COURSE OBJECTIVE

This course will enable students to;

- Describe Computer Architecture
- Measure the performance of architectures in terms of right parameters
- Summarize parallel architecture and the software used for them

• • •

PRE REQUISITE(s)

- Basic Computer Architectures
- Microprocessors and Microcontrollers

COURSE OUTCOME

This students should be able to;

- Explain the concepts of parallel computing and hardware technologies
- Compare and contrast the parallel architectures
- Illustrate parallel programming concepts

. . .

COURSE APPLICATIONS

- Parallel Computing
- Research in hardware technologies
- Research in Parallel computing, etc.,

MODULE – 1

. . .

THEORY OF PARALLELISM

In this chapter...

- THE STATE OF COMPUTING
- MULTIPROCESSORS AND MULTICOMPUTERS
- MULTIVECTOR AND SIMD COMPUTERS
- PRAM AND VLSI MODELS
- ARCHITECTURAL DEVELOPMENT TRACKS

THE STATE OF COMPUTING
Computer Development Milestones

- How it all started…
  - 500 BC: Abacus (China) – The earliest mechanical computer/calculating device.
    - Operated to perform decimal arithmetic with carry propagation digit by digit
  - 1642: Mechanical Adder/Subtractor (Blaise Pascal)
  - 1827: Difference Engine (Charles Babbage)
  - 1941: First binary mechanical computer (Konrad Zuse; Germany)
  - 1944: Harvard Mark I (IBM)
    - The very first electromechanical decimal computer as proposed by Howard Aiken
- Computer Generations
  - 1^{st} 2^{nd} 3^{rd} 4^{th} 5^{th}
  - Division into generations marked primarily by changes in hardware and software technologies

THE STATE OF COMPUTING
Computer Development Milestones

- First Generation (1945 – 54)
  - **Technology & Architecture:**
    - Vacuum Tubes
    - Relay Memories
    - CPU driven by PC and accumulator
    - Fixed Point Arithmetic
  - **Software and Applications:**
    - Machine/Assembly Languages
    - Single user
    - No subroutine linkage
    - Programmed I/O using CPU
  - **Representative Systems:** ENIAC, Princeton IAS, IBM 701

THE STATE OF COMPUTING
Computer Development Milestones

- Second Generation (1955 – 64)
  - **Technology & Architecture:**
    - Discrete Transistors
    - Core Memories
    - Floating Point Arithmetic
    - I/O Processors
    - Multiplexed memory access
  - **Software and Applications:**
    - High level languages used with compilers
    - Subroutine libraries
    - Processing Monitor
  - **Representative Systems:** IBM 7090, CDC 1604, Univac LARC

THE STATE OF COMPUTING
Computer Development Milestones

- Third Generation (1965 – 74)
  - **Technology & Architecture:**
    - IC Chips (SSI/MSI)
    - Microprogramming
    - Pipelining
    - Cache
    - Look-ahead processors
  - **Software and Applications:**
    - Multiprogramming and Timesharing OS
    - Multiuser applications
  - **Representative Systems:** IBM 360/370, CDC 6600, T1-ASC, PDP-8

THE STATE OF COMPUTING
Computer Development Milestones

- Fourth Generation (1975 – 90)
  - **Technology & Architecture:**
    - LSI/VLSI
    - Semiconductor memories
    - Multiprocessors
    - Multi-computers
    - Vector supercomputers
  - **Software and Applications:**
    - Multiprocessor OS
    - Languages, Compilers and environment for parallel processing
  - **Representative Systems:** VAX 9000, Cray X-MP, IBM 3090

THE STATE OF COMPUTING
Computer Development Milestones

- Fifth Generation (1991 onwards)
  - **Technology & Architecture:**
    - Advanced VLSI processors
    - Scalable Architectures
    - Superscalar processors
  - **Software and Applications:**
    - Systems on a chip
    - Massively parallel processing
    - Grand challenge applications
    - Heterogeneous processing
  - **Representative Systems:** S-81, IBM ES/9000, Intel Paragon, nCUBE 6480, MPP, VPP500

THE STATE OF COMPUTING
Elements of Modern Computers

- Computing Problems
- Algorithms and Data Structures
- Hardware Resources
- Operating System
- System Software Support
- Compiler Support

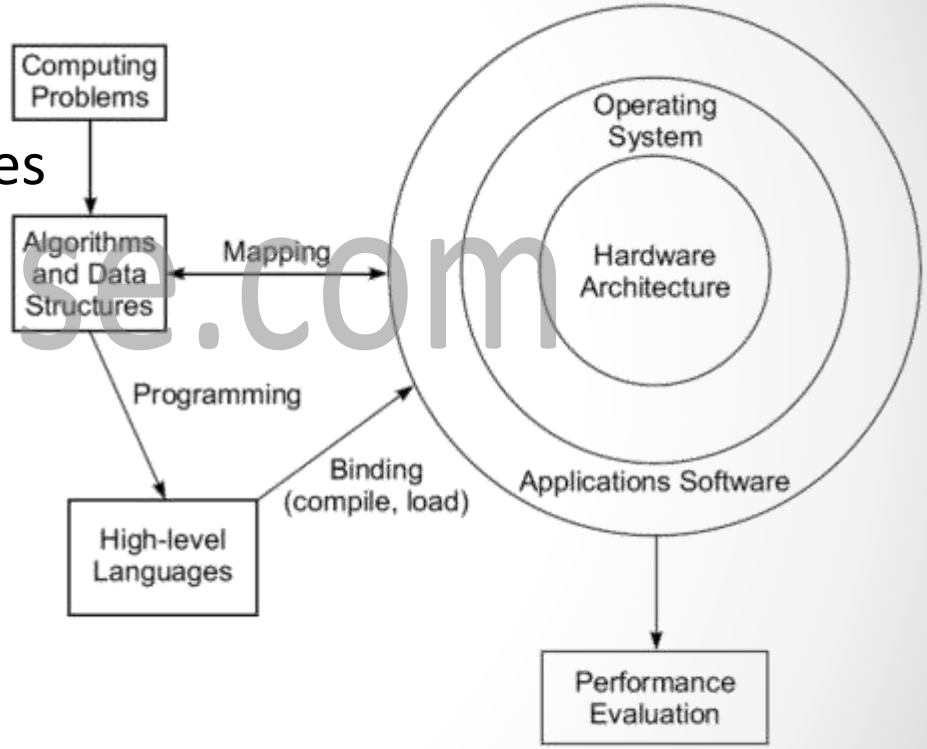
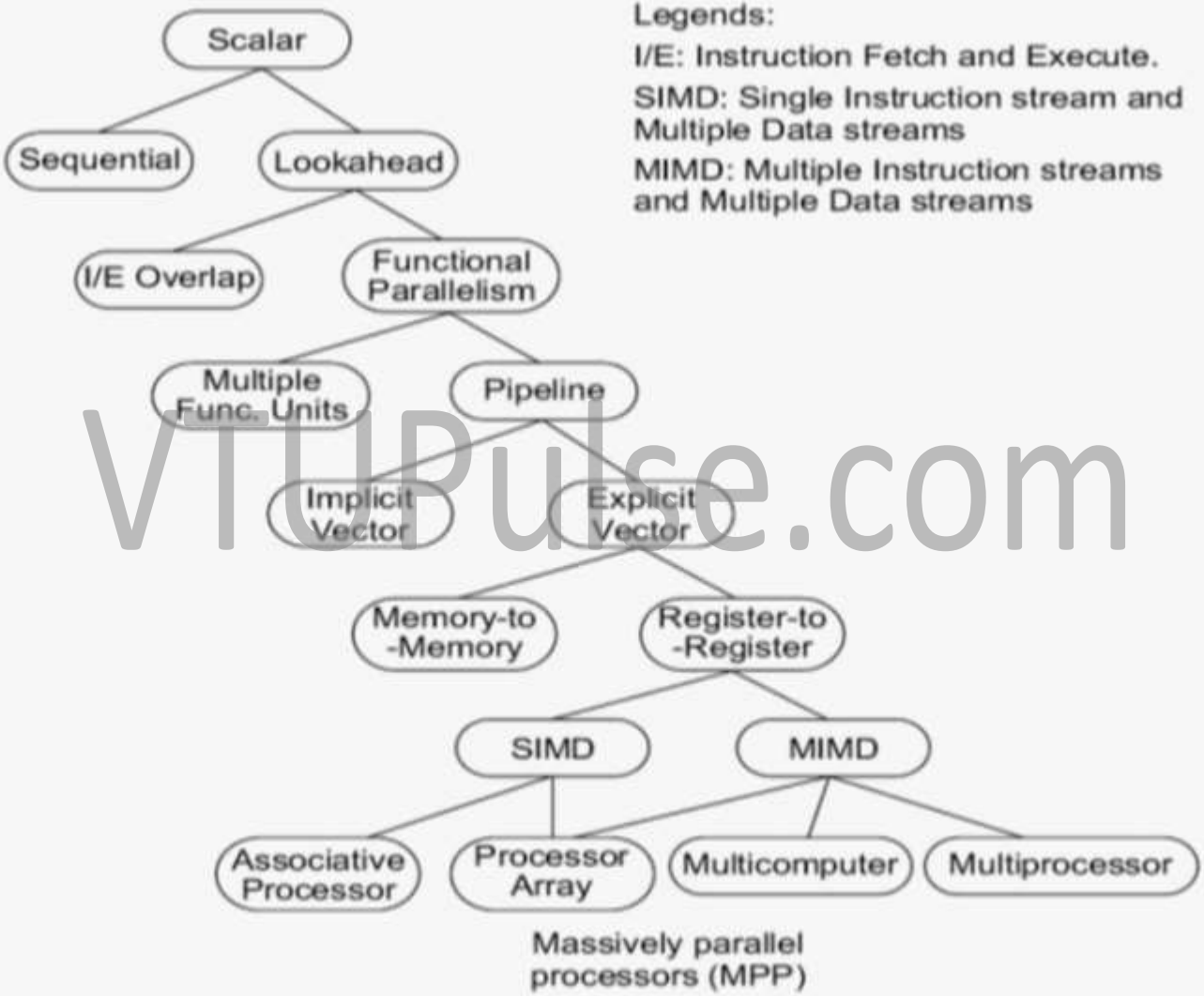


