one-to-all), permutation (one-to-one), shift (nearest-neighbor), reduction (all-to-one), all-to-all.

- Performance metrics: degree (links per node), diameter (max hops), bisection width (minimum number of links in a bisection); bisection bandwidth (aggregate bandwidth across the bisection), latency (startup + hops + bandwidth), throughput, cost, fault tolerance.
- Communication time: T = t_startup + hops × t_hop + message_size/ bandwidth.
- Trade-off: low degree (cheap) vs. low diameter (fast) vs. high bisection bandwidth (high throughput). Hypercube balances these well; mesh/torus is cheaper in wiring.
- Hypercube routing is trivial via bit-flipping; Omega networks use self-routing via destination address bits.

Objective Type Questions

1. What type of network has fixed, dedicated point-to-point links between nodes?
2. In an 8-node hypercube, what are the dimension and degree?
3. What metric measures the maximum shortest-path distance between any two nodes?
4. What metric measures the minimum number of links that must be cut to split a network into two equal halves?
5. In XY routing on a 2D mesh, which dimension is routed first?
6. What type of network uses O(N^2) switching hardware and provides non-blocking connections?
7. What is the diameter of a 16-node hypercube?
8. What is the degree of an interior node in a 2D mesh?
9. Which communication operation sends data from one node to all nodes?
10. In the communication-time formula, what does $t_{_startup}$ represent?

SGOU

Answers

1. Static network.
2. Dimension 3 and degree 3.
3. Diameter.
4. Bisection width.
5. The X dimension.
6. Crossbar network.
7. 4.
8. 4.
9. Broadcasting.
10. The fixed overhead required to prepare a message at the source.

Assignments

1. Compare the diameter and bisection width of an 8-node hypercube, an 8-node ring, and an 8-node fully connected network. Which is best for (a) worst-case latency and (b) sustained throughput? Which is cheapest?

2. A message of 1,000 bytes is sent between two PEs that are 5 hops apart. t_startup = 100 ns, t_hop = 50 ns, bandwidth = 1 GB/s. Calculate the total communication time.

3. Explain why a crossbar switch is non-blocking but a multistage Omega network is blocking. Give an example of a permutation that the Omega network cannot route simultaneously.

4. For a 64-PE SIMD system, compute the degree, diameter, and bisection width of an 8×8 mesh, an 8×8 torus, and a 6-dimensional hypercube. Present the results in a table and recommend the most suitable topology for image processing involving nearest-neighbor communication.

5. In a 4×4 mesh using XY routing, trace the path for data from (1,2) to (3,0). Determine the number of hops. Explain how the corresponding communication can be performed in a hypercube.

SGOU

6. Compare a bus, crossbar, and Omega network with respect to cost, latency, blocking, scalability, and communication capability. Present your comparison in a table.

7. Explain dimension-ordered routing, shortest-path routing, and adaptive routing. Give one advantage and one limitation of each.

Reference

1. Hwang, K., & Briggs, F. A. (1990). *Computer architecture and parallel processing*. McGraw-Hill.

2. Dally, W. J., & Towles, B. P. (2004). P*rinciples and practices of interconnection networks*. Elsevier.

3. Duato, J., Yalamanchili, S., & Ni, L. M. (2002). *Interconnection networks: An engineering approach*. Morgan Kaufmann.

4. Leighton, F. T. (2014). *Introduction to parallel algorithms and architectures: Arrays, trees, hypercubes*. Elsevier.

Suggested Reading

1. Kumar, V., Grama, A., Gupta, A., & Karypis, G. (1994). *Introduction to parallel computing* (Vol. 110). Redwood City, CA: Benjamin/Cummings.

2. Culler, D., Singh, J. P., & Gupta, A. (1998). *Parallel computer architecture* (Vol. 8). San Francisco, CA: Morgan Kaufmann.

3. Quinn, M. J. (2003). *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group.

SGOU



Parallel Algorithms for Array Processors

Learning Outcomes

After completion of this unit, learner will be able to:

- explain the principles of parallel algorithm design for SIMD systems
- design parallel algorithms for addition and multiplication using array processors
- analyze matrix operations (addition, multiplication, transpose) on array processors
- describe sorting and searching algorithms suited to SIMD architectures
- evaluate algorithm complexity in terms of parallel time and PE utilization

Prerequisite

Parallel algorithms differ fundamentally from their sequential counterparts. In sequential algorithms, the goal is to minimize the total number of operations. In parallel algorithms, the goals are: (1) minimize the number of parallel steps (time), and (2) maximize processor utilization. Some problems that are trivial sequentially (e.g., summing numbers) require careful design in parallel to achieve good performance. The interconnection network (Unit 2) directly constrains which algorithms are efficient.

Consider summing 100 numbers. Sequentially, you add them one by one, 99 additions. But imagine you have 50 people: each adds a pair of numbers simultaneously (50 additions in parallel), then 25 people add those results, then 13 people (approximately), and so on. This tree-based approach reduces the number of sequential steps from 99 to about 7 ($\log_2 100$). This is the essence of parallel algorithm design: restructuring computation to expose parallelism and reduce the critical path (the longest chain of dependent operations).

Keywords

Parallel Time, Work, Speedup, Efficiency, Critical Path, Reduction, Broadcast, Compare-Exchange, Sorting Network

SGOU

Discussion

3.3.1 Basics of Parallel Algorithm Design

Designing efficient parallel algorithms for array processors follows several principles:

1. **Decompose the Problem:** Break the problem into independent subproblems that can execute concurrently. Data decomposition (partition data among PEs) is natural for SIMD; functional decomposition (partition work by task) is less common in SIMD.

2. **Minimize the Critical Path:** The critical path is the longest sequence of dependent operations. Since all PEs execute in lockstep, the parallel time is determined by the critical path. Reducing dependencies (e.g., using tree-based reduction instead of sequential accumulation) shortens the critical path.

3. **Maximize PE Utilization:** Keep all PEs busy as much as possible. If PEs are idle (e.g., due to masking or load imbalance), efficiency drops. Aim for regular, uniform workloads.

4. **Minimize Communication:** Communication (inter-PE data transfer) is often more expensive than computation. Design algorithms to minimize data movement, and match communication patterns to the interconnection network topology.

5. **Consider the Architecture:** The efficiency of a parallel algorithm depends on the specific array processor architecture (number of PEs, interconnection network, memory access). An algorithm optimal for one topology may be poor for another.

Performance Metrics

- **Parallel Time (T_p):** Number of parallel steps. Lower is better.
- **Work (W):** The total number of operations performed by all PEs during the computation. If all PP PEs remain active for T_p parallel steps, the processor-time cost is T_pP. Processor-time cost is not necessarily equal to work because some PE cycles may involve waiting, communication, or idle time.
- **Speedup (S):** T_1 / T_p, where T_1 is the best sequential time.
- **Efficiency (E):** $E = \frac{S}{P} = \frac{T_1}{T_p P}$ = $Work_{best} / Work_{parallel}$.

  Measures how close to perfect linear speedup.
- **Cost-Optimality:** An algorithm is cost-optimal if $Work_{parallel} = O(T_1)$

SGOU

i.e., the parallel algorithm does not do asymptotically more total work than the best sequential algorithm.

3.3.2 Parallel Addition

Adding n numbers is the fundamental reduction operation. Two approaches:

Approach 1: Sequential (poor parallel)

```
sum = 0
for i = 1 to n:
        sum = sum + a[i]
```

Parallel time: O(n) - no parallelism exploited.

Approach 2: Tree-Based (Binary Tree Reduction)

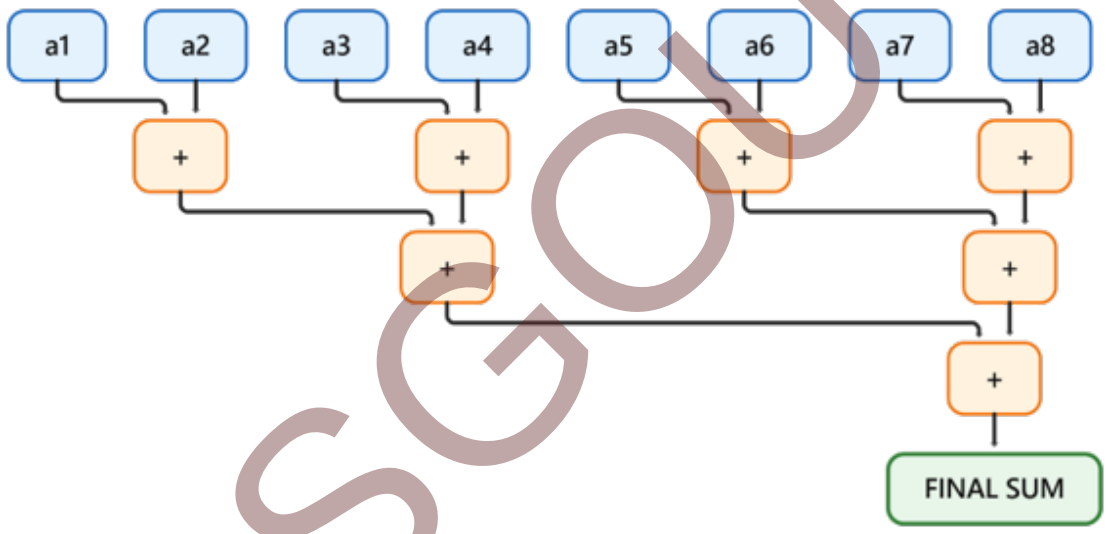


Fig. 3.3.1 Tree-based parallel reduction for summing eight elements

1. Step 1: n/2 PEs each add two numbers → n/2 partial sums.
2. Step 2: n/4 PEs each add two partial sums → n/4 partial sums.

   ...continue...

3. The reduction requires approximately log2n parallel steps.
4. Parallel time: O(log n) steps, using n/2 PEs.

SGOU

Solved Example 1: Sum 8 numbers using a tree-based reduction. Show the steps and count the parallel additions.

Solution

1. Step 1: 4 additions (a1+a2, a3+a4, a5+a6, a7+a8) → 4 partial sums.
2. Step 2: 2 additions (s1+s2, s3+s4) → 2 partial sums.
3. Step 3: 1 addition (r1+r2) → final sum.

   Total: 3 parallel steps ($\log_2 8 = 3$), 7 additions total.

   Sequential: 7 additions in 7 steps.

   Speedup: 7/3 ≈ 2.33× (using 4 PEs).

3.3.3 Parallel Multiplication

Parallel multiplication can be approached in several ways:

1. **Matrix-Vector Multiplication:** Multiply an n×n matrix A by an n-vector x to produce vector y. Each element $y[i] = \sum_{j=0}^{n-1} A[i][j] \times x[j]$

   On an array processor, each PE computes one y[i] by performing a dot product, a reduction. The multiplication of A[i][j] × x[j] for all j can be done in parallel (element-wise), followed by a tree-reduction to sum the products.

2. **Matrix-Matrix Multiplication:** C = A × B, where A, B, C are n×n matrices. Each C[i][j] = Σ A[i][k] × B[k][j]. This is n^2 independent dot products, each requiring n multiple-add operations. Parallel algorithms distribute these dot products across PEs.

3. **Systolic array approach:** A systolic array for matrix multiplication arranges PEs in a 2D grid, with data flowing rhythmically through the array. Each PE accumulates a partial sum as data flows through it.

4. **Integer Multiplication via Shifting:** For SIMD, multiplication can be decomposed into repeated addition and shifting. If the hardware multiplier is limited, software routines using shift-and-add (for binary multiplication) can be vectorized across PEs.

3.3.4 Matrix Operations Using Array Processors

Matrix operations are the bread-and-butter of array processors.

1. **Matrix Addition (C = A + B):** Trivially parallel. Each PE holds corresponding elements A[i][j] and B[i][j] and computes C[i][j] = A[i][j] + B[i][j] in one step.

SGOU

- **Parallel time:** O(1) for element-wise, if n^2 PEs available.
- **With n PEs (one row each):** O(n) steps (each PE processes n columns).

2. Matrix Multiplication (C = A × B):

The classic algorithm:

```
C[i][j] = Σ(k=0 to n-1) A[i][k] × B[k][j]
```

3. Parallel approach with n^2 PEs (2D array processor):

1. Distribute A and B so that PE(i,j) holds A[i][j] and B[i][j].
2. For k = 0 to n-1:
   - Each PE(i,j) computes A[i][k] × B[k][j] and accumulates.
   - At each iteration kk, the required elements A[i][k] and B[k][j] are made available to PE(i,j), which computes their product and adds it to its partial sum.
   - Parallel time: O(n) steps under this n^2- PE broadcast-based model.
   - Parallel time: O(n) steps (n iterations, each with a broadcast and multiply-add).

4. Systolic Array for Matrix Multiplication: A more elegant approach uses a 2D systolic array where data flows through PEs:

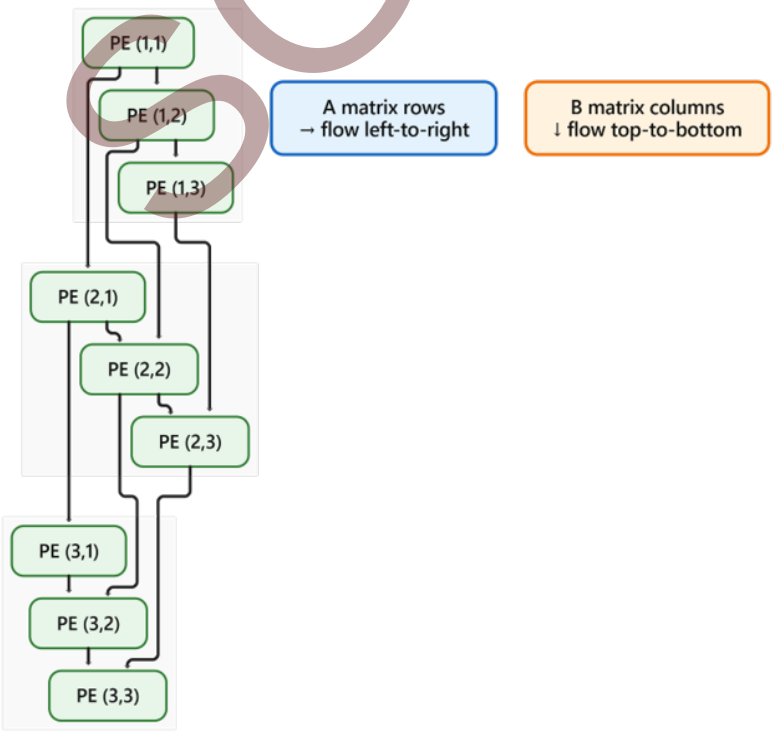


Fig. 3.3.2 2D systolic array for matrix multiplication

SGOU

Each PE(i,j) receives A elements from the left and B elements from the top, multiplies them, and accumulates the result. After the data flows through, C[i][j] is computed. The systolic array achieves O(n) time with O(n^2) PEs, with all communication being local (nearest-neighbor), ideal for VLSI implementation.

5. **Matrix Transpose:** Transposing an n×n matrix A into A^T requires moving A[i][j] to position (j,i). On a 2D mesh, this is a permutation operation. The efficiency depends on the interconnection network:

   - On a 2D mesh: elements move along rows/columns; the transpose takes O(n) time.
   - On a hypercube: the transpose maps to a bit-permutation (swap row and column bits), taking O(log n) time.

Solved Example 2: Multiply two 2×2 matrices A = [[1,2],[3,4]] and B = [[5,6],[7,8]] using 4 PEs (one per element of C). Show the steps.

Solution

- PE(1,1) computes C[1][1] = 1×5 + 2×7 = 5 + 14 = 19.
- PE(1,2) computes C[1][2] = 1×6 + 2×8 = 6 + 16 = 22.
- PE(2,1) computes C[2][1] = 3×5 + 4×7 = 15 + 28 = 43.
- PE(2,2) computes C[2][2] = 3×6 + 4×8 = 18 + 32 = 50.
- Result: C = [[19, 22], [43, 50]].
- Each PE performs a dot product (2 multiplications + 1 addition), all in parallel.

3.3.5 Sorting and Searching Algorithms for SIMD Systems

3.3.5.1 Sorting Networks

A sorting network is a fixed arrangement of comparators (compare-exchange elements) that sorts any input sequence. The network structure is data-independent the same comparisons are performed regardless of input values. This makes sorting networks ideal for SIMD, since there are no data-dependent branches.

1. **Bitonic Sort:** A bitonic sequence first increases then decreases (or vice versa). The bitonic sorter recursively merges bitonic sequences to produce a sorted sequence.

SGOU

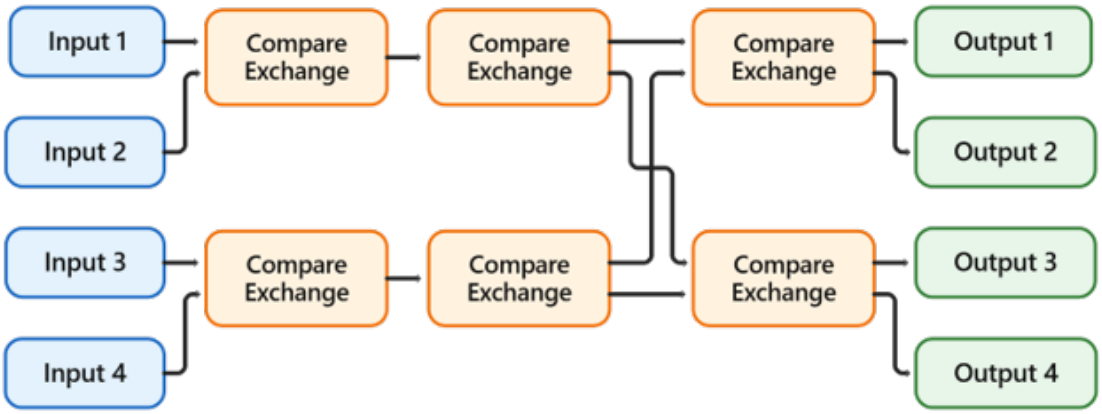


Fig. 3.3.3 Basic structure of a bitonic sorting network

Complexity: Bitonic sort has $O(\log^2 n)$ parallel time using $O(n \log^2 n)$ comparators. It is a classic SIMD sorting algorithm.

2. **Odd-Even Transposition Sort:** A simple SIMD sorting algorithm that alternates between comparing odd-even and even-odd pairs of adjacent elements:

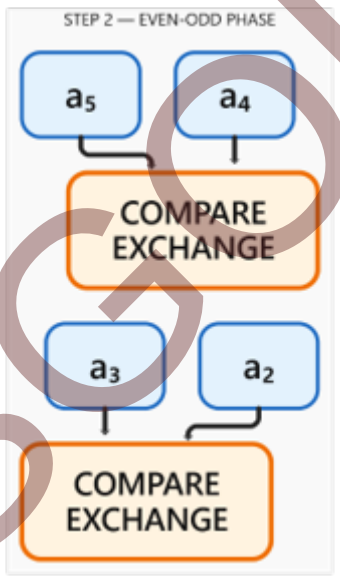


Fig. 3.3.4 Even-odd phase of odd-even transposition sorting

- **Step 1:** Compare-exchange (1,2), (3,4), (5,6), ...
- **Step 2:** Compare-exchange (2,3), (4,5), (6,7), ...
  - Repeat until sorted.
  - **Complexity:** O(n) steps for n elements on n PEs (each step compares n/2 pairs in parallel).

3.3.5.2 Searching on SIMD

1. **Parallel Linear Search:** Distribute the data across PEs. Each PE compares its element to the search key simultaneously. If any PE finds a match, it reports (via

SGOU

a reduction/broadcast). O(1) time to check all n elements using n PEs (though reporting the match requires a reduction).

2. **Parallel Binary Search:** When multiple search keys must be located in the same sorted data, the search operations can be distributed across PEs so that several searches proceed concurrently. Each individual binary search requires O(log n) comparisons.

3. **Associative Search (Content-Addressable):** If the array processor has associative memory (CAM, covered in Unit 4), searching is even faster. If an array processor provides associative memory (CAM), all stored entries can be compared with the search key in parallel, potentially allowing the match operation to complete in O(1) time.

Solved Example 3: Sort the sequence [3, 1, 4, 1, 5, 9, 2, 6] using odd-even transposition sort. Show the first two steps.

Solution

- Initial: [3, 1, 4, 1, 5, 9, 2, 6]
- Step 1 : (odd-even, pairs (1,2),(3,4),(5,6),(7,8))
  - Compare (3,1): swap → 1,3
  - Compare (4,1): swap → 1,4
  - Compare (5,9): no swap
  - Compare (2,6): no swap
  - Result: [1, 3, 1, 4, 5, 9, 2, 6]
- Step 2 : (even-odd, pairs (2,3),(4,5),(6,7))
  - Compare (3,1): swap → 1,3
  - Compare (4,5): no swap
  - Compare (9,2): swap → 2,9
  - Result: [1, 1, 3, 4, 5, 2, 9, 6]
- Continue alternating until sorted. Final: [1, 1, 2, 3, 4, 5, 6, 9].

Batcher's Sorting Networks

Kenneth Batcher, working at Goodyear Aerospace in the 1960s, developed bitonic sort and odd-even merge sort, both based on sorting networks. These algorithms were specifically motivated by SIMD architectures like the Goodyear MPP (Massively Parallel Processor). Batcher's insight, that a fixed, data-independent network of comparators can sort any input, remains fundamental to parallel sorting and is used in modern GPU sorting (e.g., NVIDIA's bitonic sort implementations in CUDA).

SGOU

The BSP Model and Parallel Algorithm Design

Leslie Valiant's Bulk Synchronous Parallel (BSP) model (1990) provides a framework for designing and analyzing parallel algorithms. BSP views computation as a sequence of "supersteps," each consisting of local computation followed by global communication and a barrier synchronization. This abstraction maps naturally to SIMD array processors (and modern GPUs), where lockstep execution and barrier synchronization are inherent. BSP helps algorithm designers reason about communication costs and scalability independently of specific hardware.

Recap

- Parallel algorithm design principles: decompose the problem, minimize the critical path, maximize PE utilization, minimize communication, and consider the architecture.
- Performance metrics: parallel time (Tp), work (total operations performed), speedup, efficiency, and cost-optimality.
- Parallel addition via tree-based reduction: O(log n) time with n/2 PEs, vs. O(n) sequentially.
- Parallel multiplication: matrix-vector and matrix-matrix multiplication map naturally to array processors; systolic arrays use local data flow for O(n) time matrix multiplication.
- Matrix operations: addition is O(1) (element-wise), multiplication is O(n) (with n^2 PEs), transpose depends on
- interconnection (mesh: O(n), hypercube: O(log n)).
- Sorting networks: bitonic sort ($O(\log^2 n)$ time), odd-even transposition sort (O(n) time). Both are data-independent, ideal for SIMD.
- Searching: Parallel linear search can check all n elements in O(1) comparison time using n PEs, although reporting or combining matches may require additional communication. Parallel binary search requires O(log n) comparisons per search, while associative search can potentially perform the match operation in O(1) time with CAM.
- The interconnection network topology (Unit 2) directly determines which algorithms are efficient.

SGOU

Objective Type Questions

1. What is the term for the longest chain of dependent operations in a parallel algorithm?
2. How many parallel steps are required to sum n numbers using binary tree reduction?
3. What is the ratio of speedup to the number of PEs called?
4. In odd-even transposition sort, which pairs are compared in the first step?
5. What is the parallel time complexity of bitonic sort?
6. In matrix addition on an array processor with n^2 PEs, what is the parallel time?
7. What type of array uses local data flow (nearest-neighbor) for matrix multiplication?
8. What is a sorting network's key property that makes it ideal for SIMD?
9. What search operation is O(1) using content-addressable memory?
10. What does "work" measure in parallel algorithm analysis?

Answers

1. Critical path.
2. O(log2 n) parallel steps.
3. Efficiency.
4. Odd-even pairs: (1,2), (3,4), (5,6), ...
5. O(log2n) parallel time.
6. O(1).
7. Systolic array.
8. Data-independence (same comparisons regardless of input).
9. Associative search.
10. The total number of operations performed by all PEs.

SGOU

Assignments

1. Design a parallel algorithm to find the maximum of n numbers on an array processor. Show the tree-based reduction steps and analyze the parallel time and work. Compare it with sequential maximum finding.

2. Write a detailed step-by-step description of 2×2 matrix multiplication on a 2×2 systolic array. Trace the flow of data through the PEs for specific matrices. Draw the systolic array with data flow.

3. Design a tree-based parallel algorithm to add 16 numbers using 8 PEs. Show all parallel steps, calculate the total number of additions, and determine the speedup and efficiency.

4. Explain how matrix addition, matrix multiplication, and matrix transpose can be mapped onto an array processor. Compare their parallel-time complexities.

5. Compare bitonic sort and odd-even transposition sort for n elements in terms of parallel time, communication pattern, and suitability for SIMD architectures.

6. Design a parallel linear-search algorithm using n PEs. Explain how the result can be combined when multiple PEs find the search key.

7. Explain how the interconnection network affects the efficiency of a parallel matrix or sorting algorithm. Compare a 2D mesh and a hypercube.

8. Implement bitonic sort in CUDA (or use a GPU sorting library). Measure sorting time for arrays of increasing size and compare the results with a CPU sorting implementation.

SGOU

Reference

1. Hwang, K., & Jotwani, N. (1993). *Advanced computer architecture: parallelism, scalability, programmability* (Vol. 199). New York: McGraw-Hill.

2. Quinn, M. J. (2003). *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill.

3. Kumar, V., Grama, A., Gupta, A., & Karypis, G. (1994). *Introduction to parallel computing* (Vol. 110). Redwood City, CA: Benjamin/Cummings.

4. Culler, D., Singh, J. P., & Gupta, A. (1998). *Parallel computer architecture* (Vol. 8). San Francisco, CA: Morgan Kaufmann.

Suggested Reading

1. Akl, S. G., & Rheinboldt, W. (1985). *Parallel sorting algorithms* (Vol. 2). Orlando, Florida: Academic press.

2. Batcher, K. E. (1968, April). Sorting networks and their applications. In *Proceedings of the April 30--May 2, 1968, spring joint computer conference* (pp. 307-314).

3. Leighton, F. T. (2014). *Introduction to parallel algorithms and architectures: Arrays, trees, hypercubes*. Elsevier.

SGOU



Associative Array Processing

Learning Outcomes

After completion of this unit, learner will be able to:

- define associative processing and contrast it with address-based (von Neumann) memory access
- explain the structure and operation of Content Addressable Memory (CAM)
- describe the architecture of associative array processors and their search capabilities
- analyze applications of associative processing in pattern matching and database operations
- evaluate the trade-offs (cost, speed, power) of associative memory versus conventional memory

Prerequisite

Conventional memory (RAM) is address-based: you provide an address, and the memory returns the data stored at that address. Associative memory inverts this paradigm: you provide the data (or part of it), and the memory returns the address(es) where matching data is found. This content-based access is the foundation of associative processing, which is particularly powerful for search, pattern matching, and database query operations.

Think of a library catalog. A conventional (address-based) catalog: "give me the book at shelf position A-42." An associative (content-based) catalog: "find all books titled 'Parallel Computing' or authored by 'Michael Flynn'." The associative search examines all entries in parallel rather than scanning them one by one. This is exactly what Content Addressable Memory does in hardware.

Keywords

Associative Processing, Content Addressable Memory (CAM), Match Line, Mask Register, Ternary CAM (TCAM), Associative Array Processor, Parallel Search,Pattern Matching

SGOU

Discussion

3.4.1 Concept of Associative Processing

Associative processing fundamentally differs from conventional processing in how data is accessed:

1. Conventional (Address-Based) Processing

```
Address → Memory → Data
```

You know the address (location) and retrieve the data. Finding data by content requires a sequential scan of memory (or a data structure like a hash table, binary tree, etc., with their own overheads).

2. Associative (Content-Based) Processing

You provide the content (or a fragment), and the memory searches all locations in parallel, returning where matches are found. This "search by content" is the defining characteristic.

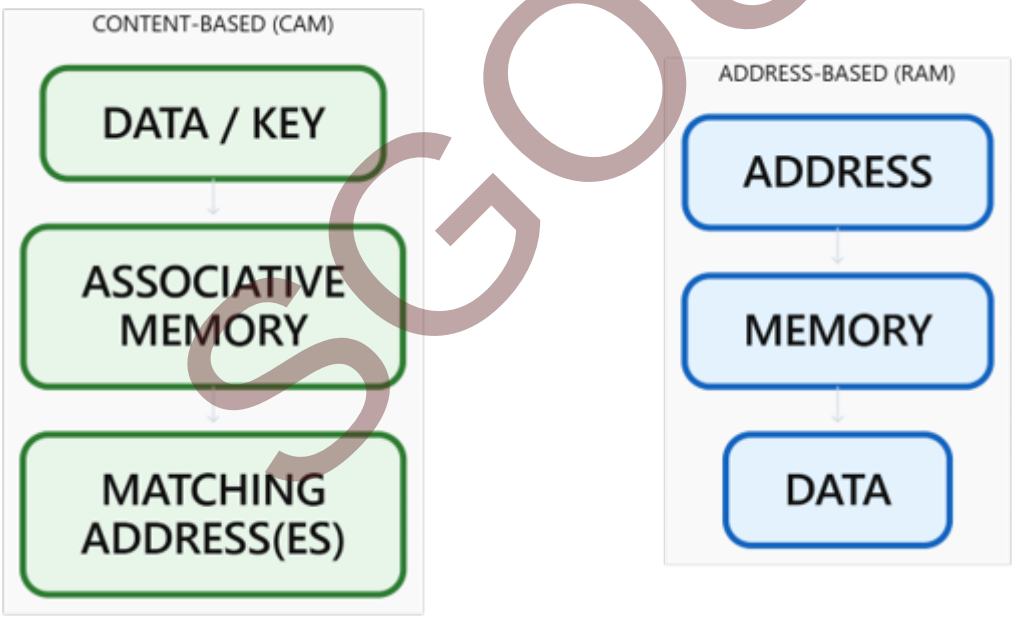


Fig. 3.4.1 Comparison of content-based (CAM) and address-based (RAM) memory access

Advantages of Associative Processing

1. Under the parallel CAM hardware model, content-based search can be completed in O(1) comparison time, because all stored words are compared with the search key simultaneously.

2. **No Separate Index Structures:** No need for hash tables, B-trees, or other indexing structures, the memory itself performs the search.

SGOU

3. **Simplicity of Programming:** Search operations are a single instruction rather than a complex algorithm.

Disadvantages

1. **High Hardware Cost:** CAM cells require additional comparison circuitry along with storage, making them larger and more expensive than conventional memory cells.

2. **High Power Consumption:** All cells compare simultaneously, consuming power proportional to the memory size.

3. **Limited Capacity:** Due to cost and power, CAMs are typically small (kilobytes to a few megabytes), not the gigabytes of conventional memory.

3.4.2 Content Addressable Memory (CAM)

CAM Cell Structure

A basic binary CAM cell stores one bit and includes comparison logic:

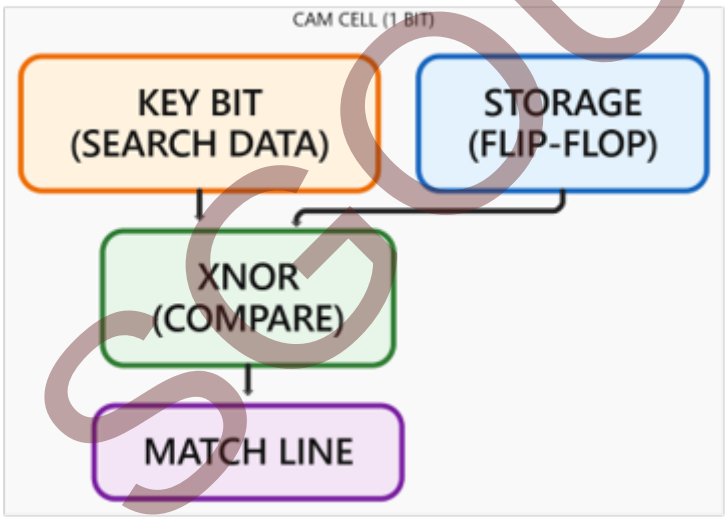


Fig. 3.4.2 Basic structure of a 1-bit CAM cell

The XNOR gate outputs 1 when the stored bit equals the key bit, 0 otherwise. The match line for a word is the AND of all its cell match results, it's 1 only if all bits match.

CAM Word Structure

A CAM word (row) consists of:

- N storage cells (one per bit).
- N comparison circuits (one per bit).
- One match line that indicates whether all bits match.
- A match flag indicating whether the word matched.

SGOU

Operation of CAM

1. **Search:** The search key is applied to all words simultaneously. Each word compares its stored data to the key. Words whose data matches the key assert their match line.
2. **Priority Encoder:** If multiple words match, a priority encoder can select one matching word according to a predefined priority.
3. **Result:** The address(es) of matching words are returned.

Ternary CAM (TCAM)

A TCAM cell supports three states: 0, 1, and "don't care" (X). The don't-care state matches both 0 and 1 in the search key. This enables wildcard matching essential for applications like routing table lookup (where the prefix matters and the rest is "don't care").

Solved Example 1: A 4-word CAM stores the following values at addresses 0-3: [1011, 0110, 1011, 1100]. A search key of 1011 is applied. Which addresses match? If a priority encoder is used, which address is reported?

Solution

- Address 0: 1011 matches → match line asserted.
- Address 1: 0110 does not match the search key 1011 because bits 3, 2, and 0 differ.
- Address 2: 1011 matches → match line asserted.
- Address 3: 1100 does not match.

  Matches: addresses 0 and 2.

  Priority encoder: selects address 0 (lowest address among matches).

3.4.3 Structure and Operation of Associative Array Processors

An associative array processor combines the array processor concept (Unit 1) with associative memory, each PE has content-addressable memory.

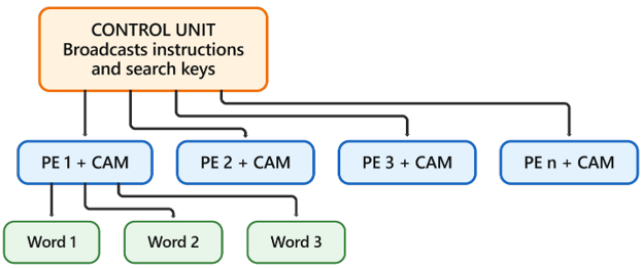


Fig. 3.4.3 Associative memory integrated with processing elements in an array processor

SGOU

Key Components

1. **Control Unit (CU)**
    - Broadcasts instructions and search keys to all PEs.
    - Manages the search operation and collects results.
2. **Processing Elements with CAM**
    - Each PE has associative memory that can search its contents in parallel.
    - PEs can perform arithmetic/logical operations on matched data.
3. **Match/Response Registers**
    - Each PE (or each word) has a match flag indicating whether it matched the search key.
    - The response register (or resolver) collects and prioritizes matches.
4. **Search Operation in an Associative Array Processor**
    - The CU broadcasts a search key (the content to find).
    - All PEs simultaneously compare the key against their local CAM contents.
    - PEs with matching content set their match flag.
    - The CU reads the match results (which PEs matched).
    - Matched PEs can then perform additional operations (read, write, arithmetic) on the matched data.

Associative Operations

Beyond simple search, associative array processors support:

1. **Associative Read:** Return the contents of matching words.
2. **Associative Write:** Write to matching words (all matches or prioritized).
3. **Associative Search:** Return the address(es) of matching words.
4. **Associative Arithmetic:** Perform operations on all matching words.
5. **Maximum/Minimum Search:** Find the maximum or minimum value by iterative comparison.

SGOU

Solved Example 2 : An associative array processor has 8 PEs, each storing one integer. The PEs store: [5, 12, 3, 12, 7, 12, 9, 4]. Find the maximum value using associative operations.

Solution

1. Iterative approach (bitwise or digit-wise maximum search):
2. The CU broadcasts a test value (starting from the highest possible bit).
3. PEs compare their stored value; those ≥ test values remain active, others are masked off.
4. Continue narrowing down until the maximum is isolated.
5. Result: The maximum is 12 (found at PEs 2, 4, and 6).
6. Under this bitwise/digit-wise search model, the maximum can be found in O(b) steps, where b is the number of bits in the stored values, assuming parallel comparison across the PEs.

3.4.4 Applications in Pattern Matching and Database Operations

1. Pattern Matching

Associative processing excels at finding patterns within data:

- **Exact Match:** Under the parallel CAM hardware model, exact matching can be completed in O(1) comparison time.
- **Wildcard Match:** TCAM supports don't-care bits, enabling patterns like "101X" (match any value starting with 101). Essential for network routing table lookup.
- **Longest Prefix Match:** In IP routing, find the routing entry with the longest matching prefix. High-speed network devices can use TCAM for longest-prefix-match routing-table lookups. Because multiple entries can be compared in parallel, TCAM can support very fast packet classification and forwarding.
- **Text Search:** For data represented in a suitable fixed-width or encoded form, associative memory can compare a search pattern with many stored records in parallel.

2. Database Operations

Associative processing maps naturally to database query operations:

- **Selection (WHERE clause):** Under the associative-search model, a selection operation can perform the matching step in O(1) comparison time, avoiding a sequential table scan.

SGOU

- **Join:** Match records from two tables based on a common field. Associative memory can find matching keys rapidly.
- **Deduplication:** Identify and remove duplicate records. Associative memory can detect duplicates by searching for existing values before insertion.
- **Aggregation:** Count, sum, or find max/min of matching records. Associative operations perform these in parallel.

3. Cache Memory

Fully associative caches use CAM to determine if an address is present in the cache. The cache searches all lines simultaneously for a tag match, a classic CAM application. (Set-associative caches use CAM on a smaller scale.)

4. Translation Lookaside Buffer (TLB)

A TLB stores recent virtual-to-physical address translations. TLBs commonly use associative or CAM-like lookup mechanisms to search virtual page numbers in parallel and obtain the corresponding physical page-frame information. This helps reduce the time required for address translation.

5. Network Routing

High-speed routers use TCAM for longest-prefix-match routing table lookup. This is perhaps the most commercially significant CAM application. TCAM can support very fast packet classification and forwarding by comparing multiple routing entries in parallel.

6. Artificial Intelligence (Symbolic)

Early AI systems (e.g., associative memories, neural network associative recall) used content-addressable access for pattern recognition and knowledge retrieval. Content-addressable memory enables "recall by partial information", retrieving a complete pattern from a partial or noisy input.

Table 3.4.1 Trade-offs Summary of CAM and Conventional RAM

Aspect	CAM	Conventional RAM
Access	By content	By address
Search by content	O(1) parallel under CAM hardware	O(n) sequential scan; faster with suitable indexing
Cell area	Higher	Lower
Power	Higher	Lower
Capacity	Typically smaller	Typically larger
Cost per bit	Higher	Lower
Best for	Search, matching, routing	General storage

SGOU

The Goodyear STARAN and MPP

Goodyear Aerospace built two landmark associative array processors: the STARAN (early 1970s) and the MPP (Massively Parallel Processor, 1983). The MPP had 16,384 PEs arranged in a 128×128 grid, each with 1 Kbit of memory, and was designed for satellite image processing for NASA. Its associative capabilities allowed ultra-fast pattern matching and image analysis, a task where traditional computers struggled. The MPP demonstrated the power of combining array processing with associative search.

CAM in Your Network Router

High-speed network devices can use TCAM (Ternary Content Addressable Memory) to perform routing-table lookups. TCAM allows multiple routing entries to be compared with a packet's destination address in parallel and is particularly useful for longest-prefix matching. This makes TCAM an important example of how specialized memory can accelerate network processing.

Recap

- Associative processing accesses data by content rather than by address, enabling parallel search in O(1) time.
- Content Addressable Memory (CAM) consists of cells with comparison logic (XNOR), where all words compare against a search key simultaneously.
- CAM cell = storage bit + XNOR comparator; match line = AND of all cell comparisons; priority encoder selects one match.
- Ternary CAM (TCAM) adds a "don't care" state, enabling wildcard and longest-prefix matching.
- Associative array processors combine array processors (Unit 1) with associative memory each PE has CAM.
- Operations: associative search, read, write, and arithmetic on matched data; maximum/minimum search in O(bits) steps.
- Applications: pattern matching (exact, wildcard, longest-prefix), database operations (selection, join, deduplication), cache memory, TLB, network routing, and symbolic AI.
- Trade-offs: CAM provides very fast parallel content-based search, but its additional comparison circuitry increases hardware area, power consumption, and cost per bit compared with conventional RAM. These factors limit its capacity.

SGOU

- The Goodyear STARAN and MPP are historical associative array processors; modern TCAM in routers is the most widespread commercial CAM application.

Objective Type Questions

1. What is the fundamental difference between associative memory and conventional RAM?
2. What logic gate is used in a CAM cell to compare the stored bit with the key bit?
3. What wire in a CAM word indicates whether that word matched the search key?
4. What does TCAM stand for, and what extra state does it support?
5. In associative processing, what is the time complexity of searching for a value by content?
6. What component in a CAM selects one match when multiple words match?
7. Which network device uses TCAM for longest-prefix-match routing?
8. What is a cache that searches all lines simultaneously for a tag match called?
9. What does the TLB (Translation Lookaside Buffer) use CAM for?
10. In associative array processors, what determines whether a PE participates in an operation?

SGOU

Answers

1. Associative memory accesses by content; RAM accesses by address.
2. XNOR gate.
3. The match line.
4. Ternary Content Addressable Memory, supports "don't care" state.
5. O(1) (constant time, all locations searched in parallel).
6. The priority encoder.

7. The router (IP router).

8. Fully associative cache.

9. To search virtual page numbers in parallel and find the corresponding physical page-frame information.

10. The PE's match/active flag determines whether it participates in the operation.

Assignments

1. Explain the operation of a binary CAM cell during a search operation. What role does the XNOR gate and match line play?

2. TCAM contains the following entries (with X = don't care): [110X, 1100, X111, 101X]. A search key 1100 is applied. Which entries match? Which is the longest prefix match?

3. Compare the hardware cost and power consumption of CAM versus SRAM. Why are CAMs typically small in capacity? When is the cost justified?

4. Design a simple 4-bit, 4-word CAM. Draw the complete circuit including CAM cells, match lines, and a priority encoder. Trace the operation for search key 1010 with stored values [1010, 0110, 1100, 1010].

5. Explain the operation of a binary CAM during a search. Show how XNOR comparison and the match line determine whether a word matches the search key.

6. A TCAM contains the entries [110X, 1100, X111, 101X]. For the search key 1100, identify all matching entries and determine the longest-prefix match.

7. Describe how an associative array processor could implement a database SELECT operation such as “find all employees with salary = 50000.” Compare this with a conventional sequential search.

8. Compare CAM and conventional RAM with respect to search time, hardware cost, power consumption, capacity, and suitable applications.

9. Explain how CAM is used in a fully associative cache and in a Translation Lookaside Buffer (TLB). Compare the two uses.

10. Explain how TCAM supports longest-prefix matching in IP routing. Discuss the advantages and limitations of using TCAM in routers.

SGOU

11. Research a modern application of content-addressable memory other than IP routing. Explain the application, why associative search is useful, and the major hardware trade-offs.

Reference

1. Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9), 948–960.

2. Foster, C. C. (1976). *Content addressable parallel processors*. Van Nostrand Reinhold.

3. Hennessy, J. L., Patterson, D. A., & Kozyrakis, C. (2025). *Computer architecture: A quantitative approach*. Morgan Kaufmann.

4. Pagiamtzis, K., & Sheikholeslami, A. (2006). Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3), 712–727.

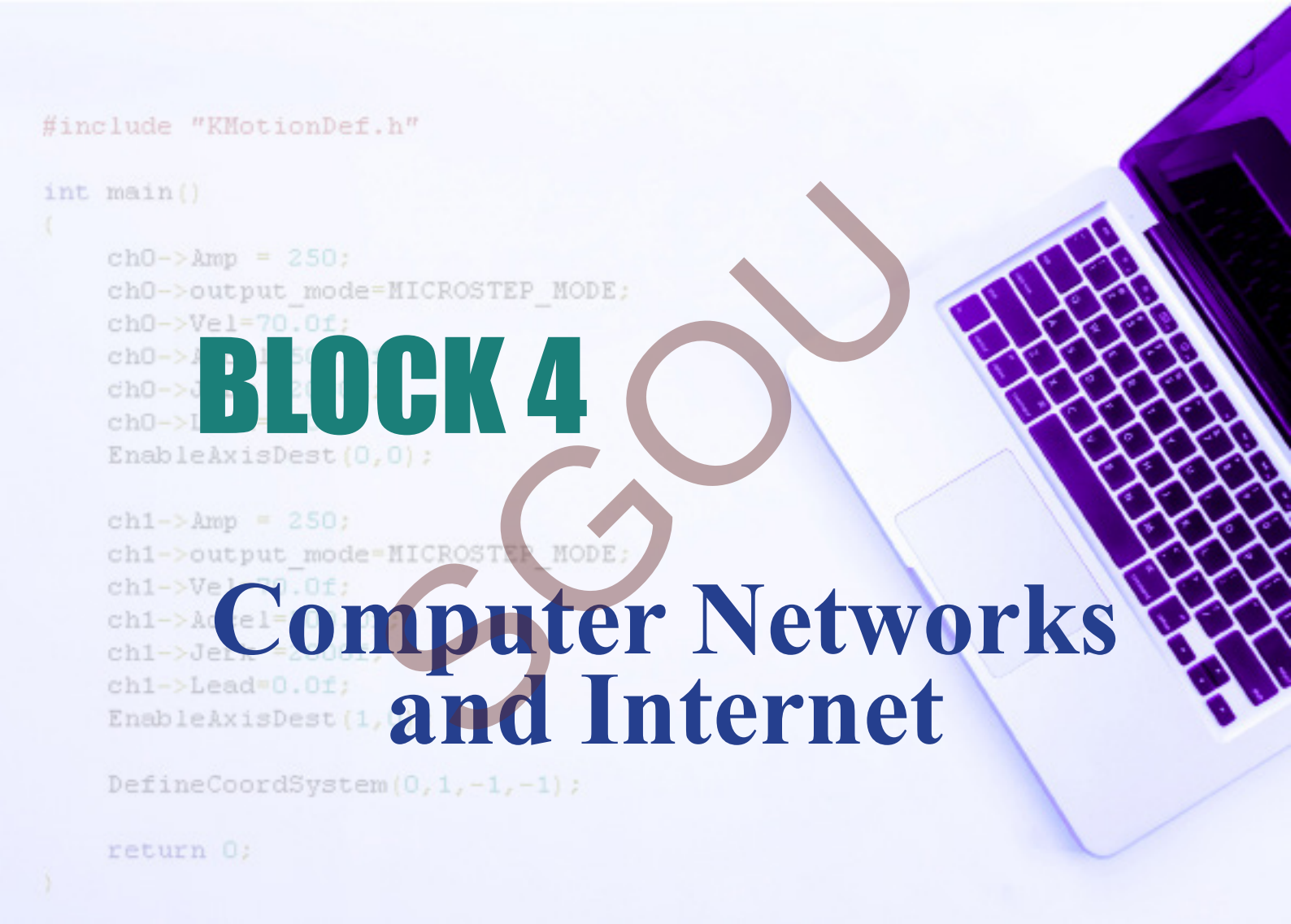
Suggested Reading

1. Kohonen, T. (1980). Content-Addressable Processors. In *Content-Addressable Memories* (pp. 281-330). Berlin, Heidelberg: Springer Berlin Heidelberg.

2. Pagiamtzis, K., & Sheikholeslami, A. (2006). Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE journal of solid-state circuits*, *41*(3), 712-727.

3. Lakshminarayanan, K., Rangarajan, A., & Venkatachary, S. (2005). Algorithms for advanced packet classification with ternary CAMs. *ACM SIGCOMM Computer Communication Review*, *35*(4), 193-204.

SGOU



BLOCK 4

Computer Networks and Internet

SGOU



Functional Structures

Learning Outcomes

After completion of this unit, learner will be able to:

- explain the rationale behind organising network functions into layers
- describe the seven layers of the OSI reference model and the role of each layer
- compare the OSI reference model with the TCP/IP model
- illustrate how data is encapsulated and de-encapsulated as it moves through the protocol stack
- define the basic elements of a communication protocol and identify common protocol examples

Prerequisite

Have you ever wondered how an email sent from one computer can reach another computer in a different country, or how a website loads correctly despite information travelling through multiple networks? Behind these everyday activities, several important tasks take place, including addressing devices, transmitting data, detecting errors, and delivering information to the correct destination. Handling all these tasks as one large process would make network design complex and difficult to maintain. To overcome this challenge, computer networks divide communication into separate layers, with each layer performing a specific responsibility and working together with the others.

Understanding this approach helps us discover how network communication is organised, how different devices communicate smoothly, and how network problems can be identified and resolved efficiently. It also provides the foundation for understanding how data travels across the internet, how protocols govern communication, and how information is prepared for transmission and received at its destination.

Let us now explore Network Layered Architecture, its principles and advantages, the OSI and TCP/IP models, and the processes and protocols that make modern computer communication possible.

SGOU

Keywords

Layered Architecture, Protocol, OSI Model, TCP/IP Model, Encapsulation, Interface, Service Access Point, Peer-to-Peer Communication

Discussion

4.1.1 Why Do We Need a Layered Architecture?

Consider the task of sending an e-mail attachment from a laptop in Kochi to a server in Frankfurt. Between the instant a user clicks "send" and the instant the file is written to disk on the far end, dozens of distinct problems must be solved: the file must be broken into manageable chunks, each chunk must be addressed so that intermediate routers know where to send it, each physical link along the way must convert the data into electrical, optical or radio signals, and errors introduced by noisy links must be detected and corrected. If a single piece of software had to handle every one of these concerns simultaneously, it would be enormous, fragile, and almost impossible to modify: changing the cabling technology would force a rewrite of the e-mail program itself.

Network designers solve this problem the same way that software engineers solve any large problem — by dividing it into smaller, self-contained pieces. Each piece, called a layer, is responsible for one well-defined aspect of communication and offers its services to the layer above through a fixed interface, while itself using the services of the layer below. This organisation is called a layered architecture, and it is the single most important structural idea in the whole of computer networking.

4.1.2 Principles of Layered Architecture

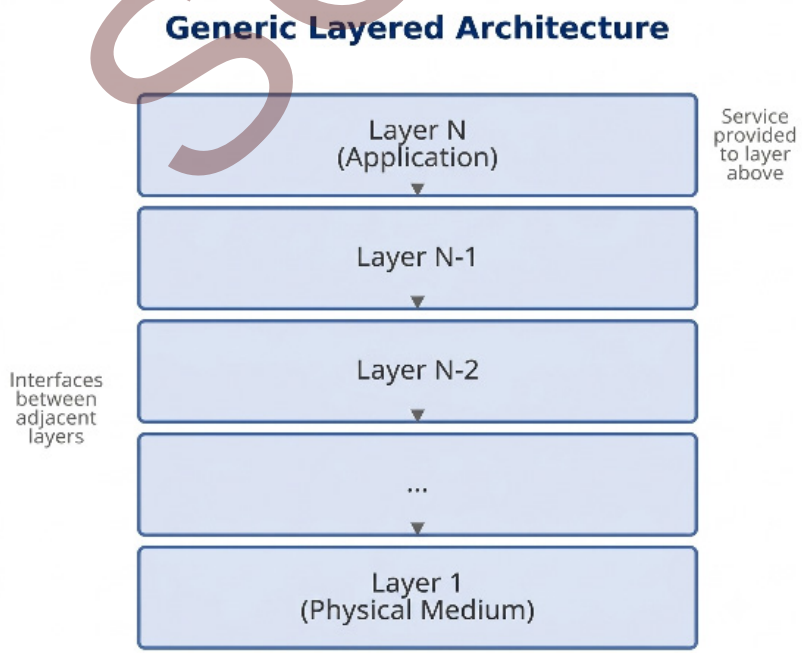


Fig. 4.1.1 Generic layered architecture: each layer offers a service to the layer above and consumes a service from the layer below

SGOU

A layered architecture rests on a small number of simple rules, illustrated in Figure 4.1.1. Every layer performs a specific function and exposes that function as a service to the layer immediately above it. To use that service, the upper layer does not need to know how the lower layer implements it — only what the service does. This separation between what a layer does (its service) and how it does it (its internal implementation) is what allows one layer to be redesigned or replaced (for example, replacing copper cabling with optical fibre) without disturbing any other layer.

Three terms recur throughout the study of layered systems:

Entity refers to an active element within a layer, typically implemented in software, such as an application program or a piece of protocol software.

Interface refers to the boundary between two adjacent layers on the same machine; it defines the primitive operations and services that the lower layer offers to the layer above.

Protocol refers to the set of rules and conventions used by peer entities on different machines to communicate; whereas the interface governs communication between adjacent layers on the same machine, a protocol governs communication between the same layer on two different machines.

Did you know?

The layering idea did not originate in computer networking. Postal systems, courier services and even biological signalling in the human body use exactly the same trick: a sender does not need to know how a courier company routes a parcel internally, only that handing the parcel over with the correct address will get it delivered. Layering is simply the engineering discipline of "divide and conquer" applied to communication.

4.1.2.1 Advantages of the Layered Approach

Organising a communication system into layers, rather than building it as one large monolithic program, brings several concrete engineering benefits:

- Modularity: each layer can be designed, implemented and tested in isolation, without needing to understand the internal workings of any other layer.
- Ease of modification: a layer's internal implementation can be changed — for example, replacing a wired link with a wireless one — as long as the interface it presents to the layer above remains unchanged.
- Interoperability: because each layer follows a published standard, hardware and software built by different manufacturers can interoperate, provided each conforms to the same interface.
- Simplified troubleshooting: network faults can be isolated to a specific layer (for example, a cabling fault at the Physical layer versus an application misconfiguration at the Application layer), which greatly speeds up diagnosis.

SGOU

- Reusability: a lower layer, once built, can be reused by many different upper-layer protocols; for instance, the same Ethernet Data Link layer carries web traffic, e-mail, and video conferencing without any changes.

The one disadvantage commonly cited against strict layering is overhead: every layer adds its own header, and processing each header consumes CPU time, so an idealised, unlayered, purpose-built protocol could in principle be marginally faster. In practice, the engineering benefits of layering so heavily outweigh this small overhead that every major network in use today, including the internet, is built as a layered system.

4.1.3 The OSI Reference Model

The Open Systems Interconnection (OSI) model was proposed by the International Organization for Standardization (ISO) in 1984 as a conceptual framework for understanding and designing network protocols. The OSI model itself is not a protocol; it is a reference model that describes what each of seven layers ought to do, leaving the actual implementation to individual protocols. Figure 4.1.2 shows the seven layers, numbered from the Physical layer (layer 1, closest to the transmission medium) up to the Application layer (layer 7, closest to the end user). The relevant information is presented in Table 4.1.1.

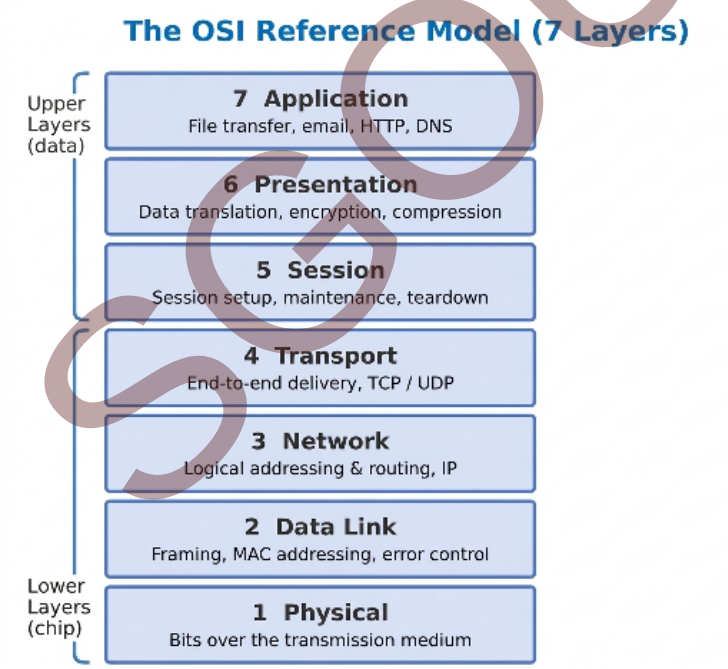


Fig. 4.1.2 The seven layers of the OSI reference model

Memory aid

Generations of networking students have remembered the order of the seven OSI layers, from Physical up to Application, using the mnemonic "Please Do Not Throw Sausage Pizza Away." Reading the first letters bottom-up gives Physical, Data Link, Network, Transport, Session, Presentation, Application.

SGOU

4.1.3.1 Functions of the Seven Layers

Table 4.1.1 summarises the principal responsibility of each layer together with representative protocols or technologies associated with it.

Table 4.1.1 Functions of the Seven OSI Layers

Layer	Primary Responsibility	Typical Protocols / Examples
7. Application	Provides network services directly to end-user applications	HTTP, FTP, SMTP, DNS
6. Presentation	Translates, encrypts and compresses data so that the application layer can interpret it correctly	SSL/TLS, JPEG, ASCII/ EBCDIC conversion
5. Session	Establishes, manages and terminates dialogues (sessions) between two communicating hosts	RPC, NetBIOS, PPTP
4. Transport	Provides reliable or best-effort end-to-end delivery, segmentation and flow control	TCP, UDP
3. Network	Provides logical addressing and determines the best path (routing) across multiple networks	IP, ICMP, routing protocols
2. Data Link	Organises bits into frames, performs physical addressing (MAC) and basic error detection on a single link	Ethernet, PPP, Wi-Fi (802.11)
1. Physical	Transmits raw bits over a physical medium as electrical, optical or radio signals	Copper cable, optical fibre, radio spectrum

SGOU

4.1.3.2 A Closer Look at Each Layer

The Physical layer is concerned purely with the mechanical, electrical and timing characteristics of the transmission medium: the voltage levels that represent a 0 or a 1, the shape and pinout of a connector, the frequency of a radio carrier, and the raw bit rate the medium can sustain. It has no concept of a "message" or an "address" — it simply moves a stream of bits from one point to another.