Fig. 1.1 Elements of a modern computer system

THE STATE OF COMPUTING
Evolution of Computer Architecture

- The study of computer architecture involves both the following:
  - Hardware organization
  - Programming/software requirements
- The evolution of computer architecture is believed to have started with **von Neumann** architecture
  - Built as a **sequential machine**
  - Executing **scalar data**
- Major leaps in this context came as…
  - Look-ahead, parallelism and pipelining
  - Flynn's classification
  - Parallel/Vector Computers
  - Development Layers



THE STATE OF COMPUTING
Evolution of Computer Architecture

Flynn's Classification of Computer Architectures

In 1966, Michael Flynn proposed a classification for computer architectures based on the number of instruction steams and data streams (Flynn's Taxonomy).

- Flynn uses the stream concept for describing a machine's structure.
- A stream simply means a sequence of items (data or instructions).

THE STATE OF COMPUTING
Evolution of Computer Architecture

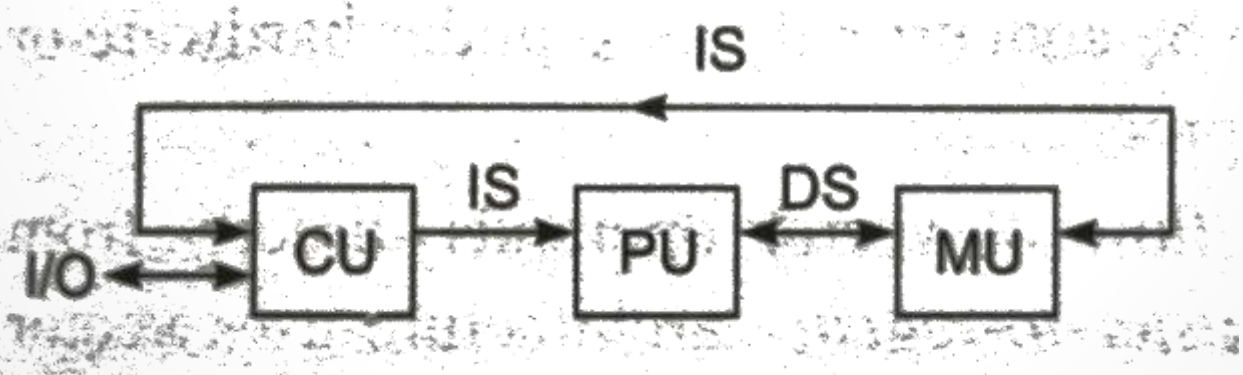
Flynn's Taxonomy

- SISD: Single instruction single data
  - Classical von Neumann architecture
- SIMD: Single instruction multiple data
- MIMD: Multiple instructions multiple data
  - Most common and general parallel machine
- MISD: Multiple instructions single data

THE STATE OF COMPUTING
Evolution of Computer Architecture

SISD

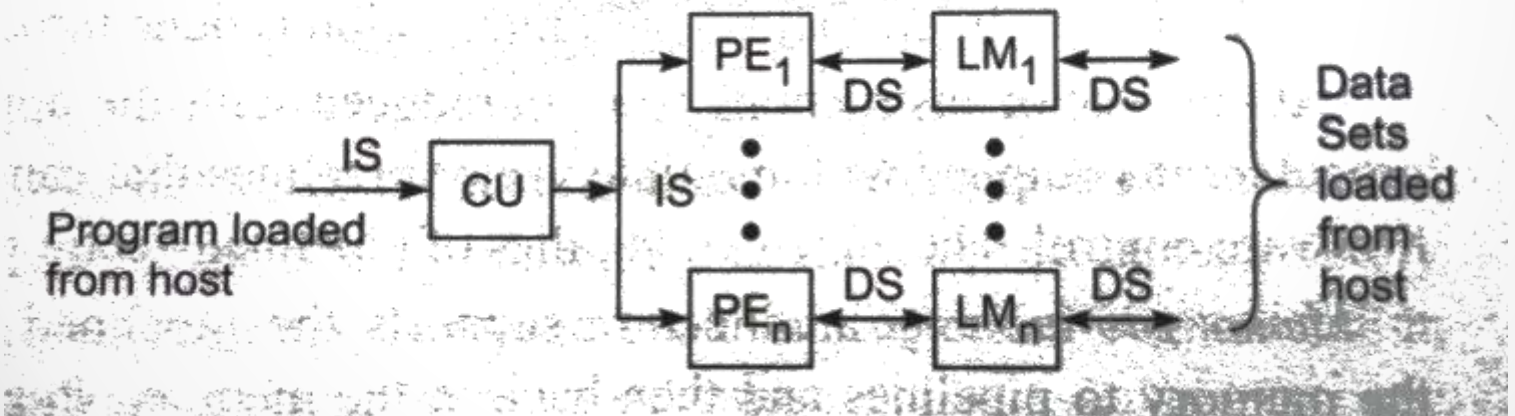
- SISD (Singe-Instruction stream, Singe-Data stream)
- SISD corresponds to the traditional mono-processor (von Neumann computer). A single data stream is being processed by one instruction stream
- A single-processor computer (uni-processor) in which a single stream of instructions is generated from the program.



THE STATE OF COMPUTING
Evolution of Computer Architecture

SIMD

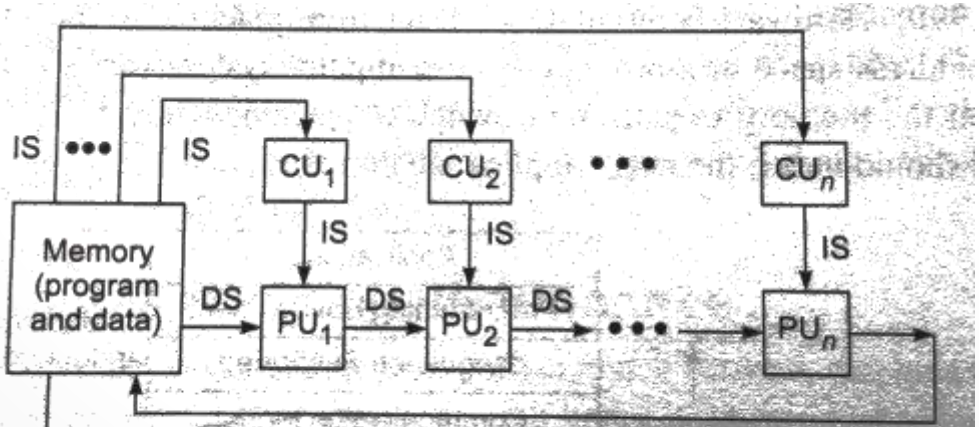
- SIMD (Single-Instruction stream, Multiple-Data streams)
- Each instruction is executed on a different set of data by different processors i.e multiple processing units of the same type process on multiple-data streams.
- This group is dedicated to array processing machines.
- Sometimes, vector processors can also be seen as a part of this group.



THE STATE OF COMPUTING
Evolution of Computer Architecture

MIMD

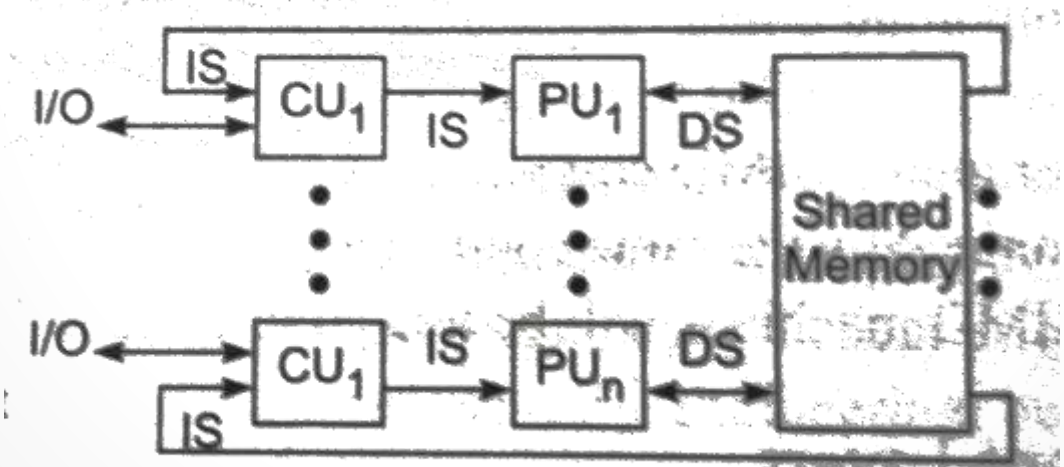
- MIMD (Multiple-Instruction streams, Multiple-Data streams)
- Each processor has a separate program.
- An instruction stream is generated from each program.
- Each instruction operates on different data.
- This last machine type builds the group for the traditional multi-processors.
- Several processing units operate on multiple-data streams



THE STATE OF COMPUTING
Evolution of Computer Architecture

MISD

- MISD (Multiple-Instruction streams, Singe-Data stream)
- Each processor executes a different sequence of instructions.
- In case of MISD computers, multiple processing units operate on one single-data stream .
- In practice, this kind of organization has never been used



THE STATE OF COMPUTING
Evolution of Computer Architecture

Development Layers

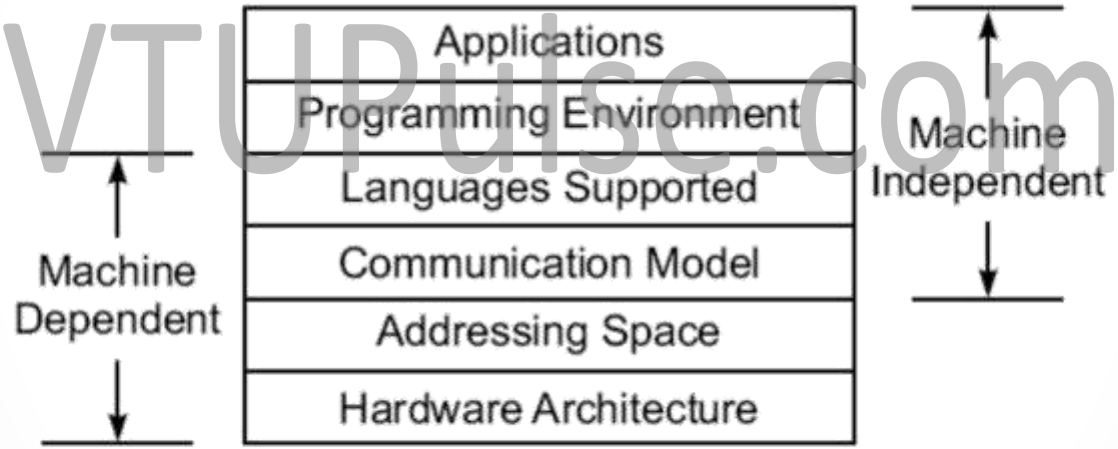


Fig. 1.4 Six layers for computer system development (Courtesy of Lionel Ni, 1990)