The Data Link layer takes the raw, unreliable bit stream offered by the Physical layer and organises it into frames, adds physical (MAC) addresses so that a frame can be delivered to the correct device on a shared local link, and performs basic error detection (and sometimes correction) using techniques such as a cyclic redundancy check. On a shared medium it also arbitrates access, deciding which device may transmit at any given instant.

The Network layer is responsible for moving a packet across multiple, possibly dissimilar, networks from source to final destination. It assigns logical (IP) addresses that are independent of the underlying hardware, and it determines the best path through the network using routing algorithms precisely the topic taken up in Unit 2 of this material.

The Transport layer provides end-to-end communication between the actual applications running on the source and destination hosts, as opposed to the Network layer's host-to-host delivery. It is responsible for segmenting long messages, and, depending on the protocol used, for reliability (retransmitting lost segments), ordering (reassembling segments in the correct sequence), and flow control (preventing a fast sender from overwhelming a slow receiver).

The Session layer establishes, synchronises and terminates a dialogue between two communicating applications, including inserting checkpoints into a long data transfer so that, if the connection fails, retransmission can resume from the last checkpoint rather than from the very beginning.

The Presentation layer ensures that data sent by the application on one machine can be read by the application on another, even if the two machines internally represent data differently; its responsibilities include character-encoding translation, data compression, and encryption.

The Application layer is the layer with which the end user (or the user's software) directly interacts, providing services such as file transfer, electronic mail, and web browsing. It is important to note that the Application layer refers to the network support that a user program calls upon and not the user program itself.

It is useful to think of the seven layers as falling into two broad groups. Layers 1 through 4 — Physical, Data Link, Network and Transport — are collectively called the lower layers or data-transport layers; they are concerned with moving bits reliably from one machine to another and are typically implemented in hardware or operating-system software. Layers 5 through 7 — Session, Presentation and Application — are the upper layers, concerned with ensuring that the data received makes sense to the application, and are usually implemented in software running on the end-user's machine.

4.1.3.3 Standards Organisations

The protocols that populate each OSI layer are not invented arbitrarily by individual vendors; they are defined and maintained by international standards bodies so that equipment from different manufacturers can interoperate. Table 4.1.2 lists the organisations most frequently encountered in networking.

SGOU

Table 4.1.2 Standards Organisations in Networking

Organisation	Area of Responsibility	Representative standard / example
ISO (International Organization for Standardization)	Published the OSI reference model itself and many other international standards	ISO/IEC 7498-1 (the OSI Basic Reference Model)
IEEE (Institute of Electrical and Electronics Engineers)	Standardises Data Link and Physical layer technologies such as Ethernet and Wi-Fi	IEEE 802.3 (Ethernet), IEEE 802.11 (Wi-Fi)
IETF (Internet Engineering Task Force)	Develops and publishes the protocols that run the internet as RFCs (Requests for Comments)	RFC 791 (IP), RFC 9293 (TCP)
ITU-T (International Telecommunication Union)	Standardises telecommunication technologies, including many aspects of the Physical layer	ITU-T X.25 (packet-switched networks), G.711 (voice coding)
W3C (World Wide Web Consortium)	Develops web-related standards at the Application layer	HTML and CSS specifications

4.1.3.4 Where Layers are Implemented in Real Systems

The seven OSI layers are a conceptual convenience; real computers do not necessarily contain seven separate pieces of software. Table 4.1.3 shows, approximately, where each layer's functionality is typically implemented on an ordinary personal computer.

Table 4.1.3 Typical Implementation of OSI Layers on a Personal Computer

Layer(s)	Typically Implemented In
Application, Presentation, Session	The application program itself and supporting software libraries (e.g. a web browser and its TLS library)
Transport, Network	The computer's operating-system kernel (e.g. the TCP/IP stack built into Windows, Linux or macOS)
Data Link, Physical	The network interface card (NIC) and its driver software

4.1.4 The TCP/IP Model

While the OSI model is an excellent teaching tool, the protocol suite that actually runs the internet predates it and follows a simpler four-layer structure known as the TCP/IP

SGOU

model (also called the Internet model), named after its two most important protocols: the Transmission Control Protocol (TCP) and the Internet Protocol (IP). Figure 4.1.3 shows how the four TCP/IP layers map onto the seven OSI layers.

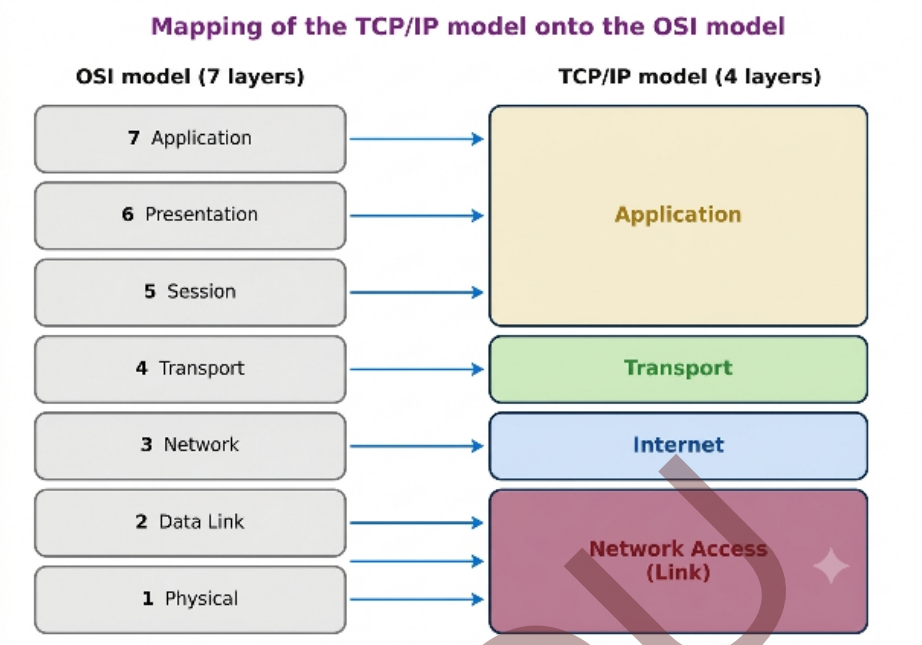


Fig. 4.1.3 Mapping of the four-layer TCP/IP model onto the seven-layer OSI model

The Network Access layer (sometimes split into separate Physical and Data Link layers) combines the functions of the OSI Physical and Data Link layers. The Internet layer corresponds directly to the OSI Network layer and is where the IP protocol operates. The Transport layer is identical in name and purpose to its OSI counterpart, hosting TCP and UDP. Finally, the Application layer in the TCP/IP model absorbs the responsibilities of the OSI Session, Presentation and Application layers into a single layer, on the pragmatic grounds that in practice these three concerns are usually handled together within a single application program. The relevant information is presented in Table 4.1.4.

Table 4.1.4 Comparison of the OSI and TCP/IP Models

Aspect	OSI Model	TCP/IP Model
Number of layers	Seven	Four
Developed by	ISO, as a general reference model	DARPA, alongside the ARPANET / early internet
Approach	Model defined first; protocols designed later to fit it	Protocols (TCP, IP) built first; model described afterwards
Layer boundaries	Strict separation of Session, Presentation and Application	Session and Presentation functions are left to the application
Practical use	Used mainly as a teaching and design reference	Used as the actual protocol suite running the internet

SGOU

4.1.5 Encapsulation and De-encapsulation

When an application hands data down through the protocol stack for transmission, each layer adds its own control information typically a header, and in the case of the Data Link layer also a trailer in front of (or around) the data it receives from the layer above. This wrapping process is called encapsulation, and the resulting unit of data at each layer is generically called a Protocol Data Unit (PDU), though it takes specific names at specific layers: a segment at the Transport layer, a packet (or datagram) at the Network layer, and a frame at the Data Link layer. Figure 4.1.4 illustrates this process.

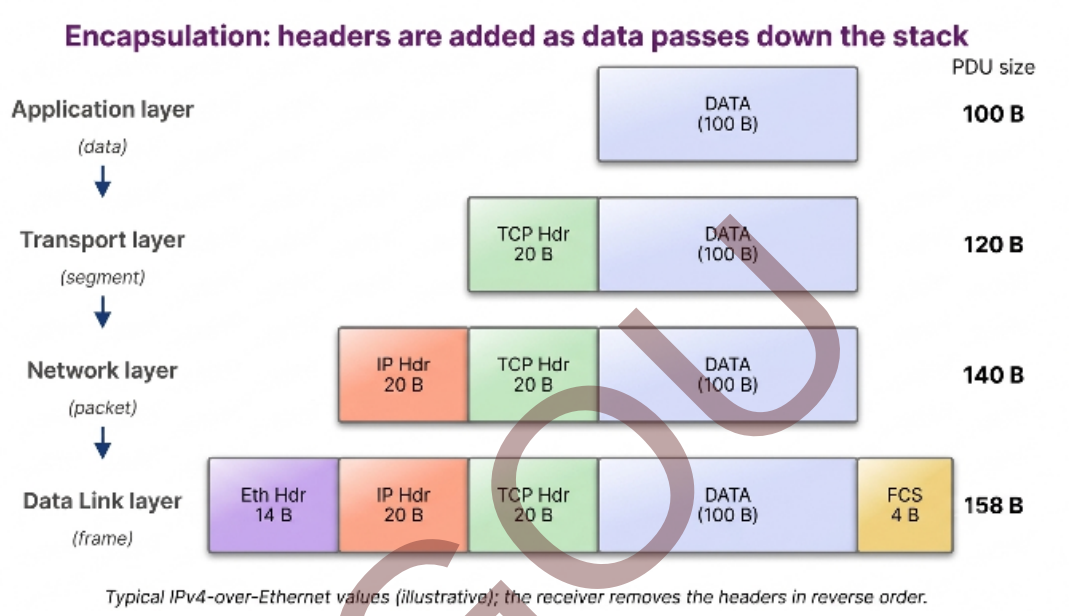


Fig. 4.1.4 Encapsulation of application data as it passes down the protocol stack; headers are added at each layer

At the receiving machine, the exact reverse process, de-encapsulation, takes place: as the frame travels up the stack, each layer strips off and interprets the header that its peer layer on the sending machine attached, before passing the remaining payload up to the next layer. Neither encapsulation nor de-encapsulation requires a layer to understand the contents of the payload it is carrying — the Network layer, for instance, routes an IP packet correctly regardless of whether the payload is an e-mail, a video frame, or a file chunk. This ignorance of payload content, by design, is exactly what allows a single internet to carry every kind of traffic without modification.

4.1.5.1 A Worked Example of Encapsulation

To make the idea of encapsulation concrete, consider a small 100-byte chat message typed by a user and sent over a TCP/IP network. Table 4.1.5 traces the size of the resulting protocol data unit as it passes down through the stack, assuming typical header sizes for IPv4 over Ethernet. The sizes used here are illustrative, typical values for this particular scenario (TCP over IPv4 over Ethernet, with no optional header fields); they may vary with protocol options and configurations.

SGOU

Table 4.1.5 Growth of a Protocol Data Unit Through the Layers (Encapsulation Example)

Layer	PDU Name	Header Added	Total Size
Application	Data	(application-specific formatting only)	100 bytes
Transport (TCP)	Segment	20-byte TCP header	120 bytes
Network (IP)	Packet	20-byte IP header	140 bytes
Data Link (Ethernet)	Frame	14-byte Ethernet header + 4-byte trailer (FCS)	158 bytes

By the time the original 100 bytes of chat text reach the Physical layer and are converted into an electrical or optical signal, they are accompanied by 58 bytes of header and trailer information contributed by three separate layers — none of which the application itself had to create or even know about. This is the practical payoff of layering: the chat application's programmer needed to think only about the 100 bytes of text, while the Transport, Network and Data Link layers transparently took care of reliable delivery, addressing and framing respectively.

How these sizes can vary

A TCP header is 20 bytes only when no options are used (it can grow to 60 bytes); an IPv4 header is likewise 20 bytes without options (up to 60); an IPv6 header is a fixed 40 bytes; a UDP header is only 8 bytes; and Ethernet adds a further 4-byte tag when a VLAN is used. The physical-layer preamble is also not counted in the 158 bytes. The calculation therefore shows the principle of encapsulation, not fixed values.

Worked illustration (header overhead)

The overhead added by the three lower layers is 20 + 20 + 14 + 4 = 58 bytes whatever the message size. For the 100-byte chat message the frame is 158 bytes, so overhead is 58/158 ≈ 37% of what is sent. For a 1,000-byte file chunk the frame is 1,058 bytes and the overhead falls to 58/1,058 ≈ 5.5%. Small messages are therefore expensive to send; larger payloads share the same header cost.

4.1.5.2 Connection-Oriented and Connectionless Service

Layers can offer their services to the layer above in one of two styles. In a connection-oriented service, the sender and receiver first establish a logical connection (a well-known example is the TCP three-way handshake), exchange data over that connection, and then formally close it; the network keeps track of the state of each connection. Setting up a connection does not by itself make delivery reliable: reliability and in-order delivery are extra features that a particular protocol adds. TCP is the standard example, because it numbers every byte, acknowledges what arrives and retransmits what is lost, so the application receives its data complete and in the order sent. In a connectionless service, each unit of data is transmitted independently, carrying its full destination address, with no prior setup and no guarantee that units will arrive in order or at all; UDP at the

SGOU

Transport layer and IP at the Network layer are classic connectionless services. The relevant information is presented in Table 4.1.6.

Key point

"Connection-oriented" and "reliable" are two different properties. TCP is both: it establishes a connection and guarantees reliable, in-order delivery by means of sequence numbers, acknowledgements and retransmission. Another connection-oriented service need not do so (the virtual circuits of Frame Relay, for instance, set up a path but did not recover lost frames). Likewise a connectionless service such as UDP leaves reliability, if it is needed, to the application.

Table 4.1.6 Comparison of Connection-Oriented and Connectionless Service

Aspect	Connection-Oriented (e.g. TCP)	Connectionless (e.g. UDP, IP)
Setup	Requires a handshake before data transfer	No setup required
Delivery guarantee	Reliable, in-order delivery, with retransmission of lost data	Best-effort; no guarantee of delivery or order
Overhead	Higher, due to connection state and acknowledgements	Lower, minimal per-packet overhead
Typical use	File transfer, web browsing, e-mail	Video streaming, online gaming, DNS lookups

The TCP three-way handshake

Establishing a TCP connection is a small protocol in its own right, consisting of three segments: the client sends a SYN (synchronise) segment to request a connection, the server responds with a SYN-ACK segment acknowledging the request and proposing its own starting sequence number, and finally the client replies with an ACK segment to confirm. Only after this three-way exchange completes does either side send any actual application data, which is precisely why connection-oriented service carries more setup overhead than a connectionless one.

4.1.6 Peer-to-Peer (Virtual) Communication

Although data physically travels down through the layers on the sending host, across the medium, and up through the layers on the receiving host, it is conceptually useful to imagine each layer as communicating directly with its counterpart — its peer — on the other machine. This is called peer-to-peer or virtual communication, shown as the dashed horizontal arrows in Figure 4.1.5. The Transport layer on the sending host, for example, behaves as though it is talking directly to the Transport layer on the receiving

SGOU

host, even though the data must actually descend to the Physical layer and cross the medium to get there. Only at the Physical layer does communication actually happen along the horizontal path, since only there does a real physical medium connect the two machines.

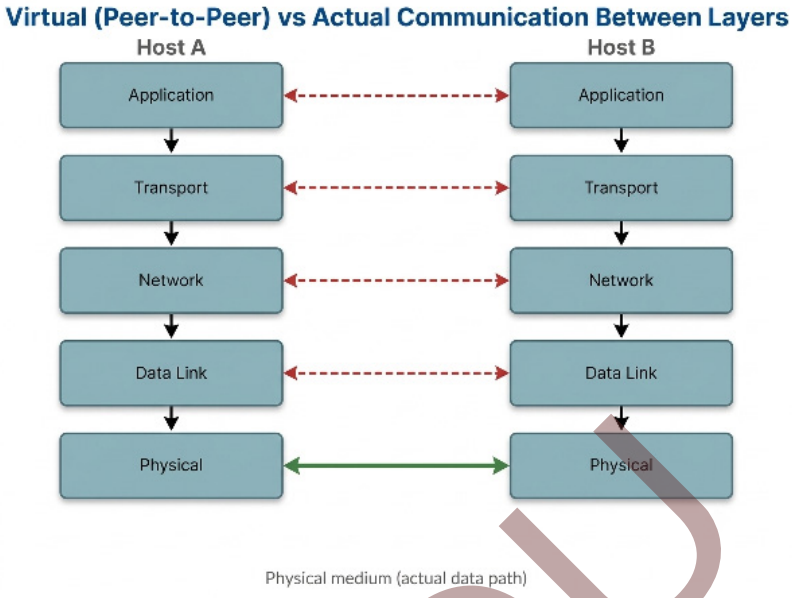


Fig. 4.1.5 Peer-to-peer (virtual) communication between corresponding layers versus the actual path taken by the data

4.1.7 Protocol Basics

A protocol is a formal set of rules and conventions that governs how two or more communicating entities exchange information. A protocol must specify three things precisely for communication to succeed:

Syntax refers to the structure or format of the data, including the order in which fields appear and how they are encoded. For instance, the fixed 20-byte header layout of an IPv4 packet.

Semantics refers to the meaning of each field or control bit. For example, what a particular flag in a TCP segment header indicates about the state of a connection.

Timing refers to when data should be sent and how fast, including rules for sequencing and for what a sender should do while waiting for an acknowledgement.

Everyday analogy

A protocol is much like the etiquette of a telephone call. The syntax is the sequence of actions — dial the number, wait for a ring, speak a greeting once the call is answered. The semantics is the shared understanding that "hello" signals readiness to talk, while a busy tone means try again later. The timing rule is the expectation that if there is no answer within a certain number of rings, the caller should hang up and try later. Change any one of these three elements without the other party's agreement and the call breaks down — exactly as it would if two computers used incompatible versions of a network protocol.

SGOU

One or more protocols govern every layer of the OSI and TCP/IP models. Table 4.1.7 lists a handful of widely used protocols and the layer at which each principally operates.

Table 4.1.7 Commonly used Network Protocols and their Layers

Protocol	Layer	Purpose
HTTP / HTTPS	Application	Transfers web pages between browser and server
DNS	Application	Translates human-readable domain names into IP addresses
TCP	Transport	Provides reliable, connection-oriented, ordered delivery
UDP	Transport	Provides fast, connectionless, best-effort delivery
IP	Network	Provides logical addressing and routes packets between networks
Ethernet (802.3)	Data Link	Frames data and controls access to a shared wired medium
Wi-Fi (802.11)	Data Link / Physical	Frames data and transmits it over a wireless radio medium

4.1.8 Multiplexing at the Transport Layer: Ports

A single host running the Transport layer protocols TCP or UDP usually communicates with several remote hosts and runs several network applications at once—a web browser, an e-mail client, and a chat application might all be active simultaneously. The Transport layer distinguishes between these simultaneous conversations using a 16-bit port number carried in the TCP or UDP header, a mechanism called multiplexing. A particular combination of source IP address, source port, destination IP address, destination port and protocol uniquely identifies one conversation, allowing a single physical network connection to carry many independent streams of data at once. Certain port numbers, known as well-known ports, are reserved by convention for specific services, as shown iWn Table 4.1.8.

Table 4.1.8 Well-Known Port Numbers and Associated Services

Port Number	Protocol	Service
20 / 21	TCP	FTP (data / control)
25	TCP	SMTP (outgoing e-mail)
53	TCP / UDP	DNS (name resolution)
80	TCP	HTTP (unencrypted web traffic)
443	TCP	HTTPS (encrypted web traffic)

SGOU

When a browser on a laptop opens two tabs to the same website, both connections share the same destination IP address and destination port 443, yet the operating system keeps their data streams perfectly separate because each connection is assigned a different, randomly chosen source port. This small mechanism — the humble port number — is what allows a single computer with a single network connection to sustain hundreds of simultaneous conversations without any of them becoming confused with one another.

Recap

- Layered architecture divides network communication into smaller, manageable layers.
- Each layer provides services to the layer above and uses services from the layer below.
- An interface connects adjacent layers on the same machine, while a protocol governs communication between peer layers on different machines.
- The OSI model has seven layers: Physical, Data Link, Network, Transport, Session, Presentation and Application.
- The TCP/IP model has four layers: Network Access, Internet, Transport and Application.
- The OSI Network layer corresponds to the TCP/IP Internet layer, while the upper three OSI layers are combined into the TCP/IP Application layer.
- Encapsulation adds headers and, at the Data Link layer, a trailer to data. De-encapsulation reverses this process at the receiver.
- A Protocol Data Unit (PDU) is the data unit at a layer. Common names include segment, packet and frame.
- Peer-to-peer communication describes how corresponding layers on different machines appear to communicate directly.
- A protocol defines syntax (format), semantics (meaning) and timing (sequence and speed).
- Common protocols include HTTP and DNS at the Application layer, TCP and UDP at the Transport layer, IP at the Network layer, and Ethernet and Wi-Fi at the Data Link and Physical layers.
- Connection-oriented services establish a connection before data transfer, while connectionless services send data independently. TCP provides reliable, ordered delivery.
- Standards bodies such as ISO, IEEE, IETF, ITU-T and W3C develop standards that support interoperability between different systems.

SGOU

- Port numbers help the Transport layer manage multiple simultaneous communications. Common examples are port 80 for HTTP and port 443 for HTTPS.

Objective Type Questions

1. Name the concept in which network functions are organised into independent layers, each offering a service to the layer above.
2. What term describes the boundary and set of operations offered by a lower layer to the layer immediately above it on the same machine?
3. What is the term for the rules governing communication between peer entities on two different machines?
4. Which organisation proposed the OSI reference model, and in which year?
5. How many layers does the OSI reference model have?
6. Which OSI layer is responsible for logical addressing and routing?
7. Which OSI layer is responsible for framing and MAC addressing?
8. How many layers does the TCP/IP model have?
9. Which single TCP/IP layer corresponds to the OSI Session, Presentation and Application layers?
10. What is the generic name given to a unit of data at any layer of a protocol stack?
11. What is the process called by which a layer strips off the header added by its peer layer at the sender?
12. Name the three components that a protocol must precisely define.
13. Which style of service establishes a logical connection before any data is transferred?
14. Name the three segments exchanged during the TCP three way handshake.
15. Which standards organisation publishes internet protocols such as TCP and IP as RFCs?
16. What mechanism allows the Transport layer to distinguish between several simultaneous conversations on a single host?

SGOU

17. Which well known port number is conventionally reserved for encrypted (HTTPS) web traffic?

18. Which OSI layer handles MAC addresses?

19. Which header field distinguishes two browser connections to the same HTTPS website?

20. Which transport protocol is most suitable for downloading a large file reliably?

21. Which transport protocol is generally preferred for a real time video call?

22. Which protocol property ensures reliable, in order delivery in TCP?

23. Which OSI layer must be modified when copper cable is replaced with optical fibre?

24. Which transport protocol is suitable for a single DNS lookup when low overhead is preferred?

25. Which protocol mechanism enables TCP to recover lost data?

26. Which OSI layer is responsible for logical addressing and routing packets between networks?

27. Which layer allows multiple applications to communicate simultaneously using port numbers?

Answers

1. Layered architecture.

2. Interface.

3. Protocol.

4. The International Organization for Standardization (ISO), in 1984.

5. Seven.

6. The Network layer.

7. The Data Link layer.

8. Four.

SGOU

9. The Application layer.
10. Protocol Data Unit (PDU).
11. De encapsulation.
12. Syntax, semantics and timing.
13. Connection oriented service.
14. SYN, SYN ACK and ACK.
15. The Internet Engineering Task Force (IETF).
16. Port numbers.
17. Port 443.
18. Data Link layer.
19. Source port.
20. TCP.
21. UDP.
22. Reliability.
23. Physical layer.
24. UDP.
25. Retransmission.
26. Network layer.
27. Transport layer.

SGOU

Assignments

1. Explain, with reasons, why network functions are organised as a layered architecture rather than as a single monolithic program.
2. Describe the seven layers of the OSI reference model and briefly state the function of each layer.

3. Distinguish between an interface and a protocol, giving one example of each.

4. Compare the OSI reference model with the TCP/IP model, highlighting at least four points of difference.

5. With the help of a labelled diagram, explain the process of encapsulation as data descends the protocol stack.

6. What is meant by peer-to-peer (virtual) communication between layers? Illustrate your answer with a diagram.

7. Define a protocol. Explain the significance of syntax, semantics and timing in protocol design.

8. List any four commonly used network protocols and identify the layer at which each operates.

9. Distinguish between connection-oriented and connectionless service, giving one real-world protocol example of each.

10. List the standards organisations responsible for Ethernet, internet protocols, and the World Wide Web respectively.

11. Explain the concept of multiplexing at the Transport layer and describe the role played by port numbers.

12. Draw a labelled diagram of encapsulation for a 250-byte message sent over TCP/IPv4/Ethernet. Show the PDU name and size at every layer and calculate the total overhead.

13. Trace a request to a secure web page through the TCP/IP model. For each layer state one protocol, the PDU name and the address or identifier used (port, IP address or MAC address).

14. Compare TCP and UDP for three applications of your choice (for example e-mail, online gaming and video streaming). Justify each choice and discuss the trade-off between reliability and delay.

15. Analyse why the TCP/IP model, rather than the OSI model, became the basis of the internet. Use the "Approach" and "Practical use" rows of Table 4.1.4 in your answer.

16. TCP/IP merges the OSI Session, Presentation and Application layers into one. Analyse the advantages and disadvantages of this design decision.

SGOU

Reference

1. Forouzan, B. A. (2013). *Data communications and networking* (5th ed.). McGraw-Hill Education.

2. International Organization for Standardization. (1994). *Information technology—Open Systems Interconnection—Basic reference model* (ISO Standard No. 7498-1).

3. Kurose, J. F., & Ross, K. W. (2017). *Computer networking: A top-down approach* (7th ed.). Pearson.

4. Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer networks* (5th ed.). Pearson Education.

Suggested Reading

1. Stallings, W. (2014). *Data and computer communications* (10th ed.). Pearson.

2. Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer networks* (5th ed.). Pearson Education.

SGOU



Interconnection Networks

Learning Outcomes

After completion of this unit, learner will be able to:

- identify common network topologies and evaluate their relative strengths and weaknesses
- distinguish the roles of switches and routers in an interconnection network
- compare circuit switching and packet switching as approaches to data transfer
- apply the shortest-path (Dijkstra) routing algorithm to a small weighted network
- describe how the flooding algorithm propagates packets through a network and evaluate its trade-offs

Prerequisite

Have you ever wondered how a message sent from your mobile phone reaches a friend on the other side of the world within seconds? How does the internet find a way to deliver your message, even when many computers and networks are connected? Behind this everyday experience lies the fascinating world of interconnection networks.

Before exploring this unit, think about how your computer connects to Wi-Fi, how a router helps you access the internet, and how information travels between different devices. You may already be familiar with basic networking concepts such as computers, communication links, IP addresses and data transmission. These simple ideas will help you understand how networks are organised, how switches and routers direct data, and how different techniques are used to transfer information efficiently. With this foundation, you are ready to explore network topologies, switching techniques and routing algorithms that help data reach its destination.

Keywords

Topology, Switch, Router, Circuit Switching, Packet Switching, Routing Algorithm, Shortest Path, Flooding

SGOU

Discussion

4.2.1 Introduction to Interconnection Networks

An interconnection network is the arrangement of communication links and switching elements that connects a set of nodes like computers, processors, memory modules, or entire local networks, so that any node can exchange data with any other. Three questions must be answered when an interconnection network is designed: How should the nodes be physically or logically arranged (topology)? What intermediate devices should forward data between nodes, and at what layer should they operate (switches and routers)? And by what technique should data actually move through the shared infrastructure by reserving a dedicated path in advance (circuit switching), or by breaking the data into independently routed packets (packet switching)? Once data can move through the network at all, a further question arises: given many possible paths between a source and a destination, which one should be chosen, a question answered by routing algorithms such as shortest-path routing and flooding. This unit addresses each of these questions in turn. The relevant information is presented in Table 4.2.1.

Table 4.2.1 Classification of Networks by Scale

Network Type	Typical Span	Typical Ownership	Example
LAN	A building or campus (up to a few km)	Single organisation	Office Ethernet, campus Wi-Fi
MAN	A city	One organisation or a consortium of providers	Municipal broadband, cable TV network
WAN	A country, continent, or the globe	Multiple carriers; links usually leased	The internet

4.2.1.1 Classification of Networks by Scale

Before examining topology, it is useful to classify interconnection networks by their physical scale, since scale strongly influences which topology and switching technique are practical. A Local Area Network (LAN) spans a single building or campus, typically up to a few kilometres, and is usually owned and administered by a single organisation — the office Ethernet or campus Wi-Fi network are common examples. A Metropolitan Area Network (MAN) spans an entire city, often connecting several LANs belonging to branches of the same organisation, or providing cable-television and broadband internet service across a municipality. A Wide Area Network (WAN) spans a country, a continent, or the entire globe, typically leasing communication links from telecommunication carriers rather than owning them outright; the internet itself is the largest WAN in existence.

SGOU

4.2.1.2 Point-to-Point versus Broadcast Links

The links that make up any of the topologies discussed below can themselves be classified into two categories. A point-to-point link connects exactly one pair of nodes, so any data placed on the link is intended for, and can be received only by, the node at the other end — the link between a router and its neighbouring router is a typical example. A broadcast link, by contrast, is shared by multiple nodes, so a transmission from any one node is potentially received by every other node connected to the same medium, and some mechanism (an address field, together with a medium-access control protocol) is required so that only the intended recipient acts on the data; classic Ethernet over a shared coaxial cable and Wi-Fi over shared radio spectrum are both broadcast-link technologies. As will be seen below, the bus and ring topologies are naturally built from broadcast-style links, whereas the star and mesh topologies are naturally built from point-to-point links.

Clarification

A broadcast link is a shared communication medium: one cable or radio channel that several nodes use at the same time (for example the original shared-cable Ethernet, or a Wi-Fi radio channel). Modern switched Ethernet networks generally use point-to-point links between hosts and switches, while Ethernet still supports the concept of broadcast at the network/protocol level: a frame sent to the broadcast address is delivered by the switch to every port of the LAN

4.2.2 Network Topologies

The topology of a network describes the pattern in which nodes and links are arranged, either physically (the actual layout of cabling) or logically (the pattern in which data is allowed to flow). Fig. 4.2.1 illustrates the six topologies most commonly discussed in networking and parallel-computing literature.

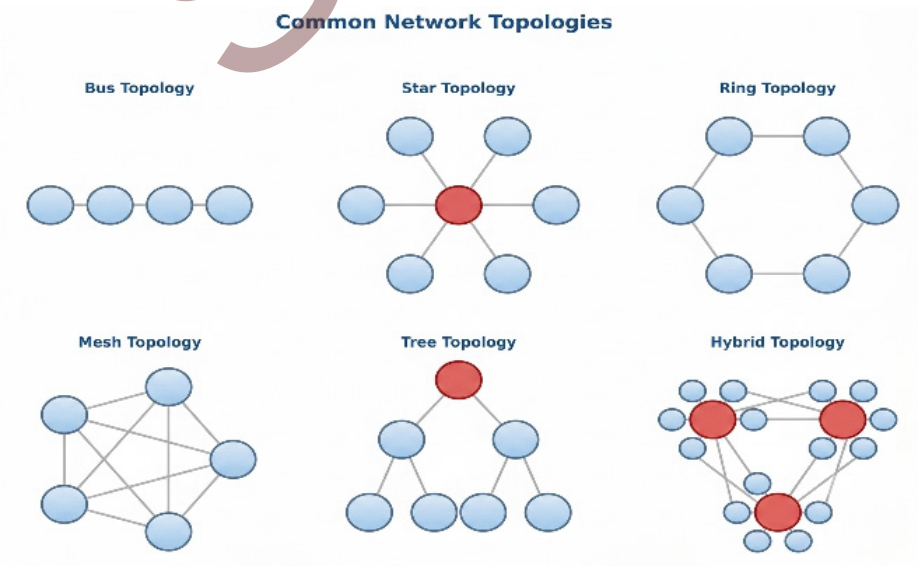


Fig. 4.2.1 Common network topologies: bus, star, ring, mesh, tree and hybrid

SGOU

4.2.2.1 Bus Topology

In a bus topology, every node is attached to a single shared communication line (the "bus") that runs the length of the network. A message transmitted by any node travels along the bus and is seen by every other node, but is accepted only by the node whose address matches the destination. Bus topology is inexpensive and simple to cable, but a fault anywhere along the shared bus can bring down the entire network, and performance degrades as more nodes contend for the same shared medium.

4.2.2.2 Star Topology

In a star topology, every node is connected by its own dedicated link to a central device such as a hub or switch, and all traffic passes through that central point. A cable fault affects only the single node attached to it, making the star topology far more fault-tolerant at the edges than a bus, but the central device becomes a single point of failure for the network as a whole and its capacity limits overall performance.

4.2.2.3 Ring Topology

In a ring topology, each node is connected to exactly two neighbours, forming a closed loop, and data travels from node to node around the ring until it reaches its destination. Ring networks distribute access to the medium fairly (as in the token-ring protocol), but a single broken link can disrupt the entire ring unless a dual-ring or self-healing design is used.

4.2.2.4 Mesh Topology

In a fully connected mesh topology, every node has a direct link to every other node. This provides the highest possible fault tolerance and the shortest possible path between any pair of nodes, since traffic never has to be relayed through an intermediate node, but the number of links required grows as $n(n-1)/2$ for n nodes, which quickly becomes impractical for large networks; partial mesh designs, which connect only some pairs of nodes directly, are used in practice, for example in wide-area backbone networks and in some high-performance-computing interconnects.

4.2.2.5 Tree Topology

A tree topology arranges nodes hierarchically, with a root node connected to one or more levels of child nodes, resembling several star networks connected together. It scales well to a large number of nodes and is easy to expand by adding further branches, but nodes higher up the tree (particularly the root) can become traffic bottlenecks and single points of failure for the branches beneath them. In a conventional tree every link at a given level has the same capacity, so the links near the root must carry the combined traffic of everything beneath them: if the 32 nodes in one half of the tree each send at 1 Gbps to the other half, the root must carry 32 Gbps, and a single 1 Gbps root link would be 32 times oversubscribed. A fat-tree (Section 4.2.7) keeps the tree structure but makes the links "fatter" — higher capacity or several parallel links — at each level closer to the root, so that the capacity at every level matches the traffic below it and the root is no longer a bottleneck.

SGOU

4.2.2.6 Hybrid Topology

A hybrid topology combines two or more basic topologies to draw on the strengths of each — for example, several star-configured local networks whose central switches are themselves connected in a partial mesh. Most large real-world networks, including corporate campus networks and the internet backbone itself, are hybrid in this sense. The relevant information is presented in Table 4.2.2.

Table 4.2.2 Comparison of Network Topologies

Topology	Key Advantage	Key Limitation
Bus	Low cabling cost, simple to install	Single shared medium is a performance bottleneck and single point of failure
Star	Fault isolation; easy to add or remove nodes	Central device is a single point of failure
Ring	Fair, orderly access to the medium	A single broken link can disrupt the whole ring
Mesh	Highest fault tolerance and shortest direct paths	Number of links grows quadratically with the number of nodes
Tree	Scales well; easy hierarchical expansion	Upper-level nodes can become bottlenecks
Hybrid	Combines strengths of multiple topologies	More complex to design and manage

In practice

The internet backbone is best described as a partial-mesh hybrid: major internet exchange points and backbone routers are richly interconnected in a mesh-like pattern for resilience, while the "last mile" that reaches individual homes and offices typically follows a star or tree topology. Many high-performance-computing clusters use specialised topologies such as the fat-tree or the torus/mesh, which are engineered variants of the tree and mesh topologies described above, chosen specifically to minimise the number of hops between any two compute nodes.

4.2.3 Switches and Routers

Once a topology has been chosen, intermediate devices are needed to forward data between nodes that are not directly connected. The two most important such devices are the switch and the router, and although both devices forward data, they operate at different layers of the protocol stack and over different scopes, as shown in Fig. 4.2.2.

SGOU

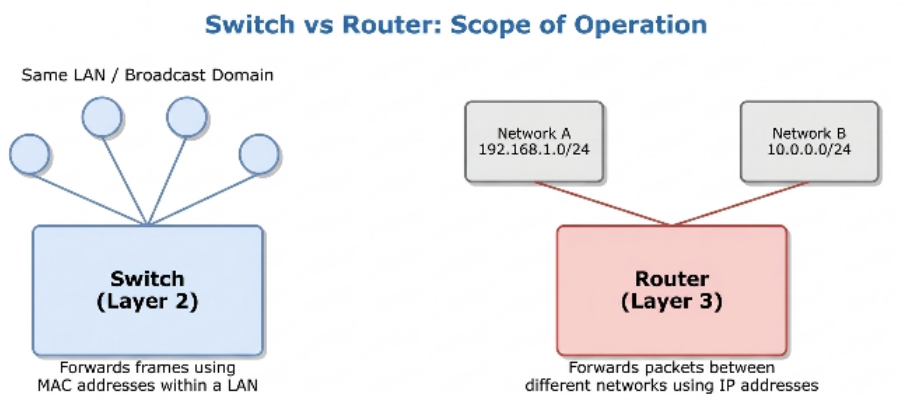


Fig. 4.2.2 A switch forwards frames within a single LAN using MAC addresses; a router forwards packets between distinct networks using IP addresses

A switch is a Data Link layer (Layer 2) device that connects multiple nodes within a single local area network (LAN). It examines the destination MAC address of each incoming frame and forwards the frame only out of the port that leads to the intended recipient, rather than broadcasting it to every port as an older hub would. By learning which MAC addresses are reachable through which ports, a switch dramatically reduces unnecessary traffic and collision, and it keeps all connected nodes within a single broadcast domain.

A router is a Network layer (Layer 3) device that connects two or more distinct networks — for example, a home network and the internet, or two different campus subnets. A router examines the destination IP address of each incoming packet and consults a routing table to decide the best next hop toward the destination network (the next hop is the immediate neighbouring router to which the packet is handed; each router-to-router transfer along the path is one hop), forwarding the packet accordingly. Unlike a switch, a router separates broadcast domains and can connect networks that use entirely different Data Link layer technologies. The relevant information is presented in Table 4.2.3.

Table 4.2.3 Comparison of Switches and Routers

Aspect	Switch	Router
OSI Layer	Data Link (Layer 2)	Network (Layer 3)
Forwarding decision based on	MAC (physical) address	IP (logical) address
Scope of operation	Within a single LAN	Between different networks
Broadcast domain	Single (all ports share one)	Separates broadcast domains
Typical use	Connecting PCs, printers and servers within an office	Connecting a home or office network to the internet

SGOU

A note on layer-3 switches

Modern networking equipment often blurs this distinction: a "layer-3 switch" is a single physical device that performs high-speed Layer-2 switching for most traffic but also has basic routing capability built in, allowing it to route between VLANs without the traffic ever leaving the device. Conceptually, however, the functional distinction between switching (Layer 2) and routing (Layer 3) described above still holds and remains the correct way to reason about how such a device behaves.

4.2.4 Circuit Switching and Packet Switching

Beyond deciding which device forwards data, a network must decide how data is moved through the shared infrastructure. Two fundamentally different strategies are used: circuit switching and packet switching, contrasted in Fig. 4.2.3.

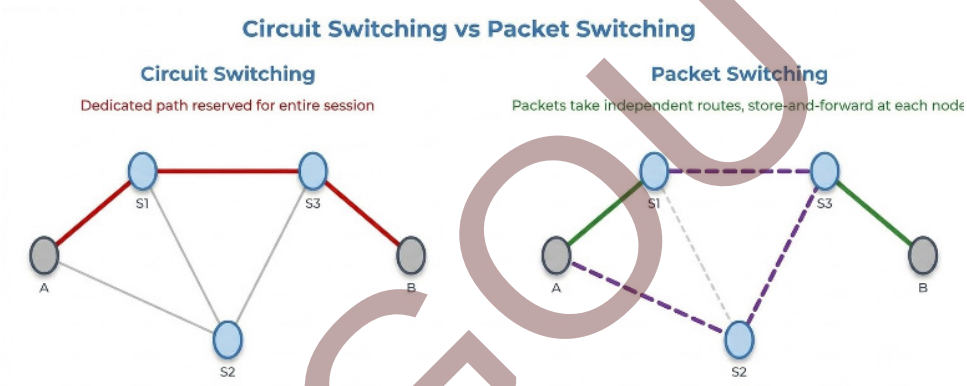


Fig. 4.2.3 Circuit switching reserves a dedicated path for an entire session; packet switching allows independent packets to take independent routes

In circuit switching, a dedicated communication path is established between the source and destination before any data is transferred, and that path, with its full bandwidth, remains reserved exclusively for the duration of the session, regardless of whether data is actually being sent at every instant. The traditional analogue telephone network is the classic example: dialling a number causes a chain of physical switches to connect a dedicated electrical path from caller to callee, which remains reserved (and unavailable to any other call) until one party hangs up. Every circuit-switched session proceeds through three distinct phases: circuit establishment, during which the path is set up hop by hop and resources are reserved along it; data transfer, during which the caller and callee exchange information over the now-dedicated path; and circuit teardown, during which the connection is released and the reserved resources are freed for other users once either party ends the session.

In packet switching, data is broken into discrete units called packets, each carrying its own destination address, and different packets belonging to the same message may travel by different routes through the network, being individually queued and forwarded at each intermediate node, a technique known as store-and-forward. No bandwidth is

SGOU

reserved in advance; the network's capacity is shared dynamically among all active senders. The internet is built entirely on packet switching. The relevant information is presented in Table 4.2.4.

Table 4.2.4 Comparison of Circuit Switching and Packet Switching

Aspect	Circuit Switching	Packet Switching
Path setup	Dedicated path established before data transfer	No path reserved; packets routed independently
Resource use	Bandwidth reserved for the whole session, even during silence	Bandwidth shared dynamically among many senders
Delay characteristics	Constant delay once the circuit is set up	Variable delay; subject to queuing at each node
Efficiency for bursty traffic	Low — idle capacity is wasted	High — idle capacity can be used by other traffic
Failure resilience	Call fails if any link in the fixed path fails	Packets can be rerouted around a failed link
Classic example	Traditional (analogue) telephone network	The internet

4.2.4.1 Message Switching

A third, largely historical technique, message switching, is worth mentioning because it sits conceptually between the other two. In message switching, an entire message (with no upper limit on size) is transmitted as a single unit from node to node using store-and-forward: each intermediate node receives the complete message, stores it (traditionally on disk or tape), and forwards it onward only when the next link is free. Message switching, once used in early telegraph and electronic-mail-relay systems, shares packet switching's lack of a pre-reserved dedicated path, but because an entire message must be stored at each hop before forwarding, it introduces far greater delay than modern packet switching, which forwards small packets rather than an entire message and can therefore begin transmitting the next packet before an earlier one has finished its journey.

Worked illustration (store-and-forward delay)

A 12,000-bit message must cross 3 links of 10 Mbps each (ignore propagation and queuing delay). Sending 12,000 bits on one link takes 12,000 / 10,000,000 = 1.2 ms. With message switching each node must receive the whole message before forwarding it, so the total is 3 × 1.2 = 3.6 ms. With packet switching the message is cut into four packets of 3,000 bits (0.3 ms per link). The first packet needs 3 × 0.3 = 0.9 ms to arrive, and the other three follow one after another, each 0.3 ms later, so the whole message arrives after (3 + 4 − 1) × 0.3 = 1.8 ms — half the time, because the links work in parallel on different packets.

SGOU

Why the internet chose packets

Early network researchers in the 1960s, including Paul Baran and Donald Davies working independently, argued that packet switching would make networks far more resilient to damage than a circuit-switched design, since traffic could automatically be rerouted around any node that was destroyed or overloaded, rather than depending on one fixed, pre-established path. This resilience argument, originally motivated by military concerns, is a large part of why the ARPANET, and ultimately the modern internet, adopted packet switching rather than the circuit-switched model of the telephone network.

4.2.5 Routing Algorithms: Shortest-Path Routing

A routing algorithm is the procedure a network uses to decide, for each destination, which outgoing link a packet should be forwarded on. Routing algorithms are broadly classified as static, in which routes are computed once (often by a human administrator) and rarely change, or adaptive (dynamic), in which routers continuously exchange information and recompute routes as network conditions change. Adaptive algorithms are further divided into two major families: distance-vector algorithms, in which each router periodically tells its immediate neighbours its entire routing table (its "distance vector") so that routes propagate hop by hop across the network, and link-state algorithms, in which each router instead floods a small piece of information, the state and cost of its own directly attached links, to every other router in the network, so that every router ends up with an identical, complete map of the topology and can independently compute the shortest path to every destination. Shortest-path computation, discussed next, is the mathematical technique that both families ultimately rely on, though distance-vector algorithms apply it in a distributed, iterative fashion while link-state algorithms apply it centrally at each router once the full topology map has been assembled by flooding. The relevant information is presented in Table 4.2.5.

Table 4.2.5 Comparison of Distance-Vector and Link-State Routing

Aspect	Distance-Vector Routing	Link-State Routing
Information exchanged	Entire routing table, sent to immediate neighbours only	Only its own link costs, flooded to every router in the network
Each router's view of the topology	Partial and indirect (learned hop by hop)	Complete (every router builds the full topology map)
Convergence after a change	Slower; can suffer from the "count-to-infinity" problem	Faster; every router recomputes independently once flooding completes
Example protocol	RIP (Routing Information Protocol)	OSPF, IS-IS

SGOU

The remaining distinctions in the original comparison concern static versus adaptive routing, the first classification described above; they are summarised in Table 4.2.6.

Table 4.2.6 Comparison of Static and Adaptive Routing

Aspect	Static routing	Adaptive (dynamic) routing
How routes are determined	Configured manually by a network administrator	Computed automatically by routers exchanging information
Response to network changes	Does not adjust automatically; requires manual reconfiguration	Automatically recalculates routes as topology or traffic changes
Overhead	None at run time; minimal router CPU and memory use	Ongoing overhead from periodic updates or flooding
Suitability	Small, stable networks with few links	Large networks where links and traffic conditions change frequently

In a packet-switched network with many possible paths between a source and a destination, a routing algorithm decides which path each packet should follow. One of the most important families of routing algorithm is shortest-path routing, in which the network is modelled as a weighted graph: each node represents a router, each edge represents a link between two routers, and the weight of each edge represents a cost, which might be distance, delay, or the inverse of available bandwidth. The task of the routing algorithm is to find the path between any two nodes whose total weight is minimum.

Dijkstra's algorithm, published by Edsger W. Dijkstra in 1959, is the best-known algorithm for this task. Starting from a source node, it repeatedly selects the unvisited node with the smallest known distance from the source, marks it as visited, and updates ("relaxes") the distances of its neighbours if a shorter path through the newly visited node has been found. The algorithm terminates once every node has been visited, at which point the shortest distance, and the corresponding path, from the source to every other node is known. The algorithm assumes that every edge weight is non-negative. With a negative weight, a node already marked visited could later be reached more cheaply, so its distance would no longer be final and the algorithm could return wrong answers (other algorithms, such as Bellman-Ford, are used in that case). Link costs such as distance, delay and the inverse of bandwidth are never negative, so the condition holds for network routing. The concept is illustrated in Fig. 4.2.4.

SGOU

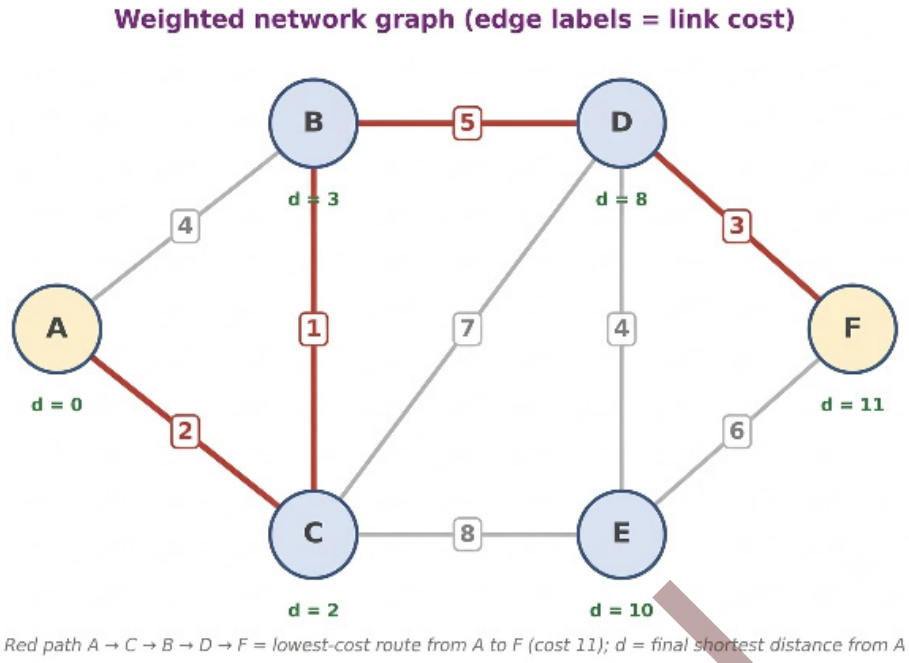


Fig. 4.2.4 A weighted network graph; the highlighted path A → C → B → D → F is the lowest-cost route from source A to destination F

Consider the network in Figure 4.2.4, in which edge weights represent link cost. Although the direct-looking path A → B → D → F has a cost of 4 + 5 + 3 = 12, the algorithm discovers that the path A → C → B → D → F, at first glance longer because it passes through more nodes, actually costs only 2 + 1 + 5 + 3 = 11, and is therefore the shortest path once all edge weights are taken into account. This illustrates the key insight of shortest-path routing: the number of hops is not what matters, the sum of the weights along the path is what matters. The relevant information is presented in Table 4.2.7.

Table 4.2.7 traces the tentative shortest distance from source A to every other node as Dijkstra's algorithm proceeds. At each step, the node with the smallest tentative distance among the unvisited nodes (shown in bold) is marked visited, and the distances of its neighbours are relaxed if a shorter path has been discovered through it.

Worked illustration (the first iterations in detail)

To relax a neighbour N of the node just visited, compute new = d(visited) + w(visited, N) and keep it if it is smaller than the current d(N). Start: d(A) = 0 and every other distance is ∞. Step 1 (visit A): d(B) = 0 + 4 = 4 and d(C) = 0 + 2 = 2. Step 2: the unvisited node with the smallest distance is C (2), so visit C. Relax its neighbours: for B, 2 + 1 = 3 is smaller than the current 4, so d(B) becomes 3 and its route is now via C; for D, 2 + 7 = 9 is smaller than ∞, so d(D) = 9; for E, 2 + 8 = 10 is smaller than ∞, so d(E) = 10. Step 3: the smallest unvisited distance is now B (3), so visit B and relax D: 3 + 5 = 8 is smaller than 9, so d(D) becomes 8. These are exactly rows 2 and 3 of Table 4.2.7; the improvement of d(B) from 4 to 3 shows why the route A → C → B (cost 3) beats the direct link A → B (cost 4).

SGOU

Quick check: what is d(F) after Step 4, when D (8) is visited? Answer: 8 + 3 = 11, which is smaller than ∞, so d(F) = 11; E stays at 10 because 8 + 4 = 12 is not smaller.

Table 4.2.7 Dijkstra's Algorithm: Step-by-Step Trace from Source Node A

Step	Node Visited	Dist. to B	Dist. to C	Dist. to D	Dist. to E	Dist. to F
1	A (start)	4	2	∞	∞	∞
2	C	3	—	9	10	∞
3	B	—	—	8	10	∞
4	D	—	—	—	10	11
5	E / F	—	—	—	—	11

The final row confirms that the lowest total cost from A to F is 11, achieved via the path A → C → B → D → F identified earlier, and the table also reveals, as a useful by-product, the shortest distance from A to every other node in the network.

The end result of running a shortest-path algorithm at a router is stored locally in a routing table, a simple data structure that the router consults for every packet it must forward. Table 4.2.8 shows the routing table that node A would build from the results traced above: for each possible destination, it records only the cost and the next hop to send a packet to — not the entire path — since each subsequent router along the way will make its own forwarding decision in exactly the same manner.

Table 4.2.8 Routing Table Constructed at Node A

Destination	Next Hop	Total Cost
B	C	3
C	C	2
D	C	8
E	C	10
F	C	11

Where shortest-path routing is used

Shortest-path computation is not merely an academic exercise: it is the algorithmic core of widely deployed link-state routing protocols such as OSPF (Open Shortest Path First) and IS-IS, which are used inside large enterprise and internet-service-provider networks to decide how packets should be forwarded from router to router.

SGOU

4.2.6 Routing Algorithms: Flooding

Flooding is a far simpler and far less efficient routing technique in which a node, on receiving a packet, forwards a copy of it out on every link except the one on which it arrived. Every neighbouring node does the same, and the packet propagates outward through the network like ripples on a pond until it eventually reaches every reachable node, as shown in Fig. 4.2.5.

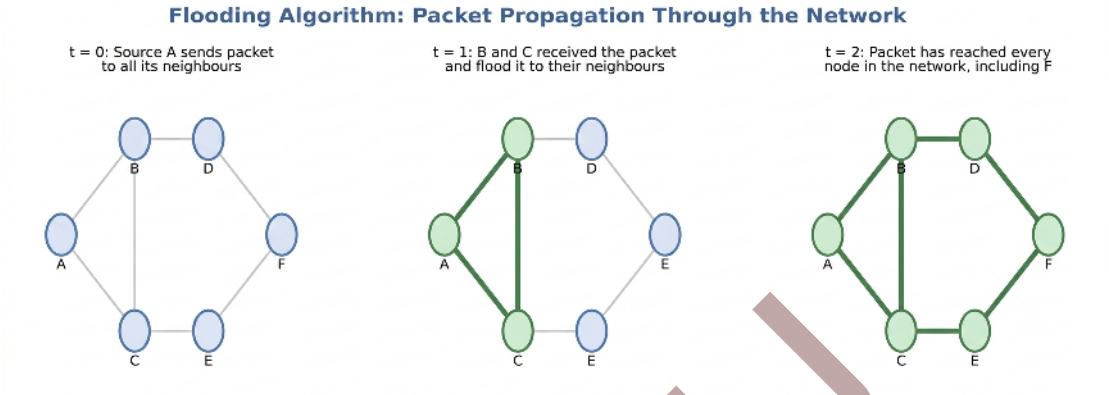


Fig. 4.2.5 Flooding: a packet injected at source node A is forwarded outward at each hop until it has reached every node in the network

In a connected network, flooding greatly increases the probability that a packet reaches its destination, because copies travel along every possible path. Delivery is not absolutely guaranteed, however: it can still fail if links or nodes fail on every path, if the network is disconnected, or if the packet expires (its hop-count or time-to-live limit reaches zero) before it arrives. Flooding, and it does so without the network needing any prior knowledge of topology or routing tables, which makes it extremely robust to node and link failures. This robustness, however, comes at a heavy cost: without safeguards, the same packet can circulate indefinitely around loops in the network, and the total volume of traffic generated grows explosively with the number of links, since every node re-transmits every packet it receives.

To make flooding practical, two common controls are applied. In uncontrolled flooding, no attempt is made to limit re-transmission, and the technique is used only in very small networks or as a fallback in emergencies. In controlled flooding, a limit is imposed: a sequence-number-controlled approach has each node remember the packets it has already forwarded (identified by source address and sequence number) and discard any duplicate, while a hop-count (or time-to-live) approach attaches a counter to each packet that is decremented at every hop, and the packet is discarded once the counter reaches zero, preventing it from circulating forever. The relevant information is presented in Table 4.2.9.

SGOU

Table 4.2.9 Advantages and Disadvantages of Flooding

Advantages of Flooding	Disadvantages of Flooding
Very high chance of delivery in a connected network, because copies follow every possible path (not guaranteed under failures, disconnection or packet expiry)	Generates a very large volume of duplicate traffic
Requires no prior knowledge of network topology or routing tables	Without control, packets can circulate forever around loops
Extremely robust to node and link failures	Consumes significant bandwidth and processing at every node
Simple to implement, with no routing computation needed	Impractical as the primary routing method in large networks

Where flooding is used

Despite its inefficiency for ordinary data traffic, controlled flooding is used deliberately in situations where reliability matters more than efficiency. Link-state routing protocols such as OSPF use a sequence-number-controlled flood to distribute link-state advertisements to every router in an area, ensuring that every router eventually builds an identical, complete picture of the network topology, which it then uses to run Dijkstra's shortest-path algorithm locally.

4.2.7 Evaluating Interconnection Networks for High-Performance Computing

In a high-performance-computing cluster, the interconnection network is often the single biggest factor limiting how well an application scales across many processors, so network designers evaluate candidate topologies using a small set of quantitative metrics rather than choosing a topology by intuition alone. Diameter is the greatest number of hops (a hop is one link traversal, from a node or router to the immediate neighbouring node or router to which the packet is forwarded) that must be traversed between the two most distant nodes in the network; a smaller diameter means that, in the worst case, data reaches its destination faster and with fewer opportunities for congestion along the way. Bisection bandwidth is the total capacity of the links that must be cut to split the network into two equal halves; a higher bisection bandwidth means the network can sustain heavy communication between arbitrary halves of the machine without becoming a bottleneck, which is critical for collective operations (such as an all-to-all data exchange) that are common in parallel scientific computing. Cost, measured as the total number of links required, must be weighed against both of the above, since a network with excellent diameter and bisection bandwidth but a prohibitive number of links (such as a full mesh) is rarely practical at scale. The relevant information is presented in Table 4.2.10.