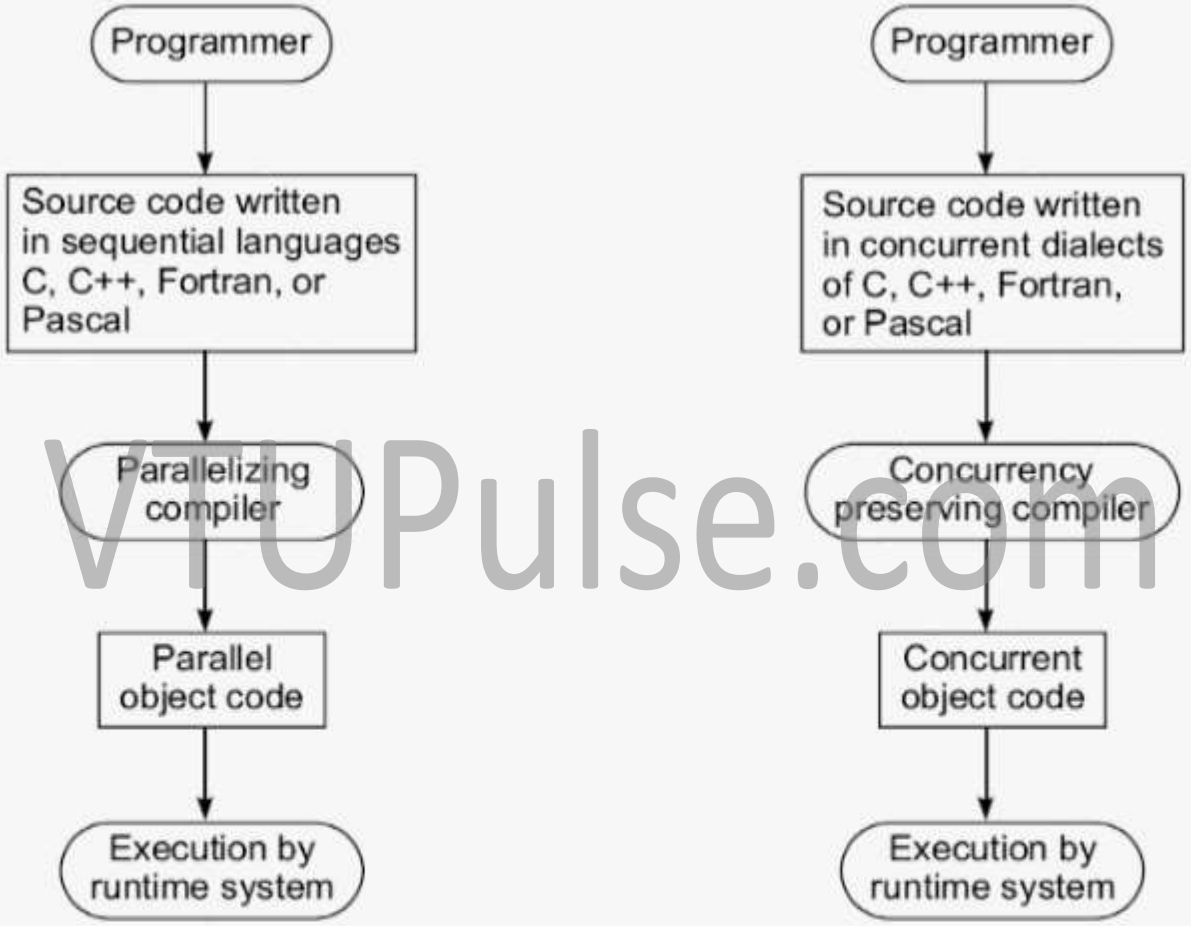


Fig. 1.5 Two approaches to parallel programming (Courtesy of Charles Seitz; adapted with permission from

THE STATE OF COMPUTING
System Attributes to Performance

- Machine Capability and Program Behaviour
  - Better Hardware Technology
  - Innovative Architectural Features
  - Efficient Resource Management
  - Algorithm Design
  - Data Structures
  - Language efficiency
  - Programmer Skill
  - Compiler Technology
- Peak Performance
  - Is impossible to achieve
- Measurement of Program Performance: **Turnaround time**
  - Disk and Memory Access
  - Input and Output activities
  - Compilation time
  - OS Overhead
  - CPU time

THE STATE OF COMPUTING
System Attributes to Performance

- Cycle Time (t), Clock Rate (f) and Cycles Per Instruction (CPI)
- Performance Factors
  - Instruction Count, Average CPI, Cycle Time, Memory Cycle Time and No. of memory cycles
- MIPS Rate and Throughput Rate
- Programming Environments – Implicit and Explicit Parallelism

THE STATE OF COMPUTING
System Attributes to Performance

- Cycle Time (processor) τ
- Clock Rate $f = 1/\tau$
- Average no. of cycles per instruction CPI
- No. of instructions in program I_c
- CPU Time $T = I_c \times CPI \times \tau$
- Memory Cycle Time $k\tau$
- No. of Processor Cycles needed p
- No. of Memory Cycles needed m
- Effective CPU Time $T = I_c \times (p + m \times k) \times \tau$

THE STATE OF COMPUTING
System Attributes to Performance

- MIPS Rate
- $\tau = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6} = \frac{f \times I_c}{C \times 10^6}$
- Throughput Rate
- $Wp = \frac{MIPS \times 10^6}{I_c} = \frac{f}{CPI \times I_c}$

THE STATE OF COMPUTING
System Attributes to Performance

- *A benchmark program contains 450000 arithmetic instructions, 320000 data transfer instructions and 230000 control transfer instructions. Each arithmetic instruction takes 1 clock cycle to execute whereas each data transfer and control transfer instruction takes 2 clock cycles to execute. On a 400 MHz processors, determine:*
  - *Effective no. of cycles per instruction (CPI)*
  - *Instruction execution rate (MIPS rate)*
  - *Execution time for this program*

THE STATE OF COMPUTING
System Attributes to Performance

	Performance Factors				
System Attributes	Instruction Count (I_c)	Average Cycles per Instruction (CPI)			Processor Cycle Time (τ)
		Processor Cycles per Instruction (CPI and p)	Memory References per Instruction (m)	Memory Access Latency (k)	
Instruction-set Architecture	☑	☑			
Compiler Technology	☑	☑	☑		
Processor Implementation and Control		☑			☑
Cache and Memory				☑	☑

Multiprocessors and Multicomputers

- Shared Memory Multiprocessors
  - The Uniform Memory Access (UMA) Model
  - The Non-Uniform Memory Access (NUMA) Model
  - The Cache-Only Memory Access (COMA) Model
  - The Cache-Coherent Non-Uniform Memory Access (CC-NUMA) Model
- Distributed-Memory Multicomputers
  - The No-Remote-Memory-Access (NORMA) Machines
  - Message Passing Multicomputers
- Taxonomy of MIMD Computers
- Representative Systems
  - **Multiprocessors:** BBN TC-200, MPP, S-81, IBM ES/9000 Model 900/VF,
  - **Multicomputers:** Intel Paragon XP/S, nCUBE/2 6480, SuperNode 1000, CM5, KSR-1