SGOU

Table 4.2.10 Comparison of Interconnection Network Topologies by Performance Metrics

Topology	Diameter	Bisection Bandwidth	Number of Links (n nodes)
Bus	O(1) (shared medium)	Low (one shared channel)	n (single shared line)
Ring	O(n)	Low (only 2 links cross any cut)	n
Star	O(1) (2 hops via centre)	Limited by central-node capacity	$n - 1$
Fully-connected Mesh	O(1) (1 hop)	Very high	$n(n-1)/2$
Tree (fat-tree, HPC variant)	O(log n)	High, if link capacity increases toward the root	O(n log n)

This is precisely why specialised HPC interconnects rarely use a plain bus, ring or star: the fat-tree topology, for instance, is engineered specifically to keep diameter small (like a tree) while increasing link capacity at higher levels of the hierarchy so that bisection bandwidth remains high, achieving a practical balance between cost, diameter and bandwidth that a simple topology cannot match at the scale of thousands of compute nodes.

Real HPC interconnects

Production supercomputers commonly use engineered variants of the topologies introduced in this unit. The fat-tree, used in many InfiniBand-based clusters, is a tree in which link bandwidth increases at higher levels to avoid bottlenecks near the root. The torus (a mesh whose edges wrap around), used in systems such as IBM's Blue Gene series, connects each node to a fixed small number of neighbours in multiple dimensions, keeping the number of links manageable while still offering multiple alternative paths between any two nodes.

Recap

- An interconnection network is the arrangement of links and switching elements that connects a set of nodes so that any node can communicate with any other.
- Common network topologies include bus, star, ring, mesh, tree and hybrid, each with its own trade-off between cost, performance and fault tolerance.

SGOU

- A switch is a Layer-2 device that forwards frames within a single LAN using MAC addresses; a router is a Layer-3 device that forwards packets between distinct networks using IP addresses.
- Circuit switching reserves a dedicated path for the entire duration of a session, as in the traditional telephone network.
- Packet switching breaks data into independently routed packets that share network capacity dynamically, as in the internet.
- Shortest-path routing models the network as a weighted graph and finds the minimum-total-weight path between source and destination; Dijkstra's algorithm is the classic technique for computing it.
- Flooding forwards every incoming packet out of every link except the one it arrived on, greatly increasing the chance of delivery (though not guaranteeing it under failures, disconnection or packet expiry) at the cost of very high traffic volume.
- Controlled flooding (using sequence numbers or a hop-count/time-to-live field) prevents packets from circulating indefinitely around loops in the network.
- Link-state routing protocols such as OSPF combine controlled flooding (to distribute topology information) with shortest-path computation (to decide actual forwarding).
- Networks are classified by scale into LAN (building/campus), MAN (city) and WAN (country/global); links themselves are classified as point-to-point or broadcast.
- Distance-vector routing (e.g. RIP) exchanges entire routing tables with immediate neighbours; link-state routing (e.g. OSPF) floods only local link costs so every router can build a complete topology map.
- Message switching, a historical technique, forwards an entire message using store-and-forward at each node, incurring more delay than packet switching.
- Interconnection networks for high-performance computing are evaluated using diameter (worst-case hop count), bisection bandwidth (capacity across the network's midpoint) and cost (number of links); engineered topologies such as the fat-tree and torus balance these factors better than simple bus, ring or star designs.

SGOU

Objective Type Questions

1. Name the topology in which every node is connected to a single shared communication line.
2. Which topology connects every node to a central hub or switch, making the central device a single point of failure?
3. In which topology does every node have a direct link to every other node?
4. At which OSI layer does a switch typically operate?
5. At which OSI layer does a router typically operate?
6. Name the switching technique in which a dedicated path is reserved for the entire duration of a call.
7. Name the switching technique used by the internet, in which data is broken into independently routed units.
8. Name the classical algorithm used to compute the shortest path in a weighted network graph
9. What routing technique forwards every incoming packet out of every link except the one on which it arrived?
10. Name the field commonly attached to a flooded packet to prevent it from circulating indefinitely.
11. Name a link-state routing protocol that uses controlled flooding to distribute topology information.
12. Give an example of a real-world network that uses circuit switching.
13. Which type of network spans an entire city, often connecting several LANs belonging to the same organisation?
14. Name the two major families of adaptive (dynamic) routing algorithms.
15. Name a routing protocol that uses the distance-vector approach.
16. Which historical switching technique forwards an entire message using store-and-forward at each node?

SGOU

Answers

1. Bus topology
2. Star topology.
3. Mesh topology
4. The Data Link layer (Layer 2).
5. The Network layer (Layer 3).
6. Circuit switching.
7. Packet switching.
8. Dijkstra's algorithm.
9. Flooding.
10. A hop-count / time-to-live (TTL) field.
11. OSPF (Open Shortest Path First).
12. The traditional public switched telephone network (PSTN).
13. A Metropolitan Area Network (MAN).
14. Distance-vector routing and link-state routing.
15. RIP (Routing Information Protocol).
16. Message switching.

SGOU

Assignments

1. Describe the six common network topologies discussed in this unit and, for each, state one advantage and one limitation.
2. Distinguish between a switch and a router in terms of the OSI layer at which each operates, the addressing scheme used, and the scope of the network each connects.
3. Compare circuit switching and packet switching, and explain why the internet was built using packet switching rather than circuit switching.

4. Explain, with a worked numerical example, how Dijkstra's shortest-path algorithm computes the lowest-cost path in a weighted network graph.

5. What is flooding? Describe the difference between uncontrolled and controlled flooding.

6. Explain how the hop-count (time-to-live) field prevents a flooded packet from circulating indefinitely around a network.

7. Describe how a link-state routing protocol such as OSPF combines flooding and shortest-path computation.

8. Discuss the interconnection-network requirements of a high-performance-computing cluster and explain why topologies such as the fat-tree or torus are preferred over a simple bus or ring.

9. Distinguish between LAN, MAN and WAN, and between point-to-point and broadcast links, with examples of each.

10. Compare distance-vector and link-state routing, explaining how each builds its view of the network topology.

11. Define diameter and bisection bandwidth, and use them to explain why a fat-tree or torus topology is preferred over a simple bus or ring in a large high-performance-computing cluster.

12. A network has links A–B = 3, A–C = 6, B–C = 2, B–D = 5, C–D = 1, C–E = 7 and D–E = 4. Apply Dijkstra's algorithm from A, show each step in a table like Table 4.2.7, and write the routing table at A (destination, next hop, cost).

13. A 20,000-bit message crosses 4 links of 5 Mbps each. Calculate the transmission time under message switching and under packet switching with five equal packets. Ignore propagation and queuing delay.

14. A 64-node cluster is to be wired as a ring, an 8 × 8 torus, a fat-tree or a full mesh. Compare the four in terms of diameter, number of links and bisection bandwidth, and recommend one.

15. Analyse the conditions under which flooding can fail to deliver a packet, and explain how sequence numbers and hop counts address (or fail to address) each condition.

16. An enterprise network has frequent link failures. Compare static routing, distance-vector routing and link-state routing for it, and justify your recommendation.

SGOU

Reference

1. Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer networks* (5th ed.). Pearson Education.

2. Forouzan, B. A. (2013). *Data communications and networking* (5th ed.). McGraw-Hill Education.

3. Kurose, J. F., & Ross, K. W. (2017). *Computer networking: A top-down approach* (7th ed.). Pearson.

4. Stallings, W. (2014). *Data and computer communications* (10th ed.). Pearson.

Suggested Reading

1. Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to parallel computing* (2nd ed.). Pearson.

2. Dally, W. J., & Towles, B. (2004). *Principles and practices of interconnection networks.* Morgan Kaufmann.

SGOU



Parallel Memory Organizations

Learning Outcomes

After completion of this unit, learner will be able to:

- distinguish between shared-memory and distributed-memory parallel architectures and identify their respective strengths and limitations
- explain the organisation and working principle of Uniform Memory Access (UMA) multiprocessor systems
- describe Non-Uniform Memory Access (NUMA) architectures and analyse how memory placement affects performance
- state the cache coherence problem in multiprocessor systems and outline the hardware protocols used to solve it, and
- differentiate between cache coherence and memory consistency

Prerequisite

In everyday life, we often depend on computers to perform tasks quickly and efficiently, such as checking weather forecasts, making online payments, watching videos, and searching for information. As technology advances, many applications need to process large amounts of data within a short period. For example, weather forecasting systems analyse information collected from satellites and sensors to predict changes in atmospheric conditions. Similarly, hospitals use computer systems to analyse medical images and assist doctors in identifying health problems. Online shopping platforms process numerous customer requests simultaneously, while banks handle thousands of transactions within seconds. These activities require computers to perform multiple operations efficiently rather than completing every task one after another. A single processor may take considerable time to process such large workloads, particularly when the required data cannot be accessed quickly enough. We may have noticed that a computer becomes slow when several applications are running or when large files are being processed. The speed of a computer depends not only on its processor but also on how quickly data can be stored, retrieved, and shared. In high performance computing, several processors or computing units work together to solve complex problems, making efficient memory access essential. For instance, scientific research centres use powerful

SGOU

computing systems to simulate climate patterns, study molecular structures, and analyse astronomical data. These tasks involve large datasets that must be accessed by multiple processors during computation. Parallel memory organization helps computing systems arrange and access memory efficiently so that different processors can obtain the data they need with less waiting time. Understanding this concept provides a foundation for exploring how high performance computing systems improve processing speed, support large scale computations, and handle complex real world problems efficiently.

Keywords

Shared Memory, Distributed Memory, UMA, NUMA, Cache Coherence, Memory Consistency, Interconnection Network, MESI Protocol, False Sharing

Discussion

4.3.1 Parallel Memory Organizations: An Overview

A parallel computer contains more than one processing element that can execute instructions simultaneously. The way these processors access memory, which stores the data and instructions they process, is one of the most important design decisions in high performance computing. It determines how processors communicate, how programs are developed, and how efficiently the system performs as more processors are added. Broadly, parallel memory organizations are classified into two types: shared memory systems, in which all processors can directly access a common memory space, and distributed memory systems, in which each processor has its own private memory and processors exchange data explicitly through an interconnection network.

4.3.1.1 Shared Memory Architecture

In a shared memory multiprocessor, all processors are connected to a single global memory through a shared bus or a switching network, and every processor can read or write any memory location using ordinary load and store instructions. Because there is one logical address space, communication between processors happens implicitly: one processor writes a value, and another simply reads it from the same address. This makes shared memory systems comparatively easy to program, since the programmer does not need to explicitly move data between processors. The concept is illustrated in Figure 4.3.1.

SGOU

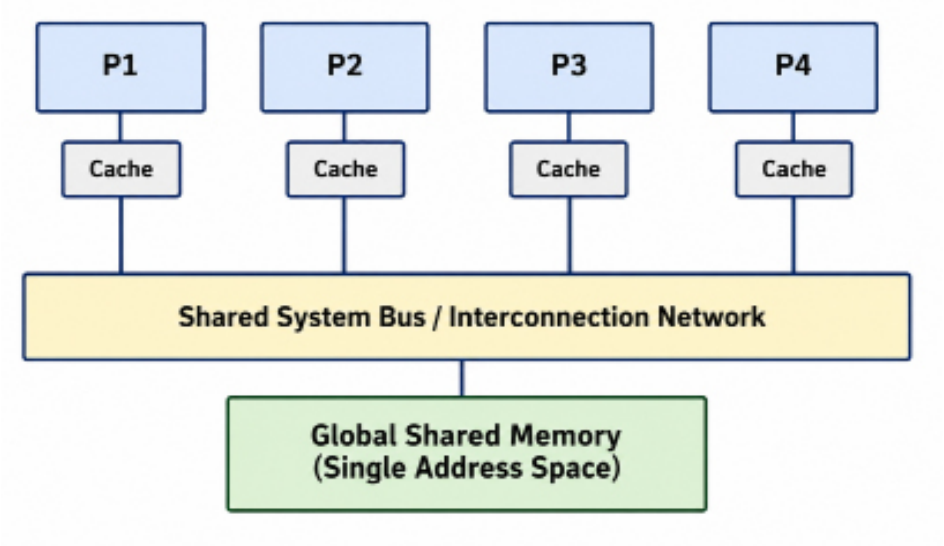


Fig. 4.3.1 A tightly coupled shared-memory multiprocessor: every processor accesses a single global memory through a shared interconnect.

Shared memory systems are also called tightly coupled systems because the processors are closely bound through the common memory and the shared bus. Symmetric multiprocessors (SMP), in which every processor has equal access to memory and I/O, are the most common commercial example — most present-day laptop and server CPUs with multiple cores are, internally, small shared-memory multiprocessors.

Did you know?

The dual-core and quad-core processors found in ordinary laptops and smartphones are themselves miniature shared-memory multiprocessors — all cores on the chip share the same physical RAM through an on-chip interconnect.

The principal advantage of shared memory is programming simplicity and low communication latency, since data does not need to be copied between processors. However, as more processors are added, contention for the shared bus and shared memory banks increases, which limits scalability — this is why shared memory systems are typically built with a modest number of processors (a few to a few dozen), whereas distributed memory is preferred for very large-scale machines.

4.3.1.2 Distributed Memory Architecture

In a distributed memory multiprocessor, each processor has its own local, private memory, and there is no single global address space that every processor can see directly. If one processor needs data held in another processor's memory, the two processors must communicate explicitly by exchanging messages over an interconnection network — this is why distributed memory systems are also called message-passing systems, or loosely coupled systems. The concept is illustrated in Figure 4.3.2.

SGOU

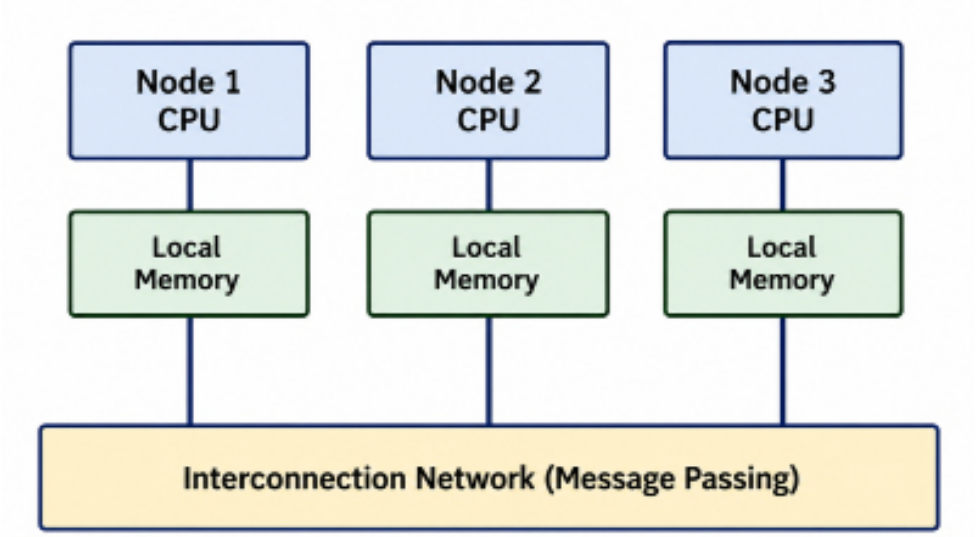


Fig. 4.3.2 A distributed-memory system: each node has its own local memory; nodes communicate only through explicit message passing over the interconnect

Because each node manages only its own memory, distributed memory architectures scale to very large numbers of processors — this is the basis of most of the world's largest supercomputers and computing clusters, where thousands of independent nodes, each with local memory, are connected by a high-speed network such as InfiniBand or a custom torus/fat-tree topology. The trade-off is programming complexity: the programmer (or the parallel programming library, such as MPI — the Message Passing Interface) must explicitly decide what data to send, to whom, and when, and communication over the network is typically much slower than a local memory access.

4.3.1.3 Interconnection Networks

Whichever memory organisation is chosen, processors, memory modules, and I/O devices must be physically connected by an interconnection network, and the choice of network has a direct bearing on cost, scalability, and achievable bandwidth. Three commonly discussed topologies illustrate the trade-offs involved. The concept is illustrated in Figure 4.3.3.

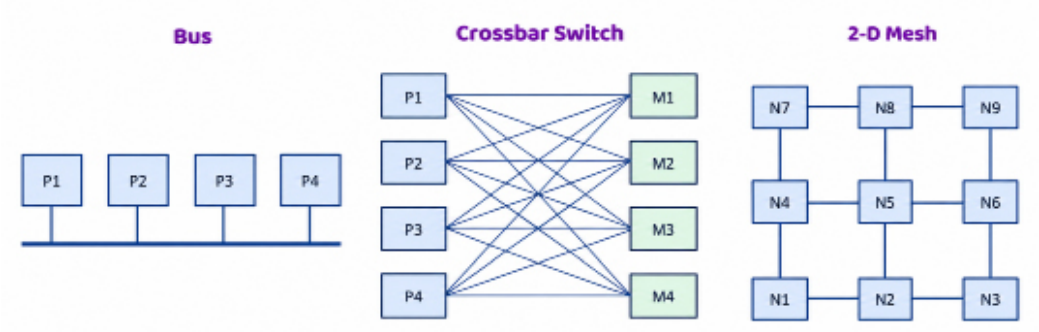


Fig. 4.3.3 Three common interconnection topologies: a shared bus, a crossbar switch, and a two-dimensional mesh

- Bus: a single shared communication path connects all processors and memory. It is inexpensive and simple, but only one transfer can proceed at a time, so it quickly becomes a bottleneck as the number of processors grows.
- Crossbar switch: every processor is connected to every memory module through a dedicated switch point, allowing many simultaneous, non-conflicting transfers. This gives excellent performance but the cost of the switch grows quadratically with the number of processors, making it impractical for very large systems.
- Multistage / mesh networks: processors are arranged in a grid (or a multistage switching network), and each node connects only to its immediate neighbours. Any two nodes can still communicate, but a message may need to pass through several intermediate nodes, trading some latency for much better scalability and lower cost than a full crossbar.

In general, bus-based interconnects suit small UMA systems, crossbar switches suit medium-sized, high-bandwidth shared-memory systems, and mesh, torus, or fat-tree networks are the interconnects of choice for large distributed-memory clusters and supercomputers, where thousands of nodes must be connected cost-effectively. The relevant information is presented in Table 4.3.1.

4.3.2 Comparing Shared and Distributed Memory

Table 4.3.1 Shared memory versus distributed memory architectures

Aspect	Shared Memory	Distributed Memory
Address space	Single, global address space	Multiple private address spaces
Communication	Implicit, via load/store	Explicit, via message passing
Programming model	Threads, OpenMP	MPI, message-passing libraries
Scalability	Limited (bus/memory contention)	High (scales to thousands of nodes)
Coupling	Tightly coupled	Loosely coupled
Typical example	Multicore CPU, SMP server	Compute cluster, supercomputer
Cost of communication	Low latency	Higher latency (network dependent)

SGOU

4.3.3 Uniform Memory Access (UMA)

Shared memory multiprocessors are further classified according to how memory access latency varies (or does not vary) across processors. In a Uniform Memory Access (UMA) architecture, all processors share a single physical memory, and — critically — every processor experiences the same access time to any memory location, regardless of which processor issued the request. This uniformity is usually achieved by connecting all processors and the memory through a shared bus or a symmetric crossbar switch, so that the physical distance and the number of intermediate stages between any processor and memory are effectively the same. The concept is illustrated in Figure 4.3.4.

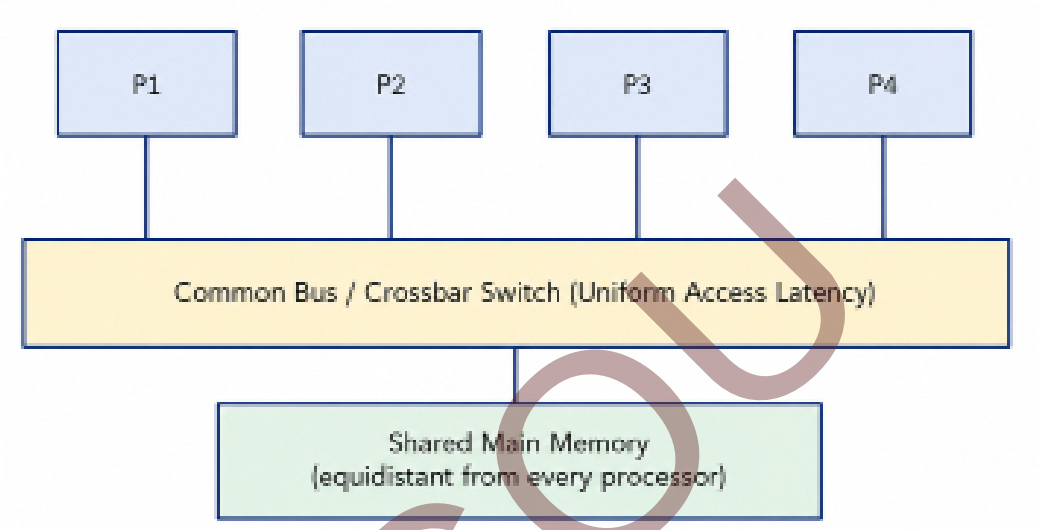


Fig. 4.3.4 Uniform Memory Access (UMA): every processor is equidistant from the shared main memory, giving identical access latency

UMA machines are usually built as Symmetric Multiprocessors (SMP). Because access time does not depend on which processor is asking, the operating system can freely migrate a process from one processor to another without any memory-access penalty, which simplifies load balancing. The main limitation is that the shared bus becomes a bottleneck as the number of processors grows — beyond roughly 16 to 32 processors, contention for the shared interconnect severely limits performance, which is why UMA systems are used for small to medium multiprocessors rather than very large machines.

Worked illustration

Suppose four processors P1–P4 in a UMA system each issue a memory read at the same instant. Because the bus is shared, the memory controller must serialise these requests one after another (say 50 ns each). The last processor to be served therefore waits roughly 150 ns longer than the first — even though every individual access, once granted, takes the same 50 ns. This illustrates how UMA gives uniform per-access latency but not necessarily uniform total waiting time under contention.

SGOU

4.3.4 Non-Uniform Memory Access (NUMA)

As the number of processors increases, a single shared bus cannot supply memory bandwidth fast enough. Non-Uniform Memory Access (NUMA) architectures address this by physically partitioning memory into several modules, each attached locally to a group (node) of processors. A processor can access its own node's local memory quickly, but accessing memory that belongs to a different node must travel across an inter-node interconnect and therefore takes noticeably longer. Hence the access time is 'non-uniform' — it depends on where, physically, the requested data resides relative to the requesting processor. The concept is illustrated in Figures 4.3.5 and 4.3.6.

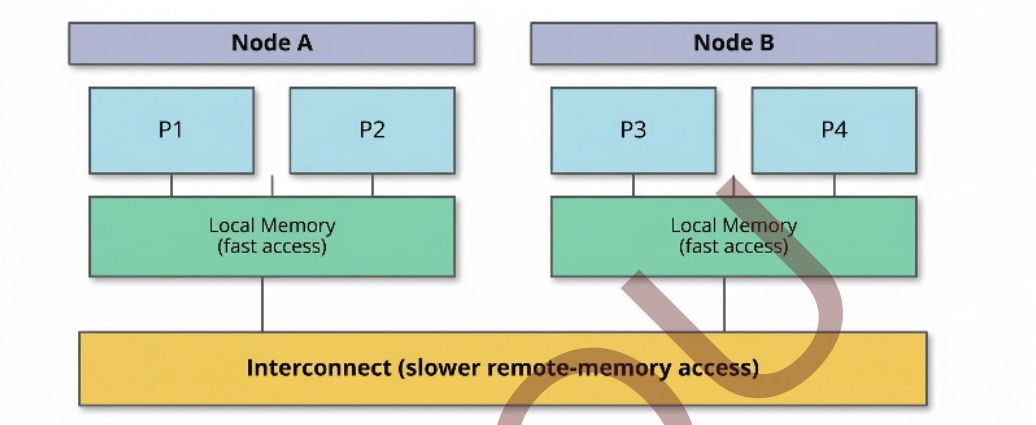


Fig. 4.3.5 NUMA architecture: processors access their own node's local memory quickly, while access to another node's memory must cross the interconnect and incurs additional latency

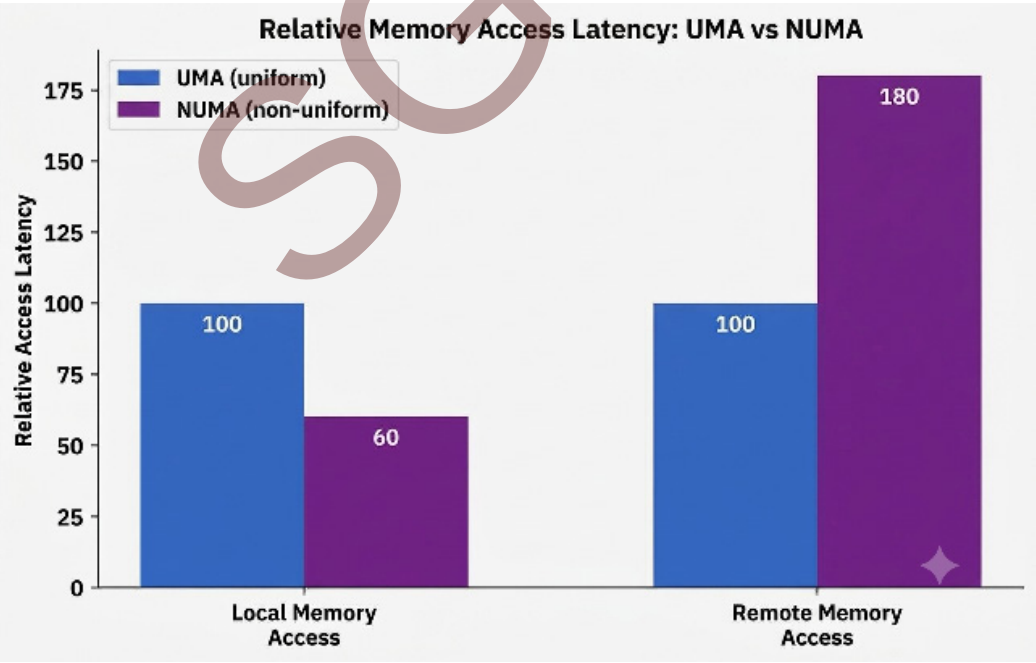


Fig. 4.3.6 Relative memory-access latency: UMA gives the same latency for every access, whereas NUMA gives fast local access but slower remote access

SGOU

NUMA systems scale far better than UMA systems because each node's local memory bandwidth is not shared with every other processor in the machine — only remote accesses compete for the shared interconnect. However, this benefit comes at the cost of programming and operating-system complexity: for best performance, the OS scheduler and memory allocator should try to keep a process's memory allocated on the same node as the processor running it, a policy generally known as NUMA-aware scheduling or 'first-touch' memory placement.

Interesting fact

Most modern multi-socket servers (two or more physical CPU sockets on one motherboard) are NUMA machines internally, even though a single operating system image presents them to the programmer as one big shared-memory computer. Poor NUMA placement, for example, a thread on socket 0 repeatedly touching memory allocated on socket 1, can silently slow a program down by 30% or more without any change to the code itself.

Operating systems expose the relative cost of remote access through a NUMA distance matrix, in which each node reports a numeric 'distance' to every other node, by convention, a node's distance to its own local memory is normalised to 10, and larger numbers indicate proportionally slower access. Table 4.3.2 shows an illustrative two-node example.

Table 4.3.2 An illustrative NUMA distance matrix — remote access is roughly twice as costly as local access in this example

Requesting Node	Distance to Node 0's memory	Distance to Node 1's memory
Node 0	10 (local)	21 (remote)
Node 1	21 (remote)	10 (local)

Worked illustration (average access time)

A program running on Node 0 makes 70% of its memory accesses to local memory (assume 100 ns) and 30% to Node 1's memory (about 210 ns, as the distances 10 and 21 suggest). The average access time is 0.7 × 100 + 0.3 × 210 = 70 + 63 = 133 ns. If NUMA-aware placement raises the local share to 95%, it becomes 0.95 × 100 + 0.05× 210 = 95 + 10.5 = 105.5 ns, about 21% lower, without changing the program at all.

A NUMA-aware scheduler and memory allocator use exactly this kind of information: whenever possible, a process's memory pages are allocated on the same node as the processor running it (a policy often called 'first-touch' allocation, since the node that first writes to a page becomes its home node), and the scheduler tries to keep a process on that same node across context switches rather than migrating it freely, as it would on a UMA system. The relevant information is presented in Table 4.3.3.

SGOU

4.3.5 UMA vs NUMA — A Summary Comparison

Table 4.3.3 UMA versus NUMA

Feature	UMA	NUMA
Memory organisation	Single shared memory pool	Memory physically partitioned by node
Access latency	Uniform for all processors	Non-uniform: fast local, slow remote
Typical interconnect	Common bus / crossbar	Node-to-node interconnect (e.g., mesh, ring)
Scalability	Limited (bus contention)	Higher, but placement-sensitive
OS scheduling concern	Minimal	Must consider processor/ memory affinity
Example systems	Small SMP servers	Large multi-socket servers, HPC nodes

4.3.6 Cache Coherence

Every processor in a shared-memory multiprocessor typically has its own private cache to reduce the average memory access time. This creates a new problem: if two or more processors keep a private, cached copy of the same memory location, and one processor updates its copy, the other processors' cached copies become stale unless they are somehow informed. The cache coherence problem is precisely this: ensuring that all processors observe a single, consistent view of the value stored at any given memory location, even though multiple cached copies of it may physically exist. The concept is illustrated in Figure 4.3.7.

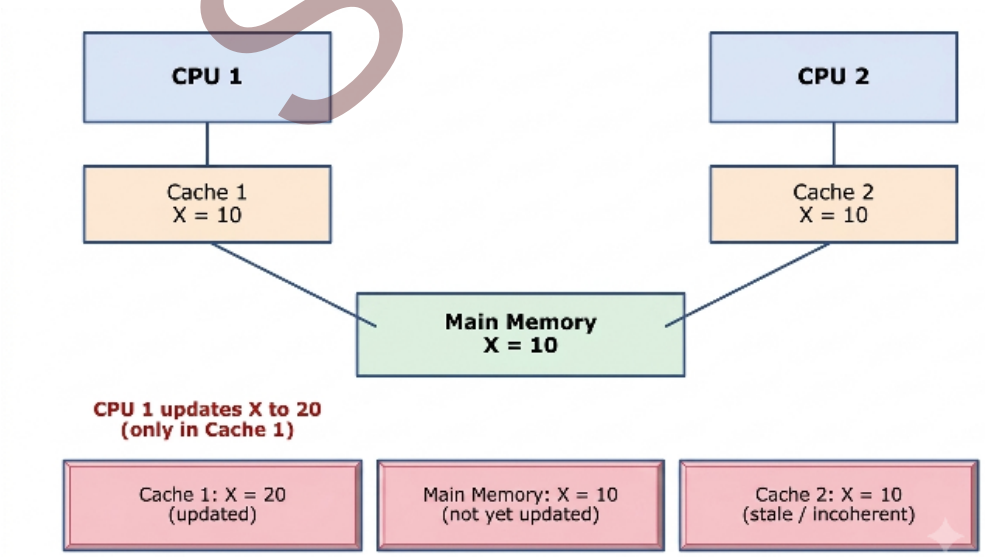


Fig. 4.3.7 The cache coherence problem: CPU 1 updates its cached copy of X to 20, while CPU 2's cache and main memory still hold the stale value 10

SGOU

A memory system is said to be coherent if, informally, (i) a read by any processor of a location always returns the most recently written value to that location, and (ii) writes to the same location by different processors are seen in the same order by all processors. Without a coherence mechanism, a parallel program could silently compute wrong results simply because different processors disagree about the current value of a shared variable.

4.3.6.1 Cache Coherence Protocols

Coherence is normally enforced in hardware using one of two broad protocol families:

- Write-invalidate protocols: when a processor writes to a shared location, all other cached copies of that location are invalidated (marked invalid). The next time another processor needs that data, it must fetch the fresh value from memory or from the writer's cache.
- Write-update (write-broadcast) protocols: when a processor writes to a shared location, the new value is immediately broadcast and updated in every other cache holding a copy, so that no cache ever becomes stale.

Write-invalidate protocols are far more common in commercial processors because they generate less bus/interconnect traffic than write-update protocols, especially when a variable is written repeatedly by the same processor before being read elsewhere.

4.3.6.2 The MESI Protocol

MESI is a widely used write-invalidate cache coherence protocol, named after the four states a cache line can be in: Modified, Exclusive, Shared, and Invalid.

- Modified (M): the cache line is present only in this cache and has been changed; it differs from main memory, and this cache is responsible for eventually writing it back.
- Exclusive (E): the cache line is present only in this cache but matches main memory (it has not been modified).
- Shared (S): the cache line may be present in multiple caches simultaneously, and all copies match main memory.
- Invalid (I): the cache line is not valid in this cache and must be fetched before use. The concept is illustrated in Figure 4.3.8.

SGOU

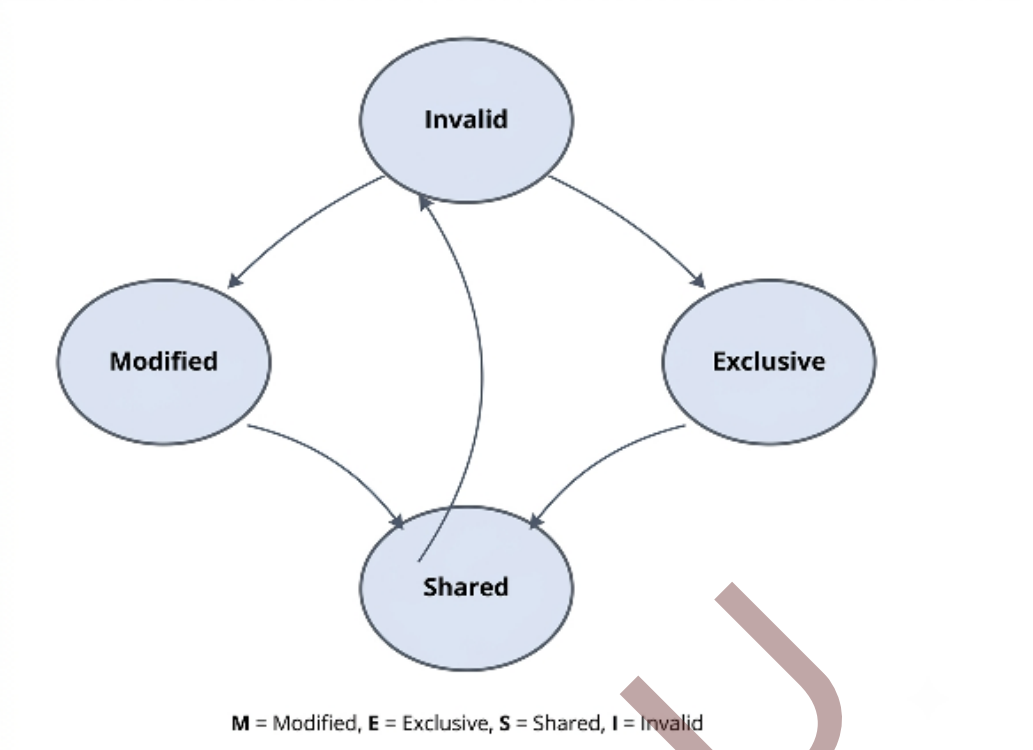


Fig. 4.3.8 State transition diagram for the MESI cache coherence protocol

Worked illustration

Processor A reads variable Y — its cache line moves to Exclusive since no one else has it. Processor B then also reads Y; A's copy and B's copy both move to Shared. Now A writes to Y: A's copy moves to Modified, and a bus 'invalidate' signal forces B's copy to Invalid. If B tries to read Y again, it must fetch the updated value, and both copies settle into Shared once more (assuming A's dirty value is written back or supplied directly to B).

4.3.6.3 Snooping vs Directory-Based Coherence

Coherence protocols are also classified by how a processor discovers that a location it has cached has been modified elsewhere. In a snooping protocol, every cache is connected to a shared broadcast medium (typically a bus) and continuously 'listens' (snoops) to every memory transaction that crosses it, invalidating or updating its own copy of a line whenever it sees another processor writing to that address. Snooping is simple and has low latency but does not scale well, because a shared bus that every cache must monitor becomes congested as more processors are added.

A directory-based protocol solves this scalability problem by maintaining a directory — a centralised or distributed data structure that records, for every cache line, which caches currently hold a copy and in what state. When a processor writes to a line, it consults the directory (rather than broadcasting to everyone) and the directory sends invalidation messages only to the specific caches that actually hold a copy. This point-to-point signalling scales to far larger numbers of processors than a shared broadcast

SGOU

bus, which is why directory-based coherence is the standard choice in large NUMA and distributed shared-memory systems, while snooping remains common in smaller bus-based UMA/SMP systems. The concept is illustrated in Figure 4.3.9. The relevant information is presented in Table 4.3.4.

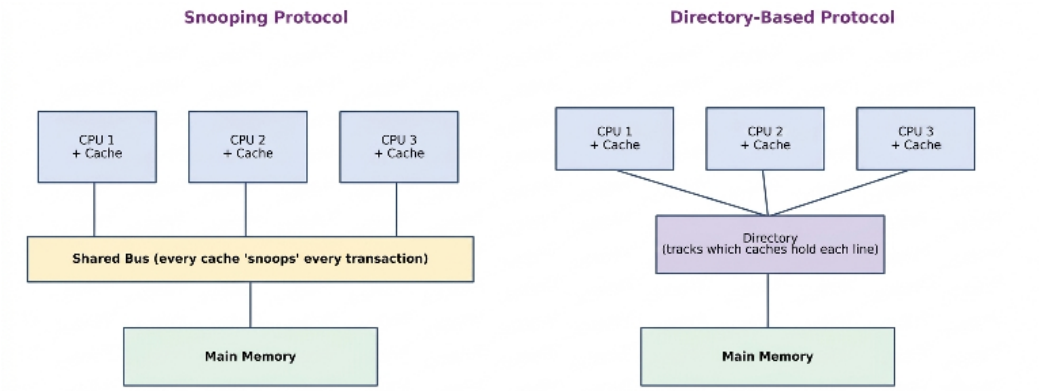


Fig. 4.3.9 Snooping coherence (every cache monitors a shared bus) versus directory-based coherence (a directory tracks sharers and sends targeted invalidations)

Table 4.3.4 Snooping versus directory-based cache coherence

Feature	Snooping Protocol	Directory-Based Protocol
Communication	Broadcast over shared bus	Point-to-point via directory
Scalability	Poor beyond a small bus	Good — scales to many nodes
Latency (small systems)	Low	Slightly higher (directory lookup)
Typical use	Small SMP / UMA systems	Large NUMA / distributed shared memory

Related concept (false sharing)

Even when two processors are logically working on entirely different variables, if those variables happen to fall within the same cache line, a write by one processor will invalidate the other's copy of the whole line, causing needless coherence traffic. This performance pitfall, called false sharing, can be mitigated by padding or aligning frequently updated variables so that each lands on its own cache line.

Worked illustration (false sharing)

Four threads on four cores each increment their own counter, and the four 4-byte counters sit side by side in one array. Together they occupy 16 bytes, so they share a single 64-byte cache line. Every increment by one core invalidates that line in the other three caches, so the line bounces between cores even though no counter is actually shared, and the program may run slower on four cores than on one. Padding each counter to its own 64-byte cache line removes the interference.

SGOU

4.3.7 Memory Consistency

Cache coherence guarantees that all processors eventually agree on the value of a single memory location. Memory consistency is a related but distinct concern: it defines the rules governing the order in which memory operations — to potentially different locations, issued by potentially different processors — appear to occur to the rest of the system. In other words, coherence is about one location; consistency is about the ordering of operations across many locations.

The strongest and most intuitive consistency model is sequential consistency, which requires that the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. Sequential consistency is easy to reason about but restricts many hardware and compiler optimisations (such as reordering instructions or buffering writes), so many real processors implement relaxed consistency models — for example, allowing a processor's own reads to be reordered ahead of its own writes to different addresses — and rely on the programmer to insert explicit memory barriers (fence instructions) at points where strict ordering genuinely matters, such as when releasing a lock.

Note the distinction

Cache coherence is typically enforced automatically by hardware and is invisible to the programmer, whereas memory consistency models are part of the programmer-visible contract of the architecture — a multi-threaded program that assumes sequential consistency may behave incorrectly on hardware that implements a relaxed consistency model unless proper synchronisation primitives are used.

Worked illustration (coherent but not consistent)

Processor 1 executes data = 42; flag = 1. Processor 2 executes while (flag == 0) { }; print(data). Each location is coherent, yet under a relaxed consistency model Processor 2 may see flag = 1 before it sees data = 42 (for example because Processor 1's writes were reordered or buffered) and print the old value 0. Under sequential consistency the program always prints 42. On relaxed hardware the programmer inserts a memory barrier between the two writes.

4.3.8 Bringing It Together

The choice of memory organisation is one of the earliest and most consequential decisions in designing a parallel or high-performance computer. Shared memory (further divided into UMA and NUMA) favours ease of programming at the expense of scalability, while distributed memory favours scalability at the expense of programming convenience. Cache coherence protocols such as MESI make shared memory practical by hiding the existence of multiple cached copies from the programmer, while memory consistency models determine what ordering guarantees the programmer can actually rely on when writing correct parallel programs. A solid grasp of these ideas is essential

SGOU

before studying parallel programming models and multiprocessor operating systems, which is the subject of the next unit.

Recap

- Shared memory multiprocessors provide a single global address space accessed by all processors; distributed memory systems give each processor a private local memory, with communication via explicit message passing.
- Shared memory is tightly coupled and easier to program; distributed memory is loosely coupled and scales to far larger numbers of processors.
- Uniform Memory Access (UMA) gives every processor identical latency to shared memory, typically via a common bus or crossbar; it is simple but does not scale well beyond a modest number of processors.
- Non-Uniform Memory Access (NUMA) partitions memory by node; local access is fast, remote access is slower, giving better scalability at the cost of placement sensitivity.
- The cache coherence problem arises when multiple processors cache copies of the same memory location and one is updated without informing the others.
- Write-invalidate protocols (such as MESI, with states Modified, Exclusive, Shared, Invalid) are the dominant hardware mechanism for maintaining cache coherence.
- Memory consistency defines the ordering guarantees for operations to different memory locations across processors; sequential consistency is the strongest and most intuitive model, but many systems use relaxed models with explicit synchronisation for performance.
- Coherence is generally automatic and hardware-managed, whereas consistency is part of the architecture's programmer-visible contract.
- Interconnection networks (bus, crossbar switch, mesh/multistage) trade off cost, bandwidth, and scalability, and underpin both shared- and distributed-memory designs.
- Snooping coherence protocols broadcast invalidations over a shared bus and suit small systems; directory-based protocols use targeted, point-to-point signalling and scale to much larger systems.
- False sharing occurs when unrelated variables share a cache line, causing unnecessary coherence traffic even without genuine data sharing.

SGOU

Objective Type Questions

1. Which memory architecture allows multiple processors to access a common address space?

2. Which shared memory architecture provides equal memory access latency to all processors?

3. What is the main scalability limitation of a UMA architecture?

4. In which memory architecture does local memory generally have lower access latency than remote memory?

5. Which problem occurs when multiple processors cache copies of the same memory location?

6. Which cache coherence protocol uses the states Modified, Exclusive, Shared and Invalid?

7. Which cache coherence protocol invalidates other cached copies when a processor writes to a cache block?

8. Which concept defines the ordering of memory operations as observed by multiple processors?

9. Which memory consistency model requires operations to appear in a single global order that preserves each processor's program order?

10. Which cache coherence approach uses a directory to track the caches holding a memory block?

11. What is the term for performance degradation caused by different variables sharing the same cache line?

12. Which interconnection topology provides a dedicated path between each processor and each memory module?

13. What is the average memory access time when local access takes 100 ns, remote access takes 200 ns, and 80% of accesses are local?

14. What technique places a thread on the processor or NUMA node best suited to its workload?

15. Which communication model is commonly used by processes in a distributed memory system to exchange data?

SGOU

Answers

1. Shared memory
2. UMA
3. Contention
4. NUMA
5. Cache coherence
6. MESI
7. Write invalidate
8. Memory consistency
9. Sequential consistency
10. Directory based
11. False sharing
12. Crossbar
13. 120 ns
14. Processor affinity
15. Message passing

SGOU

Assignments

1. Compare and contrast shared memory and distributed memory architectures, giving suitable examples of systems that use each.
2. With a neat diagram, explain the organisation and working of a Uniform Memory Access (UMA) multiprocessor system.
3. Explain Non-Uniform Memory Access (NUMA) architecture with a diagram, and discuss why NUMA-aware scheduling matters for performance.
4. Tabulate at least six points of difference between UMA and NUMA architectures.

5. What is the cache coherence problem? Illustrate it with a suitable example involving two processors sharing a variable.

6. Describe the MESI cache coherence protocol, including its four states and the transitions between them.

7. Differentiate between cache coherence and memory consistency, and explain why a system can be coherent yet still produce unexpected results without proper synchronisation.

8. Explain sequential consistency and briefly discuss why many modern processors implement relaxed consistency models instead.

9. Describe bus, crossbar, and mesh interconnection networks, and discuss the scalability trade-offs of each.

10. Distinguish between snooping and directory-based cache coherence protocols, and explain why directory-based protocols are preferred in large NUMA systems.

11. On a two-node NUMA system with local distance 10 and remote distance 21, calculate the average memory distance when a process keeps (a) 60%, (b) 90% of its pages on its own node. Explain what this implies for the scheduler.

12. Trace the MESI states for three processors P1, P2 and P3 in the sequence: P1 reads X, P2 reads X, P1 writes X, P3 reads X, P2 writes X. Present your answer as a table showing the state in each cache after every step.

13. Compare bus, crossbar and mesh interconnects for 8, 64 and 1,024 processors in terms of cost and bandwidth, and choose a suitable interconnect for each size.

14. Using a flag-and-data example, analyse why a program can be coherent yet still incorrect without proper synchronisation.

15. A scientific code speeds up 3.1 times on four sockets of a NUMA server, but only 1.4 times when one thread initialises all the data. Analyse the result using the first-touch placement policy.

SGOU

Reference

1. Hennessy, J. L., & Patterson, D. A. (2017). *Computer architecture: A quantitative approach* (6th ed.). Morgan Kaufmann.

2. Culler, D. E., Singh, J. P., & Gupta, A. (1999). *Parallel computer architecture: A hardware/software approach*. Morgan Kaufmann.

3. Stallings, W. (2016). *Computer organization and architecture: Designing for performance* (10th ed.). Pearson.

Suggested Reading

1. Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to parallel computing* (2nd ed.). Pearson.

2. Hwang, K., & Jotwani, N. (2016). *Advanced computer architecture: Parallelism, scalability, programmability* (3rd ed.). McGraw-Hill Education.

SGOU



Multiprocessor Operating Systems

Learning Outcomes

After completion of this unit, learner will be able to:

- explain the basic organisation and responsibilities of a multiprocessor operating system
- describe the process synchronization problem and apply mutexes and semaphores to solve classic synchronization problems
- explain the principal approaches to task scheduling on multiprocessor systems, including load balancing and processor affinity
- describe the major inter-process communication (IPC) mechanisms and identify which mechanism suits a given scenario, and
- analyse simple multiprocessor OS scenarios and identify potential race conditions, deadlocks, or scheduling inefficiencies

Prerequisite

This unit builds on Unit 3's treatment of shared and distributed memory organisations, UMA/NUMA architectures, and cache coherence, since the operating system's design choices are shaped directly by the underlying hardware. A basic understanding of single-processor operating system concepts, including processes, threads, the process control block, and critical sections, is assumed, along with elementary familiarity with how the CPU scheduler chooses which ready process runs next. No prior knowledge of any specific programming language's threading API is required, though the pseudocode presented here will be easiest to follow with basic programming background. A familiar real-world analogy runs through the unit: a multiprocessor operating system managing several cores is much like several bank tellers sharing one queue of customers and one till of cash, they must take turns safely accessing shared resources (synchronization), share the workload fairly (scheduling), and pass information to one another (inter-process communication) without causing errors or delays.

SGOU

Keywords

Multiprocessing, Mutex, Semaphore, Critical Section, Race Condition, Load Balancing, Processor Affinity, Starvation, Spinlock, Gang Scheduling, Inter-Process Communication (IPC)

Discussion

4.4.1 Multiprocessing Operating Systems: Basics

A multiprocessor operating system manages a computer that contains more than one processor sharing memory, I/O devices, and a single copy of the operating system (or a coordinated set of kernel instances). Its central job is to manage processes, memory, and devices while coordinating task execution across processors, ensuring safe access to shared kernel data structures, and maintaining communication and synchronisation among processes running on different processors.

Multiprocessor operating systems are broadly organised in one of three ways. The concept is illustrated in Fig 4.4.1.

4.4.1.1 Master-Slave Organisation

In a master-slave organisation, one designated processor (the master) runs the operating system kernel and makes all scheduling and resource-management decisions; the remaining processors (slaves) execute only user-level application code and request kernel services from the master. This design is simple to implement, since kernel data structures are only ever touched by one processor and therefore need no locking, but the master processor is a bottleneck and a single point of failure, if it becomes overloaded, or fails, the entire system suffers.

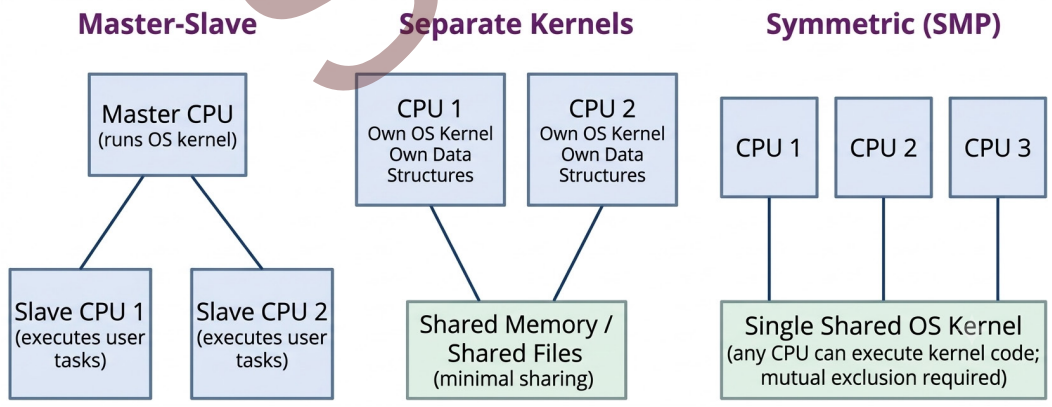


Fig. 4.4.1 Three common ways of organising a multiprocessor operating system: master-slave, separate kernels, and symmetric multiprocessing (SMP)

SGOU

4.4.1.2 Separate Kernels (Asymmetric per-node OS)

In this architecture, each processor runs its own copy of the operating system kernel with private data structures, while processors cooperate loosely by sharing selected files or small memory regions. This avoids kernel-level lock contention entirely but sacrifices the ability to load-balance processes across processors transparently, since each processor's OS instance largely manages its own workload independently.

4.4.1.3 Symmetric Multiprocessing (SMP)

In Symmetric Multiprocessing (SMP), used by modern operating systems such as Linux and Windows, all processors share a single operating system kernel, and any processor can execute kernel code, handle interrupts, or run processes. This gives excellent load balancing, since any idle processor can pick up any ready process, but it requires the kernel's internal data structures (such as the ready queue, open file table, and memory allocator) to be protected with locks so that two processors never corrupt shared kernel state by updating it simultaneously.

In practice

Nearly every laptop, desktop, and server operating system in use today(Linux, Windows, macOS) is a symmetric multiprocessing (SMP) design internally, even on machines with just two CPU cores, because SMP gives the best combination of flexibility and hardware utilisation.

4.4.1.4 Responsibilities of a Multiprocessor OS

Regardless of its organisation, a multiprocessor operating system must handle several responsibilities that have no counterpart, or only a much simpler counterpart, on a single-processor system. It must maintain a consistent view of shared kernel data structures across processors that may be executing kernel code at the same instant; it must decide, for every runnable task, both when it should run and on which processor, taking into account cache and memory locality; it must coordinate interrupt handling so that hardware interrupts can be routed to, and serviced by, an appropriate processor rather than always the same one; and it must provide synchronization primitives, the subject of the next section, that work correctly even when the 'other' process really is running at the same physical instant on a different core, rather than merely appearing to via time-slicing.

Contrast with uniprocessor systems

On a single processor, disabling interrupts is often sufficient to protect a short critical section, since no other code can run until interrupts are re-enabled. On a multiprocessor, this trick does not work, because another processor can still be executing concurrently and accessing the same shared data genuine mutual-exclusion primitives such as locks are unavoidable.

SGOU

4.4.2 Process Synchronization

When multiple processors (or multiple threads on the same processor) can execute concurrently and access shared data, the order in which their instructions actually interleave becomes unpredictable. If two threads update a shared variable without coordination, the final result can depend on the precise timing of their execution; a bug known as a race condition. The segment of code in which a process accesses shared data that must not be concurrently accessed by another process is called a critical section, and process synchronization refers to the set of mechanisms used to ensure that only one process is executing inside a critical section for a given resource at any time (mutual exclusion), while avoiding deadlock and starvation.

Worked illustration (race condition)

Suppose a shared counter variable count = 5, and two threads each execute count = count + 1 concurrently. If both threads read count as 5 before either writes back the incremented value, both will write back 6 even though two increments occurred, the counter only advances by one. This is a classic race condition caused by an unprotected critical section.

4.4.2.1 Mutex

A mutex (short for 'mutual exclusion') is the simplest synchronization primitive: a lock that can be in one of two states, locked or unlocked. A process wishing to enter a critical section must first acquire (lock) the mutex; if it is already locked by another process, the requesting process blocks (waits) until the mutex is released. Crucially, a mutex has ownership semantics; only the process or thread that locked a mutex is normally permitted to unlock it, which helps prevent accidental or malicious release by an unrelated thread.

4.4.2.2 Semaphores

A semaphore, introduced by Edsger Dijkstra, is a more general synchronization tool: an integer variable that can only be accessed through two atomic operations, conventionally called wait() (also written P()) and signal() (also written V()). wait() decrements the semaphore's value and blocks the calling process if the value becomes negative; signal() increments the value and, if there are waiting processes, wakes one of them. A binary semaphore (restricted to values 0 and 1) behaves much like a mutex, but a counting semaphore, whose value can range over any non-negative integer, can additionally control access to a pool of several identical resources. For example, allowing up to N processes to use N available printers concurrently. The concept is illustrated in Fig. 4.4.2.

SGOU

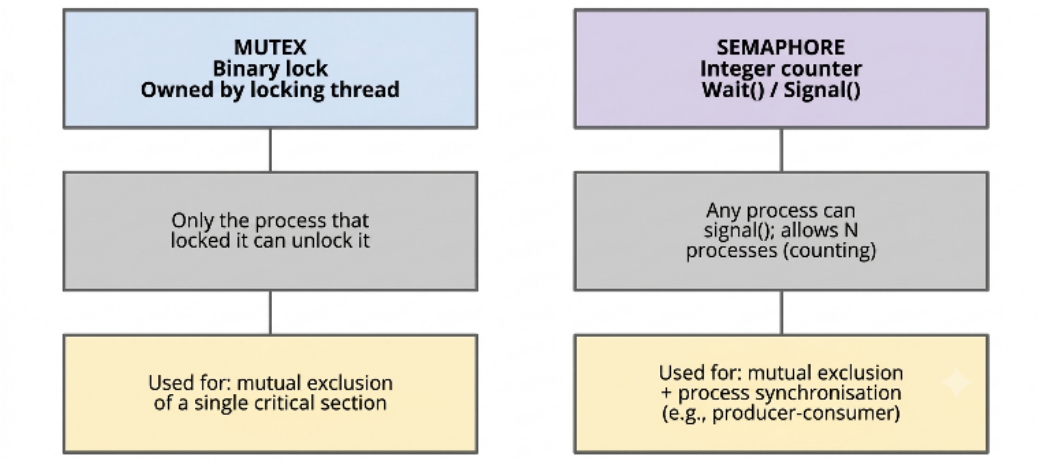


Fig. 4.4.2 Mutex versus semaphore: a mutex is a binary lock owned by its locking thread, while a semaphore is a more general counting primitive usable for both mutual exclusion and process synchronisation

4.4.2.3 The Producer–Consumer Problem

The producer–consumer (bounded-buffer) problem is a classic illustration of semaphore-based synchronization. A producer process generates data items and places them into a fixed-size shared buffer; a consumer process removes items from the buffer for processing. Two conditions must be enforced: the producer must not add an item when the buffer is full, and the consumer must not remove an item when the buffer is empty. This problem is solved using two counting semaphores, empty (initialised to the buffer size) and full (initialised to zero), along with a binary semaphore, mutex, to protect the buffer's internal pointers from simultaneous access.

The concept is illustrated in Fig. 4.4.3.