Multiprocessors and Multicomputers

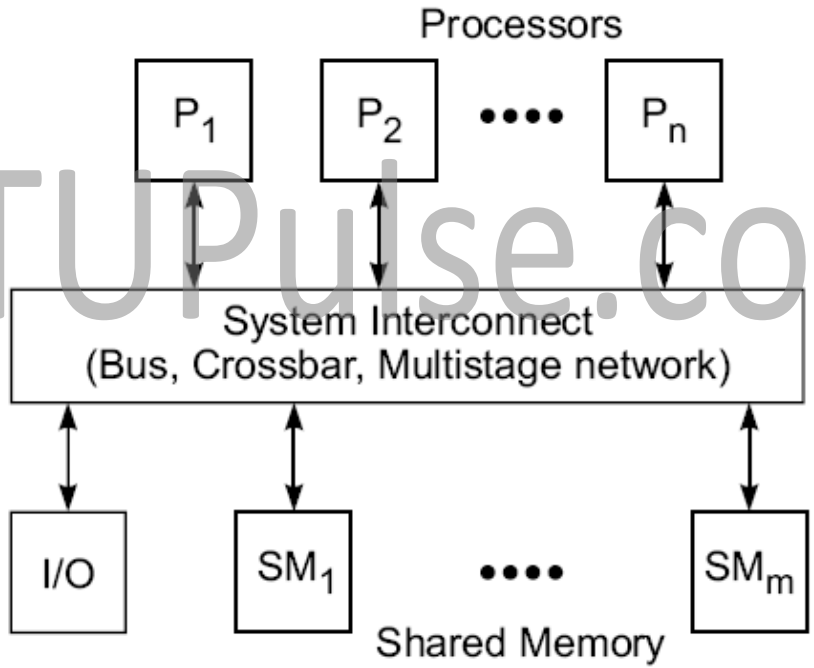
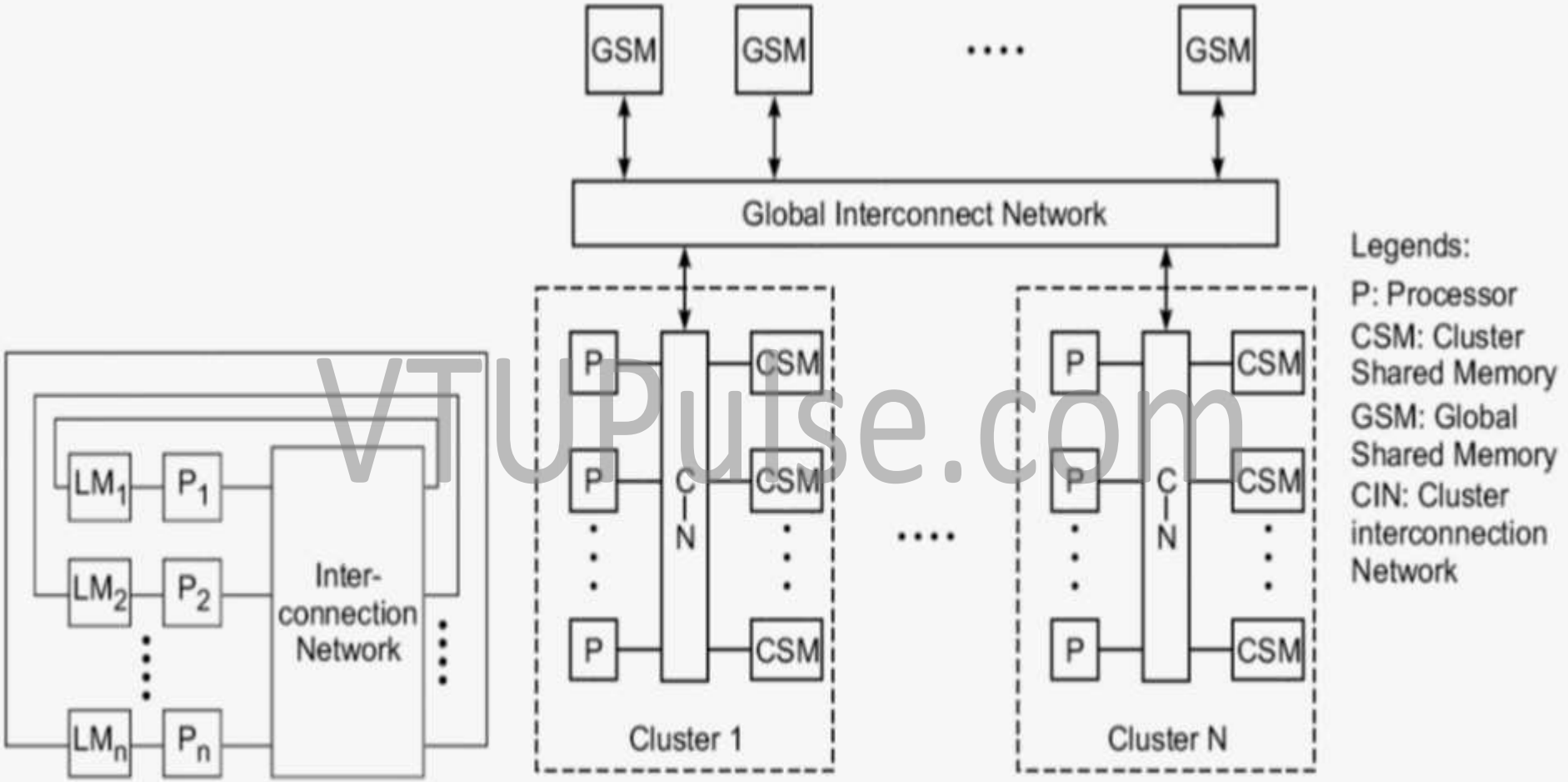


Fig. 1.6 The UMA multiprocessor model



(a) Shared local memories (e.g. the N Butterfly)

(b) A hierarchical cluster model (e.g. the Cedar system at the University of Illinois)

Multiprocessors and Multicomputers

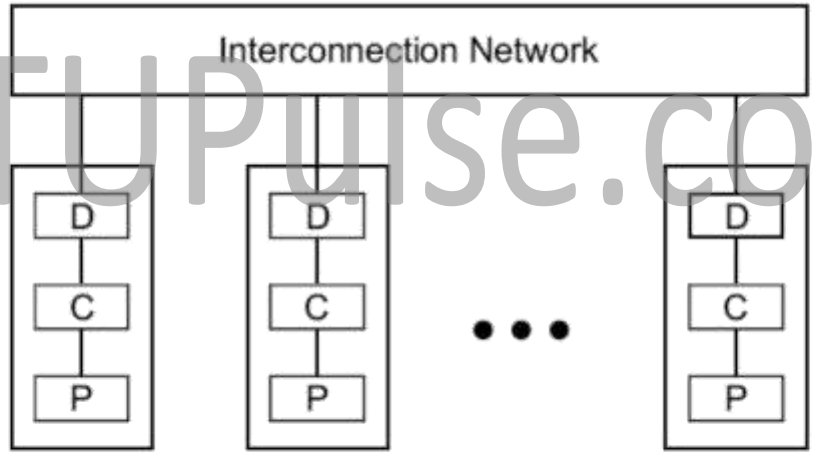


Fig. 1.8 The COMA model of a multiprocessor (P: Processor, C: Cache, D: Directory; e.g. the KSR-1)

Multiprocessors and Multicomputers

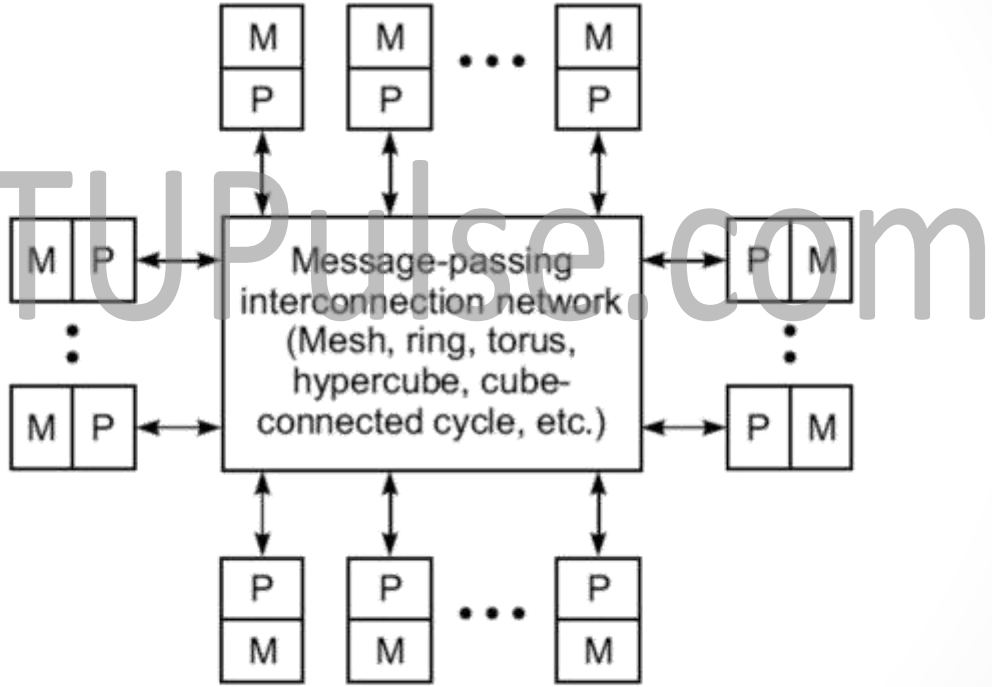
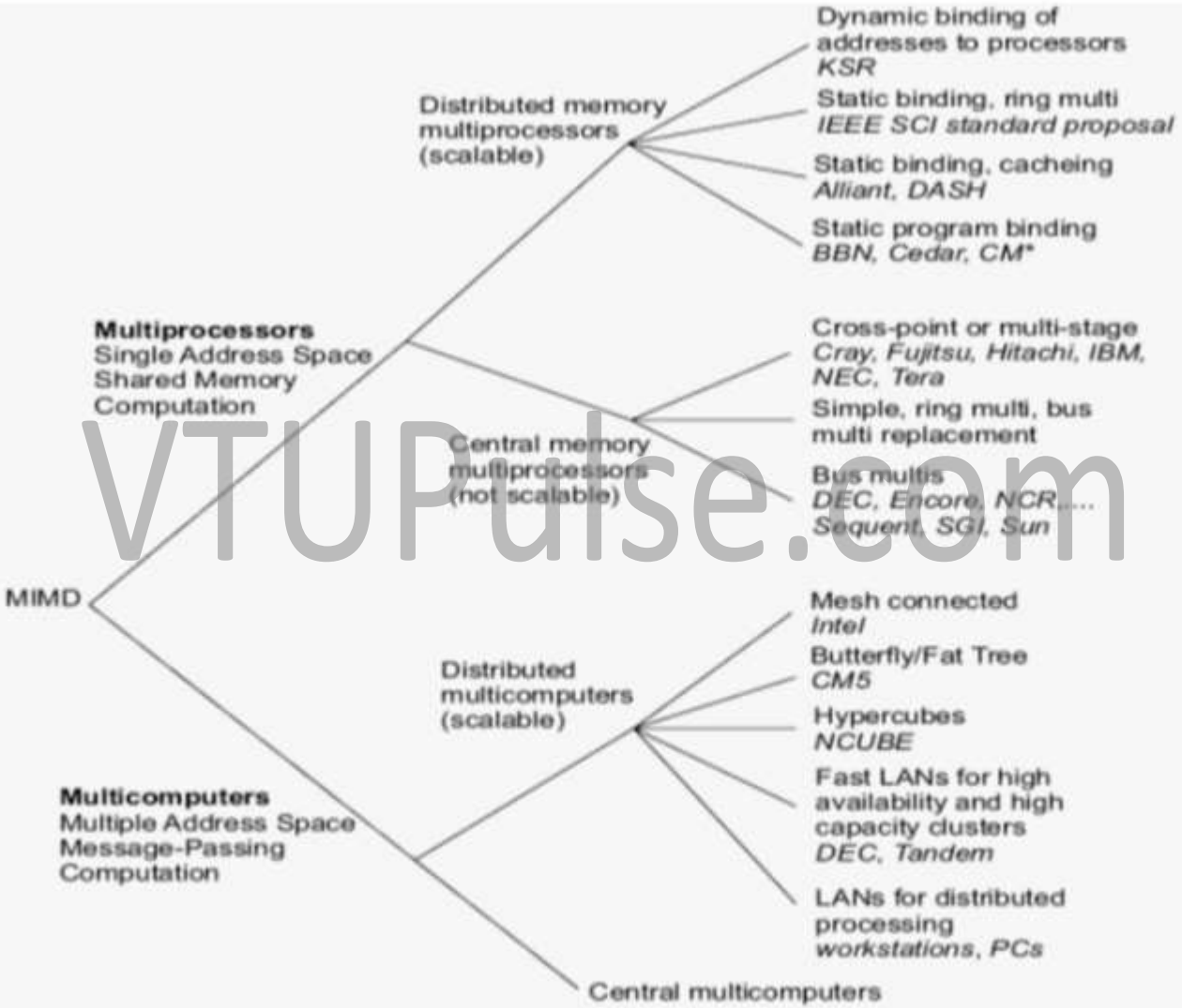
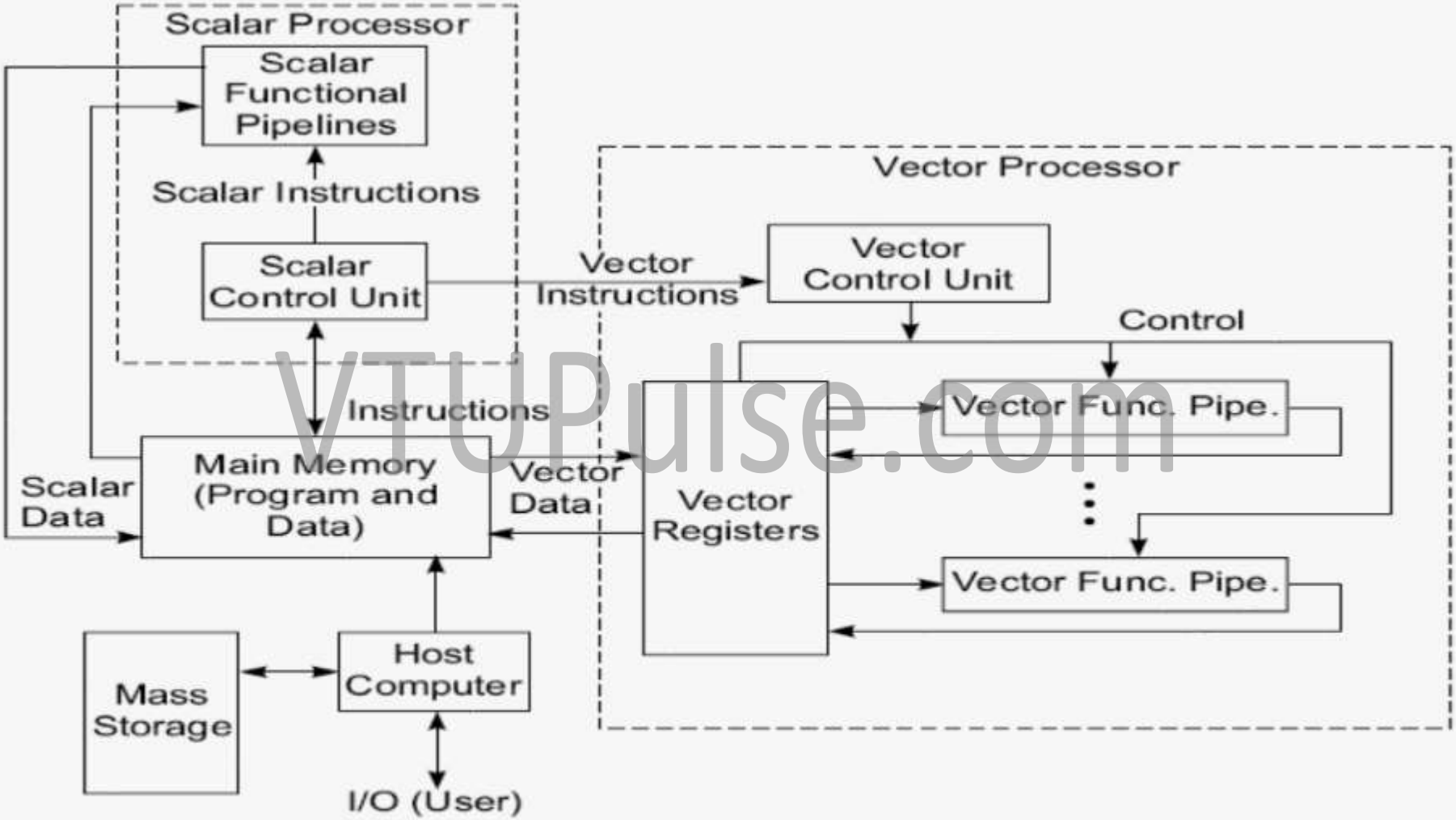