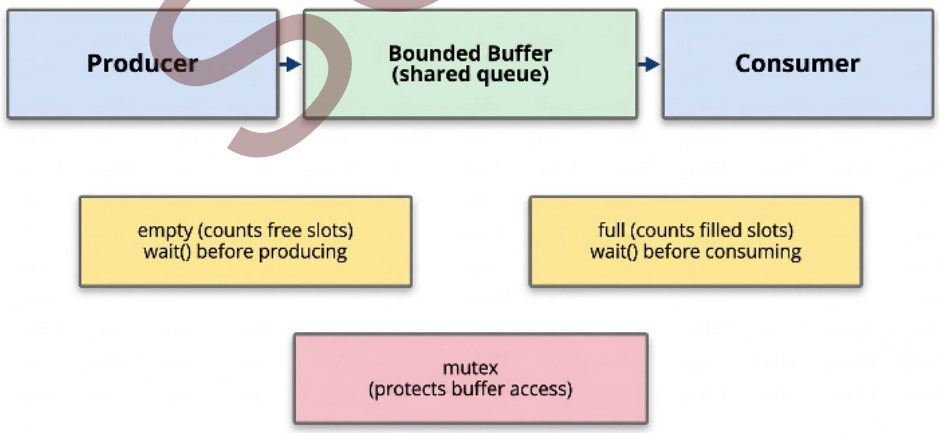


Fig. 4.4.3 The producer–consumer problem solved using two counting semaphores (empty, full) and a binary mutex protecting the shared bounded buffer

SGOU

Worked illustration

With a buffer of size 5, empty starts at 5 and full starts at 0. Before inserting an item, the producer executes wait(empty), this succeeds four times without blocking, but on the sixth attempt (buffer already full) the producer blocks until the consumer calls signal(empty) by removing an item. Symmetrically, the consumer calls wait(full) before removing an item and blocks if the buffer is empty, until the producer calls signal(full).

4.4.2.4 Spinlocks

On a multiprocessor system, an additional locking option becomes attractive: a spinlock. Instead of blocking (and being descheduled) when a lock is unavailable, a process repeatedly tests the lock in a tight loop ('busy-waiting') until it becomes free. Busy-waiting wastes CPU cycles on a single-processor system, since the lock cannot be released while the only CPU is spinning, but on a multiprocessor, the process holding the lock continues running on a different processor and will release it shortly, so a short spin can be cheaper than the overhead of a full context switch. For this reason, spinlocks are widely used inside multiprocessor operating system kernels to protect short critical sections, while ordinary blocking mutexes and semaphores remain preferable for longer critical sections.

What is a short or a long critical section?

A critical section is short when the lock is held for about as long as a context switch takes, or less: a few hundred nanoseconds to a few microseconds, for example updating a counter or linking a node into a kernel list. It is long when the holder may run for much longer or may itself block, for example while waiting for disk or network input/output or performing a lengthy computation. A spinlock keeps the waiting processor busy doing nothing useful, so it consumes CPU cycles for the whole wait. It is therefore appropriate mainly when the expected wait is short and the processor holding the lock is likely to run promptly, so that the lock will be released soon. For longer waits, blocking synchronisation (a mutex or semaphore) is generally preferable, because the waiting thread sleeps and its processor can run other work.

Working illustration

Suppose a context switch costs about 5 µs (an illustrative figure). A thread that needs a lock held for 0.2 µs wastes at most about 0.2 µs by spinning, far less than the 5 µs of blocking and waking. If the lock is held for 5 ms (while a log record is written to disk), spinning wastes 5 ms of CPU time, about a thousand times the cost of one context switch, and blocking is clearly better. If the lock holder is itself pre-empted and descheduled, even a "short" lock can be held for an entire time slice, which is one reason spinlocks are used only in short, carefully controlled kernel paths.

SGOU

4.4.2.5 The Readers–Writers Problem

The readers–writers problem models a shared resource, such as a database record or a shared data structure, that multiple reader processes may safely examine simultaneously (since reading does not modify the data), but that a writer process must access exclusively, with no readers or other writers present at the same time. A correct solution typically uses a counting mechanism to track the number of active readers together with a mutex protecting that count, and a second lock that a writer must hold for the entire duration of its access; the first reader to arrive locks out writers, and the last reader to leave releases the lock for a waiting writer. The concept is illustrated in Fig. 4.4.4.

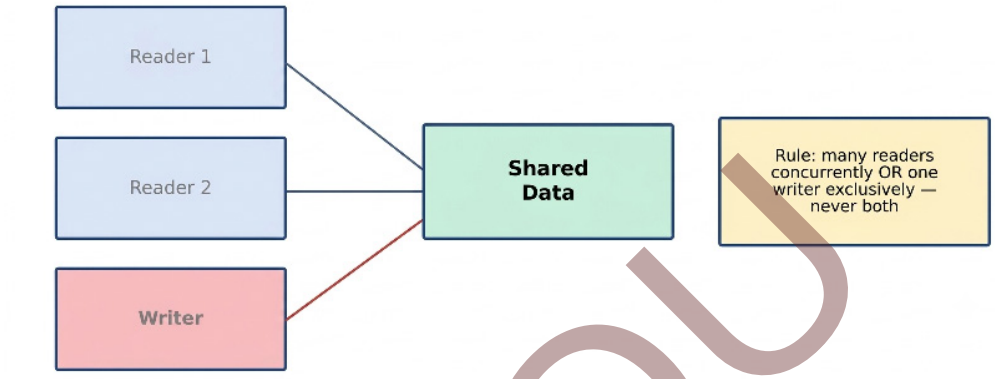


Fig. 4.4.4 The readers–writers problem: any number of readers may access shared data concurrently, but a writer requires exclusive access

Design note

A naive readers-writers solution can starve writers indefinitely if readers keep arriving (reader preference). A fair solution typically ensures that once a writer is waiting, no new readers are admitted until that writer has had its turn.

4.4.2.6 Deadlock: A Related Hazard

Synchronization primitives solve the race-condition problem, but used carelessly they can introduce a different hazard: deadlock, a state in which two or more processes are each waiting for a resource held by another, so that none of them can ever proceed. A deadlock can occur only if four necessary conditions hold simultaneously: mutual exclusion (resources cannot be shared), hold-and-wait (a process holds one resource while waiting for another), no preemption (a resource cannot be forcibly taken away), and circular wait (a cycle of processes each waiting on the next). The concept is illustrated in Fig. 4.4.5.

SGOU

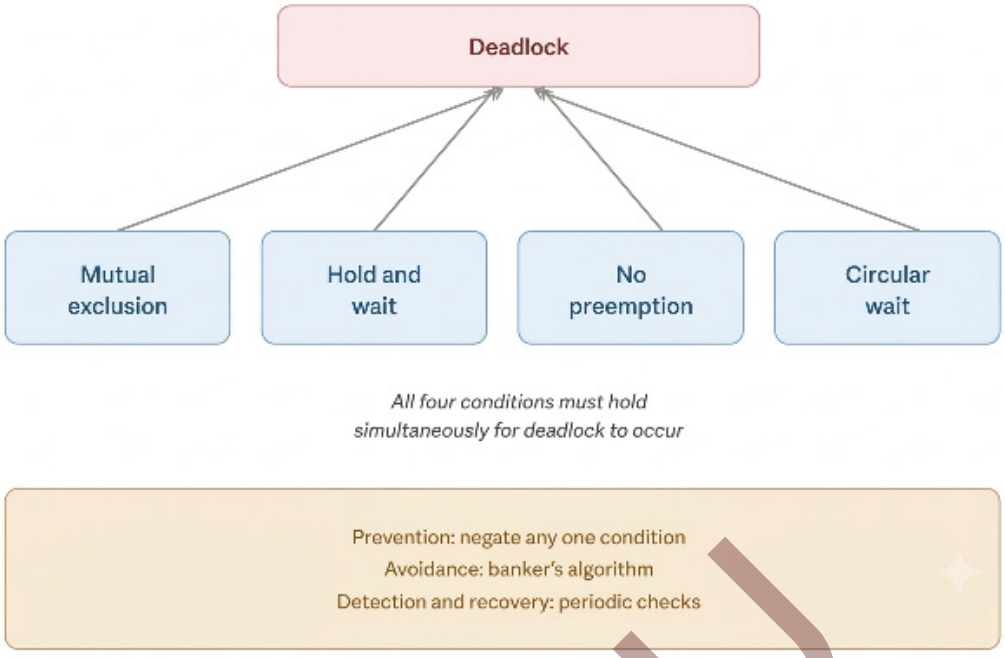


Fig. 4.4.5 The four necessary conditions for deadlock; breaking any single condition prevents deadlock from occurring

Operating systems generally address deadlock in one of three ways: prevention (designing the system so that at least one necessary condition can never hold. For example, requiring all processes to request resources in a fixed global order, which rules out circular wait), avoidance (using an algorithm such as the Banker's algorithm to grant resource requests only if the resulting state is still guaranteed safe), or detection and recovery (allowing deadlock to occur but periodically checking for it and then terminating or rolling back one of the involved processes to break the cycle).

4.4.2.7 Deadlock versus Starvation

Deadlock and starvation are both failures of progress, and they are easily confused, but they are different problems with different remedies. In a deadlock, a group of processes is stuck in a cycle: each holds a resource that another needs, so none of them can ever proceed unless the system intervenes. In starvation (also called indefinite postponement), the system as a whole keeps making progress, but one particular process is repeatedly passed over and never obtains the processor or resource it needs. The relevant information is presented in Table 4.4.1.

SGOU

Table 4.4.1 Deadlock versus Starvation

Aspect	Deadlock	Starvation
What happens	A set of processes wait for one another in a cycle; none can proceed	One process waits indefinitely while others continue to run
Cause	Mutual exclusion, hold-and-wait, no pre-emption and circular wait all hold together	Unfair policy: priorities, a scheduler that always favours others, or reader preference in readers-writers
Is the system making progress?	Not for the processes in the cycle (possibly none at all)	Yes, other processes progress normally
Resolves on its own?	No; needs prevention, avoidance or detection and recovery	Only if the competing load stops; usually needs a fairness mechanism
Typical remedy	Fixed resource ordering, Banker's algorithm, abort or roll back a process	Aging (raise priority with waiting time), FIFO or ticket queues, writer preference
Multiprocessor example	Thread 1 holds lock A and waits for B while thread 2 holds B and waits for A	A task on a busy processor's private queue never runs because higher-priority tasks keep arriving; a non-FIFO spinlock lets one CPU lose every race

Working illustration

On a two-processor system, thread T1 locks A then requests B, while thread T2 locks B then requests A. Neither can continue: this is a deadlock. In contrast, suppose a writer is waiting for a database record while new readers keep arriving and are always admitted first: the readers progress, but the writer never does. This is starvation, and it is cured by a fair rule such as "once a writer is waiting, admit no new readers". Note that a deadlocked process is also starved of its resource, but a starved process is not necessarily part of a deadlock.

4.4.3 Task Scheduling in Multiprocessors

On a single-processor system, scheduling means choosing one process from the ready queue to run next. On a multiprocessor system, the scheduler must additionally decide on which processor a given task should run, which introduces several new considerations not present in uniprocessor scheduling. The concept is illustrated in Fig 4.4.6.

SGOU

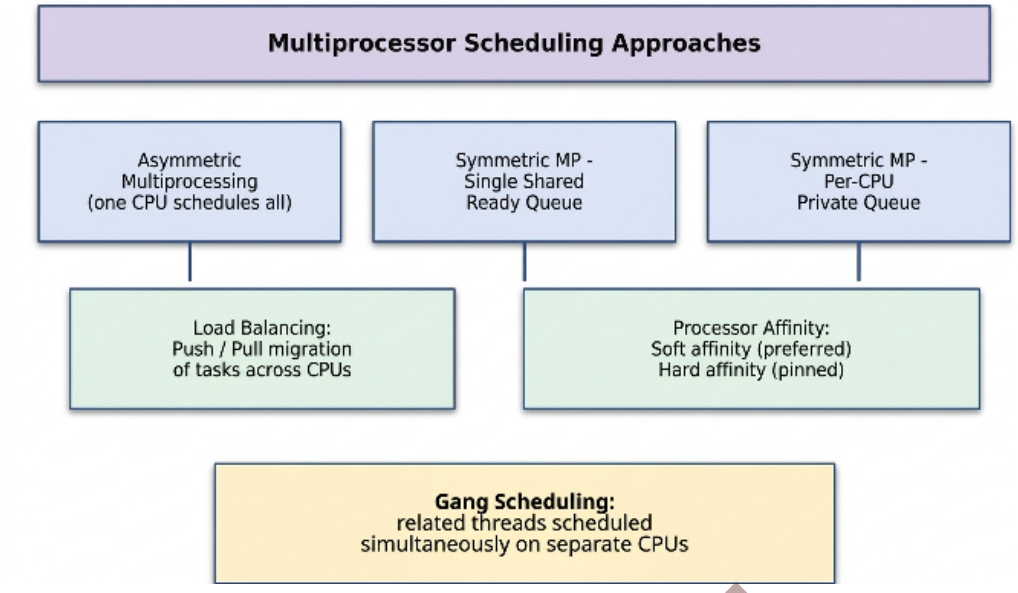


Fig. 4.4.6 Key concerns in multiprocessor task scheduling: organisation of the ready queue, load balancing, processor affinity, and gang scheduling

4.4.3.1 Ready Queue Organisation

In an asymmetric multiprocessing scheme, a single designated processor handles all scheduling decisions for the whole system. In symmetric multiprocessing, scheduling can use either a single shared ready queue, from which any idle processor pulls the next runnable task (simple, but the queue itself must be protected by a lock, and can become a contention point), or per-processor private ready queues, in which each processor mostly schedules from its own queue, reducing contention but requiring an explicit load-balancing mechanism to keep queues roughly equal in length.

4.4.3.2 Load Balancing

Load balancing keeps work evenly distributed among processors so that no processor sits idle while another has a long backlog. Two general strategies exist: push migration, in which a dedicated system task periodically checks processor loads and pushes (moves) tasks from overloaded processors to idle ones; and pull migration, in which an idle processor actively pulls a waiting task from a busy processor's queue. Many real operating systems, including Linux, combine both strategies.

4.4.3.3 Processor Affinity

Because a processor's cache retains data from the process that most recently ran on it, migrating a process to a different processor discards this 'cache warmth' and forces the process to reload its working data into the new processor's cache, an effect that is even more pronounced on NUMA systems, discussed in Unit 3, where the process's memory itself may now be physically remote from the new processor. To reduce this cost, schedulers implement processor affinity: soft affinity means the scheduler will try, but is not guaranteed, to keep a process on the same processor, whereas hard affinity allows a process (or administrator) to explicitly pin a process to a specific subset of processors, and the scheduler must honour that restriction.

SGOU

4.4.3.4 Gang Scheduling

Some parallel applications consist of a set of closely cooperating threads that frequently synchronise with one another (for example, all waiting at a barrier). If these threads are scheduled independently and one is delayed, the others may spend most of their time waiting for it. Gang scheduling addresses this by scheduling all threads of such a group simultaneously across separate processors, so that they progress together and synchronisation overhead is minimised.

Fig. 4.4.7 shows a two-processor system running job A, whose two threads A1 and A2 must meet at a barrier, and job B, whose threads B1 and B2 are independent. Each time slot lasts a few milliseconds.

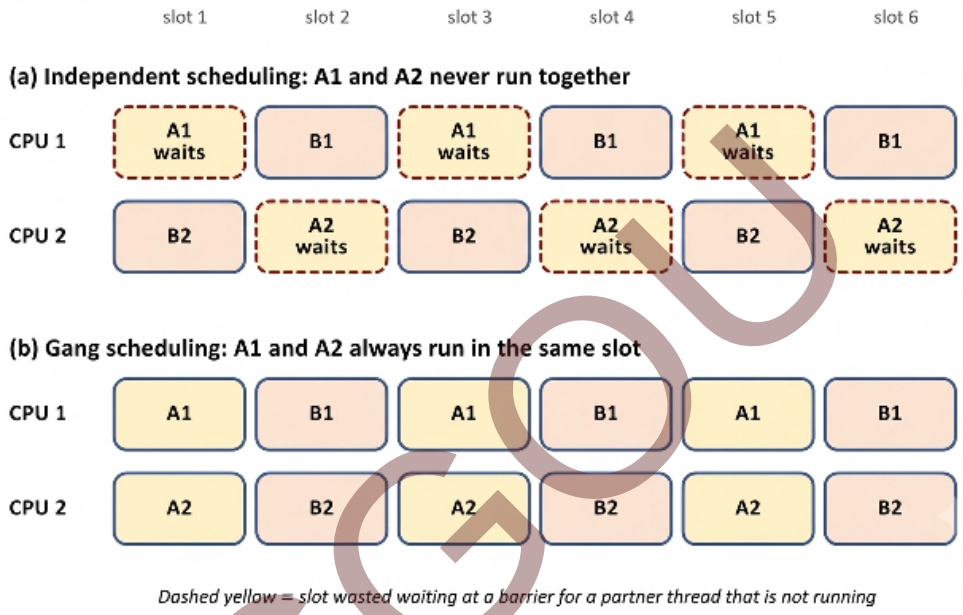


Fig. 4.4.7 Independent scheduling versus gang scheduling of two cooperating threads A1 and A2 on two CPUs; dashed slots are wasted waiting at a barrier

With independent scheduling (part a) the scheduler happens to place A1 and A2 in different slots. Whenever A1 runs, A2 is absent, so A1 reaches the barrier and can only wait; the same happens to A2 in the other slots. Half of all the time slots (6 of the 12 processor-slots) are therefore wasted. With gang scheduling (part b) both threads of A always receive the two CPUs together, in slots 1, 3 and 5, so they pass the barrier immediately, while B1 and B2 receive slots 2, 4 and 6. Each job still receives half of the machine; no slot is wasted waiting. The price is some inflexibility: while job A holds both CPUs, they cannot be given to other work even if one A thread is briefly idle.

SGOU

4.4.4 Scheduling Approaches (A Summary Comparison)

The relevant information is presented in Table 4.4.2.

Table 4.4.2 Multiprocessor scheduling approaches compared

Approach	Key Idea	Main Advantage	Main Drawback
Asymmetric (master-slave)	One CPU makes all scheduling decisions	Simple, no locking of scheduler data	Master CPU is a bottleneck
Shared ready queue (SMP)	All CPUs pull from one common queue	Naturally balanced load	Queue lock can bottleneck at scale
Private ready queues (SMP)	Each CPU has its own queue	Low contention	Needs active load balancing
Gang scheduling	Related threads scheduled together	Efficient synchronisation	Complex to implement fairly

4.4.5 Inter-Process Communication (IPC)

Processes executing concurrently, either on the same processor or on different processors in a multiprocessor system, often need to share data and coordinate their activities to ensure correct and orderly execution. This is known as process cooperation. Inter-process communication (IPC) is the collective term for the operating system facilities that allow this. IPC mechanisms fall into the same two broad categories already introduced for hardware memory organisation in Unit 3: shared memory and message passing. The concept is illustrated in Fig. 4.4.8.

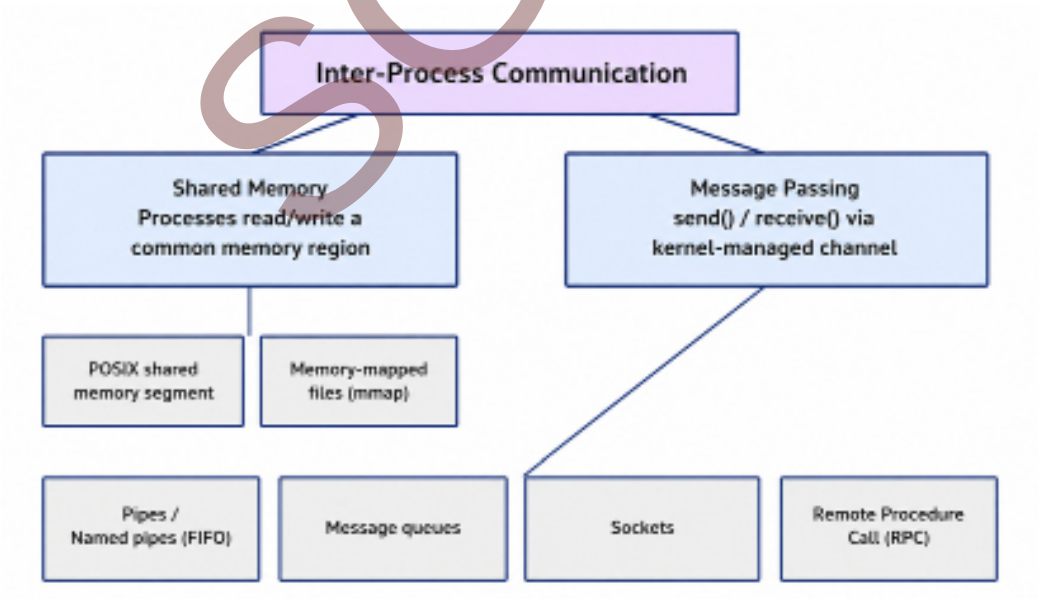


Fig. 4.4.8 Major inter-process communication mechanisms grouped under the shared-memory and message-passing models

SGOU

4.4.5.1 Shared Memory IPC

In shared memory IPC, the operating system sets up a region of memory that is mapped into the address space of two or more cooperating processes. Once mapped, the processes can read and write this region directly, exactly like ordinary memory, without further operating-system intervention which makes shared memory IPC extremely fast once set up. The cost of this speed is that the cooperating processes are entirely responsible for synchronising their own access to the shared region (typically using the mutexes or semaphores discussed earlier), since the operating system does not enforce any ordering on the accesses.

4.4.5.2 Message Passing IPC

In message passing IPC, processes communicate by sending and receiving discrete messages through the operating system, using two basic operations, send(message) and receive(message), rather than by directly sharing memory. Because the OS mediates every exchange, message passing is simpler to get right for processes that do not otherwise share memory (including processes on different machines in a distributed system), though it is generally slower than shared memory because every message typically involves at least one copy through the kernel. Common message-passing mechanisms include:

- **Pipes:** a unidirectional (or, for named pipes, bidirectional) byte stream connecting two related processes, commonly used to chain the output of one program into the input of another.
- **Named pipes (FIFOs):** similar to pipes but identified by a name in the file system, allowing unrelated processes to communicate.
- **Message queues (for example, POSIX message queues):** the OS maintains a queue of discrete, typed messages; senders and receivers need not be running at exactly the same time, and messages persist in the queue until read.
- **Sockets:** an endpoint for communication, usable both between processes on the same machine and across a network, forming the basis of most client-server and distributed applications.
- **Remote Procedure Call (RPC):** allows a process to invoke a procedure that executes in another process's (possibly remote) address space, hiding the underlying message exchange behind an ordinary-looking function call.
- **Signals:** a limited, asynchronous form of IPC in which the operating system notifies a process that a particular event has occurred (for example, a timer expiry or a request to terminate); signals carry no data payload beyond the event type itself.

Local IPC versus distributed communication

The mechanisms above fall into two groups according to their scope (Fig. 4.4.9). Local IPC (pipes, named pipes, POSIX message queues and shared memory) connect processes

SGOU

running on the same machine under one operating system. Distributed communication (sockets and remote procedure calls) connects processes that may run on different machines across a network. Sockets can also be used between two processes on one machine, but they are the basic mechanism for communication over a network. The two groups are compared in Table 4.4.3.

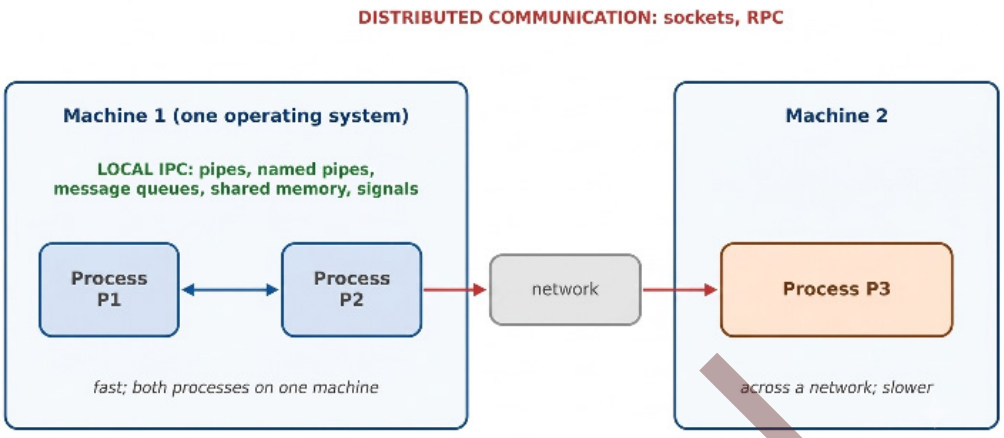


Fig. 4.4.9 Local IPC (pipes, named pipes, POSIX message queues, shared memory) versus distributed communication (sockets, RPC)

Table 4.4.3 Local IPC versus Distributed Communication

Aspect	Local IPC (pipes, POSIX message queues, shared memory)	Distributed communication (sockets, RPC)
Where the processes run	On the same machine, under one operating system	On the same or different machines connected by a network
How the peer is identified	File descriptor, a name in the file system or a queue name	IP address and port number; for RPC, a remote procedure name
Data format	Same machine, same data representation	May need marshalling (conversion) of data, especially for RPC
Speed	High: no network is involved	Lower: limited by network latency and bandwidth
Failures	Rare; both ends normally fail together	Messages can be lost or delayed and one end can fail alone, so time-outs and retries are needed
Typical use	Shell pipelines such as ls | grep, producer-consumer on one host	Web client and server, e-mail, distributed applications

Shared memory and message-passing IPC are compared in Table 4.4.4.

SGOU

Table 4.4.4 Shared memory versus message-passing IPC

Aspect	Shared Memory IPC	Message Passing IPC
Speed (after setup)	Very fast: direct memory access	Slower: data typically copied via the kernel
OS involvement	Minimal, only for initial setup	Every message goes through the OS
Synchronisation	Programmer's responsibility (mutex/semaphore)	Largely handled by the OS/IPC mechanism
Suitable for	Processes on the same machine, frequent data exchange	Related or unrelated processes, including across a network
Examples	POSIX shared memory, mmap	Pipes, message queues, sockets, RPC

Interesting fact

Modern web browsers such as Chrome run each browser tab as a separate operating-system process for stability and security, and rely heavily on message-passing IPC (built on top of sockets and pipes) so that a crash in one tab's rendering process cannot directly corrupt the memory of the main browser process or other tabs.

Multiprocessor operating systems must extend all traditional operating system responsibilities, such as process scheduling, memory management, and inter-process communication, to support multiple processors working concurrently and sharing system resources. Correct process synchronization using mutexes and semaphores prevents race conditions when multiple processors touch shared data, whether that data belongs to the kernel itself or to user-level cooperating processes. Multiprocessor-aware scheduling, incorporating load balancing, processor affinity, and, where appropriate, gang scheduling, determines how effectively the available processors are actually used. A wide range of Inter-Process Communication (IPC) mechanisms, including shared memory, sockets, and Remote Procedure Calls (RPC), enables processes to communicate and cooperate according to the requirements of an application. Together with the memory organisations studied in Unit 3, these ideas form the operating-system foundation on which high-performance parallel applications are built.

SGOU

Recap

- A multiprocessor operating system can be organised as master-slave, as separate per-processor kernels, or as symmetric multiprocessing (SMP). SMP, used by most modern OSes, allows any processor to run any task or kernel code.
- A critical section is a code segment that accesses shared data and must not be executed by more than one process at a time; unsynchronised concurrent access causes race conditions.
- A mutex is a simple binary lock with ownership, only the locking thread may unlock it.
- A semaphore is a more general integer synchronization primitive (wait()/ signal()) that supports both mutual exclusion (binary semaphore) and coordinating access to multiple resource instances (counting semaphore).
- The producer–consumer problem is a classic synchronization scenario solved using empty and full counting semaphores plus a mutex protecting the shared buffer.
- Multiprocessor task scheduling must decide not only which task runs next but on which processor, using shared or private ready queues, load balancing (push/pull migration), processor affinity (soft/hard), and, for cooperating thread groups, gang scheduling.
- Inter-process communication (IPC) allows processes to exchange data and coordinate; the two broad models are shared memory (fast, but synchronisation is the programmer's responsibility) and message passing (mediated by the OS, more portable, including pipes, message queues, sockets, RPC, and signals).
- A spinlock busy-waits instead of blocking, which can be efficient on a multiprocessor for short critical sections since the lock holder keeps running on another CPU.
- The readers–writers problem allows concurrent readers but requires exclusive access for writers; careless solutions can starve writers.
- Deadlock requires four conditions simultaneously; mutual exclusion, hold-and-wait, no preemption, and circular wait and is addressed through prevention, avoidance (e.g., Banker's algorithm), or detection and recovery.
- Deadlock and starvation both stop progress, but in a deadlock a cycle of processes waits for one another and none can proceed, whereas in starvation the system keeps working and only one process is repeatedly passed over; starvation is cured by fairness mechanisms such as aging.

SGOU

- Gang scheduling runs all the threads of a cooperating job in the same time slot, so that none wastes its slot waiting at a barrier for a partner that is not running.
- Local IPC (pipes, named pipes, POSIX message queues, shared memory) connects processes on one machine; distributed communication (sockets, RPC) connects processes across a network and must cope with delay and failure.

Objective Type Questions

1. What is the main characteristic of symmetric multiprocessing?
2. What is a critical section?
3. What is a race condition?
4. What are the atomic semaphore operations?
5. Which synchronization mechanism has ownership?
6. What is the initial value of the empty semaphore in a buffer of size N?
7. What is the tendency to execute a process on the same processor called?
8. What is the migration of processes from an overloaded processor called?
9. What is the simultaneous scheduling of related threads called?
10. Name one message passing the IPC mechanism.
11. What is a spinlock?
12. Name the four necessary conditions for deadlock.
13. Why do multiple readers access shared data concurrently?
14. What is the loss of an update due to concurrent execution called?
15. What happens when a bounded buffer becomes full?
16. Which lock is suitable for a long critical section?
17. Which IPC mechanism allows processes to share a memory region?
18. Which IPC mechanism connects the output of one program to another?

SGOU

19. What condition occurs when processes wait indefinitely for one another's resources?

20. What is the indefinite postponement of a process called?

21. Which scheduling approach uses separate ready queues for individual processors?

Answers

1. SMP
2. Critical section
3. Race condition
4. Wait and signal
5. Mutex
6. N
7. Processor affinity
8. Push migration
9. Gang scheduling
10. Sockets
11. Spinlock
12. Mutual exclusion, hold and wait, no preemption, circular wait
13. Read only
14. Lost update
15. Blocking
16. Blocking mutex
17. Shared memory
18. Pipe
19. Deadlock
20. Starvation
21. Per processor queues

SGOU

Assignments

1. Explain the three principal ways of organising a multiprocessor operating system, with the advantages and disadvantages of each.

2. What is process synchronization? Explain the concept of a critical section and the general requirements a solution must satisfy (mutual exclusion, progress, and bounded waiting).

3. Explain the working of semaphores with wait() and signal() operations, and distinguish between binary and counting semaphores.

4. Using semaphores, describe how the producer–consumer (bounded-buffer) problem is solved. Include the roles of the empty, full, and mutex semaphores.

5. Discuss the various approaches to task scheduling in multiprocessor systems, including shared versus private ready queues and load balancing strategies.

6. What is processor affinity and why does it matter more on NUMA systems? Relate your answer to the memory organisation concepts studied in Unit 3.

7. Compare shared memory IPC and message passing IPC, giving suitable examples of each and situations where one would be preferred over the other.

8. Describe any three message-passing IPC mechanisms (such as pipes, message queues, sockets, or RPC) with their typical use cases.

9. Explain the readers-writers problem and outline a synchronization solution using semaphores.

10. What is a deadlock? Explain its four necessary conditions and briefly describe how prevention, avoidance, and detection-and-recovery strategies each address it.

11. Write pseudocode for the bounded-buffer problem with N = 4 using the semaphores empty, full and mutex, and trace the values of empty and full for the first six operations of a producer and a consumer.

12. Four threads of a job must meet at a barrier every 10 ms on a 4-CPU machine shared with a second job. Draw a timeline like Figure 4.4.7 with and without gang scheduling and calculate the wasted processor time in each case.

13. A multiprocessor hosts a database server, a video-encoding tool and a web front end. Recommend an IPC mechanism for each interaction between them, and state whether each interaction is local or distributed.

SGOU

14. Distinguish deadlock from starvation using the readers-writers problem and a resource-ordering example, and propose one fix for each.

15. Analyze the trade-offs between spinlocks and blocking mutexes in an operating-system kernel on a 2-core system and on a 64-core system, including the effect of a lock holder being pre-empted

Reference

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating system concepts* (10th ed.). Wiley.

2. Stallings, W. (2018). *Operating systems: Internals and design principles* (9th ed.). Pearson.

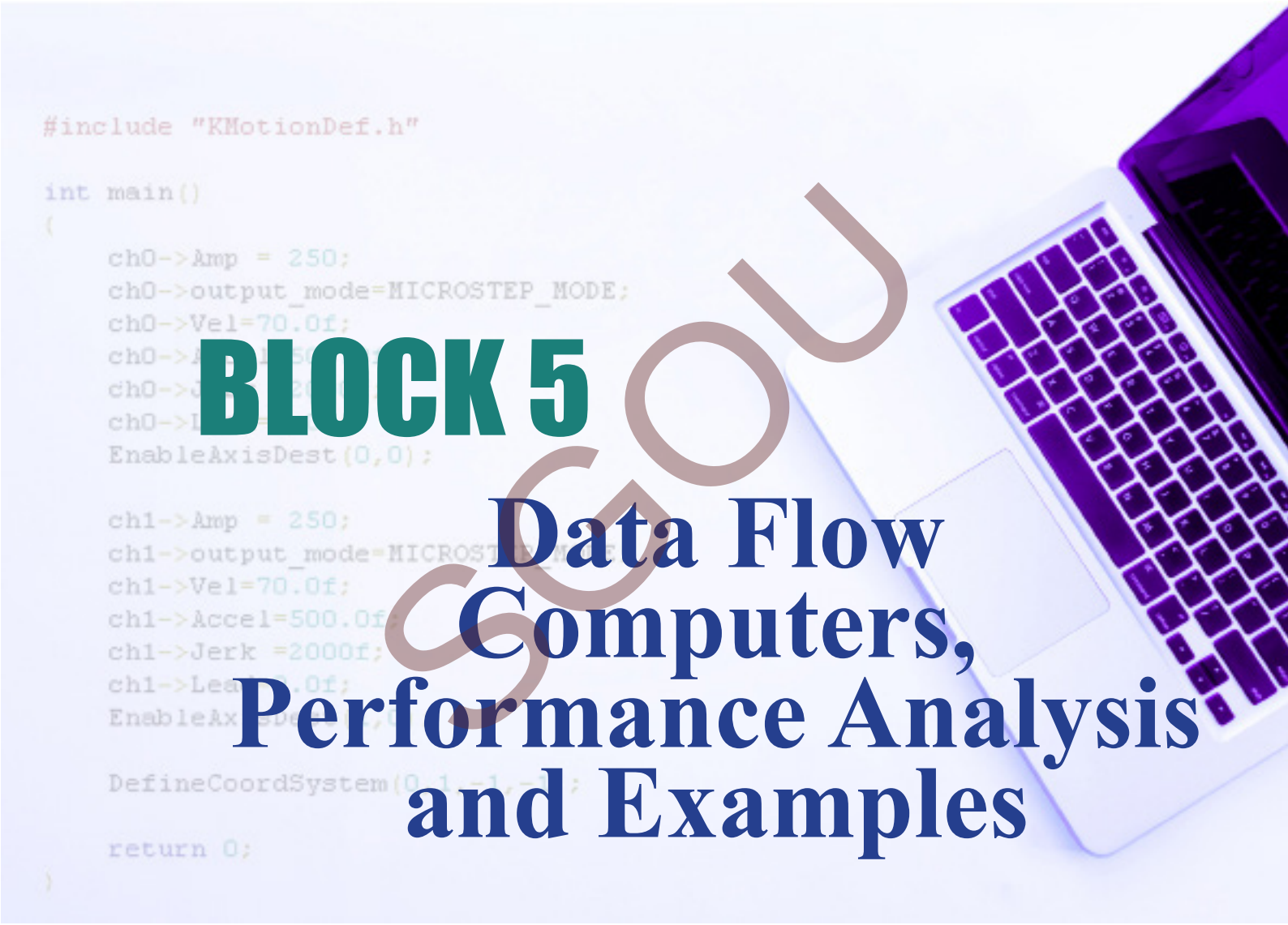
3. Tanenbaum, A. S., & Bos, H. (2015). *Modern operating systems* (4th ed.). Pearson.

Suggested Reading

1. Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to parallel computing* (2nd ed.). Pearson.

2. Love, R. (2010). *Linux kernel development* (3rd ed.). Addison-Wesley.

SGOU



BLOCK 5

Data Flow Computers, Performance Analysis and Examples

SGOU



Data-Driven Computing and Languages

Learning Outcomes

After completion of this unit, learner will be able to:

- explain the concept of data-driven (data-flow) computing and contrast it with the conventional control-flow model of execution
- distinguish declarative and functional programming languages from imperative languages, with suitable examples
- describe data-centric programming paradigms and explain how they support parallel and distributed data processing
- identify commonly used tools and frameworks that implement data-driven or data-centric computing, and
- represent a simple arithmetic computation as a data-flow graph and trace its execution

Prerequisite

This unit assumes familiarity with the conventional von Neumann model of computing, in which a program counter steps through a sequence of instructions stored in memory, as covered in earlier units on computer organisation. Basic exposure to any programming language — particularly the distinction between writing a sequence of explicit steps versus expressing a result as an equation or query — will help in appreciating the contrast between imperative and declarative styles. No prior knowledge of functional programming or any specific big-data framework is assumed; the necessary ideas are introduced from first principles. A familiar everyday illustration runs through this contrast: asking a taxi driver to "turn left, then right, then stop at the second building" is an imperative instruction (a sequence of explicit steps), whereas simply naming the destination address is a declarative one (specifying what is wanted, and leaving the driver free to choose how to get there) -- and this is exactly the shift in thinking that data-driven computing asks of a programmer.

SGOU

Keywords

Data-Driven Computing, Data-Flow Graph, Declarative Language, Functional Language, Data-Centric Paradigm, Immutability, Lazy Evaluation, MapReduce

Discussion

5.1.1 The Concept of Data-Driven Computing

Every computer executes instructions, but the principle that decides which instruction runs next can differ fundamentally from machine to machine. In the conventional control-flow model, used by virtually every general-purpose processor since the earliest von Neumann machines, a program counter (PC) determines execution order: instructions are fetched and executed strictly in the sequence laid out by the program (subject to explicit jumps and branches), regardless of whether the data an instruction needs is actually ready. Data-driven computing — more commonly called data-flow computing — inverts this principle. An instruction in a data-flow system has no fixed 'turn'; instead, it becomes eligible to execute (it is said to 'fire') as soon as all of its required input operands (tokens) have arrived, irrespective of any notion of program order or a program counter. Figure 5.1.1 contrasts the two execution models directly.

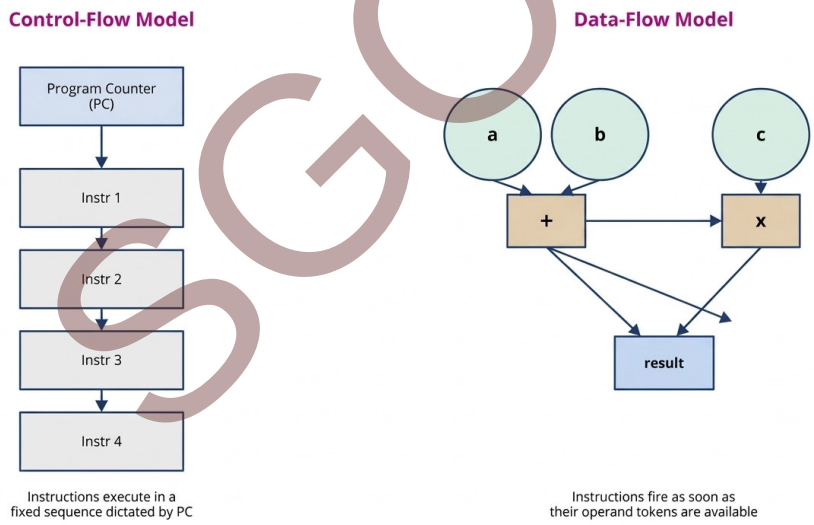


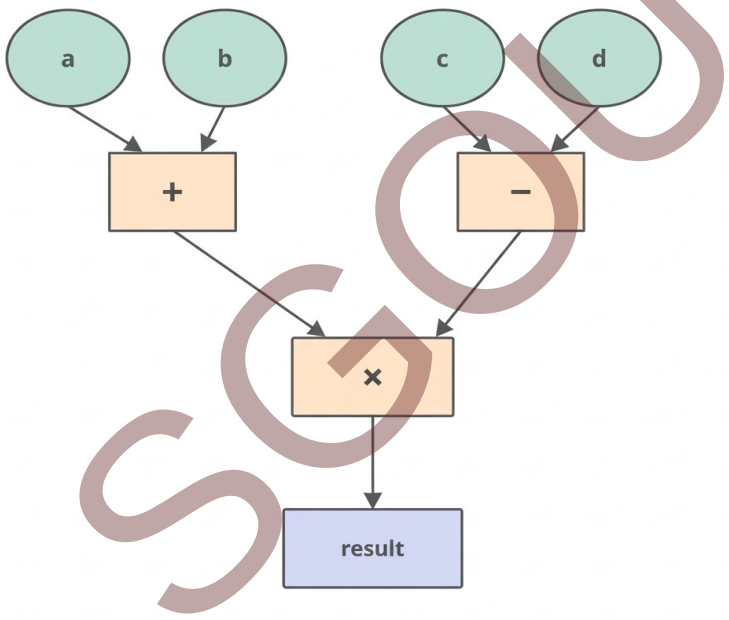
Fig. 5.1.1 Control-flow execution follows a program counter through a fixed sequence, while data-flow execution fires each instruction as soon as its operand tokens are available

This shift has a profound consequence: in a data-flow model, any two instructions whose data dependencies are already satisfied may execute simultaneously, with no artificial ordering constraint between them. Parallelism is therefore exposed naturally and automatically by the structure of the computation itself, rather than being something a programmer or compiler must explicitly engineer on top of an inherently sequential program. This is precisely why data-flow computing became an influential idea in the design of early high-performance and parallel computers, and why data-flow thinking still underlies many of today's large-scale parallel data-processing frameworks.

SGOU

The term 'data-driven' is used at two related but distinct levels in computing: at the hardware/architecture level, it refers to data-flow computer designs (studied in Unit 2 of this block), where the presence of data literally triggers instruction execution; at the software/paradigm level (the focus of this unit), it refers to programming and processing models that describe computation in terms of data transformations rather than explicit control steps.

A data-flow computation is naturally represented as a directed graph, called a data-flow graph, in which nodes represent operators (or actors) and directed arcs represent the flow of data values (tokens) from the output of one operator to the input of another. Consider the simple arithmetic expression (a + b) × (c − d). This has no inherent sequential order beyond what the data dependencies themselves require: the addition and the subtraction are entirely independent of one another and can be computed in either order, or simultaneously, while the multiplication must wait for both to finish. Figure 5.1.3 shows the resulting data-flow graph.



Nodes = operators; arcs = data tokens flowing between them

Fig. 5.1.2 A data-flow graph for the expression (a + b) × (c − d): the '+' and '−' nodes may fire in parallel since neither depends on the other's result.

Suppose a, b, c, and d are all available at time t = 0. In a data-flow machine with two available processing elements, the '+' node and the '−' node can both fire at t = 0, each producing their result by, say, t = 1. The '×' node then fires at t = 1, since both of its input tokens are now available, producing the final result at t = 2. A conventional sequential processor executing the same four operations one at a time would require four full instruction slots; the data-flow graph completes the same computation in three time steps precisely because it exploits the independence between '+' and '−'.

5.1.2 Declarative and Functional Languages

Programming languages are broadly grouped along a spectrum from imperative to declarative. An imperative language (such as C, Java, or Python written in its ordinary style) requires the programmer to specify how a result is to be computed — an explicit, step-by-step sequence of statements that update program state. A declarative language instead lets the programmer specify what result is wanted, leaving the underlying system free to decide how to compute it. SQL is a familiar example: a query such as SELECT name FROM students WHERE marks > 80 states the desired result without specifying any particular algorithm for scanning the table or building an index — the database engine decides the actual execution strategy. Figure 5.1.3 places these styles on a single spectrum.

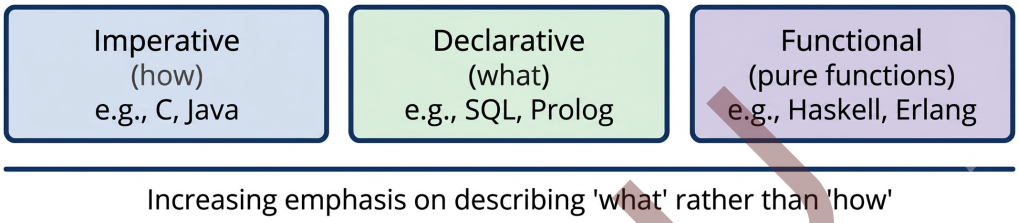


Fig. 5.1.3 A spectrum of programming paradigms, from imperative (control-oriented) through declarative to purely functional (data/value-oriented) styles.

Table 5.1.1 makes the imperative/declarative distinction concrete by contrasting how each style expresses the same simple task — selecting all values greater than a threshold from a collection.

Table 5.1.1 Imperative versus declarative programming styles, illustrated with a filtering task

Aspect	Imperative Style	Declarative Style
Focus	How to compute the result (steps)	What result is wanted
State	Explicit, mutable variables updated over time	Typically immutable; new values rather than updates
Example task	for each x in list: if x > 10: append x to result	SELECT x FROM list WHERE x > 10
Execution order	Fixed by the sequence of statements	Decided by the underlying engine/runtime
Parallelism	Must be explicitly engineered by the programmer	Can often be inferred automatically by the runtime

5.1.2.1 Functional Languages

Functional programming is an especially important declarative style because of its close conceptual relationship with data-flow computing. In a purely functional language (such

SGOU

as Haskell or, historically, pure Lisp), programs are built entirely out of mathematical functions: a function's output depends only on its inputs, and evaluating it produces no side effects — it does not modify any external, shared state. Two central properties distinguish functional programs from imperative ones:

- Immutability: once a value is created, it is never modified in place; a 'change' is expressed by producing a new value rather than mutating the old one. This eliminates an entire class of bugs caused by unexpected shared-state modification, and — importantly for this unit — it means that two independent computations can never interfere with each other through shared, mutable memory.
- Referential transparency: an expression can always be replaced by its computed value without changing the meaning of the program, because a function call always produces the same output for the same input and has no hidden side effects.

These two properties make functional programs a very natural match for the data-flow execution model: because there is no mutable shared state and no enforced statement ordering beyond genuine data dependency, a functional program's sub-expressions can, in principle, be evaluated in any order or in parallel — exactly the freedom that a data-flow graph exploits. Many data-flow architectures were, historically, designed with functional or applicative languages as their intended programming model for precisely this reason.

> In an imperative language, computing the sum of squares of a list might be written as a loop that repeatedly updates an accumulator variable (a side effect on shared state). In a functional language, the same computation is expressed as sum(map(square, list)) — a composition of pure functions with no mutable accumulator. Because map applies square to each list element independently, with no dependency between elements, a functional runtime (or a data-flow machine) can safely evaluate every application of square in parallel before combining the results with sum.

SGOU

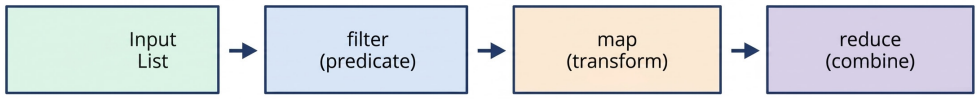
5.1.2.2 Lazy Evaluation

Many functional and declarative languages additionally support lazy evaluation, in which an expression's value is not computed until it is actually needed. This allows programs to work naturally with conceptually infinite data structures (such as an infinite stream of numbers) and closely mirrors the data-flow principle that a node fires only when its result is actually demanded and its inputs are actually available, rather than being scheduled eagerly by program order.

5.1.2.3 Higher-Order Functions and Recursion

Functional languages rely heavily on higher-order functions — functions that accept other functions as arguments or return them as results. The three most common higher-order functions in data-centric processing are map (apply a function to every element

of a collection, producing a new collection of the same length), filter (retain only the elements satisfying a predicate), and reduce or fold (combine all elements of a collection into a single summary value using a binary operator). Because each of these operates without any hidden shared mutable state, a pipeline built by chaining them together is itself a small data-flow graph: each stage is an independent, side-effect-free transformation whose output feeds the next stage's input. Figure 5.1.4 depicts a typical filter → map → reduce pipeline built this way.



Each stage is a pure function; the pipeline itself is a small data-flow graph

Fig. 5.1.4 A filter → map → reduce pipeline built from higher-order functions; each stage is a pure function, and the pipeline itself forms a small data-flow graph

A closely related point is that functional languages typically express repetition through recursion rather than through explicit loops with a mutable loop counter and accumulator, since a loop's mutable state would violate immutability. A recursive function such as factorial(n) = if n = 0 then 1 else n × factorial(n − 1) has no updateable variables at all; each recursive call simply produces a new value from its argument. This recursive style of expressing repetition turns out to map very naturally onto the dynamic data-flow architectures discussed in Unit 2, where each recursive invocation is handled as a distinctly tagged instance of the same underlying instructions.

filter(is_even, [1,2,3,4,5,6]) produces [2,4,6]; map(square, [2,4,6]) then produces [4,16,36]; and reduce(add, [4,16,36]) finally produces 56 — the sum of squares of the even numbers in the original list. Each of the three stages is a pure, independent transformation, and a data-driven runtime is free to begin processing later elements of the list through filter and map before earlier elements have even finished, since there is no shared state that could be corrupted by doing so.

5.1.2.4 Worked Example: From Imperative to Functional Style

To make the contrast between paradigms concrete, consider the task of computing the sum of the squares of the even numbers in a list. An imperative solution typically uses a mutable accumulator variable and an explicit loop:

Imperative (pseudocode): total = 0; for each x in list: { if (x is even) { total = total + x * x } }; return total. Here, total is repeatedly mutated, and the loop enforces a strict, sequential order of visiting each element — even though, mathematically, there is no reason the squaring of one even number must happen before another.

SGOU

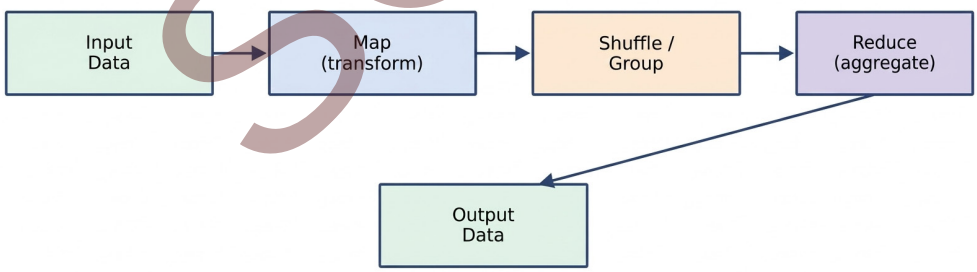
The equivalent functional solution, using the higher-order functions introduced above, has no mutable variable at all:

> Functional (pseudocode): reduce(add, map(square, filter(is_even, list))) — read from the inside out: first keep only the even numbers, then square each of the survivors, then sum the results. Every stage is a pure function of its input; there is no total variable being updated, and a data-driven runtime is free to evaluate filter and map on different elements of the list in any order, or fully in parallel, since no stage depends on the order in which sibling elements are processed.

This side-by-side comparison illustrates precisely why functional, declarative code is such a natural fit for data-driven execution: the imperative version's mutable total variable creates an artificial, purely notational dependency between loop iterations that has nothing to do with the actual mathematics of the problem, whereas the functional version exposes the problem's true, much more limited, data dependencies (each squared value depends only on its own input; only the final reduction genuinely needs all the intermediate results together).

5.1.3 Data-Centric Programming Paradigms

As datasets grew far too large to fit on a single machine, the ideas of data-flow computing and functional programming were combined and scaled up into what is now called data-centric (or data-parallel) programming — a paradigm in which a computation is expressed as a pipeline of transformations applied to large collections of data, and the underlying framework, not the programmer, is responsible for distributing that pipeline across many machines, scheduling each transformation as soon as its input data is available, and recovering automatically from individual machine failures. Figure 5.1.5 illustrates a representative data-centric pipeline.



Data flows through a pipeline of declarative transformations; the framework schedules execution across many nodes

Fig. 5.1.5 A representative data-centric processing pipeline: data flows through declarative Map, Shuffle, and Reduce stages, mirroring the data-flow firing principle at massive scale

The best-known example of this paradigm is the MapReduce programming model, popularised by Google and later implemented as open-source software in Apache Hadoop. A MapReduce program is expressed using just two user-supplied, side-effect-free functions: a map function that transforms each input record independently into zero or more intermediate key-value pairs, and a reduce function that aggregates all

intermediate values sharing the same key into a final result. Because map is applied independently to every input record, exactly like the square example above, the framework can freely fire many parallel instances of map across a large cluster of machines — the essence of data-driven, data-parallel execution.

> To count word frequencies across a huge collection of documents using MapReduce, the map function reads one document and emits a (word, 1) pair for every word it contains; the reduce function then receives, for each distinct word, the full list of 1's emitted for it across all documents, and simply sums them. Every document's map step is entirely independent of every other document's, so thousands of map tasks can fire and run in parallel across a cluster — precisely the same independence that let the '+' and '−' nodes fire in parallel in the earlier data-flow graph example.

5.1.4 Tools and Frameworks

A number of widely used software frameworks implement data-driven or data-centric execution principles, differing mainly in how they represent the data-flow graph, whether they process data in batches or as continuous streams, and how aggressively they optimise the graph before execution. Table 5.1.2 summarises four representative frameworks discussed below.

- Apache Hadoop: an open-source implementation of the MapReduce model, together with the Hadoop Distributed File System (HDFS) for storing very large datasets across many machines; well suited to large batch-processing jobs.
- Apache Spark: builds an explicit directed-acyclic-graph (DAG) of data transformations (its Resilient Distributed Datasets, or RDDs) and schedules them lazily, only actually computing a stage once its output is demanded — a direct software realisation of lazy, data-driven firing at cluster scale. Spark is typically much faster than plain Hadoop MapReduce for iterative workloads because it can keep intermediate data in memory rather than writing every stage to disk.
- Apache Flink and Apache Beam: frameworks designed around continuous, unbounded data streams rather than fixed batches; computation is expressed as a data-flow graph of operators that fire continuously as new data tokens (records) arrive, closely mirroring the classical data-flow firing rule in a streaming setting.
- TensorFlow: represents a machine-learning computation as a data-flow graph whose nodes are mathematical operations (matrix multiplications, activations, and so on) and whose edges carry multidimensional data arrays (tensors); the framework schedules each operation to run as soon as its input tensors are ready, including across multiple GPUs.

SGOU

Table 5.1.2 Representative data-driven / data-centric frameworks

Framework	Primary Model	Typical Use Case
Apache Hadoop (MapReduce)	Batch, disk-based map/ reduce	Large-scale batch processing
Apache Spark	Batch/near-real-time DAG, in-memory	Iterative analytics, machine learning
Apache Flink / Beam	Continuous streaming data-flow	Real-time stream processing
TensorFlow	Computation graph over tensors	Machine learning / deep learning

Interesting fact

Although these frameworks were developed decades after the original data-flow computer research of the 1970s and 1980s (covered in Unit 2), they independently rediscovered the same core idea — that expressing a computation as a graph of data dependencies, rather than a fixed instruction sequence, is one of the most effective ways to expose large amounts of parallelism automatically.

5.1.5 Advantages and Challenges of Data-Driven Computing

Data-driven computing offers several compelling advantages that explain its enduring influence. Because instructions fire purely on the basis of data availability, parallelism is exposed automatically, without requiring the programmer to manage threads, locks, or explicit synchronisation, and — since there is no shared mutable state to corrupt — data-driven programs are typically free of the race conditions discussed in the operating systems portion of this course. Determinism is another benefit: because the firing rule depends only on data availability and not on timing, a data-flow program (in the absence of non-deterministic 'merge' constructs) produces the same result regardless of how its independent operations happen to be interleaved or scheduled across processors, which greatly simplifies reasoning about correctness and makes such programs comparatively easy to test and debug.

These benefits come with real challenges. Fine-grained data-driven execution generates a very large number of small tokens that must be created, routed, and matched, and the overhead of this token traffic can outweigh the benefit of the parallelism exposed unless the granularity of each firing unit is chosen carefully (an issue examined in more technical hardware detail in Unit 2). Managing large, long-lived data structures such as arrays is also awkward in a purely token-based model, since treating every array element as a separate token would generate enormous numbers of tokens for even a modest array; practical systems typically introduce special non-strict, structured data types (such as I-structures) precisely to avoid this explosion. Finally, not every computation

SGOU

is naturally expressible as an acyclic graph of independent data dependencies — some algorithms are inherently sequential — so data-driven techniques, however powerful, complement rather than entirely replace conventional control-flow programming.

Design tension

The smaller (finer-grained) each data-flow node is, the more parallelism can, in principle, be exposed — but the greater the relative overhead of creating, routing, and matching the tokens that flow between nodes. Real systems, whether historical data-flow hardware or modern frameworks such as Spark, must choose a granularity — a single arithmetic operation versus an entire task or stage — that balances exposed parallelism against token/scheduling overhead for their intended workloads.

Data-driven computing replaces the rigid, sequential firing order of the control-flow model with a simple and powerful rule: an operation runs as soon as its data is ready. This single idea connects several topics that might otherwise appear unrelated — the design of data-flow computer hardware (the subject of the next unit), the mathematical discipline of functional programming with its immutability and referential transparency, and the practical, industrial-scale frameworks such as Spark, Flink, and TensorFlow that process the world's largest datasets today. Understanding data-flow graphs and the declarative/functional mindset that naturally produces them is therefore foundational to understanding both historical data-flow architectures and modern high-performance data processing systems.