Fig. 1.9 Generic model of a message-passing multicomputer



Multivector and SIMD Computers

- Vector Processors
  - **Vector Processor Variants**
    - Vector Supercomputers
    - Attached Processors
  - **Vector Processor Models/Architectures**
    - Register-to-register architecture
    - Memory-to-memory architecture
  - **Representative Systems:**
    - **Cray-I**
    - **Cray Y-MP** (2,4, or 8 processors with 16Gflops peak performance)
    - **Convex C1, C2, C3 series** (C3800 family with 8 processors, 4 GB main memory, 2 Gflops peak performance)
    - **DEC VAX 9000** (pipeline chaining support)



Multivector and SIMD Computers

- SIMD Supercomputers
  - SIMD Machine Model
    - **S = < N, C, I, M, R >**
    - N: No. of PEs in the machine
    - C: Set of instructions (scalar/program flow) directly executed by control unit
    - I: Set of instructions broadcast by CU to all PEs for parallel execution
    - M: Set of masking schemes
    - R: Set of data routing functions
  - **Representative Systems:**
    - **MasPar MP-1** (1024 to 16384 PEs)
    - **CM-2** (65536 PEs)
    - **DAP600 Family** (up to 4096 PEs)
    - **Illiac-IV** (64 PEs)

Multivector and SIMD Computers

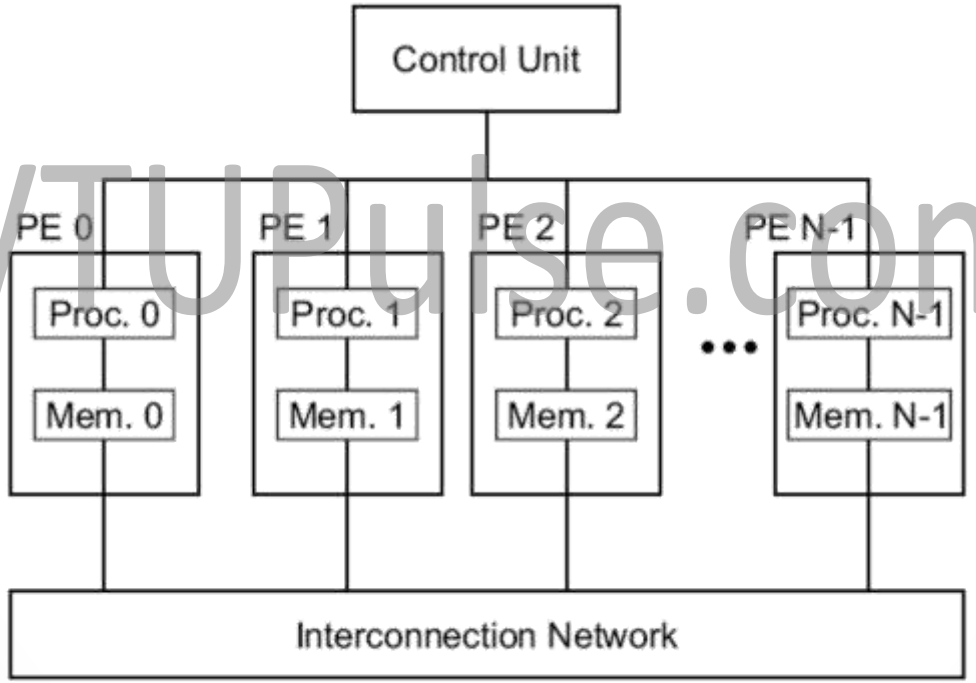


Fig. 1.12 Operational model of SIMD computers

PRAM and VLSI Models

- Parallel Random Access Machines
  - Time and Space Complexities
    - Time complexity
    - Space complexity
    - Serial and Parallel complexity
    - Deterministic and Non-deterministic algorithm
  - PRAM
    - Developed by Fortune and Wyllie (1978)
    - Objective:
      - Modelling idealized parallel computers with zero synchronization or memory access overhead
    - An n-processor PRAM has a globally addressable Memory

PRAM and VLSI Models

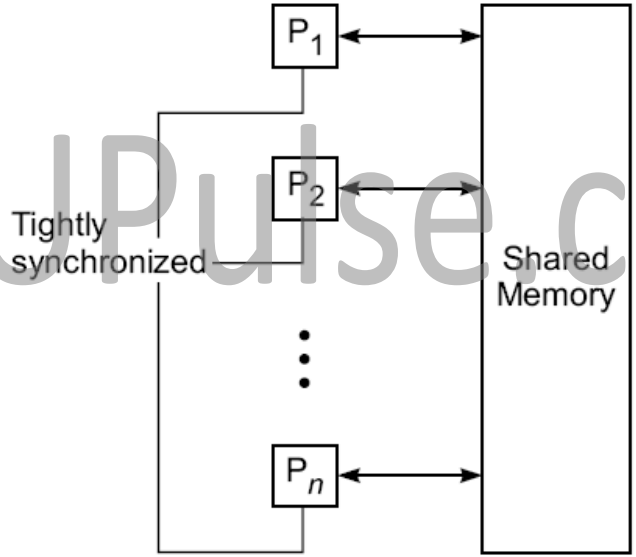


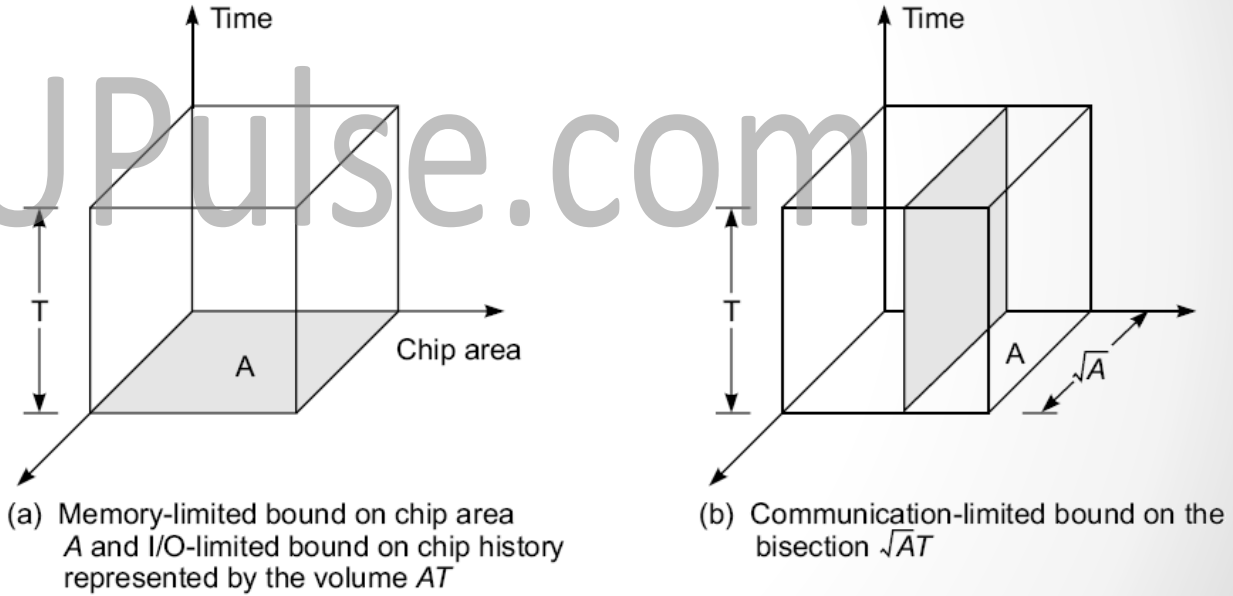
Fig. 1.14 PRAM model of a multiprocessor system with shared memory, on which all *n* processors operate in lockstep in memory access and program execution operations. Each processor can access any memory location in unit time