Recap

- Control-flow computing executes instructions in an order dictated by a program counter; data-driven (data-flow) computing instead fires an instruction as soon as all of its operand tokens are available, regardless of program order.
- A data-flow graph represents a computation as nodes (operators) connected by arcs carrying data tokens; independent sub-computations can fire in parallel.
- Imperative languages specify how to compute a result; declarative languages specify what result is wanted, leaving the how to the underlying system (e.g., SQL).
- Functional languages are a key declarative style built on immutability (values are never modified in place) and referential transparency (an expression can be replaced by its value without changing program meaning).
- Lazy evaluation delays computing a value until it is actually needed, mirroring the data-flow firing principle.

SGOU

- Data-centric paradigms such as MapReduce scale data-driven, functional-style processing (independent map operations, followed by aggregation in reduce) across large clusters of machines.
- Frameworks such as Apache Hadoop, Apache Spark, Apache Flink/Beam, and TensorFlow all implement data-driven or data-flow-graph execution models, differing chiefly in batch versus streaming semantics and in-memory versus disk-based processing.
- Higher-order functions (map, filter, reduce) let a data-processing pipeline be expressed as a chain of pure, independent transformations, forming a small data-flow graph.
- Functional languages express repetition through recursion rather than mutable loop counters, which maps naturally onto tagged, re-entrant data-flow execution.
- Data-driven computing offers automatic parallelism exposure and determinism but faces practical challenges from token-traffic overhead and the difficulty of handling large, long-lived data structures.

Objective Type Questions

1. What type of computing executes instructions based on data availability?
2. Which component determines instruction execution in the control-flow model?
3. What is the execution of a node in a data-flow graph called?
4. What do the nodes in a data-flow graph represent?
5. What do the arcs in a data-flow graph represent?
6. Which programming paradigm specifies how to perform a task?
7. Which programming paradigm specifies what result is required?
8. What is the property of preventing modification of existing values called?
9. What is the property of replacing an expression with its computed value called?
10. What is the evaluation strategy that delays computation until needed?
11. Which two functions are used in the MapReduce programming model?

SGOU

12. Name one batch-oriented data-flow framework.

13. Name one streaming data-flow framework.

14. Which three higher-order functions are commonly used in data-centric pipelines?

15. Which programming technique is commonly used instead of mutable loops in functional languages?

16. What type of parallelism is naturally supported by data-driven computing?

17. What introduces additional overhead and complexity in data-driven computing?

Answers

1. Data-driven computing
2. Program counter
3. Node firing
4. Operations
5. Data dependencies
6. Imperative programming
7. Declarative programming
8. Immutability
9. Referential transparency
10. Lazy evaluation
11. Map and Reduce
12. Apache Hadoop
13. Apache Spark Streaming
14. Map, Filter, Reduce
15. Recursion
16. Data parallelism
17. Token matching

SGOU

Assignments

1. Explain the concept of data-driven computing and contrast it with the conventional control-flow execution model, using a suitable diagram.

2. Draw the data-flow graph for the expression $(p + q) \times (r + s) - t$, and explain which nodes could fire in parallel and why.

3. Distinguish between imperative, declarative, and functional programming languages, giving an example language for each.

4. Explain immutability and referential transparency, and discuss why these properties make functional programs well suited to parallel execution.

5. What is lazy evaluation? Explain its relationship to the data-flow firing principle with a suitable example.

6. Describe the MapReduce programming model with a worked example (such as word counting), and explain how it exposes parallelism.

7. Compare Apache Hadoop and Apache Spark as data-centric processing frameworks.

8. Explain, with examples, how TensorFlow uses a data-flow graph to represent machine-learning computations.

9. Explain higher-order functions (map, filter, reduce) with a worked example, and show how their composition forms a data-flow graph.

10. Discuss the advantages and challenges of data-driven computing, including the trade-off between granularity and token-traffic overhead.

SGOU

Reference

1. Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*.

2. Zaharia, M., et al. (2016). Apache Spark: A unified engine for big data processing. *Communications of the ACM.*

3. Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys.*

Suggested Reading

1. Hwang, K., & Jotwani, N. (2016). *Advanced computer architecture: Parallelism, scalability, programmability* (3rd ed.). McGraw-Hill Education.

2. Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to parallel computing* (2nd ed.). Pearson.

SGOU



Data-Flow Computer Architectures

Learning Outcomes

After completion of this unit, learner will be able to:

- describe the general organisation of a data-flow computer and explain the token-matching and instruction-firing cycle
- explain the design and operation of a static data-flow computer, including its limitations
- explain the design and operation of a dynamic data-flow computer and how tagged tokens overcome the limitations of static designs
- compare static and dynamic data-flow architectures on the basis of re-entrancy, hardware complexity, and performance
- discuss the principal design alternatives explored in data-flow computer architecture and their trade-offs

Prerequisite

This unit builds directly on Unit 1's introduction to data-driven computing, the concept of a data-flow graph, and the firing rule that governs when a node executes; readers should be comfortable representing a small arithmetic expression as a data-flow graph before proceeding. A general understanding of computer organisation, particularly the instruction fetch, decode, and execute, and the role of an instruction store analogous to memory is also assumed, along with the notion of a loop or a recursive function call as encountered in any introductory programming course, since these constructs motivate much of the discussion of dynamic data-flow architecture in this unit. A spreadsheet is a familiar everyday example of data-flow execution already at work: when a formula cell such as "=B2+C2" is changed, the spreadsheet automatically recomputes every other cell that depends on it, in whatever order the dependencies require, without the user writing any explicit sequence of steps -- precisely the token-driven, dependency-triggered execution style that this unit's data-flow computers implement directly in hardware.

SGOU

Keywords

Static Data-Flow, Dynamic Data-Flow, Tagged Token, Matching Unit, Activity Template, Re-entrancy, Token Store, Loop Unrolling, I-Structure

Discussion

5.2.1 General Organisation of a Data-Flow Computer

A data-flow computer is a machine built specifically to execute the data-flow graphs introduced in Unit 1 directly in hardware, rather than compiling them down to a conventional sequential instruction stream. Regardless of the many specific designs proposed and built since the 1970s, almost every data-flow computer is organised around the same basic cycle of components. Fig. 5.2.1 shows this general organisation.

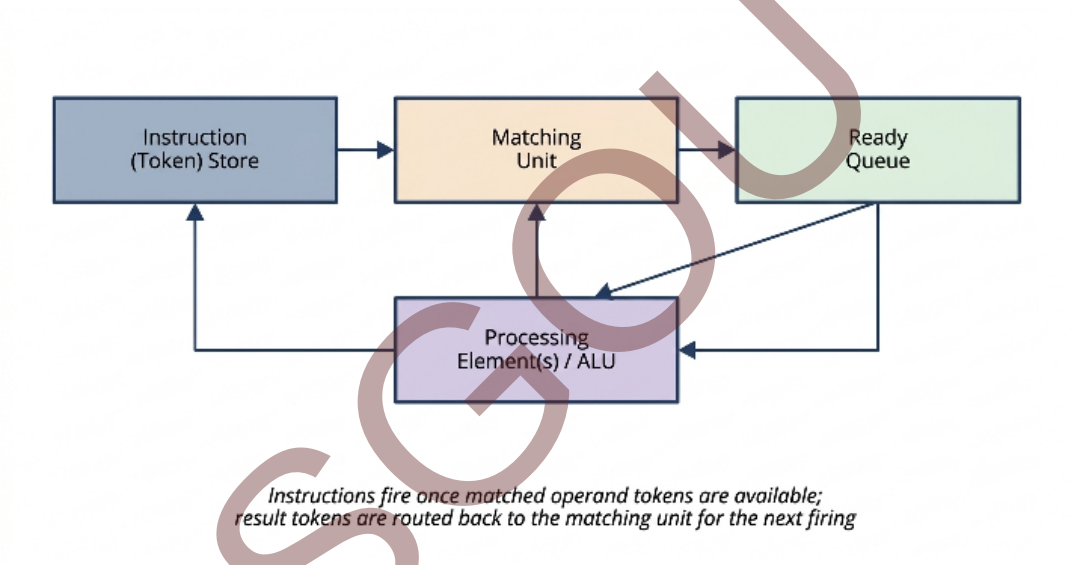


Fig. 5.2.1 General organisation of a data-flow computer

The Fig. 5.2.1 shows the general organisation of a data-flow computer: instructions (with their operands, once matched) flow from the instruction/token store through a matching unit into the processing elements, and result tokens are routed back.

- **Instruction (token) store:** holds the instructions of the data-flow graph, each instruction typically specifying an operation and the destination(s) to which its result token should be sent.
- **Matching unit:** receives incoming data tokens and checks whether all the operands required by some instruction have now arrived; when a complete set of matching operands is found, the corresponding instruction (together with its operands) is declared 'ready to fire' and passed onward.
- **Processing element(s):** one or more functional units (arithmetic logic units, ALUs — circuits that perform arithmetic and logical operations) that actually

SGOU

execute a ready instruction once it reaches them, consuming its operand tokens and producing a new result token.

- **Result / routing network:** carries newly produced result tokens back to the matching unit (or directly to waiting instructions), where they may in turn complete the operand set of one or more further instructions, continuing the cycle.

The central engineering challenge in any data-flow computer design is the matching problem: efficiently detecting, among potentially thousands of instructions each waiting for one or more operands, exactly which instruction has just had its full set of operands satisfied by an incoming token. How a machine solves this matching problem is the key distinction between the two major families of data-flow architecture discussed in this unit: static and dynamic data-flow computers.

Worked illustration (matching cycle)

Suppose instruction I3 (an addition) needs operands from I1 and I2. When I1 finishes, it sends a result token labelled 'destination: I3, left operand' into the matching unit; the matching unit records that I3's left slot is now full but its right slot is still empty, so I3 does not yet fire. When I2 later sends its token labelled 'destination: I3, right operand', the matching unit finds that I3's left slot was already filled, declares I3 ready, and dispatches both operands together to a free processing element, which executes the addition and sends I3's result onward to whichever instruction needs it next.

Fig. 5.2.2 shows the same matching cycle as three snapshots of the matching-unit entry for I3. Each instruction has one slot per operand (here a Left slot and a Right slot), and a result token names both the instruction and the slot it is addressed to.

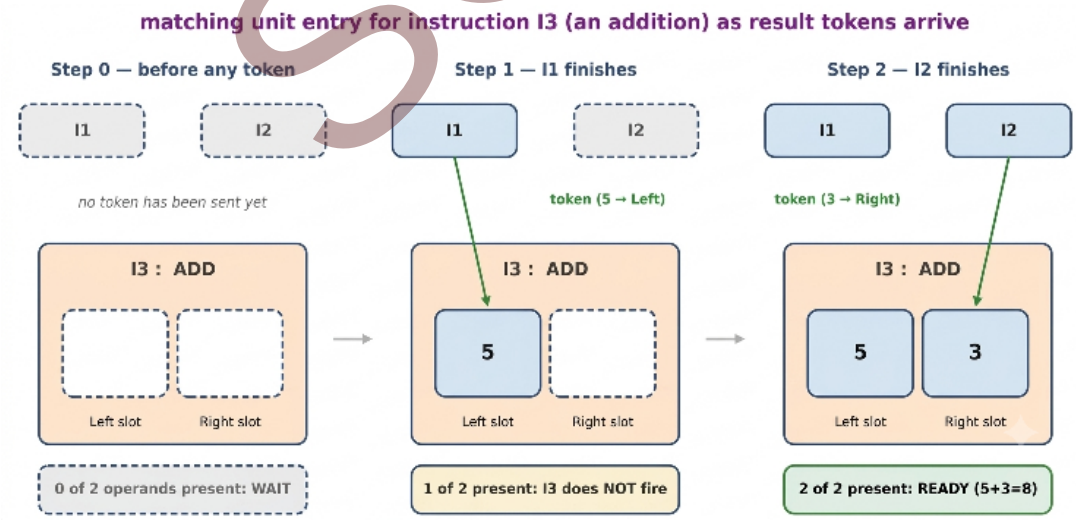


Fig. 5.2.2 Step-by-step matching for instruction I3 (an addition)

The Fig. 5.2.2 shows step-by-step matching for instruction I3 (an addition). I3 fires only when result tokens from I1 and I2 have filled both its Left and Right operand slots.

- **Step 0:** both operand slots of I3 are empty, so I3 cannot fire.
- **Step 1:** I1 finishes and sends the token [value 5 → I3, Left]. The matching unit stores 5 in I3's Left slot. Only one of the two slots is full, so I3 still waits.
- **Step 2:** I2 finishes and sends the token [value 3 → I3, Right]. Both slots are now full, so the matching unit declares I3 ready and dispatches it, with operands 5 and 3, to a free processing element, which computes 8 and sends it onward as a new token.

Try it yourself

Suppose I2's token (value 3) arrives before I1's token (value 5). What happens at each step, and does the result change? Answer: the Right slot fills first and I3 waits; when the Left token arrives the Left slot fills, I3 fires, and the result is still 8. The firing rule depends only on whether all operands are present, not on the order in which they arrive — which is why data-flow execution is deterministic.

5.2.1.1 Historical Data-Flow Machines

Data-flow computer architecture was an active research area from the mid-1970s through the early 1990s, producing several influential machines that established the static/dynamic distinction discussed in this unit and directly shaped later processor design. Figure 5.2.3 traces this history, and Table 5.2.1 lists several representative machines and their contributions.

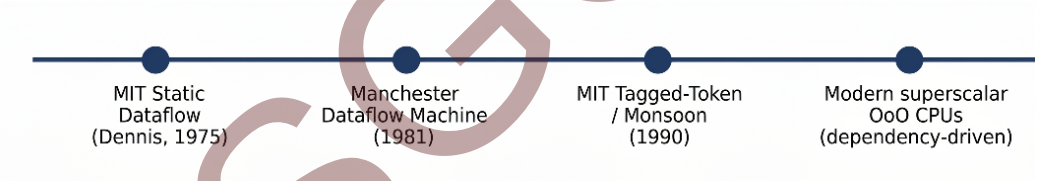


Fig. 5.2.3 A simplified timeline of influential data-flow machine designs, culminating in the dependency-driven scheduling used inside modern out-of-order processors

Table 5.2.1 Selected historical data-flow computer projects

Machine	Type	Key Contribution
MIT Static Dataflow (Dennis, 1975)	Static	First detailed static dataflow architecture proposal
Manchester Dataflow Machine (1981)	Dynamic	Working tagged-token prototype; demonstrated re-entrancy
MIT Tagged-Token / Monsoon (1990)	Dynamic	Explicit-token-store hybrid; influenced later designs
LAU System (France)	Static	Explored large-grain (block-level) dataflow actors

SGOU

5.2.2 Static Data-Flow Computer

A static data-flow computer, exemplified by the early data-flow machine designed at MIT by Jack Dennis and colleagues, takes the most direct approach to the matching problem: it simply forbids more than one token from occupying any single arc of the data-flow graph at any given time. Each instruction in the graph is represented by an activity template, a fixed hardware (or memory) slot that holds the operator, its operand slots, and the destination(s) for its result. Because each arc can carry at most one 'in-flight' token, checking whether an instruction is ready to fire reduces to a simple presence check: has a token arrived in each of this instruction's operand slots? Fig. 5.2.4 illustrates this fixed-slot arrangement.

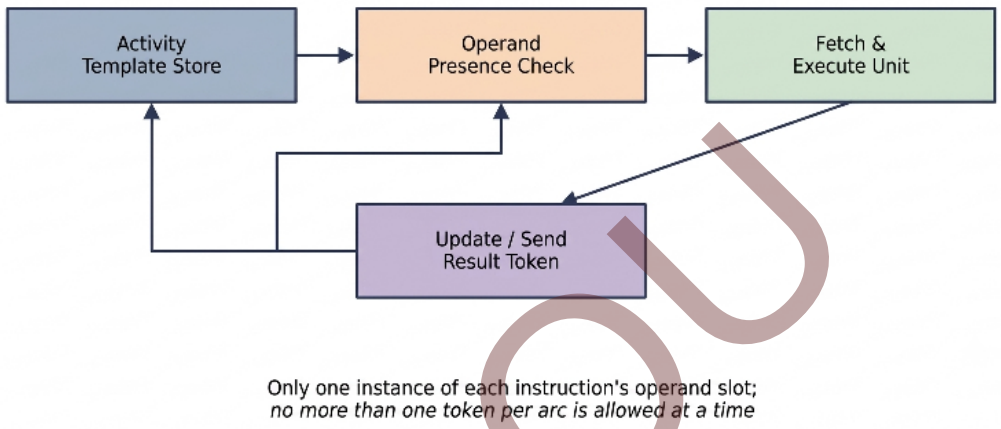


Fig. 5.2.4 Static data-flow architecture

Each activity template has a fixed operand slot per input arc, and only one token may occupy a given arc at a time. This single-token-per-arc restriction makes the hardware for matching comparatively simple and inexpensive: the presence of an operand is simply a bit that is set when a token arrives and cleared when it is consumed, with no need to compare tags or search a large associative store. Static data-flow computers were consequently easier to build with the technology of the 1970s and 1980s and served as an important proof of concept for the entire data-flow approach.

5.2.2.1 The Fundamental Limitation of Static Data-Flow

The single-token-per-arc rule, however, creates a serious limitation: it prevents re-entrancy, the ability to have several independent activations of the same code (for example, successive loop iterations or recursive calls) in progress at the same time. If the same instruction (the same node in the data-flow graph) needs to be invoked again before its previous invocation has fully completed and cleared its operand slots as happens naturally with loop iterations executing in a pipelined fashion, or with recursive procedure calls, where the 'same' instructions of the procedure body are logically invoked once per level of recursion, a static machine cannot allow the second invocation's tokens onto the same arcs until the first invocation has finished and vacated them. In effect, a static data-flow computer must serialise successive iterations of a loop or successive calls of a recursive function to the extent that they would otherwise share the same physical activity templates, which sharply limits the amount of parallelism it

SGOU

can extract from exactly the kind of repetitive, data-parallel computations (loops over large arrays, recursive divide-and-conquer algorithms) that data-flow machines were meant to accelerate.

Worked illustration

Consider a loop that computes y[i] = a[i] + b[i] for i = 0 .. 999. Every iteration uses the identical '+' node in the data-flow graph (the loop body is compiled once, not copied 1000 times). In a static machine, iteration i + 1's tokens cannot be placed on the '+' node's input arcs until iteration i's tokens have been consumed and the result token has departed — effectively forcing the 1000 iterations to execute in strict, pipelined lock-step rather than in a freely overlapped, fully parallel fashion.

5.2.2.2 Loop Unrolling: Recovering Parallelism at Compile Time

Because of the limitation just described, compilers for static data-flow machines relied on a technique called loop unrolling. Loop unrolling means rewriting a loop so that its body is copied several times, once for each iteration (or for a fixed group of iterations), instead of being executed repeatedly through a single copy. After unrolling, each copy of the body is a separate set of nodes in the data-flow graph, with its own activity templates. The example below unrolls a four-iteration loop:

```
Rolled loop:                    Unrolled by 4:
for i = 0 to 3:                 y0 = a0 + b0
    y[i] = a[i] + b[i]          y1 = a1 + b1
                                y2 = a2 + b2
                                y3 = a3 + b3
```

In the rolled loop there is a single "+" node, so on a static machine the four iterations must pass through it one after another. In the unrolled version there are four independent "+" nodes, which may all fire at the same time (Fig. 5.2.5).

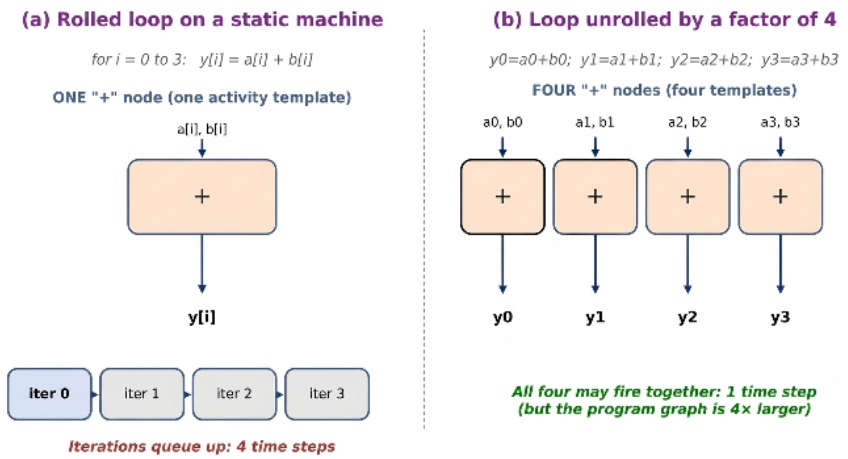


Fig. 5.2.5 (a) Rolled loop on a static machine & (b) Loop unrolled by a factor of 4

SGOU

In fig 5.2.5 (a) A rolled loop uses one "+" node, so iterations queue up on a static machine; (b) unrolling by 4 creates four independent "+" nodes that can fire together, at the cost of a graph four times larger.

Relevance and limits. Unrolling recovers parallelism at a price. The program graph grows in proportion to the number of copies (fully unrolling a 1000-iteration loop would need 1000 copies of the body), and loops whose iteration count is not known until run time cannot be fully unrolled at all. Compilers therefore often unroll partially, for example by a factor of 4 or 8. (Conventional compilers also unroll loops, mainly to reduce the cost of loop-control instructions; on static data-flow machines it was additionally a way to expose parallelism.) A dynamic machine, described next, does not need unrolling, because tags let the hardware keep many iterations in flight through the same node. This is why the need for loop unrolling appears as a comparison dimension in Section 5.2.4.

Worked illustration

The loop z[i] = x[i] × y[i] for i = 0 … 7, unrolled by a factor of 4, contains four "×" nodes and executes twice (for i = 0 … 3 and then i = 4 … 7). Unrolled fully, it needs eight "×" nodes, all of which can fire together, but the graph is twice as large.

5.2.3 Dynamic Data-Flow Computer

Dynamic data-flow computers (also called tagged-token data-flow machines) solve the re-entrancy problem directly by attaching an explicit tag to every token, in addition to its data value. A tag typically identifies the iteration number, recursion depth, or invocation context that the token belongs to. Two tokens destined for 'the same' instruction in the data-flow graph are now only matched together — allowed to jointly trigger firing — if their tags are identical; tokens from different iterations or different recursive calls, though logically bound for the same graph node, carry different tags and are kept safely apart in the matching store (the memory in which tokens wait for their partners). Fig. 5.2.6 illustrates this tagged-token matching scheme.

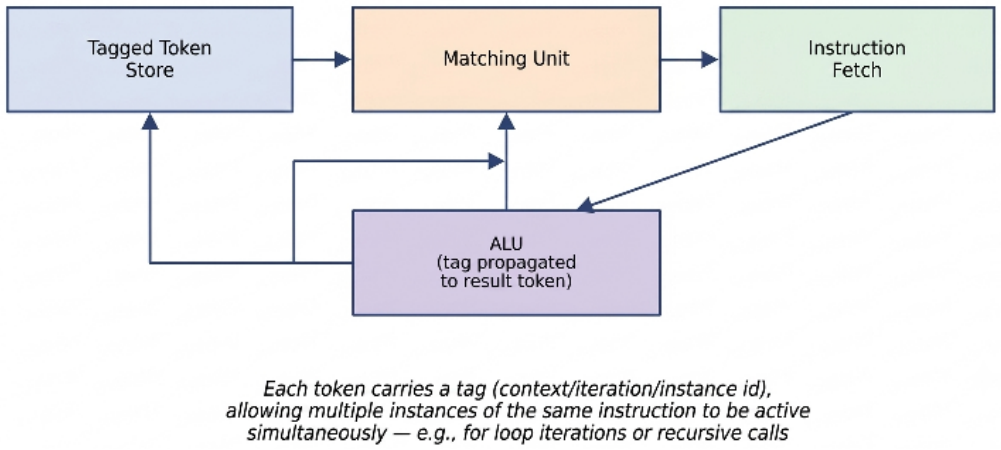


Fig. 5.2.6 Dynamic (tagged-token) data-flow architecture

SGOU

The Fig. 5.2.6 shows Dynamic (tagged-token) data-flow architecture: every token carries a tag identifying its context, so multiple invocations of the same instruction can be in flight simultaneously without interfering.

This tagging scheme means that, in principle, an unbounded number of instances of the same instruction can be 'in flight' at once — one for every distinct tag currently active — allowing successive loop iterations or nested recursive calls to proceed fully in parallel, limited only by the amount of real hardware (processing elements and matching-store capacity) available, not by any artificial single-token-per-arc restriction.

5.2.3.1 Worked Illustration: Recursion with Tags

Consider a recursive function f(n) that calls f(n − 1). When f(3) is invoked, its internal instructions are tagged with a context identifier, say <1>. When f(3) calls f(2), the nested invocation's tokens are tagged <1.1>; the subsequent call to f(1) is tagged <1.1.1>, and so on. Because every level of recursion carries a distinct tag, the matching unit never confuses a token belonging to f(3)'s instructions with a token belonging to f(2)'s or f(1)'s instructions — even though, textually, they are all invocations of the very same instructions in the compiled data-flow graph for f. Fig. 5.2.7 traces this tagging scheme across successive recursive calls.

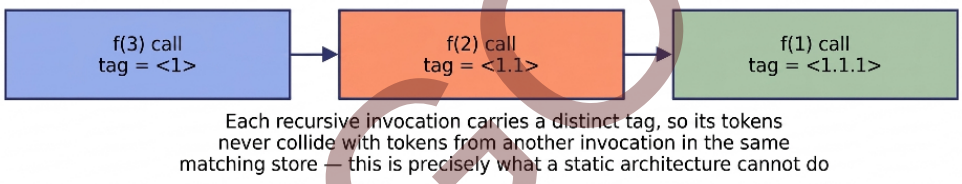


Fig. 5.2.7 Tagging successive recursive invocations of the same function: each call's tokens carry a distinct tag, so they never collide in the matching store

5.2.3.2 Worked Illustration: Loop Iterations with Tagged Tokens

Recursion is one use of tags; loops are the other. Consider the loop y[i] = a[i] + b[i] for i = 0, 1, 2 on a dynamic machine. The loop body contains a single "+" node, but the tokens of each iteration carry the iteration number as their tag, written <i>. A token is therefore described by three things: its tag, the operand slot it is addressed to (Left or Right), and its value. Suppose the tokens reach the "+" node in the following scrambled order, as can easily happen when other parts of the machine run at different speeds (Fig. 5.2.8):

SGOU

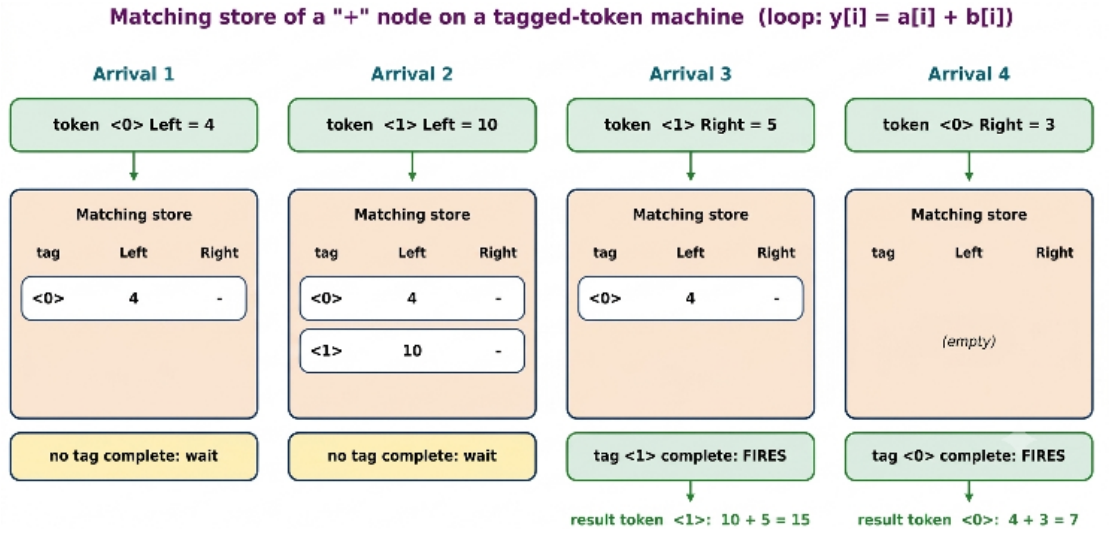


Fig. 5.2.8 Matching store of a "+" node on a tagged-token machine

Tokens of different iterations wait in separate entries, and an entry fires as soon as both of its operands (same tag) are present.

- **Arrival 1:** token <0>, Left = 4. The matching store has no entry for tag <0>, so it creates one and records Left = 4. Nothing fires.
- **Arrival 2:** token <1>, Left = 10. A separate entry is created for tag <1>. Nothing fires, because neither entry is complete.
- **Arrival 3:** token <1>, Right = 5. The entry for tag <1> already holds Left = 10, so it is complete: the node fires, 10 + 5 = 15, and the result token carries the same tag <1>. Notice that iteration 1 finishes before iteration 0.
- **Arrival 4:** token <0>, Right = 3. The entry for tag <0> is now complete: the node fires, 4 + 3 = 7, and emits a result token tagged <0>.

Two points stand out. First, a token pairs only with a token carrying the same tag, so values from different iterations (such as 4 and 5) can never be added by mistake. Second, results leave the node still tagged with their iteration numbers, so later instructions or an I-structure that stores y[0], y[1] and y[2] can place every result in the correct position even though the iterations completed out of order. A static machine, with its single token per arc, could not even have accepted the tokens of iteration 1 until iteration 0 had finished.

5.2.3.3 The Cost of Dynamic Data-Flow

This additional flexibility is not free. The matching unit of a dynamic data-flow machine must now search an associative store (conceptually, a large hash table or content-addressable memory, a memory that is searched by the content of a tag rather than by an address) for a token bearing a matching tag on the same arc, rather than simply checking a fixed presence bit, a substantially more complex and more expensive piece of hardware. Furthermore, the number of tokens that may need to be held in the matching store at any given time is, in principle, unbounded (since arbitrarily many

SGOU

iterations or recursive calls could be in flight simultaneously), which raises practical questions of finite storage that static machines, with their fixed one-token-per-arc limit, never have to confront in the same way.

5.2.4 Comparing Static and Dynamic Data-Flow Architectures

Fig. 5.2.9 contrasts the two architectures conceptually across three dimensions, each of which was introduced in the preceding sections: the need for loop unrolling (Section 5.2.2.2), meaning how much the compiler must duplicate loop bodies to expose parallelism; support for recursion (Sections 5.2.2.1 and 5.2.3.1), meaning whether several activations of the same function can be in flight together; and matching-hardware complexity (Section 5.2.3.3), meaning how costly the circuit that pairs operand tokens is. The bars show relative levels (Low, Medium, High) rather than measured values, and Table 5.2.2 summarises the comparison in tabular form.

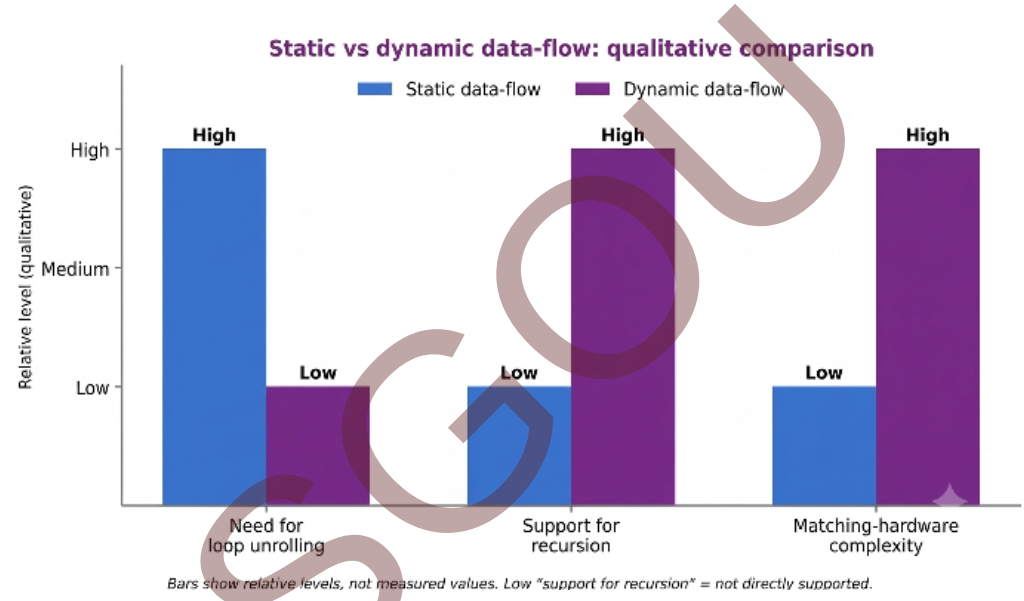


Fig. 5.2.9 Static vs Dynamic data-flow: qualitative comparison

The Fig. 5.2.9 shows a conceptual comparison of static and dynamic data-flow architectures across three dimensions: need for loop unrolling (Section 5.2.2.2), support for recursion, and matching-hardware complexity. Bars show relative levels (Low / Medium / High), not measured values.

Table 5.2.2 Static versus dynamic data-flow computer Architecture

Feature	Static Data-Flow	Dynamic Data-Flow
Tokens per arc	At most one at a time	Multiple, distinguished by tag
Matching mechanism	Simple presence check	Associative search by tag
Re-entrancy (loops, recursion)	Not directly supported	Directly supported

SGOU

Need for loop unrolling	High — compiler duplicates loop bodies to expose parallelism	Low — tags let one copy of the body serve every iteration
Hardware complexity	Lower	Higher
Matching store size	Fixed, bounded	Variable, potentially large
Representative example	MIT static data-flow machine (Dennis)	MIT Tagged-Token / Manchester Dataflow Machine

Interesting fact

The need for loop unrolling shown in Fig. 5.2.5 is a direct consequence of the single-token-per-arc rule. On a static machine the compiler had to trade program size for parallelism by unrolling loops (Section 5.2.2.2); on a dynamic machine the tags allow one copy of the loop body to serve every iteration, which is the more general solution that dynamic tagging later provided in hardware.

5.2.5 Data-Flow Design Alternatives

Beyond the fundamental static/dynamic split, data-flow computer research explored several further design alternatives, each addressing a different practical concern in building an efficient machine.

5.2.5.1 Explicit-Token-Store Architectures

A hybrid design, the explicit-token-store (ETS) architecture, pre-allocates a fixed frame of memory for each active procedure invocation (rather like a conventional activation record or stack frame — the block of memory that holds the local data of one function call), and operand slots within that frame are addressed directly by a combination of the frame's base address and an offset — avoiding the expensive associative search of a pure tagged-token design, while still supporting one live invocation per allocated frame, which in practice covers the recursion and loop-parallelism needs of most real programs without requiring unbounded associative matching hardware. This represents a middle ground between the simplicity of static data-flow and the full generality (and expense) of pure dynamic tagging.

Worked illustration (ETS addressing)

The first call of a function is given a frame starting at address 200, and the two operand slots of its addition instruction sit at offsets 0 (Left) and 1 (Right). A Left token is stored directly at address 200 + 0 = 200 and the matching Right token at 201; checking for a match is just checking whether both locations are full — no tag search is needed. A second, recursive call is given a different

SGOU

frame, say starting at 300, so its tokens go to 300 and 301 and cannot be confused with those of the first call. The frame address plays the role that the tag played in the pure tagged-token machine.

5.2.5.2 Granularity of Data-Flow Actors

Design proposals also differ in the granularity of the operations represented by a single data-flow node (actor). Fine-grained designs make each node a single machine-level operation (an addition, a comparison), maximising the parallelism exposed but incurring proportionally more matching and token-routing overhead per unit of useful work. Coarse-grained (or 'large-grain') data-flow designs instead group several operations, or even an entire subroutine, into a single actor that executes conventionally (in a sequential, control-flow manner) once its inputs arrive, reducing matching overhead at the cost of exposing somewhat less fine-grained parallelism. Many later, more practical data-flow-inspired systems (including some of the frameworks discussed in Unit 1, such as Spark's task-level DAG scheduling) adopt this coarser-grained philosophy, treating an entire function or task, rather than a single arithmetic operation, as the unit that fires once its inputs are ready.

5.2.5.3 Hybrid Data-Flow/Control-Flow Designs

A further alternative recognises that pure, fine-grained data-flow execution, while excellent at exposing parallelism, can be inefficient for the many parts of a real program that are inherently sequential (a short chain of dependent operations with no available parallelism to exploit). Hybrid architectures therefore combine data-flow scheduling at a coarse grain — deciding which large blocks or tasks are ready to run, based on data availability — with conventional control-flow (program-counter-driven) execution within each block once it starts running. This hybrid approach captures much of the parallelism-discovery benefit of data-flow scheduling while retaining the efficiency of ordinary sequential execution for code that has little exploitable data parallelism, and strongly influenced the design of later out-of-order superscalar processors (processors that start several instructions per clock cycle and may execute them in a different order from the program order), which internally use data-flow-like dependency tracking (via tools such as reservation stations — small buffers in which an instruction waits for its operands — and Tomasulo's algorithm, the classic hardware scheme for managing them) to decide when each instruction's operands are ready, even though the processor as a whole still presents a conventional, sequential, control-flow programming model to software.

Interesting fact

The influence of data-flow architecture research did not end when dedicated data-flow machines fell out of mainstream commercial use. Modern out-of-order superscalar CPUs — including virtually every high-performance processor sold today — internally track data dependencies between instructions and dispatch each instruction to execute as soon as its operands are ready, which is precisely

SGOU

the data-flow firing rule, applied within an instruction window inside an otherwise conventional control-flow processor.

Worked illustration (out-of-order execution as data flow)

Consider I1: r1 = r2 + r3; I2: r4 = r1 × r5; I3: r6 = r7 + r8. I2 needs r1, which I1 produces, but I3 depends on neither. An in-order processor executes I1, I2, I3 one after another (three cycles). An out-of-order processor starts I1 and I3 together in cycle 1, because their operands are ready — they "fire" — and starts I2 in cycle 2, as soon as I1's result arrives. The program takes two cycles instead of three (assuming one-cycle operations and two functional units).

5.2.5.4 I-Structures: Handling Large Data Structures

As noted in Unit 1, treating every element of a large array as a separate token is impractical. Data-flow architectures address this with I-structures, a special class of data structure that may be written into exactly once per element (enforcing the same single-assignment discipline as an ordinary data-flow token) but that can be read by many consumers, with any read that arrives before the corresponding write has occurred being automatically held (deferred) by the hardware until the write completes, rather than returning a wrong or undefined value. I-structures let programs work efficiently with genuinely large arrays allocating and filling elements of a matrix in parallel, for example, without violating the determinism guarantees that make data-flow programs easy to reason about, and without needing a separate token for every single element's transfer between every producer and consumer.

Worked illustration

Consider building an array A by having several parallel tasks each compute one element, and a later task that sums all of A's elements. Using an I-structure, the summing task can issue a read for A[5] as soon as it is ready to, even if the task computing A[5] has not finished yet; the read simply waits (is deferred) until the corresponding write occurs, after which the hardware automatically delivers the value — without the programmer having to explicitly synchronise the two tasks with a semaphore or a lock.

The life of a single I-structure element makes the idea clearer. Every element has a presence state — empty, full, or empty with readers waiting and Fig. 5.2.10 follows the element A[2] through its states.

SGOU

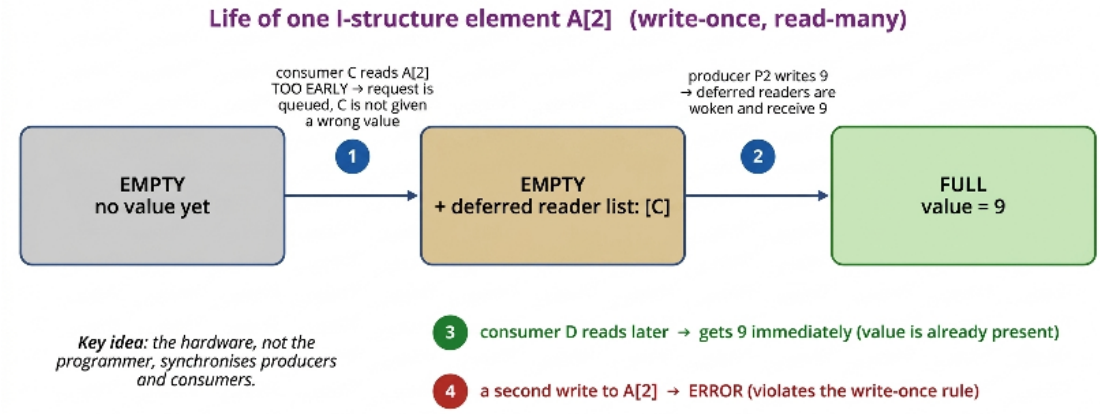


Fig. 5.2.10 State changes of one I-structure element. A read that arrives before the write is deferred and later satisfied; a second write is an error because elements are write-once

Worked illustration (a second example)

Task P2 computes A[2] = 3 × 3 = 9, while a consumer task C needs A[2] to evaluate B = A[2] + 1. (1) If C's read arrives first, A[2] is empty, so the read is queued at the element and C simply waits — it is never given a garbage value. (2) When P2 writes 9, the element becomes full and the queued read is completed: C receives 9 and computes B = 10. (3) A later consumer D that reads A[2] receives 9 at once. (4) Any attempt to write A[2] a second time is an error, because I-structure elements are write-once. In a control-flow program with an ordinary array, C would have to be made to wait explicitly (with a lock or semaphore) or risk reading an old value; the I-structure builds this waiting into the memory itself, so the result is the same whichever of P2 and C happens to run first.

5.2.6 Performance Considerations in Data-Flow Computers

The theoretical parallelism exposed by a data-flow graph is only as useful as the machine's ability to realise it in practice, and several practical factors limit how much of that theoretical parallelism a real data-flow computer can actually capture.

- Token traffic overhead: every data dependency in the graph corresponds to a token that must be created, routed through an interconnection network, and matched — for fine-grained graphs with very small actors (a single addition, say), this routing and matching overhead can rival or exceed the cost of the actual arithmetic being performed.
- Matching store capacity: a dynamic (tagged-token) machine's matching store, however large, is finite, while the number of simultaneously active tags (loop iterations, recursive calls) that a program might generate is not bounded in general; when the store fills up, the machine must throttle the creation of new tokens, limiting achievable parallelism.

SGOU

- Load balancing across processing elements: even when many instructions are simultaneously ready to fire, the machine must distribute them evenly across its available processing elements; an uneven distribution leaves some processing elements idle while others are backed up with ready work.

These factors together explain why, despite its elegant theoretical model, pure fine-grained data-flow hardware did not become the dominant commercial architecture: the overhead of managing enormous numbers of very small tokens frequently outweighed the parallelism gained, especially for programs with substantial genuinely sequential portions. This observation directly motivated the coarser-grained and hybrid design alternatives discussed above, and is a central theme taken up more quantitatively in the performance-analysis unit that follows in this block.

Preview of Amdahl's law. The last limit on parallel performance is a property of the program rather than of the hardware. Amdahl's law, named after the computer architect Gene Amdahl and derived in full in Section 5.3.3.1 of Unit 3, states that the speedup obtained by using many processors, the ratio of the running time on one processor to the running time on many is limited by the fraction of the program that cannot be parallelised. The example below previews the result informally; Unit 3 gives the exact formula.

Worked illustration (Amdahl's-law-style reasoning, see Section 5.3.3.1)

Suppose a program consists of a portion P that is perfectly parallelisable across N processing elements and a remaining portion (1 − P) that must execute sequentially, no matter how fine-grained the data-flow graph is. Even with an unlimited number of processing elements ($N \to \infty$), the best possible speedup is bounded by $1 / (1 - P)$. For example, if only 80% of a program's work is parallelisable ($P = 0.8$), the maximum possible speedup is $1 / 0.2 = 5\times$, regardless of how many processing elements the data-flow machine provides — a sobering reminder that exposing parallelism in the parallel portion of a program, however elegantly, cannot overcome the sequential portion's fundamental limit on overall speedup.

5.2.7 Bringing It Together

Data-flow computer architecture set out to build hardware that executes the data-flow graphs introduced in Unit 1 directly, exposing parallelism automatically by firing every instruction as soon as its operands are ready. Static data-flow machines demonstrated the feasibility of this idea with comparatively simple matching hardware, at the cost of forbidding re-entrant execution of loops and recursive calls. Dynamic (tagged-token) data-flow machines removed this restriction by tagging every token with its invocation context, at the cost of more complex, associative matching hardware and unbounded token-store requirements. Later design alternatives — explicit-token-store architectures, variable actor granularity, and hybrid data-flow/control-flow designs — sought practical middle grounds between these extremes, and their influence persists today in the dependency-driven instruction scheduling used inside every modern out-of-order processor and in the task-graph schedulers of large-scale data-processing frameworks such as Spark and TensorFlow.

SGOU

Recap

- A data-flow computer executes a data-flow graph directly in hardware, using an instruction/token store, a matching unit, processing elements, and a result-routing network.
- The matching problem — efficiently detecting which instruction has just received a complete set of operand tokens — is the central design challenge of any data-flow computer.
- A static data-flow computer allows only one token per arc at a time, using simple presence-bit matching in fixed activity templates; this makes matching hardware simple but prevents re-entrant execution of loops and recursive calls.
- A dynamic (tagged-token) data-flow computer attaches a context tag to every token, allowing multiple invocations of the same instruction to be in flight simultaneously, matched by tag rather than by mere presence.
- Dynamic data-flow directly supports loop pipelining and recursion but requires more expensive, associative matching hardware and a potentially unbounded token store.
- Matching is a slot-filling process: a token fills the Left or Right slot of its destination instruction (or, on a dynamic machine, the entry for its tag), and the instruction fires only when all slots are full, whatever the order of arrival.
- Loop unrolling copies a loop body so that iterations become separate nodes; static machines needed it to expose parallelism, whereas dynamic machines do not, because tags keep iterations apart.
- Explicit-token-store (ETS) architectures offer a middle ground, pre-allocating addressable frames per invocation to avoid full associative search while still supporting re-entrancy.
- Data-flow actor granularity can range from fine-grained (single operations) to coarse-grained (whole tasks or subroutines), trading exposed parallelism against matching/routing overhead.
- Hybrid data-flow/control-flow designs schedule coarse-grained tasks by data availability while executing each task's internal instructions conventionally; this idea persists in the dependency-driven scheduling of modern out-of-order superscalar processors.
- Historical machines such as the MIT static dataflow computer, the Manchester Dataflow Machine, and MIT's Tagged-Token/Monsoon architecture established and refined the static/dynamic distinction between the 1970s and early 1990s.

SGOU

- I-structures enforce single-assignment (write-once) semantics for array elements while automatically deferring reads that arrive before the corresponding write, allowing large data structures to be shared efficiently without sacrificing determinism.
- Practical data-flow performance is limited by token-traffic overhead, finite matching-store capacity, and load balancing across processing elements, which together motivated coarser-grained and hybrid designs.
- Amdahl's-law-style reasoning shows that even unlimited parallel hardware cannot exceed a speedup bound set by a program's inherently sequential portion.

Objective Type Questions

1. Which unit checks whether all the operands required by an instruction have arrived?
2. What is the basic unit of computation in a data-flow graph called?
3. What is the fixed-slot representation of an instruction in a static data-flow computer called?
4. How many tokens can occupy a single arc at a time in a static data-flow computer?
5. What is the ability to support multiple simultaneous activations of the same code called?
6. What additional information is attached to a token in a dynamic data-flow computer?
7. Which technique involves copying a loop body multiple times to expose parallelism?
8. Which architecture uses associative tag matching to distinguish different instruction invocations?
9. What is the abbreviated name of the explicit-token-store architecture?
10. Which type of data-flow actor represents an individual machine-level operation?
11. Which data structure supports write-once elements and automatically defers reads until the corresponding write occurs?
12. Which algorithm is a classic hardware scheme for managing instruction dependencies in out-of-order processors?

SGOU

Answers

1. Matching unit
2. Node
3. Activity template
4. One
5. Re-entrancy
6. Tag
7. Loop unrolling
8. Dynamic data-flow
9. ETS
10. Fine-grained
11. I-structure
12. Tomasulo's

Assignments

1. With a neat diagram, explain the general organisation of a data-flow computer, describing the role of each major component.
2. Explain the design and working of a static data-flow computer, and discuss why the single-token-per-arc restriction limits parallelism.
3. Explain the design and working of a dynamic (tagged-token) data-flow computer, illustrating with a worked example how tags allow re-entrant execution.
4. Compare static and dynamic data-flow architectures on at least five points, including hardware complexity and support for recursion.
5. Explain how loop unrolling can partially compensate for the limitations of static data-flow computers.
6. Describe the explicit-token-store (ETS) architecture and explain how it balances the trade-offs between static and dynamic data-flow designs.

SGOU

7. Discuss the distinction between fine-grained and coarse-grained data-flow actors, with examples of systems that adopt each approach.

8. Explain how hybrid data-flow/control-flow ideas have influenced the design of modern out-of-order superscalar processors.

9. Trace, step by step, how a matching unit determines that an instruction is ready to fire, using a worked example of an addition instruction awaiting two operands.

10. What are I-structures? Explain, with an example, how they allow large data structures to be shared between parallel tasks without violating determinism.

11. Discuss the practical factors that limit achievable parallelism in data-flow computers, and use Amdahl's-law-style reasoning to explain why a sequential program portion bounds overall speedup.

12. For the loop $z[i] = x[i] \times y[i] + 1$ with $i = 0 \ldots 3$, draw the data-flow graph of the loop unrolled by 4 for a static machine. Count the nodes and state the number of time steps needed if every node takes one step and all inputs are available.

13. A program contains a recursion of depth 10,000 and is to be run on a machine with a limited matching store. Compare a static, a pure tagged-token and an explicit-token-store design for this program and recommend one, justifying your choice.

14. Critically analyse the statement: "Dynamic data-flow removes the need for loop unrolling but not the need to choose a granularity."

Reference

1. Dennis, J. B., & Misunas, D. P. (1975). A preliminary architecture for a basic data-flow processor. *ACM SIGARCH Computer Architecture News.*

2. Arvind, & Nikhil, R. S. (1990). Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers.*

3. Gurd, J. R., Kirkham, C. C., & Watson, I. (1985). The Manchester prototype dataflow computer. *Communications of the ACM.*

SGOU

Suggested Reading

1. Hwang, K., & Jotwani, N. (2016). *Advanced computer architecture: Parallelism, scalability, programmability* (3rd ed.). McGraw-Hill Education.

2. Hennessy, J. L., & Patterson, D. A. (2017). *Computer architecture: A quantitative approach* (6th ed.). Morgan Kaufmann.

SGOU



Performance Analysis in Multiprocessor and Data-Flow Architectures

Learning Outcomes

After completion of this unit, learner will be able to:

- define and compute the principal performance metrics used to evaluate parallel and data-flow systems, including speedup, efficiency, and scalability
- explain the factors that limit achievable performance in multiprocessor and data-flow architectures
- apply Amdahl's Law and Gustafson's Law to analyse the theoretical limits of parallel speedup
- identify common performance bottlenecks in parallel systems and relate them to specific architectural causes, and
- describe standard benchmarks and profiling tools used to measure and compare high-performance computing systems

Prerequisite

This unit draws directly on the multiprocessor architecture concepts covered earlier in this course (shared and distributed memory, UMA/NUMA, cache coherence) and on the data-flow computing concepts introduced in Units 1 and 2 of this block. A basic comfort with elementary algebra and simple graph plotting is assumed, since several of the key results in this unit (Amdahl's Law, Gustafson's Law) are expressed as mathematical formulae and visualised as curves. No prior exposure to formal performance-analysis theory or benchmarking tools is required. A familiar everyday analogy motivates why this unit matters: adding nine more workers to a task that takes one worker ten hours does not make it finish in one hour, because some part of the job (planning, gathering materials, and other work that cannot be split among many people) must still be done sequentially -- this is exactly the intuition behind Amdahl's Law and the limits on parallel speedup studied in this unit.

SGOU

Keywords

Speedup, Efficiency, Scalability, Amdahl's Law, Gustafson's Law, Load Imbalance, Performance Bottleneck, Work-Depth Model, Roofline Model, Benchmark

Discussion

5.3.1 Performance Metrics

Before a parallel or data-flow system's performance can be improved, it must first be measured, and measured using metrics that mean the same thing across different machines and different problem sizes. Several standard metrics serve this purpose. Figure 5.3.1 gives an overview of the metrics discussed below.

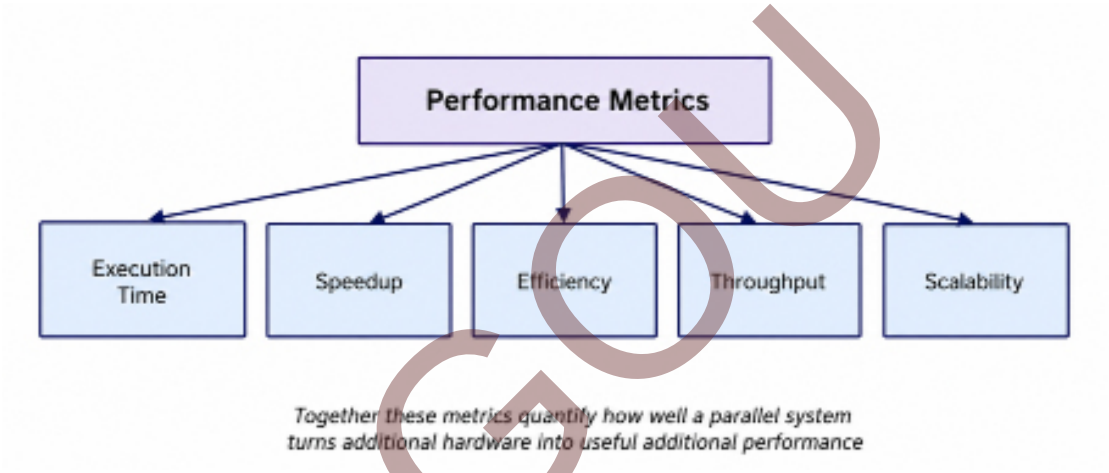


Fig. 5.3.1 The principal performance metrics used to evaluate a parallel or data-flow system

5.3.1.1 Execution Time and Speedup

The most basic metric is simply execution time (T): the wall-clock time (the elapsed real time, as read from a clock) a program takes to run to completion on a given machine. Given the execution time T1 of the best available sequential algorithm on a single processor, and the execution time TN of a parallel version of the same computation running on N processors, speedup is defined as S(N) = T1 / TN. A speedup of N (called linear or ideal speedup) would mean the parallel system used all N processors with perfect effectiveness; in practice, S(N) is almost always somewhat less than N because of the overheads discussed later in this unit.

5.3.1.2 Efficiency

Efficiency normalises speedup by the number of processors used: E(N) = S(N) / N. An efficiency of 1 (100%) corresponds to perfect linear speedup; efficiency below 1 quantifies how much of each processor's theoretical contribution is actually being realised as useful work, after accounting for overheads such as communication,

SGOU

synchronisation, and load imbalance. Efficiency is particularly useful for comparing two different parallel implementations of the same problem on different numbers of processors, since raw speedup numbers alone can be misleading when N differs.

Worked illustration

Suppose a sequential algorithm takes T1 = 100 seconds, and a parallel version running on 8 processors takes T8 = 16 seconds. Then S(8) = 100/16 = 6.25, and E(8) = 6.25/8 = 0.78 (78%). This tells us that, on average, each of the 8 processors is contributing about 78% of its theoretical share of useful work — a reasonably good, though not perfect, result.

5.3.1.3 Throughput and Scalability

Throughput measures the rate at which a system completes units of work (for example, transactions per second, or floating-point operations per second — FLOPS — a metric widely used to rate the raw arithmetic power of supercomputers). Scalability describes how well a system's performance continues to improve as more processors, memory, or nodes are added; a system is said to scale well if speedup continues to grow close to linearly as N increases, and to scale poorly if speedup quickly saturates or even declines beyond some point, typically because of the bottlenecks discussed in Section 5.3.5.

5.3.1.4 Strong Scaling and Weak Scaling

Scalability is usually assessed in one of two complementary ways, corresponding closely to the fixed-size and scaled-workload assumptions behind Amdahl's and Gustafson's laws respectively. Strong scaling measures how execution time decreases as more processors are applied to a fixed-size problem — the ideal is that doubling the processor count halves the execution time, exactly the scenario Amdahl's Law addresses. Weak scaling measures how execution time behaves as both the processor count and the problem size are increased together, proportionally, so that the amount of work per processor stays roughly constant — the ideal here is that execution time remains flat as the system scales up, exactly the scenario Gustafson's Law addresses. Figure 5.3.2 contrasts the two scaling styles directly.

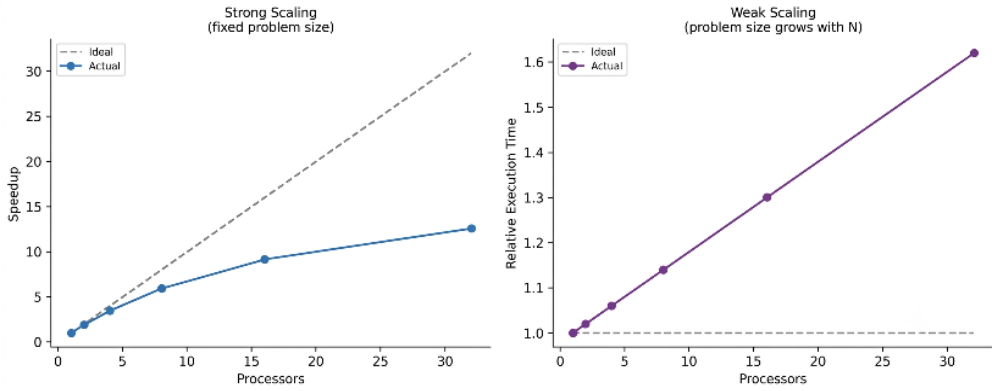


Fig. 5.3.2 Strong scaling (fixed problem size, speedup measured against processor count) versus weak scaling (problem size grows with processor count, execution time ideally stays flat)

SGOU

Worked illustration

A strong-scaling study might report 'this simulation completes in 100 minutes on 1 processor, 52 minutes on 2, and 28 minutes on 4' — the problem itself (say, a fixed 1000×1000 grid) never changes size. A weak-scaling study of the same underlying application might instead report 'this simulation of a 1000×1000 grid takes 100 minutes on 1 processor, and a simulation of a proportionally larger 2000×2000 grid also takes about 100 minutes on 4 processors' — the problem size grows with the processor count, and the goal is to keep execution time roughly constant rather than to shrink it.

Applying the metrics. In the strong-scaling study above, the speedups are 100/52 ≈ 1.92 on 2 processors and 100/28 ≈ 3.57 on 4, so the efficiencies are 1.92/2 ≈ 96% and 3.57/4 ≈ 89%. Efficiency is slipping as processors are added — a typical sign of growing communication or synchronisation overhead.

A program takes 240 s on one processor and 20 s on 16 processors. Find the speedup and the efficiency. Answer: S(16) = 240/20 = 12 and E(16) = 12/16 = 0.75 (75%).

5.3.2 Factors Affecting Performance

Several architectural and algorithmic factors act together to prevent a parallel or data-flow system from achieving ideal, linear speedup. Figure 5.3.2.1 summarises the four major contributing factors.

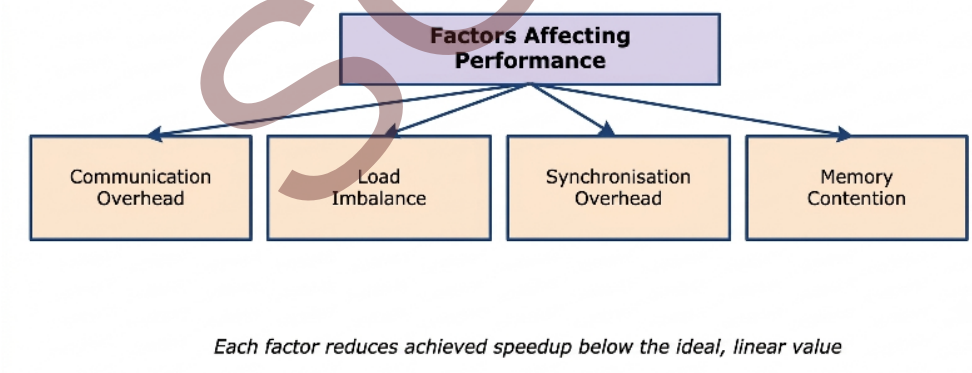


Fig. 5.3.3 Four major factors that reduce achieved speedup below the ideal, linear value in parallel and data-flow systems

- Communication overhead: time spent exchanging data between processors (message passing) or coordinating access to shared memory, rather than performing useful computation; this overhead grows with the number of processors and, in message-passing systems, with network latency (the delay before a transfer begins) and bandwidth (the amount of data the network can carry per second) limits.

SGOU

- Load imbalance: if work is not divided evenly among processors, some finish early and sit idle while others are still computing, wasting the idle processors' potential contribution to overall speedup.
- Synchronisation overhead: the mutexes (locks that admit one process at a time to a critical section), semaphores (counters that regulate access to shared resources), and barriers (points that every cooperating process must reach before any may continue) introduced in earlier units (needed to coordinate access to shared data and to align cooperating processes at particular points) themselves consume time and can cause processors to wait idly for one another.
- Memory contention: as discussed in the UMA/NUMA and cache-coherence material earlier in this course, many processors competing for a shared bus, shared memory bank, or shared cache-coherence traffic can throttle the effective memory bandwidth available to each processor.

Worked illustration (load imbalance)

A program of 100 equal tasks is shared among 4 processors as 40, 20, 20 and 20 tasks. The program finishes only when the busiest processor finishes, after 40 task-times, although perfect balance would take 25. The speedup is 100/40 = 2.5 on 4 processors, an efficiency of 62.5%: three processors sit idle for part of the run.

It is common for the communication-to-computation ratio of a parallel program to worsen as the problem is divided among more processors, because the amount of data exchanged at the boundaries between partitions often grows relative to the (shrinking) amount of computation each partition performs. This is one reason why simply adding more processors does not guarantee proportionally better performance.

5.3.3 Amdahl's Law and Gustafson's Law

Two influential mathematical models describe the theoretical limits of parallel speedup, corresponding to two different ways of thinking about how a problem's size relates to the number of processors available.

5.3.3.1 Amdahl's Law

Amdahl's Law (previewed informally in Section 5.2.6 of Unit 2), formulated by Gene Amdahl in 1967, assumes a fixed problem size and asks how much a fixed amount of work can be sped up by adding processors. If a fraction P of the total work is perfectly parallelisable and the remaining fraction (1 − P) is inherently sequential (must run on a single processor regardless of how many are available), then the speedup achievable with N processors is given by $S(N) = 1 / ((1 - P) + P/N)$. As N grows arbitrarily large, this

SGOU

expression approaches a hard ceiling of 1 / (1 − P), regardless of how many processors are added — the sequential fraction alone sets an upper bound on achievable speedup. Figure 5.3.3.1 plots this saturating speedup curve.

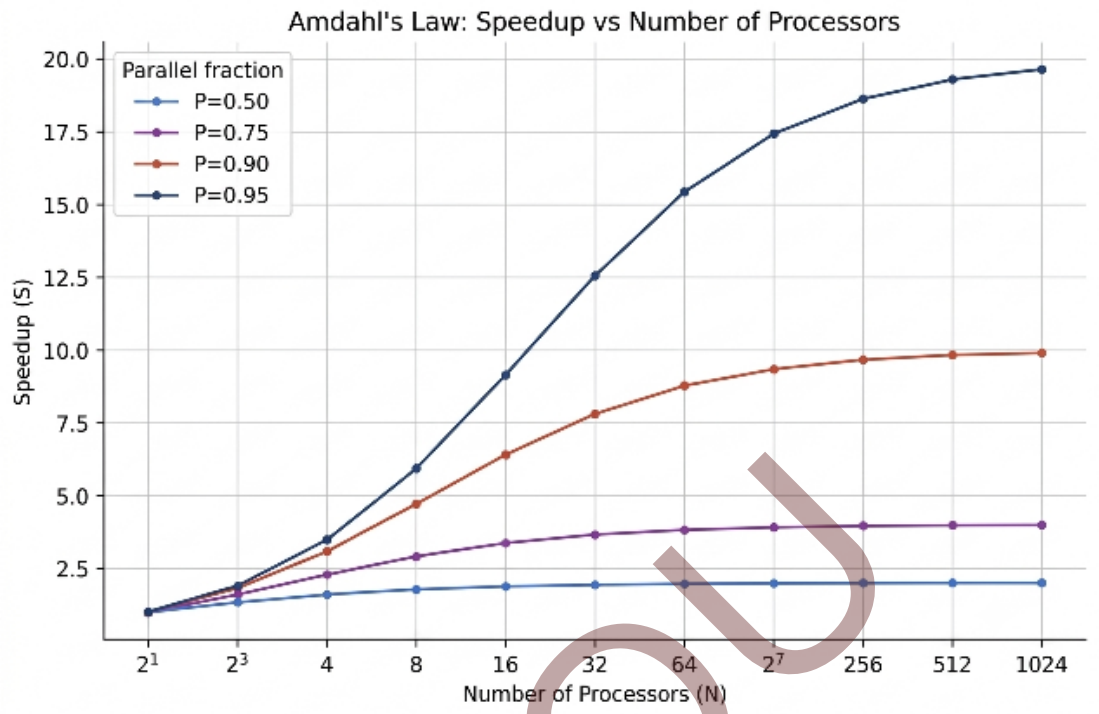


Fig. 5.3.4 Amdahl's Law: speedup saturates as the number of processors grows, with the ceiling determined entirely by the sequential (non-parallelisable) fraction of the workload

Worked illustration

If 90% of a program's execution time (P = 0.9) can be parallelised perfectly, then even with an unlimited number of processors, the maximum possible speedup is 1 / (1 − 0.9) = 1 / 0.1 = 10×. Adding a thousand processors cannot exceed this ceiling — the remaining 10% sequential portion alone caps overall speedup at 10×, no matter how much parallel hardware is thrown at the other 90%.

Where the formula comes from. Let T1 be the running time on one processor. The sequential part takes (1 − P) × T1 whatever the number of processors, while the parallel part, shared perfectly among N processors, takes P × T1 / N. Hence TN = T1 × [(1 − P) + P/N], and S(N) = T1 / TN = 1 / [(1 − P) + P/N].

SGOU

Table 5.3.1 applies the formula to the 90%-parallel program above.

Table 5.3.1 Amdahl's Law for P = 0.9: speedup and efficiency as the number of processors N grows

N (processors)	Speedup S(N)	Efficiency E(N) = S(N)/N
1	1.00	100%
2	1.82	91%
4	3.08	77%
8	4.71	59%
16	6.40	40%
64	8.77	14%
1024	9.91	1%

Reading the table. Speedup rises quickly at first, but every doubling of the processors buys less: going from 8 to 16 processors raises the speedup only from 4.71 to 6.40, and 1024 processors give barely 13% more speedup than 64. Meanwhile efficiency collapses from 91% to about 1%. The table shows why adding processors to a program with a sequential fraction quickly becomes poor value.

5.3.3.2 Gustafson's Law

Gustafson's Law, proposed by John Gustafson in 1988, challenges Amdahl's fixed-problem-size assumption by observing that, in practice, when more processors become available, users typically choose to solve a correspondingly larger problem in the same amount of time, rather than solving the same fixed-size problem faster. Under this scaled-workload assumption, if s is the (now fixed) fraction of execution time spent on inherently sequential work, the scaled speedup with N processors is $Sp(N) = N - s \times (N - 1)$, which grows roughly linearly with N rather than saturating at a fixed ceiling. Figure 5.3.5 illustrates this scaled-workload speedup.

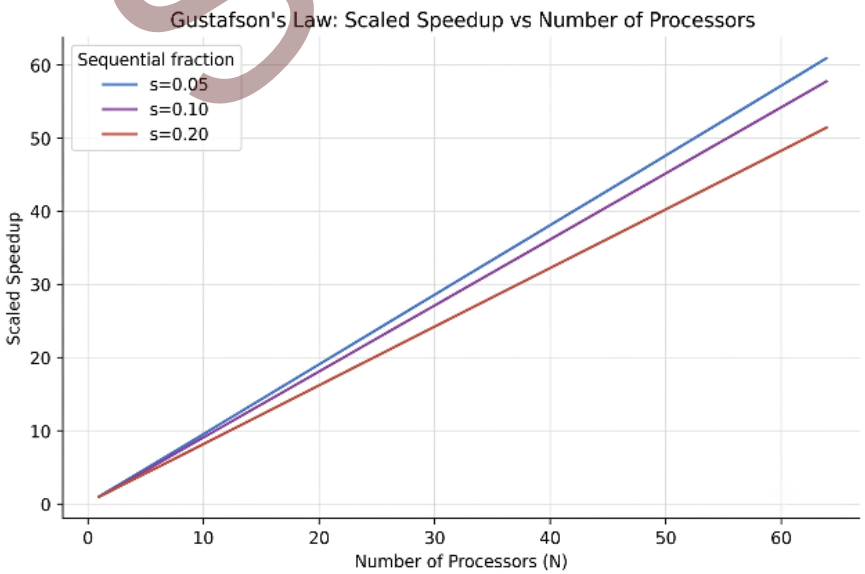


Fig. 5.3.5 Gustafson's Law: under the scaled-workload assumption, speedup continues to grow with the number of processors rather than saturating at a fixed limit

SGOU

Worked illustration

Take s = 0.1 (10% sequential). With N = 10 processors the scaled speedup is 10 − 0.1 × 9 = 9.1; with N = 100 it is 100 − 0.1 × 99 = 90.1. By contrast, Amdahl's Law says a fixed-size problem with the same 10% sequential fraction can never exceed 10×. There is no contradiction: when the problem is enlarged to use more processors, the parallel work grows while the sequential work stays about the same, so the sequential share of the total work shrinks.

Table 5.3.2 draws together the two laws for direct comparison.

Table 5.3.2 Amdahl's Law versus Gustafson's Law

	Amdahl's Law	**Gustafson's Law**
Problem size assumption	Fixed, regardless of N	Grows with N (fixed execution time)
Speedup behaviour	Saturates at a ceiling	Grows roughly linearly with N
Best suited to	Latency-sensitive, fixed-size tasks	Throughput-oriented, scalable workloads
Key insight	Sequential fraction caps speedup	Larger problems can better exploit more processors

Reconciling the two laws

Amdahl's and Gustafson's laws are not contradictory — they simply model two different real-world scenarios. Amdahl's Law is the right lens when a specific, fixed-size problem must be solved as quickly as possible (e.g., a single weather forecast that must complete before a deadline); Gustafson's Law is the right lens when more computational power is used to tackle a correspondingly larger or more detailed problem in roughly the same wall-clock time (e.g., running a climate simulation at finer spatial resolution as more processors become available).

5.3.4 Parallel Algorithm Analysis

Beyond speedup formulas, computer scientists analyse the intrinsic parallelism of an algorithm using the work-depth model, which characterises an algorithm independently of any specific machine. Work (W) is the total number of elementary operations the algorithm performs, summed across all processors — equivalent to the running time on a single processor. Depth (D), also called the critical path length, is the length of the longest chain of operations that must be executed one after another because of genuine data dependencies, and represents the minimum possible running time even with an

SGOU

unlimited number of processors. Figure 5.3.6 works through this calculation on a small example graph.

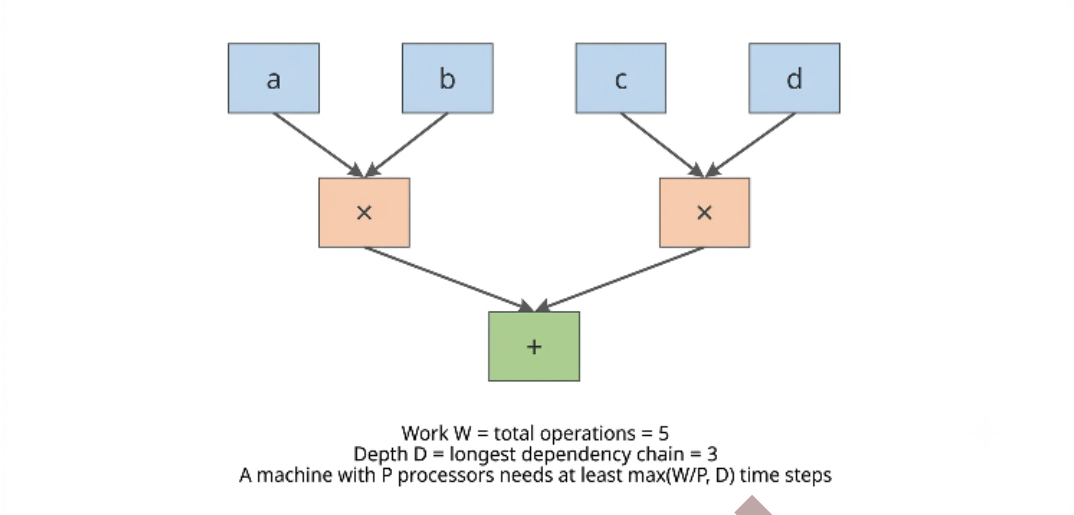


Fig. 5.3.6 The work-depth model applied to the data-flow graph for (ab)+(cd): work W = 3 (two multiplications and one addition), depth D = 2 (the longest chain: a multiplication followed by the addition)

For any algorithm, the best possible running time on P processors is bounded below by max(W/P, D): the machine can never go faster than dividing the total work evenly across all P processors (W/P), nor faster than the length of the longest dependency chain (D), whichever is larger. The ratio W/D is called the algorithm's parallelism, and it places an upper bound on how many processors can be usefully employed before adding more processors yields no further benefit, regardless of how cleverly the work is scheduled.

Worked illustration

The graph of (a + b) × (c − d) from Unit 1 has three operations ("+", "−" and "×"), so W = 3. The longest dependency chain is "+" (or "−") followed by "×", so D = 2. The parallelism is W/D = 3/2 = 1.5, so two processors are already enough: with P = 2 the lower bound is max(3/2, 2) = 2 time steps, and with P = 100 it is still max(3/100, 2) = 2. Processors beyond the parallelism cannot shorten the critical path.

The shape of an algorithm matters as much as its size. Figure 5.3.7 shows two ways of adding eight numbers x1 … x8.

SGOU

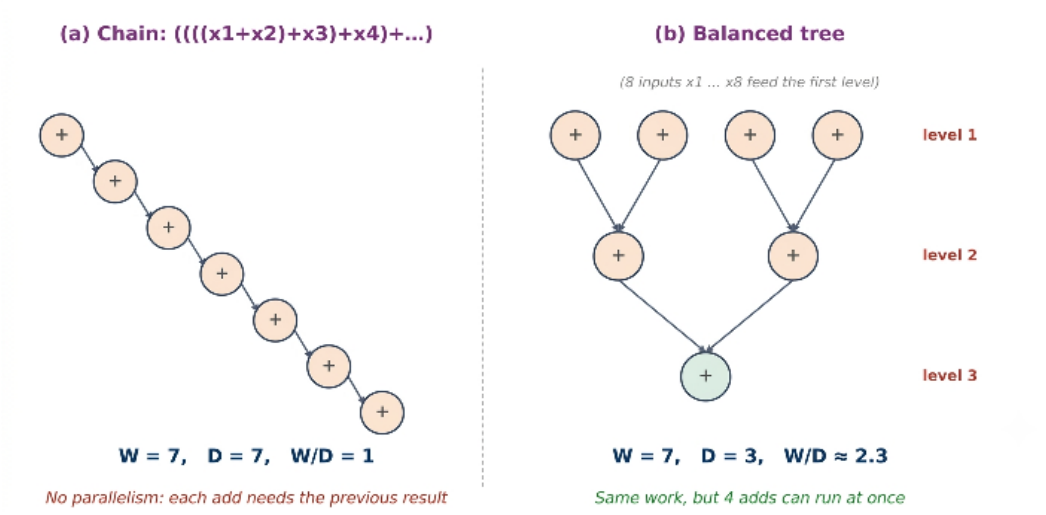


Fig. 5.3.7 Two ways of adding eight numbers: (a) a chain, with W = 7 and D = 7; (b) a balanced tree, with W = 7 and D = 3. The work is identical, but the depth is very different

Worked illustration

Both versions perform the same 7 additions, so W = 7 in each. The chain has D = 7 and parallelism W/D = 1: extra processors are useless. The tree has D = 3 (levels of 4, 2 and 1 additions) and parallelism 7/3 ≈ 2.3. On P = 4 processors the tree needs at least max(7/4, 3) = 3 steps against 7 for the chain, more than twice as fast, even though the total work is identical.

Try it yourself: find W, D and the lower bound on time with P = 4 for adding 16 numbers with a balanced tree. Answer: W = 15 additions; D = 4 levels (8 + 4 + 2 + 1); lower bound = max(15/4, 4) = max(3.75, 4) = 4 steps.

5.3.5 Performance Bottlenecks

A performance bottleneck is any single resource or dependency that limits overall system throughput or speedup, such that improving every other part of the system produces no further benefit until the bottleneck itself is addressed. Identifying bottlenecks is often more valuable than simply adding more raw hardware, since additional processors, memory, or bandwidth applied to a non-bottlenecked part of the system yield little or no improvement.

5.3.5.1 The Memory Wall

One of the most persistent bottlenecks in high-performance computing is the so-called memory wall: over several decades, processor clock speeds and arithmetic throughput have improved far faster than main-memory access latency and bandwidth, so that a processor capable of extremely fast computation may nonetheless spend much of its time idle, waiting for data to arrive from memory. Figure 5.3.8 illustrates this widening gap over time.

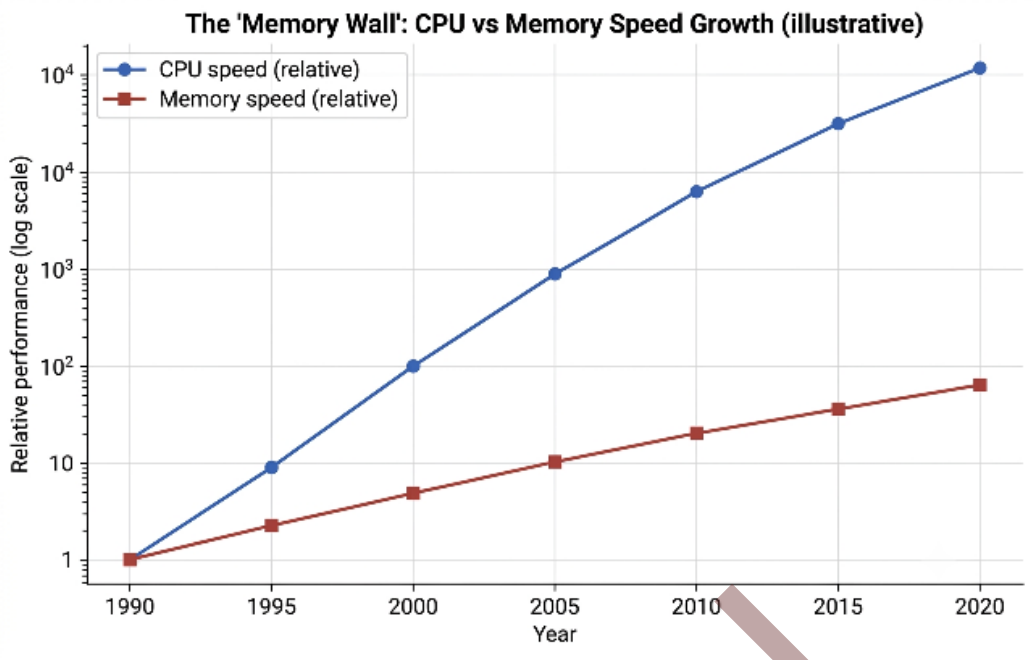


Fig. 5.3.8 The 'memory wall': illustrative relative growth of CPU speed versus memory speed over time, showing the widening gap that limits achievable performance

The memory wall directly motivates much of the cache-hierarchy and cache-coherence material covered earlier in this course: caches exist precisely to hide this growing latency gap by keeping frequently used data physically close to the processor, and the cache coherence protocols discussed there exist to keep multiple such caches consistent without forcing every access back out to slow main memory.

5.3.5.2 Other Common Bottlenecks

- Interconnection network bandwidth: in message-passing and NUMA systems, the network connecting nodes can become saturated well before individual processors are fully utilised, especially for communication-intensive algorithms.
- Serial bottleneck sections: as Amdahl's Law formalises, any portion of a program that must execute sequentially — even a small one — imposes a hard ceiling on overall speedup that no amount of additional parallel hardware can overcome.
- I/O bottlenecks: reading input data from, or writing results to, disk or network storage is often orders of magnitude slower than in-memory computation, and can dominate total execution time for data-intensive high-performance computing workloads.

SGOU

Design principle

A system should always be tuned by first identifying its current bottleneck (using the benchmarks and profiling tools discussed next) and improving that specific resource; improving any other part of the system before addressing the bottleneck typically produces negligible overall gain, a principle sometimes summarised as 'the chain is only as strong as its weakest link.'

Worked illustration (finding the bottleneck)

A profiler reports that a parallel run spends 30% of its time computing, 45% in MPI communication, 15% in file input/output and 10% idle. Halving the communication time gives a new total of 30 + 22.5 + 15 + 10 = 77.5% of the old time, a speedup of 1/0.775 ≈ 1.29×. Doubling the processor speed (halving the compute time) gives 15 + 45 + 15 + 10 = 85%, a speedup of only 1/0.85 ≈ 1.18×. Communication, the largest component, is the bottleneck worth attacking first — the same reasoning as Amdahl's Law, applied to the parts of a single run.

5.3.6 The Roofline Model

The roofline model provides a simple, visual way to diagnose whether a given program is limited by computational throughput or by memory bandwidth — precisely the memory-wall bottleneck discussed above. Each program is characterised by its operational intensity: the number of floating-point operations it performs per byte of data moved from memory. Plotting attainable performance (GFLOPs/s) against operational intensity produces a characteristic 'roofline' shape: a sloped line (bounded by available memory bandwidth) for low operational intensity, meeting a flat ceiling (bounded by peak computational throughput) for high operational intensity. Figure 5.3.9 plots this characteristic roofline shape.

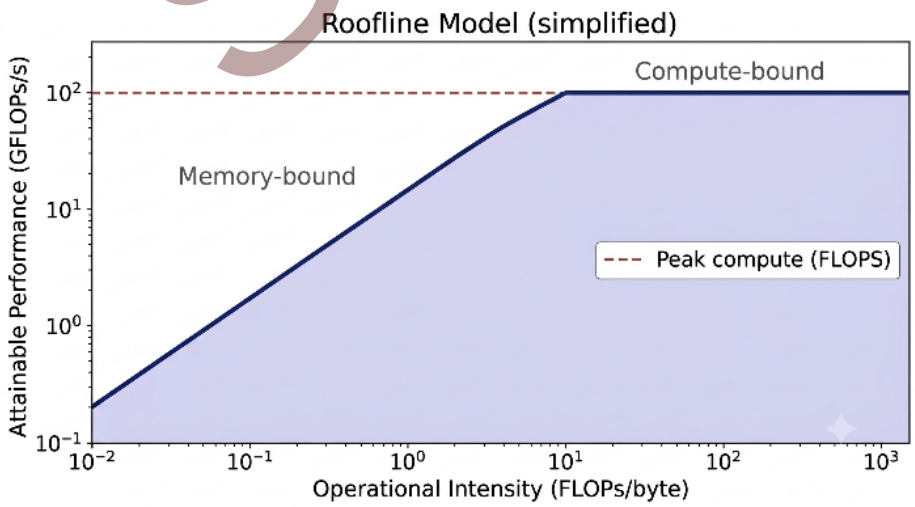


Fig. 5.3.9 The roofline model: programs with low operational intensity are memory-bandwidth-bound (below the sloped roof), while programs with high operational intensity are compute-bound (below the flat roof)

SGOU

A program that falls under the sloped part of the roof is memory-bound: no amount of additional computational hardware will help until memory bandwidth is improved or the algorithm is restructured to reuse data already in cache (increasing its operational intensity). A program under the flat part of the roof is compute-bound, and here additional computational hardware (more cores, wider vector units) genuinely helps. The roofline model therefore gives a quick, visual answer to exactly the kind of bottleneck-identification question raised earlier in this section.

Worked illustration

Take a machine with a peak of 1000 GFLOP/s (billions of floating-point operations per second) and a memory bandwidth of 100 GB/s. The sloped roof is operational intensity × bandwidth and the flat roof is 1000 GFLOP/s, so they meet at 1000/100 = 10 FLOP/byte (the "ridge point"). (a) Vector addition c[i] = a[i] + b[i] in double precision performs 1 floating-point operation while moving 24 bytes (two 8-byte reads and one 8-byte write), so its operational intensity is about 0.042 FLOP/byte and its attainable performance about $0.042 \times 100 \approx 4.2$ GFLOP/s — roughly 0.4% of peak: it is memory-bound. (b) Multiplying two n × n matrices performs about $2n^3$ operations on roughly $24n^2$ bytes of data if the cache can hold the operands, so the intensity is about n/12; for n = 1000 that is about 83 FLOP/byte, far to the right of the ridge point. The program is compute-bound and can approach the full 1000 GFLOP/s.

5.3.7 Benchmarks and Tools

Because performance depends heavily on workload characteristics, meaningful comparison between different high-performance computing systems requires standardised benchmarks — carefully specified programs or program suites run identically on different machines, so that resulting measurements are comparable.

LINPACK: measures a system's floating-point performance (in FLOPS) by solving a dense system of linear equations; the results of the LINPACK benchmark are used to compile the well-known TOP500 list, which ranks the world's fastest supercomputers twice a year.

SPEC (Standard Performance Evaluation Corporation) benchmarks: a family of benchmark suites (such as SPEC CPU) covering a broad mix of integer and floating-point workloads, widely used to compare general-purpose processor performance.

NAS Parallel Benchmarks (NPB): a suite developed by NASA specifically to evaluate the performance of parallel supercomputers on computational-fluid-dynamics-style workloads, including kernels that stress different communication patterns.

Profiling tools: software such as gprof, Intel VTune, and various MPI-aware tracing tools instrument a running parallel program to reveal where time is actually spent — computation, communication, synchronisation, or idle waiting — helping pinpoint the specific bottleneck limiting a given program's performance. Table 5.3.3 lists representative tools of each kind and what each is used for.

SGOU

Table 5.3.3 Representative benchmarks and profiling tools for high-performance computing

Tool/Bench mark	Primary Purpose	Typical Use
LINPACK	Dense linear-algebra floating-point performance	TOP500 supercomputer ranking
SPEC CPU	General-purpose integer/ floating-point workloads	Processor performance comparison
NAS Parallel Benchmarks	Parallel scientific/CFD-style kernels	Supercomputer / cluster evaluation
Profiling tools (VTune, gprof, MPI tracers)	Fine-grained time/resource breakdown	Bottleneck identification and tuning

Choosing a tool in practice. The right benchmark or profiler depends on the question being asked:

- A university wants to know how its new cluster ranks among the world's fastest machines: run LINPACK (HPL) and compare the result with the TOP500 list.
- A company wants to compare two server processors for general workloads: use published SPEC CPU results.
- A research group wants to check whether a cluster's network suits a fluid-dynamics code: run the NAS Parallel Benchmarks.
- A developer's MPI program stops scaling beyond 16 nodes and the cause is unknown: profile it with an MPI-aware tracer or VTune to see whether time goes to computation, communication or waiting.

5.3.8 Bringing It Together

Rigorous performance analysis turns the architectural ideas developed throughout this course — shared and distributed memory, cache coherence, data-flow execution — into quantitative, comparable statements about how fast a given system actually runs. Speedup, efficiency, and scalability give a common vocabulary for describing performance; Amdahl's and Gustafson's laws model the fundamental limits imposed by a program's sequential fraction under two different (fixed-size versus scaled) usage assumptions; the work-depth model formalises how much intrinsic parallelism an algorithm possesses independent of any particular machine; and standard benchmarks and profiling tools make it possible to measure real systems consistently and to locate the specific bottleneck limiting performance in a given case. Together, these tools equip the high-performance computing practitioner to reason precisely about when, why, and how much a parallel or data-flow system can be expected to outperform its sequential counterpart.

SGOU

Recap

- Speedup S(N) = T1/TN and efficiency E(N) = S(N)/N are the fundamental metrics for quantifying how effectively a parallel system uses N processors.
- Throughput and scalability describe, respectively, the rate of completing work and how well performance continues to improve as more resources are added.
- Communication overhead, load imbalance, synchronisation overhead, and memory contention are the principal factors that reduce achieved speedup below the ideal, linear value.
- Amdahl's Law models speedup under a fixed problem size and shows that the sequential fraction of a program sets a hard ceiling on achievable speedup, however many processors are added.
- Gustafson's Law models speedup under a scaled workload (larger problems as more processors become available) and shows speedup growing roughly linearly with the number of processors.
- The work-depth model characterises an algorithm's intrinsic parallelism via work (W, total operations) and depth (D, critical path length); running time is bounded below by max(W/P, D).
- The memory wall — the widening gap between processor speed and main-memory speed — is one of the most persistent performance bottlenecks in high-performance computing, motivating cache hierarchies and coherence protocols.
- Standard benchmarks (LINPACK, SPEC, NAS Parallel Benchmarks) and profiling tools enable consistent performance comparison across systems and help identify the specific bottleneck limiting a given workload.
- Strong scaling measures speedup on a fixed-size problem as processors are added (mirroring Amdahl's Law); weak scaling measures how execution time behaves as problem size and processor count grow together (mirroring Gustafson's Law).
- The roofline model plots attainable performance against operational intensity to quickly diagnose whether a program is memory-bandwidth-bound or compute-bound.
- Amdahl's Law in numbers (P = 0.9): speedup reaches only 6.4 on 16 processors and 9.9 on 1024, while efficiency falls from 91% to about 1%.
- Algorithm structure matters: adding eight numbers with a chain has depth 7, but with a balanced tree depth 3, although the work (7 additions) is the same.

SGOU

- Tuning starts with measurement: profile the run and attack the largest component first; the roofline ridge point (peak performance ÷ memory bandwidth) separates memory-bound from compute-bound programs.

Objective Type Questions

1. What is the formula for speedup?
2. What is the formula for parallel efficiency?
3. Which law limits speedup due to the sequential portion of a program?
4. What is the maximum speedup according to Amdahl's Law?
5. Which law assumes that problem size increases with the number of processors?
6. What does work (W) represent in the work-depth model?
7. What does depth (D) represent in the work-depth model?
8. What is the longest chain of dependent operations called?
9. What is the minimum running-time bound with P processors in terms of W and D?
10. What is the memory wall?
11. Which benchmark is used for the TOP500 supercomputer list?
12. Name one tool used for profiling programs.
13. What remains fixed in strong scaling?
14. What remains fixed per processor in weak scaling?
15. What does operational intensity measure?
16. A program takes 240 seconds on one processor and 20 seconds on 16 processors. What is its speedup?
17. What is the efficiency when speedup is 12 on 16 processors?
18. A program is 95% parallelisable. What is its maximum possible speedup according to Amdahl's Law?

SGOU

19. What is the scaled speedup on 64 processors when the sequential fraction is 5%, according to Gustafson's Law?

20. How many additions are required to add 16 numbers using a balanced tree?

21. What is the depth of a balanced addition tree for 16 numbers?

22. In the roofline model, what is the attainable performance when peak performance is 500 GFLOP/s, memory bandwidth is 50 GB/s, and operational intensity is 4 FLOP/byte?

23. Is a kernel with an operational intensity of 4 FLOP/byte memory-bound or compute-bound when the ridge point is 10 FLOP/byte?

24. Which type of scaling test doubles both the number of processors and the problem size?

25. What is the main purpose of a profiling tool?

Answers

1. T_1/T_P
2. $S(P)/P$
3. Amdahl's Law
4. $1/f$
5. Gustafson's Law
6. Total computation
7. Critical-path length
8. Critical path
9. $max(W/P, D)$
10. Processor-memory gap
11. LINPACK
12. gprof
13. Total problem size

SGOU

14. Workload per processor

15. Operations per byte

16. 12×

17. 75%

18. 20×

19. 60.85

20. 15 additions

21. 4 levels

22. 200 GFLOP/s

23. Memory-bound

24. Weak scaling

25. Hotspot identification

Assignments

1. Define and explain speedup, efficiency, throughput, and scalability as performance metrics for parallel systems.

2. Explain the four major factors that reduce achieved speedup below the ideal, linear value, giving an example of each.

3. State and derive Amdahl's Law, and use it to compute the maximum possible speedup for a program that is 85% parallelisable.

4. State Gustafson's Law and explain, with a suitable example, why it gives a more optimistic view of scalability than Amdahl's Law for certain workloads.

5. Explain the work-depth model of parallel algorithm analysis with a worked example, and derive the lower bound on running time with P processors.

6. Discuss the memory wall as a performance bottleneck, and explain how cache hierarchies and coherence protocols help mitigate it.

7. Describe any three standard benchmarks used in high-performance computing, explaining what each measures and how the results are used.

SGOU

8. Explain how profiling tools help identify performance bottlenecks in a parallel program, with a suitable example scenario.

9. Distinguish between strong scaling and weak scaling with suitable examples, and relate each to Amdahl's and Gustafson's laws respectively.

10. Explain the roofline model and how it is used to determine whether a program is memory-bound or compute-bound.

11. A program is 80% parallelisable. Using Amdahl's Law, compute the speedup for N = 4, 16 and 64, state the maximum possible speedup, and find the smallest N that gives a speedup of 4.

12. Compute the work, depth and parallelism of the dot product of two 8-element vectors (8 multiplications followed by a balanced tree of 7 additions). Find the lower bound on running time for P = 2, 4 and 8.

13. A weather centre must produce a forecast before a fixed deadline, while a climate group wants to run the same model at four times the resolution in the same time. Analyse which of Amdahl's and Gustafson's laws is the better guide for each, and why.

14. Two kernels run on a machine with a peak of 800 GFLOP/s and 100 GB/s of memory bandwidth: kernel A has an operational intensity of 0.5 FLOP/byte and kernel B 20 FLOP/byte. Place each on the roofline, state its limiting resource, and recommend one optimisation for each.

15. A profile of a parallel run shows 25% computation, 50% communication, 15% input/output and 10% idle time. Compare the overall speedups obtained by (a) halving the communication time and (b) halving the input/output time, and recommend where to spend effort.

SGOU

Reference

1. Amdahl, G. M. (1967). *Validity of the single processor approach to achieving large scale computing capabilities*. AFIPS Conference Proceedings.

2. Gustafson, J. L. (1988). *Reevaluating Amdahl's law. Communications of the ACM*, 31(5), 532–533. https://doi.org/10.1145/42411.42415

Suggested Reading

1. Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *Introduction to parallel computing* (2nd ed.). Pearson.

2. Hennessy, J. L., & Patterson, D. A. (2017). *Computer architecture: A quantitative approach* (6th ed.). Morgan Kaufmann.

3. Hwang, K., & Jotwani, N. (2016). *Advanced computer architecture: Parallelism, scalability, programmability* (3rd ed.). McGraw-Hill Education.

SGOU



Case Studies of Multiprocessor and Data-Flow Systems

Learning Outcomes

After completion of this unit, learner will be able to:

- identify representative commercial multiprocessor systems and situate them within the shared-memory / distributed-memory classification studied earlier
- describe real-world shared-memory architecture case studies and relate their design choices to UMA/NUMA concepts
- describe real-world message-passing multiprocessor case studies and explain how they scale to large numbers of nodes
- recall the basics of data-flow computer architecture and relate them to concrete historical machines, and
- compare and contrast the case studies covered in this unit on the basis of architecture, scalability, and intended workload

Prerequisite

This unit is a case-study-based synthesis of the entire course and assumes the learner has completed the earlier units on shared and distributed memory, UMA and NUMA architectures, cache coherence, multiprocessor operating systems, data-driven computing, and static and dynamic data-flow architectures. No new theoretical concepts are introduced here; instead, the concepts already studied are anchored to specific, real commercial and research systems so that the abstract architectural principles can be seen at work in concrete machines. For instance, the smartphone or laptop used to read this material is itself a small multiprocessor system, and the supercomputers that forecast the weather or train large AI models are direct descendants of the shared-memory, message-passing, and data-flow machines studied as case studies in this unit -- seeing how these design choices played out in real, named systems makes the earlier, more abstract units concrete and memorable.

SGOU

Keywords

Commercial Multiprocessor, Shared-Memory System, Message-Passing System, Cluster, Symmetric Multiprocessing, Data-Flow Computer, Tagged-Token Architecture, SIMD, Directory-Based Coherence

Discussion

5.4.1 Commercial Multiprocessor Systems: An Overview

The theoretical architectures discussed earlier in this course, including shared memory, distributed memory, Uniform Memory Access (UMA), and Non-Uniform Memory Access (NUMA), have been implemented and developed into commercial systems over several decades of high-performance computing (HPC) history. In UMA systems, all processors access memory with approximately equal latency, whereas in NUMA systems, memory access latency depends on whether the memory is local or remote. Examining a handful of representative commercial systems shows how the same underlying principles were adapted to the technology and market demands of their era. Fig. 5.4.1 places these systems on a timeline.



Fig. 5.4.1 A simplified timeline of representative commercial and research multiprocessor systems, spanning shared-memory, message-passing, and massively parallel designs

Commercial multiprocessor systems can be broadly classified according to their memory organisation into shared-memory systems (Section 5.4.2) and message-passing or distributed-memory systems (Section 5.4.3). This classification follows the concepts introduced earlier in the course. However, many commercial systems combine both approaches. For example, a cluster may consist of multiple shared-memory nodes connected through a message-passing network, allowing processors to share memory within each node and exchange messages between nodes.

Interesting fact

Many of the design choices that seem obvious today such as building a supercomputer from many relatively ordinary processors connected by a fast network, rather than from a single, extremely exotic, custom-built processor, represent a historical shift in high-performance computing philosophy, moving away from the highly specialised vector supercomputers (machines whose instructions operate on whole arrays of numbers at once) of the 1970s and 1980s (such as the Cray-1) toward the commodity-cluster and massively-parallel designs that dominate the field today

SGOU

5.4.2 Shared Memory Architectures: Case Studies

Shared-memory commercial systems implement the UMA and NUMA principles studied earlier using real hardware interconnects and operating systems. Figure 5.4.2 depicts a representative shared-memory server of this kind.

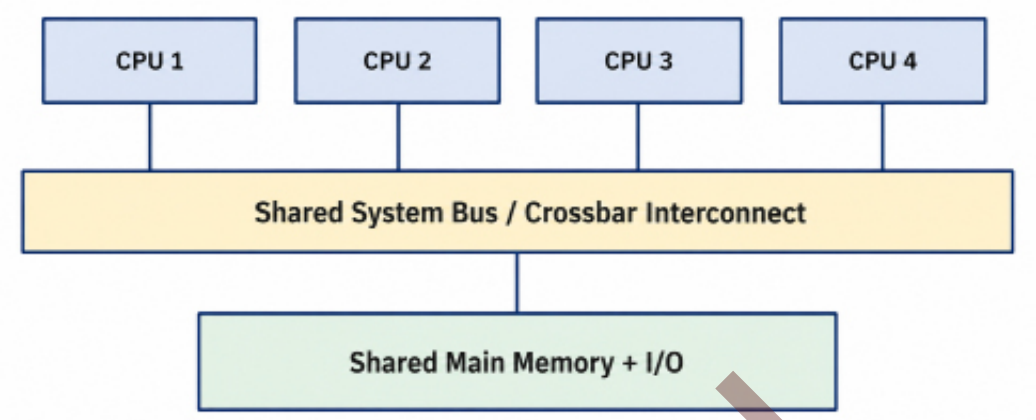


Fig. 5.4.2 A representative commercial symmetric-multiprocessor (SMP) server: multiple CPUs connected to shared memory through a common bus or crossbar interconnect

5.4.2.1 Cray X-MP and Y-MP

The Cray X-MP (1982) and its successor the Cray Y-MP were among the earliest commercially successful shared-memory multiprocessors, extending Cray's earlier single-processor vector-supercomputer designs to two, four, or eight tightly coupled processors sharing a common, high-bandwidth memory. These machines exemplify the tightly coupled, UMA-style shared-memory design discussed earlier: all processors saw the same, uniformly accessible memory, which kept programming comparatively simple while the number of processors remained modest.

5.4.2.2 SGI Origin 2000

The SGI Origin 2000 (1996) is a classic example of a commercial cache-coherent NUMA (ccNUMA) system, directly implementing the NUMA architecture and directory-based cache coherence protocol (in which a table, the directory, records which nodes hold a copy of each memory line, so that updates are sent only to those nodes) studied earlier in this course. The Origin 2000 organised processors into small groups ('nodes'), each with its own local memory, connected by a scalable interconnection network (a hypercube-like topology called CrayLink); a hardware directory tracked which nodes cached which memory lines, allowing the system to scale to well over 100 processors while still presenting programmers with a single, shared address space.

How a directory works, step by step

Figure 5.4.3 follows one read miss. Node A needs memory line X, whose "home" (the node that holds its memory and its directory entry) is Node B, while the latest, modified copy sits in Node C's cache.

SGOU

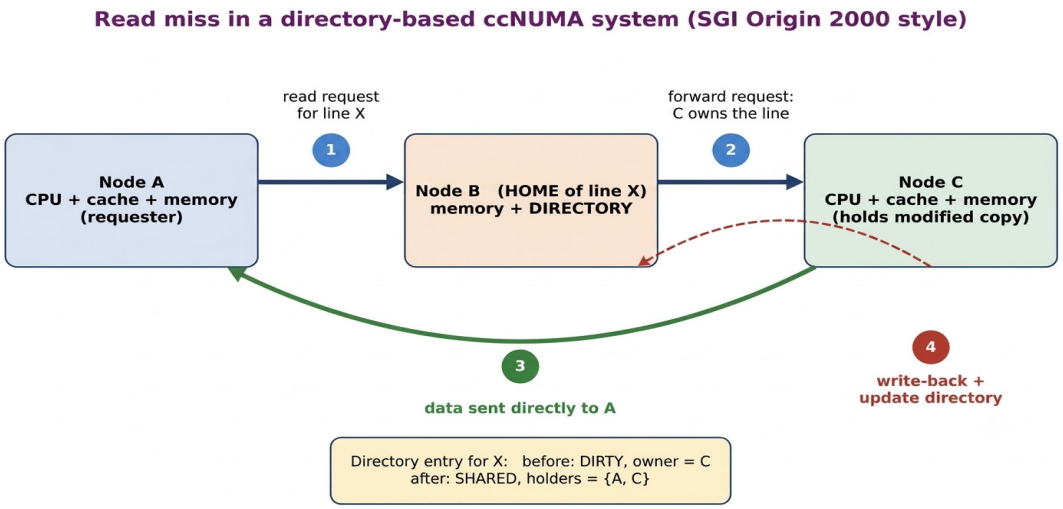


Fig. 5.4.3 A read miss in a directory-based ccNUMA system: (1) A asks the home node B; (2) B's directory shows that C owns the modified line and forwards the request; (3) C sends the data directly to A; (4) C writes the line back and the directory is updated

Because only the home node and the current owner are contacted, the traffic grows slowly as the machine gets larger; a protocol that broadcasts every miss to all nodes would flood the network once there are many nodes. This is what allowed the Origin 2000 to scale beyond 100 processors.

5.4.2.3 Modern Multi-Socket Servers

Present day commercial servers from vendors such as IBM, HPE, and Dell routinely support two, four, or eight physical CPU sockets, with each socket containing multiple processor cores. All processors share a single operating system image and a single NUMA address space. This design is a modern descendant of the Origin 2000's ccNUMA architecture, now implemented on a single motherboard rather than across multiple cabinets. These servers use the symmetric multiprocessing (SMP) operating system organisation studied earlier in this course, in which all processors are equal peers that share one operating system and memory space. Table 5.4.1 summarises these three shared-memory case studies.

Table 5.4.1 Selected shared-memory commercial system case studies

System	Era	Memory Organisation	Notable Feature
Cray X-MP / Y-MP	1980s	UMA, tightly coupled	Vector processing + shared memory
SGI Origin 2000	1990s	ccNUMA, directory-based coherence	Scaled to 100+ processors
Modern multi-socket servers	2010s–present	ccNUMA (on-board)	Many cores per socket, SMP OS

SGOU

Connecting to earlier units

The Origin 2000's directory-based coherence and NUMA distance considerations are the exact real-world instances of the directory-based coherence protocol and NUMA distance matrix discussed in the memory-organisation portion of this course — the concepts are not abstract exercises but describe how actual, commercially shipped machines were built.

Practical applications

Machines of the Cray X-MP/Y-MP class were used for weather forecasting and engineering simulation; the Origin 2000 for large scientific computations; and today's multi-socket servers commonly host in-memory databases and enterprise applications in which many threads read and update one large shared data set. Such workloads favour a single shared address space, because threads can use the same data structures without writing explicit message-passing code.

5.4.3 Message Passing Multiprocessors: Case Studies

Message-passing (distributed-memory) systems abandon the single shared address space in favour of scalability, connecting many independent nodes, each with local memory through a dedicated high-speed network, exactly as studied in the distributed-memory architecture material earlier in this course. Figure 5.4.4 depicts this cluster organisation.

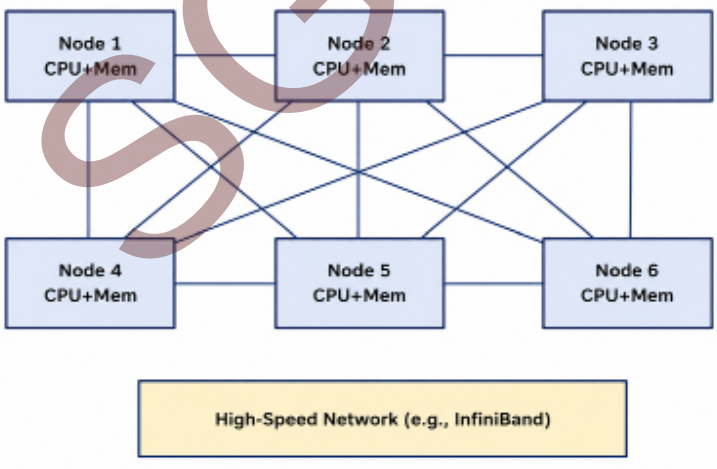


Fig. 5.4.4 A representative message-passing cluster: independent nodes, each with local memory, connected by a high-speed interconnection network

5.4.3.1 IBM SP2

The IBM SP2 (Scalable POWERparallel, 1994) connected dozens to hundreds of individual IBM RS/6000 workstation-class nodes, each with its own processor and local memory, through a dedicated high-performance switch. Programs on the SP2

SGOU

were written using explicit message passing, the software model corresponding directly to the distributed-memory hardware organisation, and the SP2 became one of the most commercially successful early distributed-memory supercomputers, illustrating that a machine assembled from many relatively standard processors could be more cost-effective and more scalable than a smaller number of exotic custom processors.

5.4.3.2 IBM Blue Gene

The IBM Blue Gene family (Blue Gene/L, 2004, and later Blue Gene/P and /Q) pushed the message-passing, distributed-memory philosophy to a much larger scale, connecting tens of thousands to over a million relatively low-power processor cores through multiple specialised interconnection networks (including a three-dimensional torus network (a grid whose edges wrap around, so every node has the same number of neighbours) for general communication and a separate tree network for collective operations such as broadcasts and reductions). Blue Gene systems held the top position on the TOP500 supercomputer list (introduced in Unit 3) for several years, demonstrating that very large numbers of modest individual processors, carefully interconnected, could outperform far fewer, individually faster processors on suitably parallel workloads; a direct, large-scale illustration of Amdahl's and Gustafson's laws in practice.

5.4.3.3 Beowulf Clusters and Commodity Computing

Alongside purpose-built supercomputers, the 1990s also saw the rise of Beowulf clusters: distributed-memory supercomputers assembled from ordinary, commodity personal-computer hardware connected by standard Ethernet networking and running open-source message-passing software (typically an implementation of MPI, the Message Passing Interface). Beowulf style clusters significantly reduced the cost of accessing high performance computing and established a widely adopted architectural model based on commodity nodes, standard networking, and MPI based software. This model forms the foundation of most academic and industrial computing clusters used today. Table 5.4.2 compares the IBM SP2, IBM Blue Gene, and Beowulf clusters on this basis.

Table 5.4.2 Selected message-passing (distributed-memory) system case studies

System	Era	Scale	Notable Feature
IBM SP2	1990s	Tens to hundreds of nodes	Early commercially successful MPP
IBM Blue Gene/L	2000s	Tens of thousands+ cores	Torus + tree interconnect; TOP500 leader
Beowulf clusters	1990s–present	Variable (commodity scale)	Commodity hardware + MPI; low cost of entry

SGOU

Worked illustration

A Beowulf cluster of 100 ordinary desktop-class nodes, each roughly 1/100th as individually powerful as a specialised supercomputer processor, can nonetheless match or exceed that supercomputer's aggregate throughput on a well-parallelised workload with a small sequential fraction, precisely the scenario Amdahl's Law shows to be favourable, and the reasoning that made commodity clusters commercially disruptive.

Message passing in practice

In an MPI program every process has its own memory and cannot read another process's memory directly. The sketch below (pseudocode) lets each of several processes add up its own share of the data and then combines the partial sums on process 0:

```
rank  = MPI_Comm_rank()                  # my process number: 0, 1, 2, ...
local = sum(my_share_of_the_data)        # each process works on its own data
total = MPI_Reduce(local, SUM, root = 0) # combine the partial sums on process 0
if rank == 0: print(total)
```

The combining step is an explicit collective message-passing operation; on Blue Gene such reductions were handled by the dedicated tree network. A typical real use is weather forecasting: the atmosphere is divided into regions, each assigned to one process, and after every time step neighbouring regions exchange their boundary values by messages.

5.4.3.4 The Connection Machine: A Data-Parallel Alternative

Not every massively parallel machine used the message-passing, multiple-instruction style discussed above. Thinking Machines Corporation's Connection Machine (CM-1, 1985, and later CM-2) took a data-parallel SIMD (Single Instruction, Multiple Data) approach: a single control unit broadcasts one instruction at a time to thousands of simple processing elements, each of which executes that same instruction simultaneously but on its own local data. This design suited problems with a very regular, uniform structure such as image processing or cellular-automaton-style simulations, extremely well, since a single instruction stream could drive enormous data parallelism without the overhead of independently scheduling and synchronising many separate instruction streams. Figure 5.4.5 depicts this data-parallel organisation.

SGOU

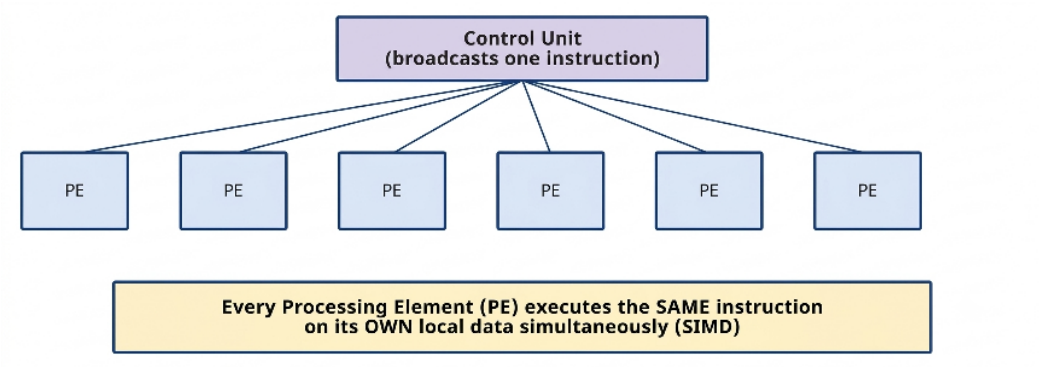


Fig. 5.4.5 The Connection Machine's data-parallel (SIMD) organisation: a single control unit broadcasts one instruction that every processing element executes simultaneously on its own local data

Contrast with message passing

In the SP2 or Blue Gene's MIMD (Multiple Instruction, Multiple Data) message-passing style, every node runs its own independent instruction stream and explicitly sends messages when it needs to communicate; in the Connection Machine's SIMD style, there is only one instruction stream for the whole machine, and communication between processing elements is handled by specialised hardware rather than explicit message-passing code. The SIMD approach is a natural match for the fine-grained, homogeneous parallelism that also characterises many map-style data-centric operations discussed in Unit 1, foreshadowing the data-parallel execution style later adopted by GPUs.

Worked illustration (SIMD)

To brighten a 1024 × 1024 image (1,048,576 pixels), every pixel needs the same operation, out = min(255, 1.2 × in). With one control unit and 1,024 processing elements, the instruction is broadcast once for each group of 1,024 pixels, so the whole image takes 1,048,576 / 1,024 = 1,024 broadcasts. A SIMD machine is inefficient, however, when elements need different operations: in an if/else, the elements that do not take the current branch must sit idle while the others execute it.

5.4.3.5 GPUs and Modern Heterogeneous Supercomputers

Today's fastest supercomputers on the TOP500 list (introduced in Unit 3) are typically heterogeneous message-passing clusters (clusters that mix different kinds of processor) in which each node combines conventional CPUs with one or more graphics processing units (GPUs). A GPU is, internally, organised very much like a modern descendant of the Connection Machine's SIMD philosophy, thousands of simple cores executing the same instruction stream in lockstep across large, regular data sets (such as matrices in deep-learning training) while the cluster of GPU-equipped nodes as a whole communicates via conventional message passing (typically MPI), exactly like the IBM SP2 or Blue

SGOU

Gene. Modern high-performance computing systems therefore often combine, within a single machine, the message-passing, shared-memory, and data-parallel SIMD principles all studied as separate historical threads in this unit.

5.4.4 Data-Flow Architecture Basics: A Recap

Before examining concrete data-flow machines, it is useful to briefly recap the architectural principles established in Unit 2: a data-flow computer executes a data-flow graph directly, firing each instruction as soon as its operand tokens are available, using a matching unit to detect when an instruction's full set of operands has arrived, and routing result tokens onward to whichever further instructions require them. Figure 5.4.6 recaps this organisation.

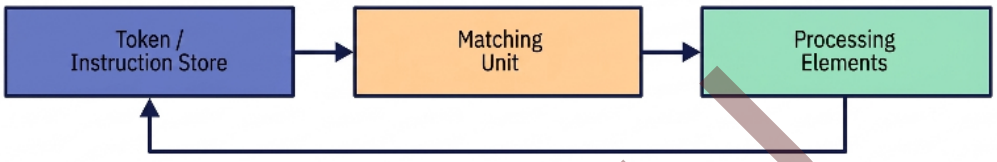


Fig. 5.4.6 A brief recap of the general data-flow computer organisation studied in Unit 2: instruction/token store, matching unit, and processing elements

As Unit 2 explained, static data-flow machines permit only one token per arc, simplifying matching hardware at the cost of preventing re-entrant execution of loops and recursive calls, while dynamic (tagged-token) data-flow machines attach a context tag to every token, allowing many instances of the same instruction to be in flight simultaneously at the cost of more complex, associative matching hardware.

5.4.5 Examples of Data-Flow Computers

Several concrete machines were built to realise the static and dynamic data-flow principles studied in Unit 2, each contributing distinct engineering lessons.

5.4.5.1 The MIT Static Data-Flow Machine

Designed by Jack Dennis and colleagues at MIT beginning in the mid-1970s, this was the first detailed, worked-out proposal for a static data-flow computer, establishing the activity-template-based matching mechanism and single-token-per-arc discipline discussed in Unit 2. Although never built at full scale as a commercial product, it served as the conceptual foundation for essentially all subsequent data-flow architecture research.

5.4.5.2 The Manchester Dataflow Machine

Built at the University of Manchester in the early 1980s under the direction of John Gurd and Ian Watson, the Manchester Dataflow Machine was one of the first working hardware prototypes of a dynamic, tagged-token data-flow computer. It successfully demonstrated hardware supported reentrancy, which allows multiple instances of a

SGOU

loop body or recursive function call to execute simultaneously. This implementation validated the tagged token matching principle discussed theoretically in Unit 2.

5.4.5.3 MIT Tagged-Token Dataflow Architecture and Monsoon

Continuing MIT's data-flow research through the 1980s, Arvind and colleagues developed the Tagged-Token Dataflow Architecture, later realised in the Monsoon machine (1990) built in collaboration with Motorola. Monsoon introduced the explicit token store (ETS) design, discussed as a data flow design alternative in Unit 2. It allocated addressable memory frames for each procedure activation, avoiding the overhead of fully associative tag searching while supporting reentrant execution. This approach provided a practical compromise between purely static and purely dynamic data flow designs.

5.4.5.4 SIGMA-1 and Other International Efforts

Japan's SIGMA 1 project, developed at the Electrotechnical Laboratory during the 1980s, aimed to build a large-scale dynamic data flow machine for scientific computing. In contrast, France's LAU System explored a coarse grained approach, in which entire blocks of instructions rather than individual operations were used as the units of data flow scheduling. These systems demonstrate the trade off between fine grained and coarse grained scheduling discussed as a data flow design alternative in Unit 2. Table 5.4.3 presents a comparison of these historical machines.

Table 5.4.3 Examples of historical data-flow computers

Machine	Institution / Year	Type	Key Lesson
MIT Static Dataflow	MIT, mid-1970s	Static	Established the activity-template matching mechanism
Manchester Dataflow Machine	Univ. of Manchester, 1981	Dynamic (tagged-token)	First working hardware demonstration of re-entrancy
Monsoon (Tagged-Token DFA)	MIT/Motorola, 1990	Dynamic, explicit-token-store	Practical hybrid of static and dynamic designs
SIGMA-1	ETL Japan, 1980s	Dynamic	Large-scale dynamic dataflow for scientific computing
LAU System	France, 1980s	Static, large-grain	Explored coarse-grained dataflow actors

SGOU

Legacy

Although none of these dedicated data-flow machines achieved lasting commercial success against conventional control-flow processors, their core insight, fire an instruction as soon as its data dependencies are satisfied, survives directly inside every modern out-of-order superscalar processor (as discussed in Unit 2) and inside the task-graph schedulers of frameworks such as Apache Spark and TensorFlow (as discussed in Unit 1), making data-flow architecture research one of the most influential 'commercially unsuccessful but conceptually victorious' threads in computer architecture history.

Worked illustration (data-flow firing inside a modern CPU)

Consider I1: r1 = r2 + r3; I2: r4 = r1 × r5; I3: r6 = r7 − r8; I4: r9 = r4 + r6. I2 needs I1 (through r1), and I4 needs both I2 (r4) and I3 (r6). Assume one-cycle operations and two functional units. In program order, one at a time, the four instructions take 4 cycles. An out-of-order processor fires I1 and I3 together in cycle 1 (their operands are ready and they do not depend on each other), I2 in cycle 2 (r1 has arrived), and I4 in cycle 3 (both r4 and r6 are available): 3 cycles. This is a data-flow graph in disguise: the register values act as tokens, and the reservation stations act as the matching unit.

5.4.6 Comparing the Case Studies

The systems examined in this unit demonstrate that the same fundamental architectural principles, such as shared versus distributed memory, static versus dynamic scheduling, and fine grained versus coarse grained parallelism, are used across different computing systems. Although these systems were developed in different periods and for different purposes, they apply these principles in various ways to improve performance, resource utilisation, and scalability. Table 5.4.4 synthesises all the case studies from this unit within the shared-memory / message-passing / data-flow classification.

Table 5.4.4 A synthesis of case studies across the shared-memory / message-passing / data-flow classification

Category	Representative Systems	Core Principle Illustrated
Shared memory (NUMA)	SGI Origin 2000, modern multi-socket servers	Directory-based coherence, non-uniform access
Message passing	IBM SP2, Beowulf clusters	Scalable, loosely coupled distributed memory
Massively parallel	IBM Blue Gene	Very large-scale message passing with specialised interconnects
Data-parallel (SIMD)	Connection Machine CM-1/ CM-2	Single instruction stream driving many processing elements

SGOU

Heterogeneous / GPU-based	Modern TOP500 systems	SIMD-style GPUs combined with MPI-based clusters
Static data-flow	MIT Static Dataflow, LAU System	Simple matching; limited re-entrancy
Dynamic data-flow	Manchester Dataflow Machine, Monsoon, SIGMA-1	Tagged tokens enabling re-entrant execution

Applying the comparison.

Table 5.4.4 classifies the systems; Table 5.4.5 shows how the same classification is used to choose an architecture for a given workload.

Table 5.4.5 Matching workloads to architectures (worked scenarios)

Workload	Key characteristic	Best-fit architecture	Reason
In-memory database serving many users	Many threads read and update the same large data set	Shared-memory multi-socket server (ccNUMA / SMP)	One address space lets threads share data without explicit messages; coherence hardware keeps caches consistent
Weather forecast on a grid of millions of cells	Regions are mostly independent and exchange boundary data with neighbours	Message-passing cluster (MPI), Beowulf or Blue Gene style	Scales to thousands of nodes; communication is regular and small compared with computation
Brightening or filtering a very large image	The same operation on every pixel	SIMD / GPU	One instruction stream drives thousands of simple cores
Training a large neural network	Regular matrix arithmetic on very large data	GPU-equipped cluster (SIMD inside a node, MPI between nodes)	Combines data-parallel speed within a node with scalability across nodes
Running a graph of tasks with irregular dependencies	Work becomes ready at unpredictable times	Data-flow-style, dependency-driven scheduling (e.g. task-graph runtimes)	Each task runs as soon as its inputs are ready, exposing parallelism automatically

SGOU

Try it yourself

A laboratory plans to run a molecular-dynamics code in which neighbouring regions send small messages to each other every time step. Which row of Table 5.4.5 is closest, and what would you check before buying the hardware?

Answer

The weather-forecast row (message-passing cluster). Check the interconnect's latency and bandwidth and the communication-to-computation ratio (Section 5.3.2), because a slow network would limit how far the code scales.

5.4.7 Bringing It Together

This unit has traced the shared-memory, message-passing, and data-flow architectural principles developed throughout this course through a sequence of concrete, historically important machines. Cray's early vector multiprocessors and SGI's Origin 2000 showed shared-memory UMA and NUMA principles realised in commercially successful products; IBM's SP2 and Blue Gene, together with the Beowulf cluster movement, showed message-passing distributed memory scaling from dozens to over a million processing cores; and the MIT, Manchester, and Monsoon data-flow machines showed the static and dynamic data-flow principles of Unit 2 validated in working hardware. Together with the performance analysis tools discussed in Unit 3, these case studies provide a comprehensive understanding of high performance computing architecture. They demonstrate how architectural principles develop from fundamental concepts and mathematical performance models into practical computing systems. They also show how ideas originally explored in dedicated data flow hardware continue to influence modern processors and data processing frameworks used in everyday computing.

Recap

- Commercial multiprocessor systems realise the shared-memory and distributed-memory (message-passing) principles studied earlier in this course using real interconnects, operating systems, and application software.
- The Cray X-MP/Y-MP exemplify early tightly coupled, UMA-style shared-memory multiprocessing; the SGI Origin 2000 exemplifies commercial ccNUMA with directory-based coherence, scaling to over 100 processors.
- Modern multi-socket servers are direct descendants of ccNUMA designs, running SMP operating systems across many cores per socket.
- The IBM SP2 and Beowulf clusters exemplify message-passing distributed memory built from relatively standard processing nodes; IBM Blue Gene scaled this philosophy to over a million processor cores using specialised torus and tree interconnects.

SGOU

- The Connection Machine exemplified a data-parallel SIMD alternative, broadcasting a single instruction stream to thousands of processing elements, each operating on its own local data.
- Modern GPU-based supercomputers combine SIMD-style data parallelism within each GPU with conventional MPI-based message passing across nodes, blending several architectural principles studied separately in this unit.
- Static data-flow principles were first detailed in the MIT Static Dataflow Machine; dynamic (tagged-token) data-flow was first demonstrated in working hardware by the Manchester Dataflow Machine.
- MIT's Monsoon machine introduced a practical explicit-token-store (ETS) design as a hybrid between static and dynamic data-flow extremes; SIGMA-1 and the LAU System further explored large-scale and coarse-grained data-flow alternatives, respectively.
- Although dedicated data-flow hardware did not achieve lasting commercial dominance, its core firing principle persists inside modern out-of-order processors and data-flow-graph-based software frameworks.
- Out-of-order execution applies the firing rule inside a CPU: independent instructions run as soon as their operands are ready (the four-instruction example takes 3 cycles instead of 4).
- Choosing an architecture starts from the workload: shared memory for tightly shared data, message passing for scalable and regular communication, and SIMD/GPU for uniform data parallelism.

Objective Type Questions

1. Name one commercially successful shared memory multiprocessor from the 1980s.
2. What memory organisation did the SGI Origin 2000 implement?
3. What type of cache coherence protocol did the SGI Origin 2000 use?
4. Name one commercial distributed memory system from the 1990s.
5. Which network topology did IBM Blue Gene use for general communication?
6. What type of cluster is built using commodity personal computers?
7. What does SIMD stand for?

SGOU

8. Which architecture is commonly used for processing millions of pixels simultaneously?

9. Which programming model is commonly used in Beowulf clusters?

10. Name the first working hardware prototype of a dynamic data flow computer.

11. Which architecture did the Monsoon machine introduce?

12. In which country was the SIGMA 1 data flow computer developed?

13. Which architecture is suitable for an in memory database shared by multiple threads?

14. Which law estimates speedup in parallel computing?

15. How many instruction broadcasts are required for 4,096 elements using 1,024 processing elements?

16. What type of processor executes independent instructions out of program order?

17. What type of coherence does the SGI Origin 2000 implement?

18. What limits SIMD performance when processing data dependent branches?

19. Which communication model does MPI support?

Answers

1. Cray X MP
2. NUMA
3. Directory
4. IBM SP2
5. Torus
6. Beowulf
7. SIMD
8. SIMD
9. MPI

SGOU

10. Manchester

11. ETS

12. Japan

13. Shared memory

14. Amdahl's Law

15. Four

16. Out of order

17. Cache coherence

18. Divergence

19. Message passing

Assignments

1. Describe the Cray X-MP/Y-MP and SGI Origin 2000 as case studies of shared-memory multiprocessor architecture, relating their design to the UMA/NUMA classification studied earlier in this course.

2. Describe the IBM SP2 and IBM Blue Gene as case studies of message-passing multiprocessor architecture, and explain how each scales to large numbers of processors.

3. What is a Beowulf cluster? Explain why commodity clusters became commercially disruptive, relating your answer to Amdahl's and Gustafson's laws.

4. Describe the Connection Machine's SIMD data-parallel architecture and contrast it with the MIMD message-passing style of the IBM SP2 or Blue Gene.

5. Explain how modern GPU-based supercomputers combine data-parallel (SIMD) and message-passing (MIMD) principles within a single system.

6. Recap the basic organisation of a data-flow computer, and explain how the MIT Static Dataflow Machine and the Manchester Dataflow Machine each realised this organisation differently.

SGOU

7. Explain the design of the Monsoon (MIT Tagged-Token Dataflow Architecture) machine, and discuss how its explicit-token-store approach balances static and dynamic data-flow trade-offs.

8. Compare the SIGMA-1 and LAU System data-flow projects, highlighting the distinct design alternative each explored.

9. Construct a table comparing at least four of the case-study systems covered in this unit on the basis of memory organisation, scalability, and intended workload.

10. Discuss the lasting influence of dedicated data-flow computer research on modern processor design and software frameworks, even though dedicated data-flow hardware did not achieve commercial dominance.

11. Choose the most suitable architecture, justifying each choice with the help of Table 5.4.5, for: (a) a bank's transaction database; (b) a 10,000-core ocean-circulation simulation; (c) real-time filtering of 100 video streams.

12. Draw the dependency graph and an out-of-order schedule (two functional units, one-cycle operations) for I1: r1 = a + b; I2: r2 = c × d; I3: r3 = r1 − r2; I4: r4 = e + f; I5: r5 = r3 × r4. State the number of cycles needed.

13. Compare IBM Blue Gene and a Beowulf cluster in terms of interconnect, scale, cost and the kinds of applications each suits.

14. Analyse how coarse-grained data-flow (as in the LAU System) addresses the token-traffic problem of fine-grained machines described in Unit 2, and relate your answer to the task-level scheduling of Spark in Unit 1.

15. A research group must choose between one 4-socket shared-memory server and a 16-node cluster with the same total number of cores, for a program that is 95% parallel but exchanges large arrays among all processes at every step. Analyse the trade-offs using Amdahl's Law and the idea of communication overhead, and recommend one.

Reference

1. Gurd, J. R., Kirkham, C. C., & Watson, I. (1985). The Manchester prototype dataflow computer. *Communications of the ACM.*

2. Papadopoulos, G. M., & Culler, D. E. (1990). Monsoon: An explicit token-store architecture. *ACM SIGARCH Computer Architecture News.*

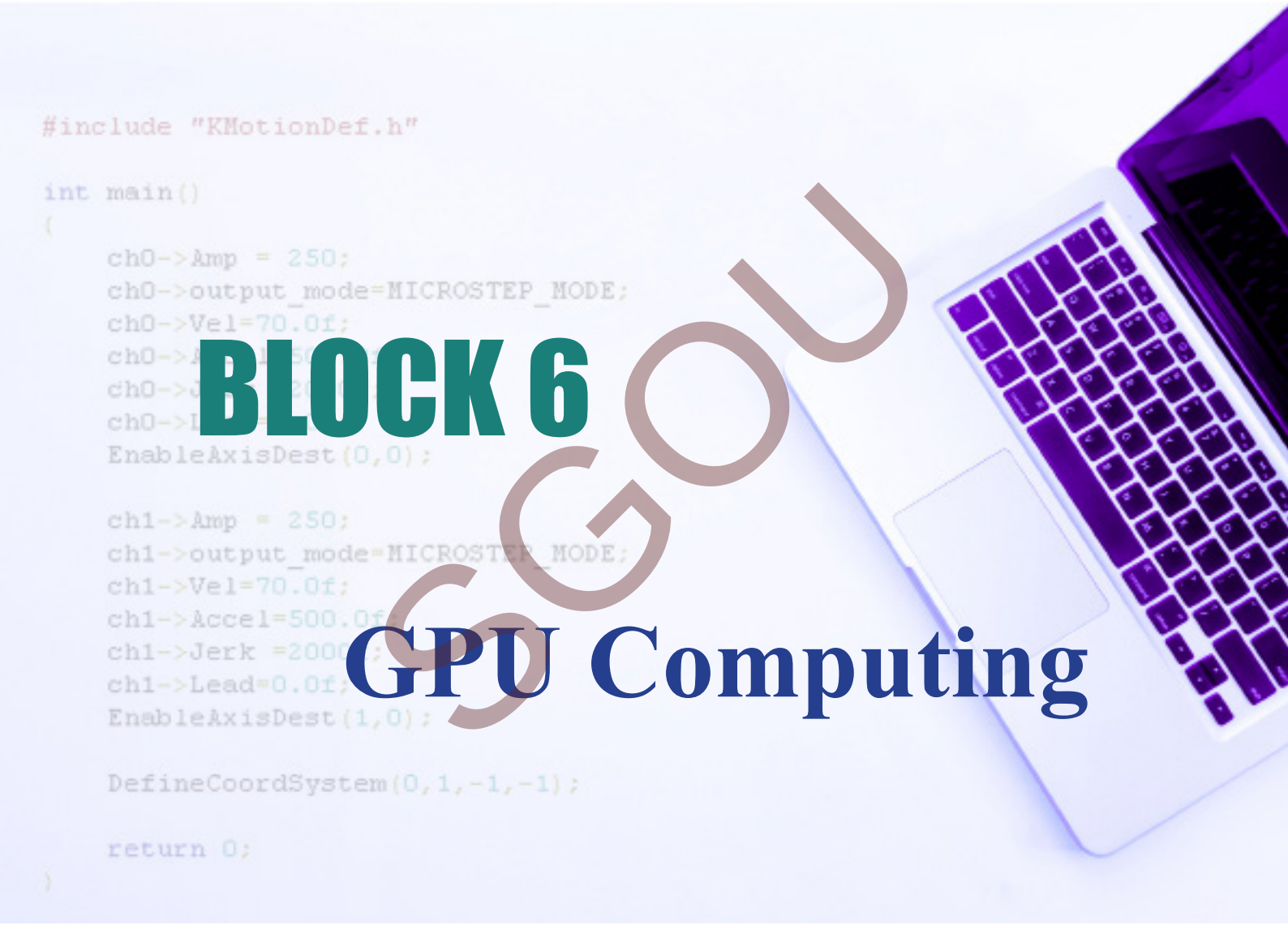
3. Sterling, T. L., et al. (2003). *Beowulf cluster computing with Linux* (2nd ed.). MIT Press.

Suggested Reading

1. Hwang, K., & Jotwani, N. (2016). *Advanced computer architecture: Parallelism, scalability, programmability* (3rd ed.). McGraw-Hill Education.

2. Hennessy, J. L., & Patterson, D. A. (2017). *Computer architecture: A quantitative approach* (6th ed.). Morgan Kaufmann.

SGOU



BLOCK 6

GPU Computing

SGOU



Heterogeneous Computing and Modern GPU Architectures

Learning Outcomes

After completion of this unit, learner will be able to:

- define heterogeneous parallel computing and explain why CPU+GPU systems are effective
- describe the architecture of a modern GPU, identifying streaming multiprocessors, cores, and memory hierarchy
- contrast CPU and GPU design philosophies (latency-oriented vs. throughput-oriented)
- explain the SIMT execution model and its relationship to SIMD
- analyze the components of the GPU memory hierarchy and their performance characteristics

Prerequisite

Traditional computing primarily relied on the Central Processing Unit (CPU), which is designed to minimize latency by executing individual threads as quickly as possible. CPUs are well suited for tasks involving complex control flow, sequential execution, and irregular operations. In contrast, Graphics Processing Units (GPUs) are designed to maximize throughput by executing thousands of relatively simple threads simultaneously, making them highly efficient for data-parallel computations. Heterogeneous computing combines the strengths of both processors by assigning serial and control-intensive tasks to the CPU while offloading computationally intensive, parallel tasks to the GPU.

This concept can be illustrated using a transportation analogy. A CPU can be compared to a Formula 1 race car, capable of completing a single delivery extremely quickly, whereas a GPU resembles a fleet of 1,000 scooters, each handling a simple task but collectively delivering a large number of packages efficiently. A heterogeneous computing system functions like a logistics company that uses race cars for urgent, individual deliveries and scooters for bulk deliveries. Similarly, by assigning each type of computation to the processor best suited for it, heterogeneous computing improves overall computational efficiency and performance.

SGOU

Keywords

Streaming Multiprocessor (SM), CUDA Core, SIMT (Single Instruction, Multiple Threads), Warp, Thread Block, Grid, Occupancy

Discussion

6.1.1 Heterogeneous Parallel Computing

In image processing, the Central Processing Unit (CPU) manages program execution and coordinates data flow, while the Graphics Processing Unit (GPU) processes multiple image pixels simultaneously to accelerate computation. This illustrates how different processing units can collaborate to solve a computational problem efficiently. Heterogeneous parallel computing is widely used in scientific simulations, artificial intelligence, and other applications that require high computational performance.

Heterogeneous Parallel Computing is a parallel computing paradigm that utilizes multiple types of processors, each optimized for specific kinds of computational tasks, to work together on a single problem. The most common heterogeneous system combines one or more CPUs with one or more GPUs, enabling workloads to be distributed according to the capabilities of each processor, thereby improving execution speed, resource utilization, and overall system efficiency.

Why Heterogeneous?

No single processor architecture is optimal for all types of computational workloads. Different processors are designed to meet different performance requirements. CPUs are optimized for low-latency execution, whereas GPUs are designed to maximize computational throughput. Combining these architectures allows a computing system to handle diverse tasks efficiently by utilizing the strengths of each processor.

1. **Central Processing Units (CPUs) – Latency-Oriented**

   - Consist of a relatively small number of powerful cores, typically 4–32 in many systems.
   - Optimized to execute individual threads as quickly as possible, minimizing execution latency.
   - Employ large caches, sophisticated branch prediction, and out-of-order execution to improve performance.
   - Well suited for complex, irregular, and control-intensive tasks, such as operating systems, compilers, and transaction processing.

SGOU

2. Graphics Processing Units (GPUs) – Throughput-Oriented

- Contain thousands of relatively simple processing cores designed for parallel execution.
- Optimized to execute numerous threads simultaneously, maximizing computational throughput.
- Typically use simpler control mechanisms and rely on massive parallelism to hide memory latency, although cache sizes and execution features vary across GPU architectures.
- Well suited for regular, data-parallel workloads, such as graphics rendering, matrix operations, and deep learning.

Thus, heterogeneous computing combines the low-latency capabilities of CPUs with the high-throughput capabilities of GPUs, enabling efficient execution of a wide range of computational tasks.

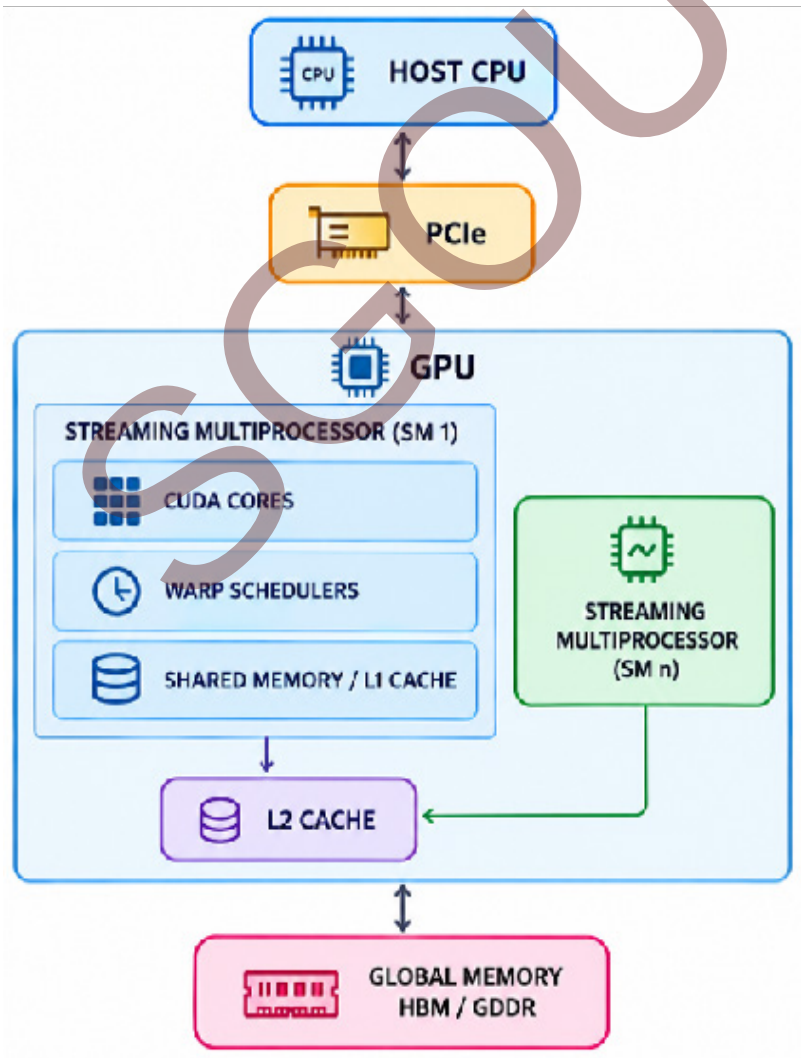


Fig. 6.1.1 Simplified architecture of a modern GPU showing streaming multiprocessors and memory hierarchy

SGOU

The 6.1.1 illustrates the architecture of a GPU-based parallel computing system and its interaction with the Host Central Processing Unit (CPU). The Host CPU communicates with the GPU through the Peripheral Component Interconnect Express (PCIe) interface, which enables the transfer of data, instructions, and computational results between them. The GPU consists of multiple Streaming Multiprocessors (SMs), each responsible for executing parallel workloads. Within an SM, CUDA cores perform arithmetic and logical operations, warp schedulers coordinate the execution of groups of threads called warps, and shared memory/L1 cache provides fast access to frequently used data for threads within the multiprocessor.

The GPU also includes an L2 cache, which is shared across the GPU's Streaming Multiprocessors and helps reduce access to slower global memory. Global memory, typically implemented using High Bandwidth Memory (HBM) or Graphics Double Data Rate (GDDR) memory, stores data and instructions required by GPU kernels during execution. When a program runs, data is generally transferred from the host system to GPU memory, processed in parallel by the GPU's SMs, and the results are returned to the CPU when required. This hierarchical organization of processing units and memory enables efficient data access, large-scale parallel execution, and high computational throughput.

6.1.1.1 The Heterogeneous Programming Model

The heterogeneous programming model divides computational tasks between the CPU and GPU according to their capabilities. The CPU manages program execution, input/output (I/O), and control flow, while the GPU performs computationally intensive, data-parallel operations using thousands of threads.

Steps involved in the programming model

1. **CPU executes the main program :** The CPU initializes the application, handles I/O operations, and manages the program's control flow.

2. **Identify parallel tasks :** The CPU identifies operations that can be performed simultaneously, such as multiplying two arrays containing one million elements.

3. **Transfer data to GPU memory :** The required input data is copied from the CPU's memory to the GPU's memory.

4. **Launch the GPU kernel :** The CPU launches a kernel, a function executed on the GPU, with a configuration specifying the GPU threads.

5. **Execute tasks in parallel :** The GPU executes the kernel across many threads, allowing multiple data elements to be processed concurrently.

6. **Transfer results back :** After GPU execution completes, the output data is copied from GPU memory to CPU memory when required.

SGOU

7. **Continue CPU execution :** The CPU uses the computed results for further processing or to produce the final output.

Amdahl's Law in Heterogeneous Computing

Amdahl's Law is a principle used to calculate the maximum overall speedup of a program when only a portion of its execution can be accelerated, such as by using a GPU while the remaining serial portion executes on the CPU.

In heterogeneous computing, Amdahl's Law helps determine how much performance improvement can be achieved by combining the CPU and GPU. Even if the GPU executes parallel tasks much faster, the serial portion of the program limits the overall speedup.

Formula:

$$S = \frac{1}{f + \frac{1-f}{P}}$$

where,

- S is the overall program speedup.
- f is the fraction of execution time spent on serial tasks.
- (1-f) is the fraction of execution time spent on parallel tasks.
- P is the speedup of the parallel portion using the GPU.

Solved Example 1: A program spends 80% of its time in a data-parallel section that can be accelerated 20× on a GPU, and 20% in serial code that cannot be accelerated. Using Amdahl's Law, calculate the maximum overall speedup.

Solution:

Step 1: Identify the given values

- Serial fraction, f = 0.20 (the 20% that cannot be accelerated).
- Parallel fraction, (1-f) = 0.80
- GPU acceleration factor, P = 20

Step 2: Apply Amdahl's Law

The overall speedup is calculated using the formula:

SGOU

$$S = \frac{1}{f + \frac{1-f}{P}}$$

Step 3: Substitute the values

$$S = \frac{1}{0.20 + \frac{0.80}{20}} = \frac{1}{0.24} = \boxed{4.17\times}$$

Result: The maximum overall speedup is approximately 4.17×. Although the GPU accelerates the parallel portion by 20 times, the serial portion remains unchanged and limits the overall performance improvement. This demonstrates the importance of minimizing serial execution in heterogeneous computing systems.

6.1.2 Architecture of a Modern GPU

Modern GPUs (such as NVIDIA Ampere and Hopper architectures) are designed to achieve high computational throughput by executing thousands of threads concurrently. Their architecture follows a hierarchical organization centered around Streaming Multiprocessors (SMs), which contain multiple processing cores and specialized hardware units. These components work together to execute parallel workloads efficiently in applications such as scientific computing, graphics rendering, and deep learning.

As shown in fig 6.1.2, the GPU contains multiple SMs connected to the memory hierarchy. The terminology used in this section, such as CUDA cores, SMs, and warps, follows NVIDIA's GPU architecture and CUDA programming model; other GPU vendors may use different architectural terminology.

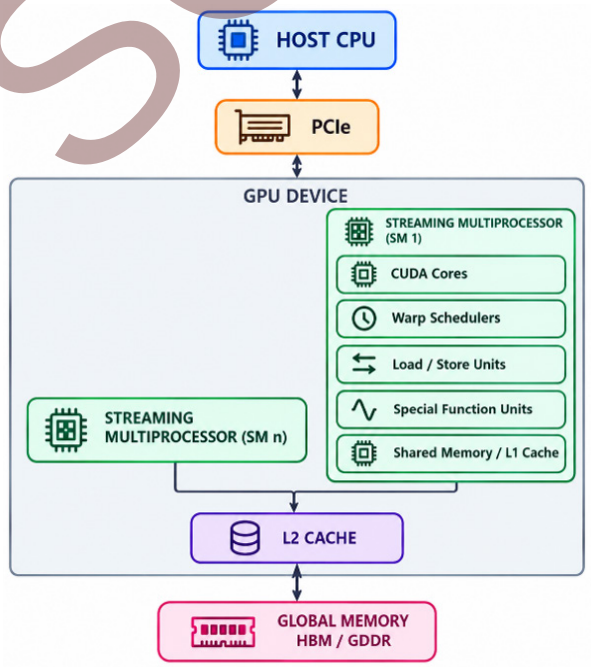


Fig. 6.1.2 Simplified architecture of a modern NVIDIA GPU

SGOU

6.1.2.1 Key Components of a Modern GPU

A modern GPU consists of several interconnected components that work together to execute thousands of threads in parallel and efficiently manage data processing and memory access.

1. Streaming Multiprocessor (SM)

The Streaming Multiprocessor (SM) is the fundamental computational unit of a GPU responsible for executing groups of threads in parallel. Each SM contains the following components:

- **CUDA cores:** Perform integer and floating-point arithmetic operations.
- **Special Function Units (SFUs):** Perform specialized mathematical operations, such as sine, cosine, and exponential functions.
- **Load/Store units:** Handle data transfers between registers and memory.
- **Warp schedulers and dispatch units:** Schedule warps and issue instructions for execution.
- **Register file:** Provides fast, thread-local storage using thousands of 32-bit registers, depending on the architecture.
- **Shared memory/L1 cache:** Provides fast on-chip data storage for threads executing on an SM.
- **Instruction cache:** Stores recently used instructions for efficient instruction fetching.

2. CUDA Core

A CUDA core is an individual scalar arithmetic execution unit within an NVIDIA SM that executes integer or floating-point instructions. Unlike a CPU core, it generally does not contain sophisticated control mechanisms such as complex branch prediction and out-of-order execution. Its relatively simple design allows many CUDA cores to be integrated into a GPU, enabling large-scale parallel computation. The supported operations and execution throughput depend on the GPU architecture and instruction type.

3. Warp and SIMT

SIMT (Single Instruction, Multiple Threads) is the execution model used by NVIDIA GPUs, in which threads are organized into groups called warps. A warp typically consists of 32 threads that execute the same instruction on different data elements. This organization enables efficient parallel execution across multiple threads. When threads within a warp follow different control-flow paths, the GPU may execute those paths separately, reducing execution efficiency.

SGOU

SIMT is similar to SIMD (Single Instruction, Multiple Data), but SIMT organizes execution around individual threads, whereas SIMD operates on vector elements or lanes.

4. Memory Hierarchy

The GPU memory hierarchy consists of several levels (Table 6.1.1) designed to balance access speed, storage capacity, and data sharing. Registers provide fast, thread-local storage, while shared memory and L1 cache provide rapid data access within an SM. The L2 cache is shared across the GPU, and global memory stores large amounts of data accessible to GPU threads.

Table 6.1.1 Illustrative GPU Memory Hierarchy

Memory level	Location	Scope
Registers	Register file within an SM	Per-thread
Shared memory	On-chip memory within an SM	Shared among threads in a block
L1 cache	Within an SM	Per-SM
L2 cache	Shared GPU cache	Device-wide
Global memory	GPU device memory (HBM or GDDR)	Accessible across the device

Memory sizes, cache configurations, and access latencies vary across GPU generations and models. Shared memory and L1 cache may also share physical resources, depending on the architecture.

Solved Example 2: An SM can support 2,048 threads (64 warps). If a kernel uses 1,024 threads per block, how many blocks can run concurrently on one SM? What is the occupancy if each thread uses 64 registers and the SM has 65,536 registers?

Solution:

Step 1 : Blocks per SM based on thread limit

$$\frac{2048 \text{ threads}}{1024 \text{ threads/block}} = 2 \text{ blocks}$$

Therefore, the thread limit allows 2 blocks per SM.

Step 2 : Register usage per block

1024 threads X 64 registers/thread = 65,536 registers

Since the SM has 65,536 registers in total, it can accommodate only 1 block at a time. (The SM has 65,536 registers total, so 2 blocks would need 131,072 registers i.e., insufficient.)

SGOU

Step 3 : Actual concurrent blocks per SM

The number of concurrent blocks is limited by the most restrictive resource (in this case, registers).

65,536 / 65,536 = 1 block.

Step 4 : Calculate occupancy

Occupancy is the ratio of active threads to the maximum number of threads supported by the SM.

$$\text{Occupancy} = \frac{\text{Active threads}}{\text{Maximum supported threads}} \times 100\%$$

$$= \frac{1024}{2048} \; X \; 100$$

$$= 50\ \%$$

Occupancy can be limited by register availability, not just by the maximum thread count. Reducing register usage per thread may allow more blocks to run concurrently, increasing occupancy and potentially improving latency hiding. However, higher occupancy does not always guarantee better performance.

6.1.3 CPU vs. GPU: Architectural Comparison

CPUs and GPUs are designed with different architectural goals to handle computational tasks efficiently. While CPUs focus on executing individual tasks quickly using a few powerful cores, GPUs use many parallel execution units to process large amounts of data simultaneously. Understanding these differences (Table 6.1.2) helps identify the most suitable processor for a given workload and explains why modern heterogeneous computing systems combine CPUs and GPUs to achieve high performance.

Table 6.1.2 CPU vs. GPU

Aspect	CPU	GPU
Cores	Few, powerful and complex cores	Many parallel execution units
Clock Frequency	Typically higher (around 3–5 GHz)	Typically lower (around 1–2 GHz)
Cache	Often has large L1, L2 and L3 caches	Uses L1/shared memory and L2 cache, alongside high-bandwidth global memory
Threads	Supports multiple threads, usually on the order of tens to hundreds per processor	Can manage thousands of active threads, depending on architecture and workload

SGOU

Control Flow	Sophisticated branch prediction and out-of-order execution	Designed to execute large numbers of threads efficiently; divergent branches can reduce performance
Latency Hiding	Uses caches, branch prediction and out-of-order execution	Uses many active warps to switch to ready work when other warps stall
Optimization Goal	Minimize latency	Maximize throughput
Best-Suited Workloads	Sequential, irregular and control-intensive tasks	Regular, data-parallel and computationally intensive tasks

This comparison is illustrative rather than universal. Actual core counts, clock frequencies, cache sizes, and supported threads vary substantially across CPU and GPU architectures and product generations.

CPUs emphasize fast execution of individual tasks, while GPUs emphasize high throughput by executing many operations in parallel. GPUs also use caches and memory hierarchy to reduce memory delays, but a major advantage is their ability to hide latency by scheduling other ready warps.

From Graphics to General-Purpose GPUs

GPUs evolved from fixed-function graphics accelerators in the 1990s into programmable processors capable of general-purpose computing. NVIDIA's GeForce 8800 (2006) and the introduction of CUDA in 2007 helped make GPU programming more accessible to C developers. Before CUDA, GPGPU programming often required expressing computations through graphics concepts such as textures, shaders, and vertices. CUDA enabled developers to write parallel programs more naturally, contributing to the growth of GPU computing in scientific simulations, data processing, and deep learning.

The Memory Bandwidth Wall

GPU computational performance (FLOPS) has grown rapidly, but memory bandwidth has not always increased at the same rate. This creates a memory wall, where the GPU's processing units may wait for data to arrive from memory. Performance therefore depends not only on computational power but also on efficient memory access. Techniques such as using shared memory, ensuring coalesced global-memory access, and reducing unnecessary data transfers help improve performance. High Bandwidth Memory (HBM) uses vertically stacked memory dies and a wide memory interface to deliver high bandwidth in a compact package.

SGOU

Recap

- Heterogeneous computing combines latency-oriented CPUs with throughput-oriented GPUs to achieve efficient overall performance.
- CPUs have a few powerful cores, large caches, and complex control logic, whereas GPUs contain many parallel execution units optimized for data-parallel workloads.
- A GPU is organized into Streaming Multiprocessors (SMs), each containing CUDA cores, special function units, warp schedulers, registers, and shared memory.
- In NVIDIA CUDA, threads are grouped into warps of typically 32 threads that execute instructions according to the Single Instruction, Multiple Threads (SIMT) model.
- The GPU memory hierarchy consists of registers, shared memory, L1/L2 caches, and global memory, with registers generally offering the fastest access and global memory providing the largest capacity.
- GPUs hide memory latency primarily through thread-level parallelism by switching among ready warps, while caches and other memory mechanisms help reduce effective memory-access delays.
- Occupancy is the ratio of active warps to the maximum number of active warps supported by an SM, and it is limited by resources such as registers, shared memory, and thread capacity.
- Amdahl's Law explains that the serial portion of a program limits its overall speedup, regardless of how much the parallel portion is accelerated by a GPU.
- High Bandwidth Memory (HBM) provides very high memory bandwidth, helping supply data to GPU processing units and reduce memory-bandwidth bottlenecks.

SGOU

Objective Type Questions

1. What is the term for using different types of processors (CPU + GPU) together for optimal performance?
2. What is the fundamental building block of a GPU, containing many CUDA cores and shared memory?
3. In NVIDIA CUDA's SIMT model, how many threads are grouped into a warp?

4. What is the primary difference in design philosophy between a CPU and a GPU?

5. Which level of the GPU memory hierarchy is fastest but smallest and per-thread?

6. What memory level allows threads within the same block to cooperate?

7. How does a GPU primarily hide memory latency?

8. What is the ratio of active warps to the maximum warps an SM can support called?

9. What memory technology stacks memory dies directly on the GPU package to increase bandwidth?

10. Which NVIDIA platform, introduced in 2006, democratized GPU computing for C programmers?

Answers

1. Heterogeneous computing.

2. Streaming Multiprocessor (SM).

3. 32 threads.

4. CPU is latency-oriented (few powerful cores); GPU is throughput-oriented (many simple cores).

5. Registers.

6. Shared memory.

7. By switching between warps (thread-level parallelism).

8. Occupancy.

9. High Bandwidth Memory (HBM).

10. CUDA.

SGOU

Assignments

1. Compare NVIDIA Volta and Hopper architectures with respect to SM organization, memory hierarchy, and specialized hardware.

2. Explain control-flow divergence within a warp and discuss methods for reducing its performance impact.

3. Compare CPU and GPU architectures with respect to cores, control flow, memory hierarchy, latency, and throughput.

4. A program has 15% serial execution and the remaining 85% can be accelerated 30× on a GPU. Calculate the maximum overall speedup.

5. An SM supports 64 warps (2,048 threads) and has 96 KB of shared memory. A kernel uses 512 threads per block and 24 KB of shared memory per block. Calculate the maximum number of blocks per SM and the resulting occupancy.

Reference

1. Hennessy, J. L., Patterson, D. A., & Kozyrakis, C. (2025). *Computer architecture: a quantitative approach*. Morgan kaufmann.

2. Kirk, D. B., & Hwu, W. W. (2016). *Programming Massively Parallel Processors: A Hands-on Approach* (3rd ed.). Morgan Kaufmann.

3. Nickolls, J., & Dally, W. J. (2010). The GPU computing era. *IEEE Micro, 30*(2), 56–69.

Suggested Reading

1. Sanders, J., & Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley.

2. NVIDIA. *CUDA C++ Programming Guide*.

3. NVIDIA. *CUDA C Best Practices Guide*.

SGOU



Fundamentals of GPU Computing

Learning Outcomes

After completion of this unit, learner will be able to:

- explain how GPU architectures scale to provide increasing computational power
- describe recent developments in GPU technology, including Tensor Cores and specialized units
- analyze trends in GPU computing: exascale, AI acceleration, and energy efficiency
- evaluate how GPU scalability enables large-scale scientific and AI applications
- identify key vendors and their architectural approaches

Prerequisite

Building on the GPU architecture fundamentals from Unit 1, this unit examines how GPUs have evolved and scaled over time, and where the technology is heading. Understanding scalability requires appreciating how adding more SMs, more memory bandwidth, and specialized units translates to higher performance for parallel workloads.

Imagine a factory that produces widgets, where production can be increased either by making each worker faster, similar to increasing the clock speed, or by hiring more workers, similar to adding more processor cores. GPUs primarily follow the second approach by increasing the number of cores to perform many operations simultaneously. However, adding more cores alone does not guarantee better performance, as the workers must receive materials quickly and be coordinated efficiently. Similarly, GPU scalability depends on improving three key factors together: computational power, memory bandwidth, and scheduling efficiency.

SGOU

Keywords

Tensor Core, Ray Tracing Core, HBM (High Bandwidth Memory, NVLink, Exascale Computing, Energy Efficiency (FLOPS/Watt)

Discussion

6.2.1 Scalable GPUs

GPU scalability is the ability to increase computational throughput by adding more processing resources. Unlike CPUs, which have hit diminishing returns from increasing clock speed (the "power wall"), GPUs scale primarily by adding more cores and more memory bandwidth.

How GPUs Scale?

GPUs achieve higher performance by increasing the number of processing cores and efficiently coordinating computation, memory bandwidth, and thread scheduling. Some key factors (Fig 6.2.1) are:

1. **More SMs:** Adding more Streaming Multiprocessors increases the number of warps that can execute concurrently. The NVIDIA A100 has 108 SMs; the H100 has 132 SMs.

2. **More Cores per SM:** Newer GPU architectures provide more execution resources and specialized functional units within each SM, increasing the amount of work that can be processed concurrently.

3. **Wider Memory Bus and HBM:** Memory bandwidth scales with the number of memory channels and the use of stacked HBM. The A100 has ~1.5 TB/s bandwidth; the H100 has ~3.3 TB/s.

4. **Specialized Units:** Tensor Cores (for AI matrix operations) and ray tracing cores (for graphics) add capabilities without consuming general-purpose core resources.

5. **Multi-GPU Interconnect (NVLink):** Connecting multiple GPUs with high-speed links (NVLink, Infinity Fabric) scales beyond a single device.

SGOU

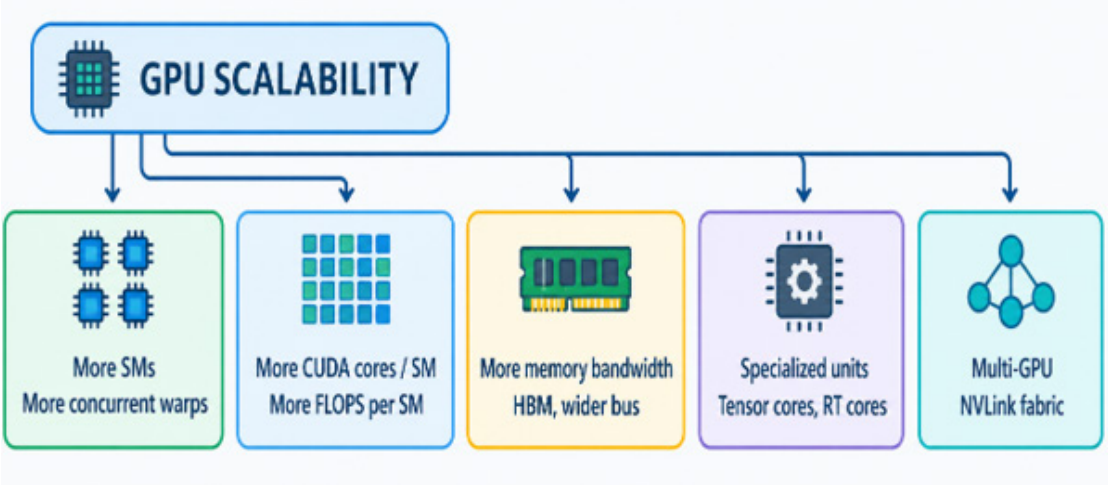


Fig. 6.2.1 Key factors affecting GPU scalability

Solved Example 1: A GPU generation has 80 SMs, each with 128 CUDA cores running at 1.5 GHz, assume, for simplicity, that each core performs one floating-point operation per cycle. Calculate the peak single-precision throughput (in TFLOPS). If the next generation doubles SMs to 160 and increases clock to 1.8 GHz, what is the new peak?

Solution:

Step 1: Calculate the initial GPU's peak throughput

Given:

- Number of SMs = 80
- CUDA cores per SM = 128
- Clock speed = 1.5 GHz
- Operations per core per cycle = 1 floating-point operation (FLOP)

Formula: Peak FLOPS = SMs × cores/SM × clock × FLOPS/core/cycle.

Substituting the values,

= 80 X 128 X 1.5 X 109 X 1

=15,360 X 109 FLOPS

Since 1 TFLOPS=1012 FLOPS,

=15.36 TFLOPS.

Step 2: Calculate the next generation's peak throughput

The next generation doubles the number of SMs to 160 and increases the clock speed to 1.8 GHz.

SGOU

Given:

- Number of SMs = 160
- CUDA cores per SM = 128
- Clock speed = 1.8 GHz
- Operations per core per cycle = 1 FLOP

Substituting into the formula,

Peak FLOPS = 160 X 128 X 1.8 X 10^9 X 1

= 36,386 X 10^9 FLOPS

= 36.384 TFLOPS.

Step 3: Compare the two generations

Parameter	Initial GPU	Next-generation GPU
Number of SMs	80	160
CUDA cores per SM	128	128
Total CUDA cores	10,240	20,480
Clock speed	1.5 GHz	1.8 GHz
Peak throughput	15.36 TFLOPS	36.864 TFLOPS

Scaling factor: 36.86 / 15.36 = 2.4× (from doubling SMs and increasing clock 20%).

i.e., The peak throughput increases from 15.36 TFLOPS to 36.864 TFLOPS, representing a 2.4× increase, or a 140% improvement.

This calculation assumes one FLOP per core per cycle. Actual GPU peak throughput may differ because of instruction capabilities, hardware design, and the number of floating-point operations supported per cycle.

6.2.2 Recent Developments and Trends

Recent developments in GPU technology (Table 6.2.1) focus on improving computational performance, energy efficiency, memory bandwidth, and scalability to meet the growing demands of artificial intelligence, scientific computing, and large-scale parallel applications. Some of them are:

1. Tensor Cores and AI Acceleration

Tensor Cores are specialized execution units designed to accelerate matrix multiply-accumulate operations (Fig 6.2.2), which are widely used in neural networks. NVIDIA introduced Tensor Cores with its Volta architecture in 2017. The supported matrix dimensions, data types, and throughput vary across GPU architectures.

SGOU



Fig. 6.2.2 Matrix multiply-accumulate operation performed by a Tensor Core

The following examples use representative NVIDIA GPU configurations; specifications vary by product variant and configuration.

- **Volta V100:** 640 Tensor Cores, 125 TFLOPS (tensor).
- **A100:** 432 Tensor Cores, 312 TFLOPS (tensor, FP16).
- **H100:** 528 Tensor Cores, ~1,979 TFLOPS (tensor, FP8 sparse).

Tensor Cores have become an important contributor to AI training and inference performance by accelerating matrix operations used extensively in neural networks.

2. Ray Tracing Cores

Introduced with NVIDIA's Turing architecture in 2018, Ray Tracing (RT) cores are specialized hardware units designed to accelerate ray–triangle intersection tests, a fundamental operation in physically based rendering. By speeding up these calculations, RT cores enable real-time ray tracing in games, producing more realistic lighting, reflections, and shadows while reducing the computational workload on general-purpose GPU cores.

3. High Bandwidth Memory (HBM)

HBM stacks multiple memory dies vertically and connects them with through-silicon vias (TSVs) and a wide interface, providing much higher bandwidth than traditional GDDR. HBM3, used in H100-class configurations, provides very high memory bandwidth; for example, the H100 SXM configuration provides about 3.35 TB/s.

4. Chiplet Design

As monolithic dies approach physical and economic limits, GPU manufacturers are increasingly adopting chiplet-based designs, which combine multiple smaller dies through high-speed interconnects to function as a single processor. Architectures such as AMD's Instinct MI250X and Intel's Ponte Vecchio use this approach to improve

SGOU

manufacturing yield, increase design flexibility, and enable scalability beyond the size limits imposed by semiconductor manufacturing reticles.

5. Multi-GPU Interconnect

NVLink (NVIDIA) and Infinity Fabric (AMD) connect multiple GPUs with high bandwidth and low latency, enabling multi-GPU systems for large-scale training. H100 systems use NVLink 4.0 to provide high-bandwidth GPU-to-GPU communication; the aggregate bandwidth depends on the GPU configuration.

6. Exascale Computing

GPUs power most of the world's fastest supercomputers. Examples of GPU- and accelerator-based exascale systems include Frontier, Aurora, and El Capitan. Their reported performance varies with benchmark and system configuration.

Frontier became the first publicly reported exascale supercomputer on the TOP500 HPL benchmark in 2022. Later systems such as Aurora and El Capitan extended GPU/accelerator-based exascale computing. Reported performance values depend on the benchmark and the date of the TOP500 release.

GPU-centric exascale systems enable unprecedented scientific simulations (climate, molecular dynamics, cosmology).

7. Energy Efficiency

As FLOPS grow, energy efficiency (FLOPS/Watt) becomes critical. Modern GPUs use:

- Lower-precision formats such as FP16, BF16, FP8, and INT8 can improve energy efficiency and throughput for suitable AI workloads, although the actual benefit depends on the hardware and workload.
- Dynamic voltage/frequency scaling.
- Power gating of idle units.

8. Software Ecosystem Maturity

CUDA is a widely adopted GPU-computing platform, particularly within NVIDIA's ecosystem, offering extensive libraries, development tools, and support for parallel applications. Other programming frameworks and APIs, such as OpenCL, HIP, SYCL, OpenMP offloading, Vulkan Compute, and Metal, provide alternative approaches to heterogeneous computing across different hardware platforms.

- **OpenCL:** A vendor-neutral framework for programming heterogeneous systems, supporting CPUs, GPUs, and other processors.
- **HIP (AMD):** AMD's C++-based GPU programming environment, designed to facilitate CUDA-like development and portability across supported GPU platforms.

SGOU

- **SYCL:** An open-standard, C++-based programming model that supports single-source development for heterogeneous computing devices.
- **OpenMP Offloading:** Extends OpenMP to enable developers to offload computationally intensive tasks from the CPU to supported accelerators, including GPUs.
- **Vulkan Compute:** A cross-platform API that enables general-purpose GPU computing through compute shaders.
- **Metal (Apple):** Apple's graphics and compute API, designed to provide efficient access to GPU capabilities on Apple platforms.

The maturity of these software ecosystems influences GPU adoption by affecting application portability, development complexity, performance optimization, and compatibility with existing tools and libraries.

Table 6.2.1 Summary of Recent GPU Trends

Trend	Description	Impact
Tensor Cores	Specialized matrix units	AI acceleration
RT Cores	Ray-triangle intersection	Real-time ray tracing
HBM	High-bandwidth stacked memory	Higher memory bandwidth
Chiplet	Multi-die packaging	Scaling and manufacturing flexibility
Multi-GPU	NVLink and other high-speed fabrics	Large-scale GPU computing
Low Precision	FP16/BF16/FP8/INT8	Higher throughput and energy efficiency
Exascale	10^{18}FLOPS systems	Large-scale scientific computing

These figures are architecture-specific peak tensor throughput values reported for different precisions and, in some cases, sparsity. They should not be interpreted as directly comparable general-purpose GPU FLOPS.

6.2.3 Multi-GPU and GPU Cluster Scaling

A single GPU can provide very high computational throughput, but many modern applications require more computation and memory capacity than one GPU can provide. Multi-GPU computing addresses this limitation by connecting several GPUs within a system or across a computing cluster. The workload is divided among the GPUs so that different parts of the computation can execute concurrently.

For example, consider the training of a large neural network. Instead of performing the entire computation on one GPU, a system with four GPUs can divide the training

SGOU

workload among them. Each GPU processes a portion of the data or model, and the GPUs exchange information when synchronization is required.

The basic idea is illustrated below:

CPU → Interconnect → GPU 1, GPU 2, GPU 3, GPU 4

Each GPU contains its own processing units and memory. Therefore, adding GPUs can increase both computational capacity and available aggregate memory. However, the GPUs must communicate with one another, and this communication introduces additional overhead.

6.2.3.1 Types of Multi-GPU Parallelism

Multi-GPU systems commonly use several forms of parallelism.

1. Data Parallelism

In data parallelism, each GPU receives a different portion of the input data while using the same computational model. For example, a training dataset can be divided into four batches and processed simultaneously by four GPUs.

After processing their respective batches, the GPUs may exchange gradients or other intermediate information before the next iteration.

2. Model Parallelism

In model parallelism, different parts of a large model are placed on different GPUs. This approach is useful when the complete model cannot fit into the memory of a single GPU.

For example, the first part of a neural network can be stored on GPU 1, while later layers are stored on GPU 2. Intermediate results must then be transferred between the GPUs.

3. Pipeline Parallelism

Pipeline parallelism divides a computation into stages. Different GPUs execute different stages simultaneously on different portions of the input. While one GPU processes one portion of data, another GPU can process an earlier or later portion.

This approach can increase hardware utilization, but it requires careful scheduling to reduce idle periods between stages.

Communication Overhead

Adding more GPUs does not always produce proportional performance improvement. When GPUs need to exchange large amounts of data, communication can become a significant part of the total execution time.

An efficient multi-GPU application should minimize unnecessary communication and synchronization.

SGOU

6.2.4 GPU Scaling Limitations

Although GPUs can scale by adding more processing resources, practical performance is limited by several factors. A GPU may contain thousands of arithmetic units, but these units must be supplied with data and useful instructions.

1. Memory Bandwidth Limitation

A computational unit may finish its arithmetic operation quickly but still have to wait for data from memory. If the application performs many memory accesses relative to the amount of computation, increasing the number of processing units may provide little additional benefit. This situation is commonly described as a memory bandwidth limitation.

2. Communication Overhead

In multi-GPU systems, data must sometimes move between GPUs. Communication consumes time and may prevent the GPUs from continuously performing useful computation. Applications with frequent communication may therefore scale poorly when additional GPUs are added.

3. Synchronization Overhead

Parallel programs often require synchronization to ensure that all participating processors have completed a particular stage before the next stage begins. If one GPU finishes early and must wait for another GPU, part of its computational capacity remains unused.

4. Load Imbalance

A parallel workload should be divided as evenly as possible. If one GPU receives significantly more work than the others, the remaining GPUs may become idle while waiting for it to finish.

For example, if four GPUs receive workloads taking 20, 20, 20, and 35 seconds respectively, the total execution time is determined largely by the GPU requiring 35 seconds.

5. Power and Thermal Constraints

Increasing the number of processing units and operating them at high utilization increases power consumption and heat generation. Large GPU systems therefore require efficient cooling and power delivery. Consequently, modern GPU design considers not only performance but also performance per watt.

6. Software Efficiency

Hardware resources can be underutilized if the software cannot expose sufficient parallelism. Poor memory access patterns, branch divergence, synchronization, and

SGOU

inefficient kernels can prevent a program from achieving the theoretical performance of the GPU.

Therefore, GPU scalability depends on both hardware resources and the ability of software to use those resources efficiently.

6.2.5 Specialized GPU Architectures

Modern GPUs combine general-purpose execution units with specialized hardware designed for frequently occurring operations.

- **Tensor Cores:** Accelerate matrix multiply-accumulate operations used in AI training and inference.
- **Ray Tracing Cores:** Accelerate ray-triangle intersection and related operations in graphics rendering.
- **Other Specialized Accelerators:** Hardware for video processing, signal processing, data movement, and AI inference can improve performance and energy efficiency for suitable workloads.

Why Specialization Matters

Specialized hardware can provide higher throughput and better energy efficiency for targeted workloads. However, its benefits depend on whether the application can use the operations supported by that hardware.

For example, a general-purpose execution unit is like a worker who performs many tasks, whereas a specialized accelerator is like a machine designed to perform one task efficiently. Modern GPUs combine both approaches to balance flexibility and performance.

6.2.6 Future Trends in GPU Computing

GPU development continues to focus on increasing computational throughput while reducing memory and communication bottlenecks. Several trends are particularly important.

1. Advanced Memory Technologies

As computational throughput increases, memory systems must provide sufficient data bandwidth. Future systems are expected to continue improving:

- memory bandwidth,
- memory capacity,
- energy efficiency,
- proximity between compute units and memory.

The objective is to reduce the gap between computation speed and data-delivery speed.

SGOU

2. 3D Packaging

Three-dimensional packaging allows different layers of semiconductor components to be placed vertically or closely together.

Compared with traditional two-dimensional layouts, 3D packaging can provide shorter connections and higher communication density. It can therefore help integrate compute, cache, and memory technologies within a compact package.

3. Energy-Efficient GPU Computing

As GPU systems grow, power consumption and cooling become important design constraints. Future systems will continue to emphasize performance per watt, efficient power management, specialized accelerators, and improved cooling and packaging technologies.

4. GPU-Accelerated Exascale and Beyond

Exascale systems demonstrate that massive parallelism can be used to perform at least 1018 floating-point operations per second under appropriate benchmark conditions.

Future supercomputers will continue combining large numbers of CPUs and GPUs or other accelerators. Applications such as climate modeling, molecular dynamics, astrophysics, artificial intelligence, and computational fluid dynamics can benefit from this computational capability.

However, achieving higher performance will require balancing computation, memory bandwidth, communication, synchronization, and energy consumption.

6.2.6.1 Emerging Heterogeneous and Accelerator Technologies

Modern computing systems increasingly combine CPUs, GPUs, and Neural Processing Units (NPUs). CPUs handle general-purpose and control-intensive tasks, GPUs accelerate data-parallel computations, and NPUs are designed to accelerate neural-network operations efficiently. Selecting the appropriate processor for each workload can improve performance and energy efficiency.

- **Neuromorphic Computing:** Neuromorphic systems use designs inspired by the structure and operation of biological neural networks. They are being explored for applications such as event-based sensing and low-power neural computation.
- **Processing-in-Memory (PIM):** Processing-in-memory aims to perform certain computations near or within memory, reducing data movement between the processor and memory. This may benefit workloads that are limited by memory bandwidth, although practical advantages depend on the technology and application.

SGOU

The Tensor Core Revolution

Before Tensor Cores, many neural-network computations on GPUs were performed using FP32 arithmetic on general-purpose CUDA cores.. Tensor Cores changed this by providing specialized hardware for high-throughput matrix operations, initially supporting FP16 computation and later additional data types such as BF16 and FP8. Tensor Cores can provide substantial speedups for suitable matrix-intensive AI workloads compared with execution using general-purpose arithmetic units alone. Published estimates place GPT-3 training compute at roughly 3.14×10^{23} floating-point operations (314 ZettaFLOPs). Tensor Cores and other GPU optimizations can substantially improve the efficiency of such matrix-intensive workloads

From Cryptomining to AI

GPUs have evolved beyond their original role in graphics processing to support general-purpose GPU computing (GPGPU), cryptocurrency mining, and artificial intelligence (AI) acceleration. Each transition has exploited their fundamental strength: massive data parallelism. The cryptocurrency-mining boom increased demand for GPUs, contributing to periods of limited availability and higher prices, while the rapid growth of AI workloads has significantly increased demand for data-center GPU computing. This adaptability demonstrates the enduring value of the GPU's throughput-oriented architecture across a wide range of computational applications.

SGOU

Recap

- GPU scalability comes from adding more SMs, more cores per SM, more memory bandwidth, specialized units, and multi-GPU interconnects.
- Peak FLOPS = SMs × cores/SM × clock × FLOPS/core/cycle.
- Tensor Cores accelerate matrix multiply-accumulate (AI workloads), achieving hundreds of TFLOPS.
- Ray tracing cores accelerate ray-triangle intersection for realistic graphics.
- HBM (stacked memory) uses a wide, high-bandwidth interface and can substantially increase memory bandwidth; the achievable bandwidth depends on the HBM generation and GPU implementation.
- Chiplet design builds GPUs from multiple smaller dies for better yield and scaling.
- Multi-GPU interconnects such as NVLink and Infinity Fabric support efficient communication in large-scale multi-GPU and accelerator-based systems.

- GPU- and accelerator-based exascale systems include Frontier, Aurora, and El Capitan.
- Energy efficiency can be improved through suitable low-precision arithmetic, specialized accelerators, and power-management techniques.
- Software ecosystem: CUDA is widely used in NVIDIA-based GPU computing, alongside alternatives such as OpenCL, HIP, SYCL, and others.
- GPUs have been repurposed from graphics to GPGPU to crypto to AI, leveraging the same data-parallel architecture.

Objective Type Questions

1. How do GPUs primarily scale computational performance?
2. What specialized GPU unit accelerates matrix multiply-accumulate for AI?
3. What does HBM stand for, and what problem does it address?
4. Which NVIDIA interconnect connects multiple GPUs with high bandwidth?
5. What is the term for computing systems capable of 10^18 operations per second?
6. What specialized unit accelerates ray-triangle intersection in graphics?
7. What is the peak FLOPS formula for a GPU?
8. Which lower-precision formats are used to improve AI energy efficiency?
9. What design approach builds a processor from multiple smaller dies?
10. Which supercomputer became the first system to achieve exascale performance on the TOP500 HPL benchmark in 2022?

SGOU

Answers

1. Adding more cores (and SMs).
2. Tensor core.
3. High Bandwidth Memory (addresses the memory bandwidth wall).
4. NVLink.

5. Exascale computing.
6. Ray tracing (RT) core.
7. SMs × cores/SM × clock × FLOPS/core/cycle.
8. FP16, BF16, FP8, INT8.
9. Chiplet design.
10. Frontier.

Assignments

1. A GPU has 132 SMs, 128 cores per SM, clock of 1.98 GHz, and performs 2 FLOPS per core per cycle. Calculate the peak throughput in TFLOPS. Explain why this "peak" is rarely achieved in practice.
2. Select one recent GPU development such as chiplets, 3D packaging, optical interconnects, or processing-in-memory and explain its potential impact.
3. Compare NVIDIA H100 and AMD Instinct MI250X in terms of architecture, matrix acceleration, memory, and interconnect.
4. Explain how adding SMs, increasing memory bandwidth, and using specialized units affect GPU performance.
5. Explain the role of NVLink or another GPU interconnect in multi-GPU computing and discuss communication bottlenecks.

SGOU

Reference

1. Hennessy, J. L., Patterson, D. A., & Kozyrakis, C. (2025). *Computer architecture: a quantitative approach*. Morgan kaufmann.
2. Kirk, D. B., & Hwu, W. W. (2016). *Programming Massively Parallel Processors: A Hands-on Approach* (3rd ed.). Morgan Kaufmann.
3. Nickolls, J., & Dally, W. J. (2010). The GPU computing era. *IEEE Micro, 30*(2), 56–69.
4. Jia, Z., Maggioni, M., Staum, J., & Scarpazza, D. P. (2018). Dissecting the NVIDIA Volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826*.

Suggested Reading

1. NVIDIA. *NVIDIA H100 Tensor Core GPU Architecture*.
2. AMD. *AMD Instinct MI250X Accelerator*.
3. NVIDIA. *CUDA C++ Programming Guide*.
4. Sanders, J., & Kandrot, E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley.

SGOU



CUDA Programming and Data Parallelism

Learning Outcomes

After completion of this unit, learner will be able to:

- explain data parallelism and how it maps to the GPU execution model
- describe the structure of a CUDA program, including host and device code
- analyze device global memory and the process of data transfer between host and device
- write basic CUDA kernels for data-parallel operations
- evaluate the performance implications of memory transfers and data layout

Prerequisite

CUDA (Compute Unified Device Architecture) is NVIDIA's parallel computing platform. It extends C/C++ with keywords and functions that let programmers write code that runs on the GPU. Understanding CUDA requires distinguishing between host code (runs on CPU) and device code (runs on GPU), and appreciating the explicit memory management (copying data between host and device) that heterogeneous computing entails.

Imagine you (the CPU) are managing a team of 1,000 data-entry workers (GPU threads). You have a stack of 1,000 forms (data) in your office (CPU memory). The workers are in a different building (GPU) with their own filing cabinets (GPU memory). To get the work done, you must: (1) ship the forms to the workers' building (host-to-device copy), (2) give each worker a simple, identical instruction (the kernel), (3) wait for them to finish, and (4) ship the completed forms back (device-to-host copy). The data transfer between host and device can become a significant performance bottleneck, especially when large amounts of data are transferred frequently.

Keywords

Data Parallelism, Kernel, Host, Device, Global Memory, Grid

SGOU

Discussion

6.3.1 Data Parallelism

Data parallelism is a form of parallelism where the same operation is applied to many data elements simultaneously. It is the natural fit for GPU computing, which has thousands of simple cores.

Example - Vector Addition:

Add two arrays A and B (each of N elements) to produce C:

```
// Sequential (CPU):
for (int i = 0; i < N; i++)
    C[i] = A[i] + B[i];
```

Each iteration is independent, element C[i] depends only on A[i] and B[i]. This is perfectly data-parallel: the same addition operation applies to all N elements.

Mapping to GPU

Instead of one thread looping through N elements, we launch N threads, each computing one element:

```
// Parallel (GPU): N threads, each does one addition

__global__ void add(int *A, int *B, int *C)
{
  int i = threadIdx.x; // This thread's index
  C[i] = A[i] + B[i];      // Each thread computes one element
}
```

The loop is replaced by thread parallelism; the thread index replaces the loop variable.

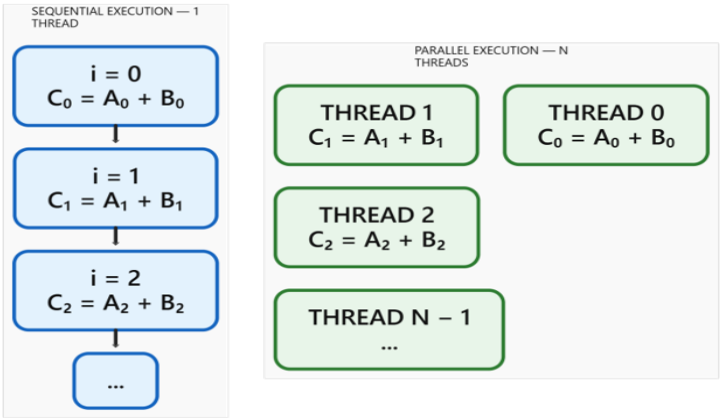


Fig. 6.3.1 Comparison of parallel and sequential vector addition

SGOU

Characteristics of Data-Parallel Problems

- The same operation applies to all elements.
- Elements are independent (no data dependency between them).
- The workload is regular and uniform.
- Examples: vector/matrix operations, image processing, Monte Carlo simulation, neural network layers.

6.3.2 CUDA Program Structure

A CUDA program has a distinct structure with host code and device code.

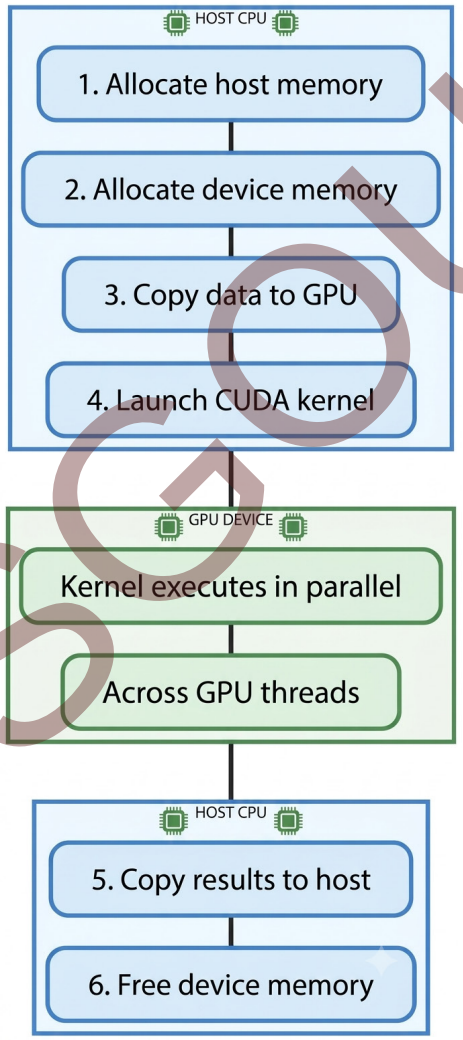


Fig. 6.3.2 Basic CUDA workflow for vector addition

SGOU

Complete CUDA Program (Vector Addition)

```
#include    <cuda_runtime.h>
#include <stdio.h>
#include <stdlib.h>
// Device code (kernel) — runs on GPU
_global_ void vectorAdd(int *A, int *B, int *C, int N) {
      int i = blockIdx.x * blockDim.x + threadIdx.x;
 if (i < N)
C[i] = A[i] + B[i];
}
// Host code — runs on CPU
 int main()
 {
int N = 1 << 20; // 1 million elements
size_t size = N * sizeof(int);
// 1. Allocate host memory
int *h_A = (int*)malloc(size);
int *h_B = (int*)malloc(size);
int *h_C = (int*)malloc(size);
// Initialize host arrays
for (int i = 0; i < N; i++)
 {
    h_A[i] = i;
h_B[i] = 2 * i;
}
// 2. Allocate device memory
int *d_A, *d_B, *d_C;
cudaMalloc(&d_A, size);
```

SGOU

```
cudaMalloc(&d_B, size);
 cudaMalloc(&d_C, size);
// 3. Copy data host → device
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
// 4. Launch kernel
int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);
// 5. Copy result device → host
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
// 6. Free device memory
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
// Free host memory
free(h_A); free(h_B); free(h_C); return 0;
}
```

Table 6.3.1 Key CUDA Keywords and Concepts

Element	Description
__global__	Declares a kernel — called from host, runs on device
__device__	Declares a device function — called from device, runs on device
__host__	CUDA function qualifier indicating that a function executes on the host (CPU). Host execution is the default for ordinary C/C++ functions.
threadIdx	Thread index within a block (3D: .x, .y, .z)
blockIdx	Block index within the grid
blockDim	Block dimensions (threads per block)
gridDim	Grid dimensions (blocks per grid)
<<<grid, block>>>	Kernel launch syntax

SGOU

Thread Indexing

The global thread index is computed as:

```
global_index = blockIdx.x * blockDim.x + threadIdx.x
```

For 2D indexing layout:

```
row = blockIdx.x * blockDim.x + threadIdx.x // row
col = blockIdx.y * blockDim.y + threadIdx.y // column
```

Solved Example 1: A kernel is launched with 64 blocks and 128 threads per block. What is the total number of threads? What is the global index of thread 50 in block 10?

Solution

- Total threads = 64 × 128 = 8,192.
- Global index of thread 50 in block 10

  = blockIdx.x × blockDim.x + threadIdx.x

  = 10 × 128 + 50 = 1,330.

6.3.3 Device Global Memory and Data Transfer

Device Global Memory

Global memory is the main GPU memory (HBM/GDDR). Characteristics:

- **Large capacity:** 8-80 GB.
- **High latency:** 400-800 cycles (vs. ~4 cycles for CPU L1 cache).
- **High bandwidth:** several hundred GB/s to multiple TB/s, depending on the GPU architecture and memory technology.
- Accessible by all threads across all blocks.
- **Persistent:** Data survives across kernel launches (until freed).

The following values are representative ranges intended for illustration; actual capacity, latency, and bandwidth depend on the GPU architecture and product.

Global memory access patterns critically affect performance

1. **Coalesced Access:** When threads in a warp access consecutive memory addresses, the memory system can combine (coalesce) these accesses into a single wide transaction. This maximizes bandwidth utilization.

SGOU

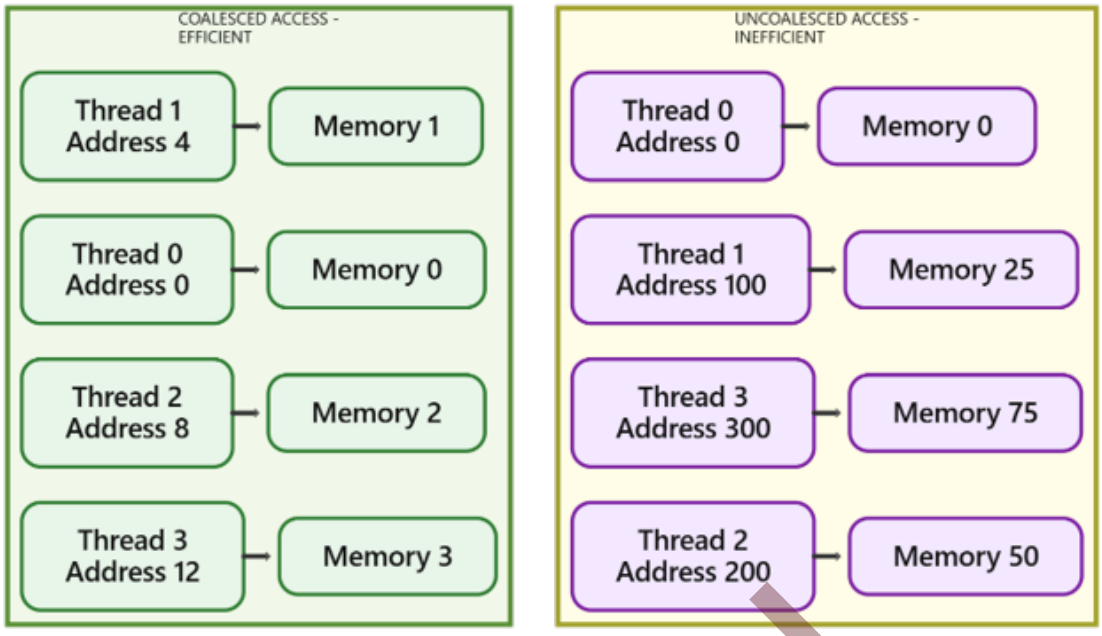


Fig. 6.3.3 Comparison of uncoalesced and coalesced global memory accesses

2. **Data Transfer (Host ↔ Device):** Data transfer between host and device is a critical bottleneck. Host-device transfers over interfaces such as PCIe can have substantially lower bandwidth than GPU global memory. GPU-to-GPU links such as NVLink can provide much higher bandwidth than PCIe, depending on the configuration.

Table 6.3.2 Bandwidth and Latency Comparison of GPU Interconnects and Memory Interfaces

Interface	Bandwidth	Latency
PCIe 4.0 x16	~32 GB/s	High
PCIe 5.0 x16	~64 GB/s	High
NVLink 3.0	~600 GB/s	Lower
GPU Global Memory (HBM3)	~3,300 GB/s	High relative to on-chip memory

The following bandwidth values are representative examples; actual values depend on the specific hardware configuration and implementation.

Key performance principle: minimize host-device data transfer

Techniques:

- **Keep data on device:** Perform multiple operations on data while it's on the GPU, rather than copying back and forth.

SGOU

- **Overlap transfer with computation:** Use asynchronous transfers (cudaMemcpyAsync) and CUDA streams to hide transfer latency behind kernel execution.
- **Batch transfers:** Transfer large contiguous blocks in one operation rather than many small transfers.

Solved Example 2: Transferring 1 GB of data from host to device over PCIe 4.0 (32 GB/s) and computing a kernel that takes 50 ms. If the transfer is synchronous (no overlap), what is the total time? If fully overlapped, what is the minimum time?

Solution

- Transfer time = 1 GB / 32 GB/s = 31.25 ms.
- Synchronous: Transfer (31.25 ms) + Kernel (50 ms) = 81.25 ms.
- Fully overlapped: max(31.25 ms, 50 ms) = 50 ms.
- Overlap saves 31.25 ms (38% reduction).

Other Memory Types (context for later)

While global memory is the main device memory, CUDA also provides:

- **Shared memory:** On-chip memory associated with a thread block and designed for fast data sharing among threads in the block. Its latency depends on the GPU architecture and access pattern.
- **Registers:** Per-thread storage providing very low-latency access.
- **Constant memory:** Read-only, cached, optimized for broadcast.
- **Texture memory:** Read-only, cached, optimized for spatial locality.

The Cost of Data Movement

In GPU computing, moving data is often more expensive than computing on it. A global-memory access can require substantially more cycles than an arithmetic operation, depending on the architecture and whether the data is already available through caches. This is why GPU programmers obsess over memory access patterns (coalescing, shared memory, caching). The "roofline model" formalizes this: performance is limited either by peak compute (for arithmetic-intensive workloads) or by memory bandwidth (for memory-bound workloads). Understanding which limit applies is the first step in optimization.

SGOU

Zero-Copy Memory and Unified Memory

Traditionally, CUDA required explicit copies between host and device memory. NVIDIA's Unified Memory (introduced in CUDA 6) abstracts this away, the programmer accesses a single address space, and the runtime automatically migrates data between host and device as needed. This dramatically simplifies programming (no explicit cudaMemcpy), at the cost of some performance overhead from page faults and migration. Zero-copy memory allows the GPU to directly access pinned host memory, useful for small, frequently-accessed data.

Recap

- Data parallelism applies the same operation to many independent data elements, the natural fit for GPU computing.
- A CUDA program has host code (CPU) and device code (GPU). The host orchestrates; the device accelerates.
- CUDA program flow: allocate host memory → allocate device memory (cudaMalloc) → copy data H2D (cudaMemcpy) → launch kernel → copy results D2H → free memory.
- Kernel is declared with __global__ and launched using the <<<grid, block>>> execution configuration syntax.
- Thread indexing: global index = blockIdx × blockDim + threadIdx.
- Global memory is the main device memory — large, high-latency, high-bandwidth, accessible by all threads.
- Coalesced access (consecutive addresses in a warp) maximizes memory bandwidth; uncoalesced access wastes bandwidth.
- Data transfer (host↔device) over PCIe is a bottleneck; minimize it, batch it, and overlap it with computation.
- Shared memory (on-chip) and registers (per-thread) are faster alternatives to global memory.
- Unified Memory abstracts explicit data movement, simplifying programming.

SGOU

Objective Type Questions

1. What is data parallelism?
2. What is a CUDA kernel?
3. What is the difference between host code and device code?
4. What does threadIdx.x represent?
5. How is the global thread index calculated in a one-dimensional CUDA grid?
6. What is the purpose of cudaMalloc()?
7. What is the purpose of cudaMemcpy()?
8. What is coalesced memory access?
9. Why can host-device data transfer become a performance bottleneck?
10. How can asynchronous data transfer improve CUDA program performance?

Answers

1. Applying the same operation to multiple data elements concurrently.
2. A function executed by many GPU threads in parallel.
3. Host code runs on the CPU, while device code runs on the GPU.
4. The thread's index within its block.
5. blockIdx.x × blockDim.x + threadIdx.x.
6. To allocate memory on the GPU device.
7. To copy data between host and device memory.
8. A memory-access pattern in which threads in a warp access nearby or consecutive memory locations efficiently.
9. Because the transfer bandwidth can be much lower than the bandwidth of GPU global memory.
10. By overlapping data transfer with kernel execution.

SGOU

Assignments

1. Explain data parallelism with vector addition and identify why vector addition is suitable for GPU execution.

2. Explain the sequence of operations in a CUDA program from host-memory allocation to freeing device memory.

3. Write a CUDA kernel for element-wise vector subtraction and explain how each thread determines the element it processes.

4. Explain the roles of threadIdx, blockIdx, and blockDim. Demonstrate their use with a one-dimensional kernel.

5. Explain the characteristics of GPU global memory and discuss how memory-access patterns influence performance.

6. Explain host-to-device and device-to-host data transfers using cudaMemcpy(). Discuss why minimizing transfers is important.

7. Compare coalesced and uncoalesced global-memory accesses using a suitable example or diagram.

8. Explain how asynchronous transfers, CUDA streams, and overlapping computation with data transfer can improve application performance.

9. A CUDA kernel is launched with 32 blocks and 256 threads per block. Calculate the total number of threads.

10. For a thread with blockIdx.x = 5, blockDim.x = 256, and threadIdx.x = 20, calculate its global thread index.

11. Explain why coalesced global-memory access generally provides better performance than uncoalesced access.

12. A program transfers 2 GB of data over a 32 GB/s interface. Calculate the approximate transfer time. Explain how overlapping the transfer with computation can reduce total execution time.

SGOU

Reference

1. Hennessy, J. L., Patterson, D. A., & Kozyrakis, C. (2025). *Computer architecture: a quantitative approach*. Morgan kaufmann.

2. Kirk, D. B., & Hwu, W. W. (2016). *Programming Massively Parallel Processors: A Hands-on Approach* (3rd ed.). Morgan Kaufmann.

3. Sanders, J., & Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley.

4. Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. *ACM Queue, 6*(2), 40–53.

Suggested Reading

1. NVIDIA. *CUDA C++ Programming Guide*.

2. NVIDIA. *CUDA C++ Best Practices Guide*.

3. NVIDIA. *CUDA Runtime API Documentation*.

SGOU



Kernel Functions and Threading

Learning Outcomes

After completion of this unit, learner will be able to:

- explain kernel basics and the GPU threading model
- distinguish between processes and threads, and describe types of threads
- describe thread management in operating systems and GPU-specific thread scheduling
- explain kernel-level thread support and its advantages
- analyze thread synchronization mechanisms and identify, prevent, and resolve deadlock

Prerequisite

So far, we have explored several important concepts in parallel computing and understood how computational tasks can be performed more efficiently. But how are these computations actually executed on a GPU? How does a program organize thousands of parallel operations, and how are these operations managed to ensure efficient execution?

To understand this, we need to look more closely at two fundamental concepts: kernel functions and threading. They help us understand how computational instructions are defined, how threads execute them, and how parallel execution is organized on GPU hardware.

Let us move ahead and explore Kernel Functions and Threading to uncover how these concepts bring parallel computing into action.

Keywords

Kernel, Process, Thread, User-Level Thread (ULT), Kernel-Level Thread (KLT), Warp, Synchronization, Deadlock, Thread Indexing, Coalesced Memory Access

SGOU

Discussion

6.4.1 Kernel Basics

The term "kernel" has two related meanings in computing:

1. **Operating System Kernel:** The core of an operating system that manages hardware resources (CPU, memory, I/O), provides system calls, and schedules processes/threads. It runs in privileged mode and is the foundation upon which all other software runs.

2. **CUDA Kernel (GPU):** A function executed on the GPU by many threads in parallel. In CUDA, a kernel is declared with __global__ and launched with `<<<grid, block>>>`. The kernel represents the data-parallel portion of a program.

This unit covers both meanings — kernel basics (the CUDA kernel and the OS kernel's thread support) — since both are relevant to GPU computing.

CUDA Kernel Characteristics:

- Executes as a grid of thread blocks.
- Each thread executes the same kernel code (SIMT).
- Threads access their unique indices via threadIdx, blockIdx, blockDim.
- Kernels cannot return values (they write results to memory).
- Kernels execute asynchronously (the host continues without waiting, unless synchronized).

Kernel Launch Configuration:

```
kernelName<<<gridDim, blockDim, sharedMemSize, stream>>>(args);
```

Table 6.4.1 CUDA Kernel Launch Parameters

Parameter	Description
gridDim	Number of blocks in the grid
blockDim	Number of threads per block
sharedMemSize	Optional dynamic shared-memory size
stream	Optional CUDA stream

SGOU

6.4.2 Process vs. Thread

Process:

- An executing program with its own virtual address space.
- Has its own virtual address space and resources such as memory mappings and file descriptors. Each thread within the process has its own registers, program counter, and stack.
- Processes are isolated — one process cannot directly access another's memory.
- Creating and switching processes is expensive (context switch involves memory mapping changes).
- Inter-process communication (IPC) is required (pipes, sockets, shared memory).

Thread:

- The smallest unit of execution within a process.
- Threads share the process's address space and resources.
- Each thread has its own stack, registers, and program counter.
- Threads are lightweight — creating and switching threads is cheap.
- Threads communicate via shared memory (no IPC needed).
- Multiple threads can run concurrently on multiple CPU cores (or GPU cores).

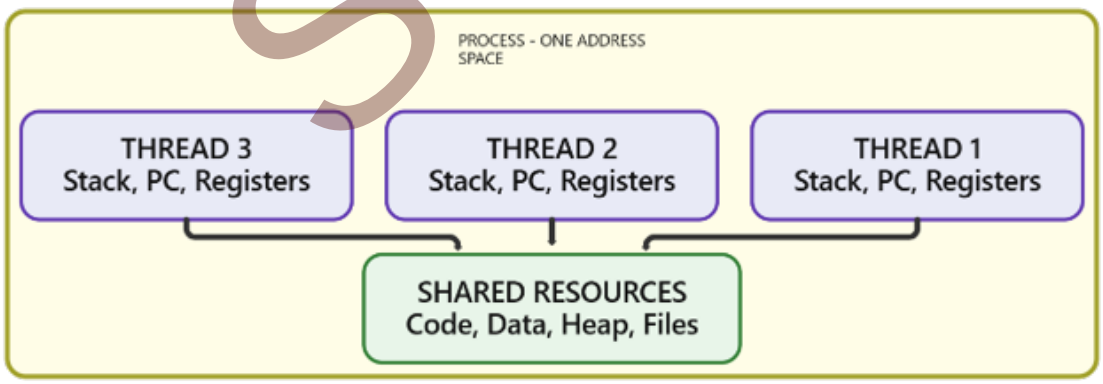


Fig. 6.4.1 Multiple threads sharing a single process address space

SGOU

Comparison:

Table 6.4.2 Comparison between Processes and Threads

Aspect	Process	Thread
Address space	Own (isolated)	Shared (within process)
Creation cost	High	Low
Context switch	Expensive	Cheap
Communication	IPC needed	Shared memory
Failure isolation	High	Low (one thread crash kills process)
Parallelism	Across cores/machines	Within process, across cores

Solved Example 1: A multi-threaded program has 4 threads on a 4-core CPU. Each thread performs a compute-bound task. What is the maximum speedup? What if the task is I/O-bound with 90% I/O wait?

Solution:

- **Compute-bound:** With 4 cores and 4 threads, ideal speedup is ~4× (linear). No speedup beyond 4 threads (no more cores).
- **I/O-bound (90% wait):** Threads spend 90% of time waiting for I/O. With many threads, the CPU can switch to another thread during I/O wait. The CPU utilization can approach 100%, and far more than 4 threads can be beneficial (each thread is mostly waiting). The speedup is limited by the I/O subsystem, not the CPU count.

6.4.3 Types of Threads

Threads are classified by how they are managed:

1. User-Level Threads (ULTs):

Managed by a user-space threading library or runtime without direct kernel scheduling of each user-level thread. Examples include green threads and user-space fibers. In a many-to-one model, multiple ULTs are mapped to a single kernel-supported thread, so the operating system sees only one schedulable thread for that process.

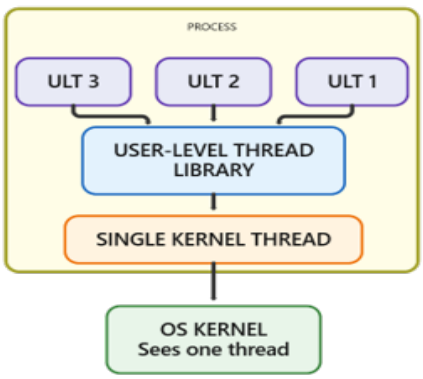


Fig. 6.4.2 Many-to-one mapping of user-level threads to a kernel thread

SGOU

Advantages:

- Fast creation and switching (no kernel mode transitions).
- Portable (library-level, not OS-dependent).
- Application-specific scheduling.

Disadvantages:

- If one ULT blocks (e.g., I/O), all ULTs in the process block (the kernel doesn't know there are multiple threads).
- Cannot exploit multiple CPU cores (the kernel schedules only one thread).

2. Kernel-Level Threads (KLTs):

Managed directly by the OS kernel. Each thread is visible to the kernel and can be scheduled independently.

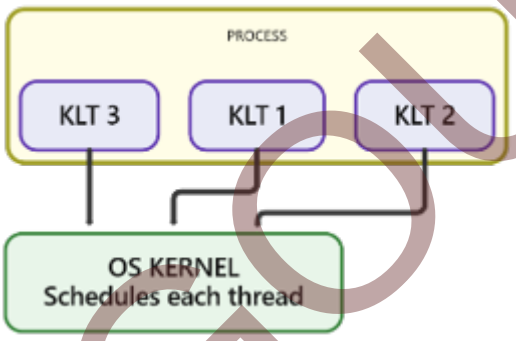


Fig. 6.4.3 One-to-one mapping of kernel-level threads to the operating system

Advantages:

- True parallelism across multiple cores (kernel schedules threads on different cores).
- If one thread blocks, others can continue.
- OS-level scheduling, priority, and preemption.

Disadvantages:

- Slower creation and switching (kernel mode transitions).
- Higher overhead.

3. Hybrid (Many-to-Many) Model:

Combines ULTs and KLTs: multiple ULTs are mapped to multiple KLTs. The application can use many cheap ULTs while still exploiting multiple cores through the KLTs. Modern runtimes such as Go and Java virtual threads use runtime schedulers that multiplex many user-level tasks or virtual threads over a smaller number of kernel-supported carrier threads.

SGOU

6.4.4 Thread Management

Thread management involves creating, scheduling, and coordinating threads.

Thread Lifecycle (OS threads):

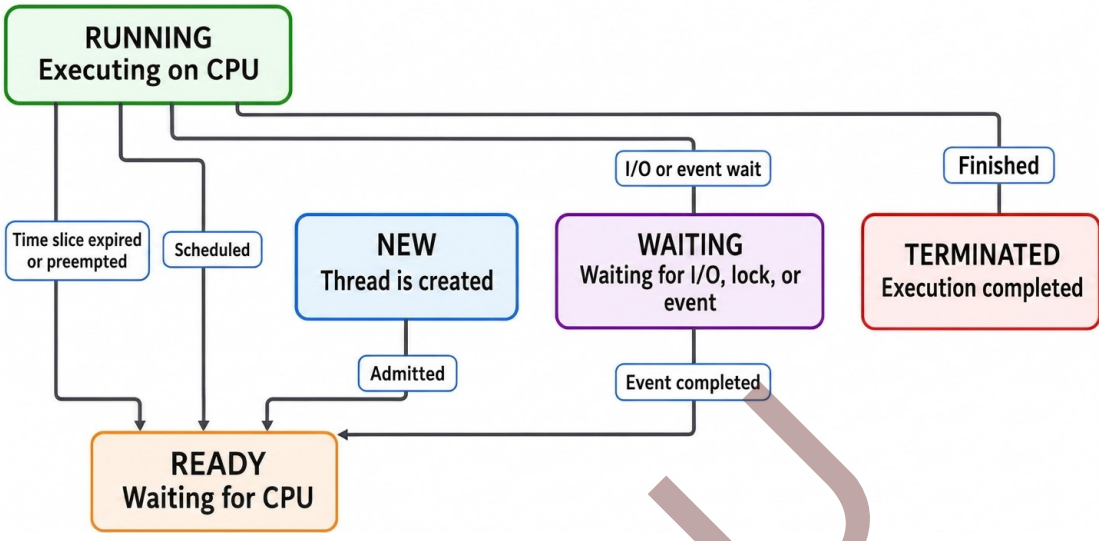


Fig. 6.4.4 Thread state transitions

Thread Operations:

- **Create:** Spawn a new thread (pthread_create, std::thread, etc.).
- **Join:** Wait for a thread to finish (pthread_join).
- **Detach:** Let a thread run independently, reclaiming resources on exit.
- **Yield:** Voluntarily give up the CPU to another thread.
- **Exit/Terminate:** End the thread's execution.

GPU Thread Management (different from OS):

GPU threads are managed differently from OS threads:

- CUDA threads are lightweight from the host operating system's perspective; their execution is managed by the GPU hardware and CUDA runtime.
- CUDA organizes threads into blocks, and hardware executes threads in groups called warps. Blocks are assigned to streaming multiprocessors (SMs), while warp schedulers select eligible warps for execution..
- Warps are scheduled by the SM's warp scheduler (in hardware).
- A CUDA kernel can launch a very large number of threads. The GPU hardware executes these threads in warps, while blocks are scheduled onto available SM resources.
- There is no per-thread OS overhead; scheduling is hardware-based.

SGOU

GPU Thread Hierarchy:

Table 6.4.3 CUDA Thread Hierarchy, Communication, and Synchronization

Level	Count (typical)	Communication	Synchronization
Thread	Thousands-millions	Registers (private)	Within warp (lockstep)
Warp	32 threads	Shuffle (warp-level)	Implicit (lockstep)
Block	Up to 1024 threads	Shared memory	syncthreads()
Grid	Up to millions	Global memory	Kernel boundary

6.4.5 Kernel-Level Thread Support

Kernel-level thread support refers to the OS kernel's ability to manage threads directly (as opposed to user-level thread libraries).

Why Kernel-Level Threads Matter:

1. **True Parallelism:** Kernel-level threads allow the OS scheduler to schedule individual threads independently on multiple CPU cores. User-level threading can also exploit multiple cores when a runtime maps user-level threads onto multiple kernel-supported threads, as in many-to-many models.

2. **Blocking Without Stalling:** If one kernel-level thread blocks on I/O, the kernel can schedule another thread — unlike user-level threads, where one blocked thread stalls the entire process.

3. **Fair Scheduling:** The kernel can enforce fairness, priorities, and preemption across threads.

4. **Integration with OS Services:** Kernel threads integrate with the OS's I/O, signal, and scheduling subsystems.

Linux Implementation:

Linux implements threads as "lightweight processes" (via the clone() system call). Each thread has:

- Its own thread ID (TID).
- Its own stack and registers.
- Shared address space, file descriptors, and signal handlers (via clone flags).

The scheduler treats threads as schedulable entities, assigning them to CPU cores independently.

SGOU

GPU Kernel-Level Support (Analogous Concept):

On the GPU, the "kernel-level" support is provided by the hardware:

- The GPU's thread scheduler manages warps directly in hardware.
- Warp scheduling is done at the SM level (each SM has warp schedulers).
- The driver and runtime (CUDA runtime) manage grid/block allocation across SMs.
- There is no OS involvement in individual GPU thread scheduling — it's entirely hardware-managed.

Solved Example 2: A CPU has 4 cores. A program creates 8 kernel-level threads, all compute-bound. How many threads run truly in parallel? What happens to the other 4 threads?

Solution:

- Only 4 threads run truly in parallel (one per core).
- The other 4 threads are in the "ready" state, waiting for a core.
- The OS scheduler time-slices: each thread gets a portion of CPU time.
- The threads run concurrently (interleaved) but not all in parallel simultaneously.

6.4.6 Thread Synchronization

Synchronization coordinates threads to ensure correct ordering and prevent race conditions.

Why Synchronization is Needed:

When multiple threads access shared data, without synchronization:

- **Race conditions:** The outcome depends on the unpredictable timing of thread execution.
- **Data corruption:** Concurrent read/write of the same data produces inconsistent results.
- **Lost updates:** One thread's update overwrites another's.

Synchronization Mechanisms:

1. **Mutex (Mutual Exclusion):** A lock that allows only one thread to access a critical section at a time.

SGOU

```
pthread_mutex_lock(&mutex);
// critical section
pthread_mutex_unlock
(&mutex);
```

2. **Semaphore:** A counter-based synchronization primitive. Binary semaphores act like mutexes; counting semaphores control access to a pool of resources.

3. **Barrier:** A synchronization point where all threads wait until all have arrived.

```
pthread_barrier_wait(&barrier);
```

4. **Condition Variable:** Allows threads to wait for a condition to become true, and to signal other threads.

GPU Synchronization:

CUDA provides specific synchronization primitives:

Table 6.4.4 Scope of CUDA synchronization primitives

Primitive	Scope	Description
syncthreads()	Block	Barrier: all threads in a block wait until all arrive
syncwarp()	Warp	Synchronize threads within a warp
Atomic operations	Device	Atomic read-modify-write (atomicAdd, etc.)
Cooperative groups	Various scopes	Provides flexible synchronization and coordination mechanisms; grid-wide synchronization requires an appropriate cooperative launch.

Example — Shared Memory Synchronization:

```
_global_ void reduction(int *input, int *output, int N)
 {
   _shared_ int sdata[256];
   int tid = threadIdx.x;
   int i = blockIdx.x * blockDim.x + threadIdx.x;
// Load into shared memory
sdata[tid] = (i < N) ? input[i] : 0;
```

SGOU

```
__syncthreads(); // Ensure all loads complete
// Reduction in shared memory
for (int s = 1; s < blockDim.x; s *= 2) {
     if (tid % (2*s) == 0)
  sdata[tid] += sdata[tid + s];
___syncthreads(); // Sync between reduction steps
}
```

The barriers ensure all threads have loaded their data before any thread starts the reduction, and `_syncthreads()` synchronize between reduction steps.

6.4.7 Deadlock in Threading

Deadlock is a situation where two or more threads (or processes) are each waiting for a resource held by another, forming a circular wait — none can proceed.

Four Necessary Conditions for Deadlock (Coffman Conditions):

1. **Mutual Exclusion:** Resources cannot be shared (at least one resource is non-shareable).
2. **Hold and Wait:** A thread holds at least one resource while waiting for another.
3. **No Preemption:** Resources cannot be forcibly taken away; only the holder can release them.
4. **Circular Wait:** A cycle of threads each waiting for a resource held by the next. All four conditions must hold simultaneously for deadlock to occur.

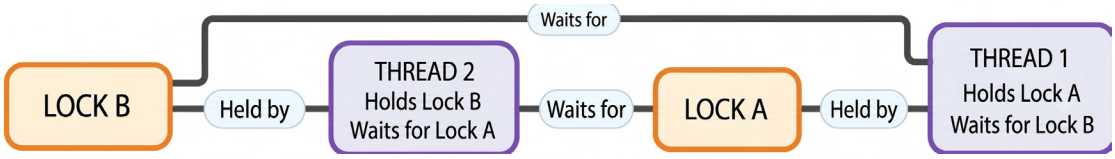


Fig. 6.4.5 Deadlock caused by circular waiting between two threads

Classic Deadlock Example:

```
// Thread 1:
pthread_mutex_lock(&mutexA);
pthread_mutex_lock(&mutexB);     // Waits for B, holds A
```

SGOU

```
// ... use A and B ...
pthread_mutex_unlock(&mutexB);
pthread_mutex_unlock(&mutexA);
```

```
// Thread 2:
pthread_mutex_lock(&mutexB);
pthread_mutex_lock(&mutexA);     // Waits for A, holds B
// ... use A and B ...
pthread_mutex_unlock(&mutexA);
pthread_mutex_unlock(&mutexB);
```

If Thread 1 acquires A and Thread 2 acquires B simultaneously, then Thread 1 waits for B (held by Thread 2) and Thread 2 waits for A (held by Thread 1) — deadlock.

Deadlock Prevention and Avoidance:

1. **Lock Ordering (Prevention):** Always acquire locks in a consistent, fixed order. If all threads lock A before B, no circular wait can form.

```
// Both threads:
pthread_mutex_lock(&mutexA);
pthread_mutex_lock(&mutexB);
// ...
pthread_mutex_unlock(&mutexB);
pthread_mutex_unlock(&mutexA);
```

2. **Lock-Free / Wait-Free Algorithms:** can avoid traditional lock-based deadlocks by using atomic operations instead of mutexes. They have their own progress and fairness considerations.

3. **Timeouts (Avoidance):** Use try-lock with a timeout; if a lock cannot be acquired, release held locks and retry.

4. **Deadlock Detection:** Maintain a resource allocation graph; detect cycles and break them (by preempting resources or terminating a thread).

SGOU

5. **Avoid Hold-and-Wait:** Acquire all required resources atomically (or release all held resources before requesting new ones).

Solved Example 3: Two threads need to update a bank account balance. Thread 1: transfer from account A to B. Thread 2: transfer from account B to A. Explain how deadlock occurs and how to prevent it.

Solution:

- Thread 1: locks A, then tries to lock B.
- Thread 2: locks B, then tries to lock A.
- Deadlock: Thread 1 holds A and waits for B; Thread 2 holds B and waits for A.
- Prevention (lock ordering): Always lock accounts in order of account ID (lower ID first). Both threads lock A before B, so no circular wait.

The Dining Philosophers Problem

Edsger Dijkstra's classic concurrency problem (1965) illustrates deadlock beautifully: five philosophers sit around a table with one fork between each pair. To eat, a philosopher needs both adjacent forks. If each philosopher picks up the left fork simultaneously, all wait for the right fork — deadlock. Solutions include: pick up forks in a consistent order (one philosopher picks up the right fork first), or use a limit on simultaneous diners. This problem remains the canonical teaching example for deadlock avoidance.

GPU Warp Divergence

The Hidden Deadlock Risk On GPUs, block. If a barrier is placed inside a conditional branch (e.g., must be reached by all threads in a), some threads reach the barrier while others skip it — the threads that reach the barrier wait forever for threads that never arrive. This can cause a GPU kernel to hang because some threads reach the barrier while others do not. Therefore, a block-level barrier such as __syncthreads() should be used only where all threads in the block can reach it according to the program's control flow.

SGOU

Recap

- A CUDA kernel is a function executed by many GPU threads, while an OS kernel manages system resources and provides core operating-system services.
- A process has its own address space and resources, while threads within the process share the address space but have their own execution state.

- User-level threads are managed by a user-space runtime, whereas kernel-level threads are managed and scheduled by the operating system.
- The GPU thread hierarchy in NVIDIA CUDA consists of threads, warps, blocks, and grids.
- Thread management includes creation, scheduling, synchronization, waiting, and termination.
- Synchronization mechanisms include mutexes, semaphores, barriers, condition variables, atomic operations, and CUDA synchronization primitives.
- In CUDA, __syncthreads() provides block-level synchronization, while __ syncwarp() synchronizes threads within a warp.
- Deadlock requires mutual exclusion, hold-and-wait, no preemption, and circular wait.
- Deadlock can be prevented through consistent lock ordering, avoiding hold-and-wait, timeouts, and suitable lock-free or wait-free designs.
- In GPU programs, barriers placed incorrectly in divergent control flow can cause a kernel to hang.

Objective Type Questions

1. What is the term for a function executed on the GPU by many threads in parallel?
2. What is the fundamental difference between a process and a thread?
3. What type of thread is managed directly by the OS kernel and can be scheduled independently on multiple CPU cores?
4. In the thread lifecycle, what state does a thread enter when it is waiting for I/O?
5. What CUDA function synchronizes all threads within a block?
6. What are the four necessary conditions for deadlock?
7. What technique prevents deadlock by always acquiring locks in the same order?
8. In the SIMT model, what is a group of 32 threads called?

SGOU

9. What is the smallest unit of execution within a process?

10. In the Dining Philosophers problem, what resource do the philosophers compete for?

Answers

1. Kernel.

2. A process has its own address space; threads share a process's address space.

3. Kernel-level thread (KLT).

4. Waiting (blocked) state.

5. __syncthreads()

6. Mutual exclusion, hold-and-wait, no preemption, circular wait.

7. Lock ordering (consistent lock acquisition order).

8. A warp.

9. Thread.

10. Forks (chopsticks).

Assignments

1. Write a multi-threaded program (C with pthreads, Java, or Python threading) that computes a parallel sum using multiple threads. Use a mutex to protect the shared sum variable. Then rewrite it using a barrier-based approach. Compare the results.

2. Identify a real-world scenario where deadlock can occur (e.g., database transaction, file locking). Describe the deadlock conditions, and propose two different prevention strategies. Evaluate the trade-offs of each.

3. Implement a simple thread pool in your preferred language. Explain how the pool manages worker threads, task queueing, and synchronization. Compare the performance with creating a new thread per task.

4. Compare processes and threads with respect to address space, resource sharing, creation cost, context switching, communication, and failure isolation.

SGOU

5. Compare ULTs and KLTs. Discuss the advantages and limitations of each and explain why modern runtimes may use a combination of user-level scheduling and kernel-supported threads.

6. Explain the relationship among a CUDA thread, warp, block, and grid. Illustrate the hierarchy with a suitable diagram.

7. Explain __syncthreads(), __syncwarp(), atomic operations, and cooperative groups. Give a suitable example showing why synchronization is necessary.

8. Develop a small multithreaded program that can produce deadlock using two mutexes. Modify the program using consistent lock ordering and explain why the modified version avoids circular wait.

Reference

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.

2. Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.). Pearson.

3. Hennessy, J. L., & Patterson, D. A. (2019). *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann.

4. Kirk, D. B., & Hwu, W. W. (2016). *Programming Massively Parallel Processors: A Hands-on Approach* (3rd ed.). Morgan Kaufmann.

Suggested Reading

1. NVIDIA. *CUDA C++ Programming Guide*.

2. NVIDIA. *CUDA C++ Best Practices Guide*.

3. Herlihy, M., & Shavit, N. *The Art of Multiprocessor Programming*.

SGOU

Model Question Paper Sets

SGOU

SREENARAYANAGURU OPEN UNIVERSITY

MODEL QUESTION PAPER - SET 1

QP CODE: ……

Reg. No:.....

Name: ……..

SIXTH SEMESTER BACHELOR OF COMPUTER APPLICATIONS (BCA) EXAMINATION

DISCIPLINE SPECIFIC ELECTIVE COURSE

B21CA05DE - HIGH PERFORMANCE COMPUTING

2023 - 24 Admission Onwards

Time: 3 Hours Max Marks:70

Section A

Answer any 10 questions. Each carries one mark

(10x1=10)

1. Why is high performance computing important?
2. What is pipeline depth?
3. State two advantages of array processors.
4. What is the technique of overlapping instruction execution called?
5. Which pipeline stage is responsible for retrieving an instruction from memory?
6. Which communication operation sends data from one node to all nodes?
7. What search operation is O(1) using content-addressable memory?
8. What is a cache that searches all lines simultaneously for a tag match called?
9. What does PDU stand for?
10. Which network topology connects each node to two neighbouring nodes in a closed loop?
11. Which programming technique is commonly used instead of mutable loops in functional languages?

SGOU

12. Which architecture uses associative tag matching to distinguish different instruction invocations?

13. Name one tool used for profiling programs.

14. What memory technology stacks memory dies directly on the GPU package to increase bandwidth?

15. How do GPUs Scale?

Section B

Answer any 5 questions. Each carries two marks

(5x2=10)

16. Write any four levels of parallelism in computing.

17. Mention any two techniques used to overcome pipeline hazards.

18. Differentiate between scalar processing and vector processing.

19. What is branch prediction? State its purpose.

20. Mention any two limitations of SIMD array processors.

21. Differentiate between a mutex and a semaphore.

22. Differentiate between LAN and WAN

23. Explain Data-Centric Programming Paradigms.

24. What is Amdahl's Law?

25. Explain Data Parallelism.

Section C

Answer any 5 questions. Each carries four marks

(5x4=20)

26. State Amdahl's Law and explain how the serial portion of a program limits the maximum speedup in parallel computing. Write the speedup formula and explain its terms.

27. Explain the applications of parallel computing. Describe the different decomposition strategies and resources required for parallel computing.

SGOU

28. Explain the advantages of pipelining in processor design.

29. Describe any four design issues involved in pipelined processors.

30. Write a detailed step-by-step description of 2×2 matrix multiplication on a 2×2 systolic array. Trace

31. Explain any four applications of SIMD array processors with suitable examples.

32. Explain Bus, Star, Ring and Mesh topologies.

33. Describe Declarative and Functional Languages.

34. What are the different software frameworks to implement data-driven or data-centric execution principles?

35. Explain different types of Threads?

Section D

Answer any 2 questions. Each carries fifteen mark

(2x15=30)

36. Explain Flynn's Taxonomy of computer architectures in detail with examples. Draw a neat, labelled diagram illustrating Flynn's Taxonomy.

37. Explain the principles of parallel algorithm design and discuss various parallel algorithms used for addition, multiplication, matrix operations, sorting, and searching in SIMD array processors.

38. Describe the explicit-token-store (ETS) architecture and explain how it balances the trade-offs between static and dynamic data-flow designs.

39. Explain the sequence of operations in a CUDA program from host-memory allocation to freeing device memory.

SGOU

SREENARAYANAGURU OPEN UNIVERSITY

MODEL QUESTION PAPER - SET 2

QP CODE: ……

Reg. No:.....

Name: ……..

SIXTH SEMESTER BACHELOR OF COMPUTER APPLICATIONS (BCA) EXAMINATION

DISCIPLINE SPECIFIC ELECTIVE COURSE

B21CA05DE - HIGH PERFORMANCE COMPUTING

2023 - 24 Admission Onwards

Time: 3 Hours Max Marks:70

Section A

Answer any 10 questions. Each carries one mark

(10x1=10)

1. What is High Performance Computing?
2. List any two stages involved in floating-point addition
3. What is the primary purpose of CUDA introduced in 2006?
4. Which type of hazard arises due to data dependency between instructions?
5. What is the technique of predicting the outcome of a branch instruction called?
6. What metric measures the maximum shortest-path distance between any two nodes?
7. What type of array uses local data flow (nearest-neighbor) for matrix multiplication?
8. What component in a CAM selects one match when multiple words match?
9. Which OSI layer is responsible for logical addressing and routing?
10. What does UMA stand for?
11. Name one batch-oriented data-flow framework.

SGOU

12. Which algorithm is a classic hardware scheme for managing instruction dependencies in out-of-order processors?

13. What is the maximum speedup according to Amdahl's Law?

14. Which level of the GPU memory hierarchy is fastest but smallest and per-thread?

15. Which lower-precision formats are used to improve AI energy efficiency?

Section B

Answer any 5 questions. Each carries two marks

(5x2=10)

16. List the five stages of instruction pipelining in a typical RISC processor.

17. Define latency and throughput in instruction pipelining.

18. What are the basic stages of an instruction pipeline?

19. What is a control hazard? Give an example.

20. Mention any four basic principles of parallel algorithm design for array processors.

21. Mention two characteristics of distributed memory architecture

22. Differentiate between a switch and a router.

23. What is the memory wall?

24. What is a Heterogeneous Programming Model?

25. What is GPU Cluster Scaling?

Section C

Answer any 5 questions. Each carries four marks

(5x4=20)

26. Discuss the advantages, limitations and applications of MIMD systems.

27. What are pipeline hazards? Explain the three major types of pipeline hazards with suitable examples.

28. Explain the basic features of a vector instruction set.

SGOU

29. Describe the three major types of pipeline hazards with examples.

30. Compare the hardware cost and power consumption of CAM versus SRAM.

31. Explain the SIMD concept and describe the basic architecture of an array processor with its major components.

32. Explain shared memory and distributed memory architectures with suitable diagrams

33. What is lazy evaluation? Explain its relationship to the data-flow firing principle with a suitable example.

34. Explain how loop unrolling can partially compensate for the limitations of static data-flow computers.

35. Compare the Processes and Threads.

Section D

Answer any 2 questions. Each carries fifteen mark

(2x15=30)

36. Explain the concept and working of arithmetic pipelines. Illustrate the execution of floating-point addition using a suitable pipeline example.

37. Explain the different network topologies, including bus, star, ring, mesh, tree and hybrid topologies. Compare their structures, advantages, limitations and applications.

38. Write a detailed note on the case studies of Multiprocessor and Data-Flow Systems.

39. Explain __syncthreads(), __syncwarp(), atomic operations, and cooperative groups. Give a suitable example showing why synchronization is necessary.

SGOU

സർവ്വകലാശാലാഗീതം

വിദ്യയാൽ സ്വതന്ത്രരാകണം
വിശ്വപൗരരായി മാറണം
ഗ്രഹപ്രസാദമായ് വിളങ്ങണം
ഗുരുപ്രകാശമേ നയിക്കണേ

കൂരിരുട്ടിൽ നിന്നു ഞങ്ങളെ
സൂര്യവീഥിയിൽ തെളിക്കണം
സ്നേഹദീപ്തിയായ്വിളങ്ങണം
നീതിവൈജയന്തി പാറണം

ശാസ്ത്രവ്യാപ്തിയെന്നുമേകണം
ജാതിഭേദമാകെ മാറണം
ബോധരശ്മിയിൽ തിളങ്ങുവാൻ
ജ്ഞാനകേന്ദ്രമേ ജ്വലിക്കണേ

കുരീപ്പുഴ ശ്രീകുമാർ

SGOU

SREENARAYANAGURU OPEN UNIVERSITY

Regional Centres

Kollam
Sreenarayanaguru Open University
Headquarters Building
BSNL Bhavan, Vellayittambalam,
Kollam, Kerala, Pin : 691012
Ph: 9188912554
email:rckollam@sgou.ac.in

Tripunithura
Govt. College
Tirpunithura, Ernakulam
Kerala, Pin : 682301
Ph: 04842927436
email:rcedirector@sgou.ac.in

Thalassery
Govt. Brennen College
Dharmadam, Thalassery
Kannur, Pin : 670106
Ph: 04952990494
email:rctdirector@sgou.ac.in

Pattambi
Sree Neelakanta Govt. Sanskrit College
Pattambi, Palakkad
Kerala, Pin : 679303
Ph: 04662912009
email:rcpdirector@sgou.ac.in

Kozhikode
Govt. Arts and Science College
Meenchantha, Kozhikode,
Kerala, Pin : 673002
Ph: 04952920228
email:rckdirector@sgou.ac.in

SAY NO TO RAGGING

BEFORE YOU EVEN THINK OF RAGGING



Humiliation

Blacklisting

Ruined Career

Download

ANTI RAGGING

App

Don't just stand and watch. Stop Ragging! Show Character

Visit UGC Website i.e. www.ugc.ac.in & www.antiraging.in to UGC Anti Ragging regulations.

Ministry of Education
Government of India

Are You Being Ragged ?

विश्वविद्यालय अनुदान आयोग
University Grants Commission

SSGOU

DON'T LET IT BE TOO LATE

SAY NO TO DRUGS

LOVE YOURSELF AND ALWAYS BE HEALTHY



SREENARAYANAGURU OPEN UNIVERSITY

The State University for Education, Training and Research in Blended Format, Kerala

High Performance Computing

COURSE CODE: B21CA05DE

SGOU

ISBN 978-81-903304-7-3

Sreenarayanaguru Open University
Kollam, Kerala Pin- 691601, email: info@sgou.ac.in, www.sgou.ac.in Ph: +91 474 2966841