PRAM and VLSI Models

- Parallel Random Access Machines
  - PRAM Variants
    - **EREW**-PRAM Model
    - CREW-PRAM Model
    - ERCW-PRAM Model
    - **CRCW**-PRAM Model
  - Discrepancy with Physical Models
    - Most popular variants: EREW and CRCW
    - SIMD machine with shared architecture is closest architecture modelled by PRAM
    - PRAM Allows different instructions to be executed on different processors simultaneously. Thus, PRAM really operates in synchronized MIMD mode with shared memory

PRAM and VLSI Models

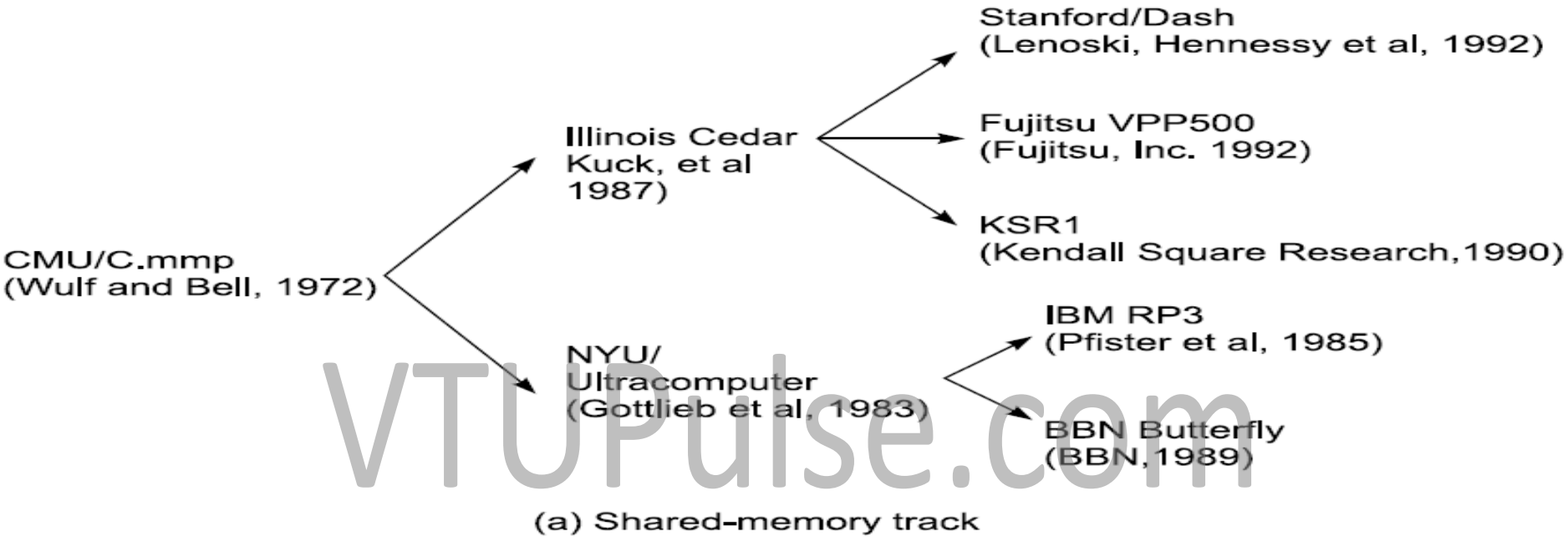
- VLSI Complexity Model
  - The AT^2 Model
    - Memory Bound on Chip Area
    - I/O Bound on **Volume AT**
    - Bisection Communication Bound **(Cross-section area)** $\sqrt{A}\,T$
      - *Square of this area used as lower bound*



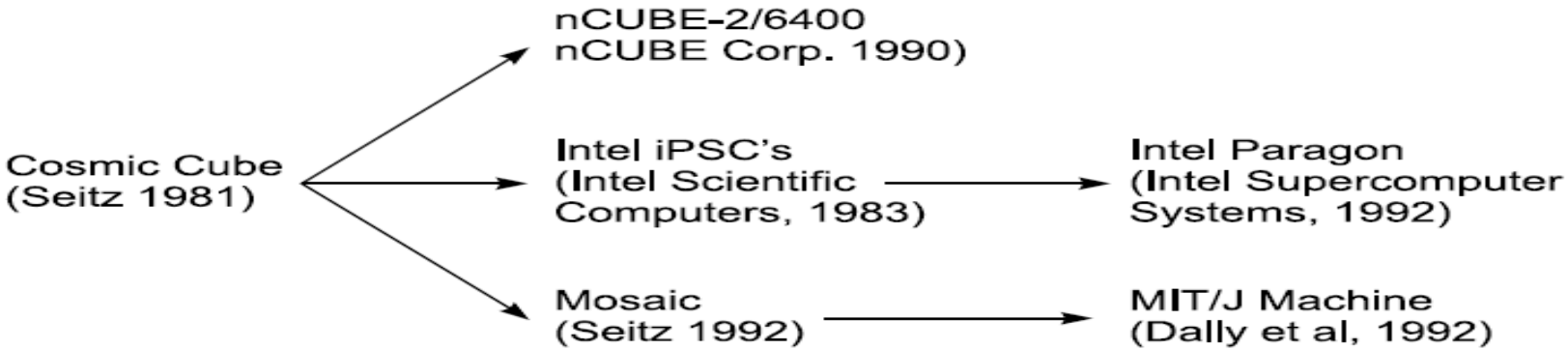
(a) Memory-limited bound on chip area A and I/O-limited bound on chip history represented by the volume AT

(b) Communication-limited bound on the bisection $\sqrt{A}T$

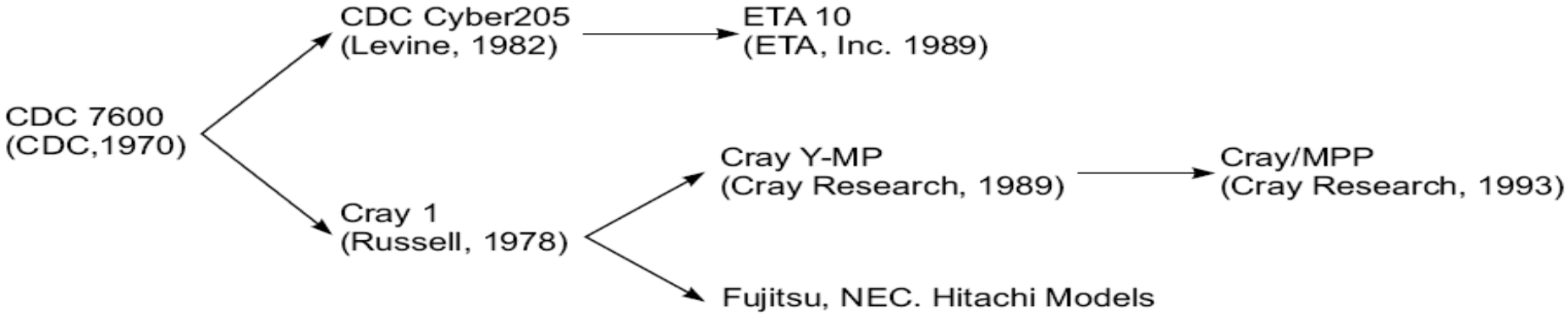
Fig. 1.15 The AT^2 complexity model of two-dimensional VLSI chips



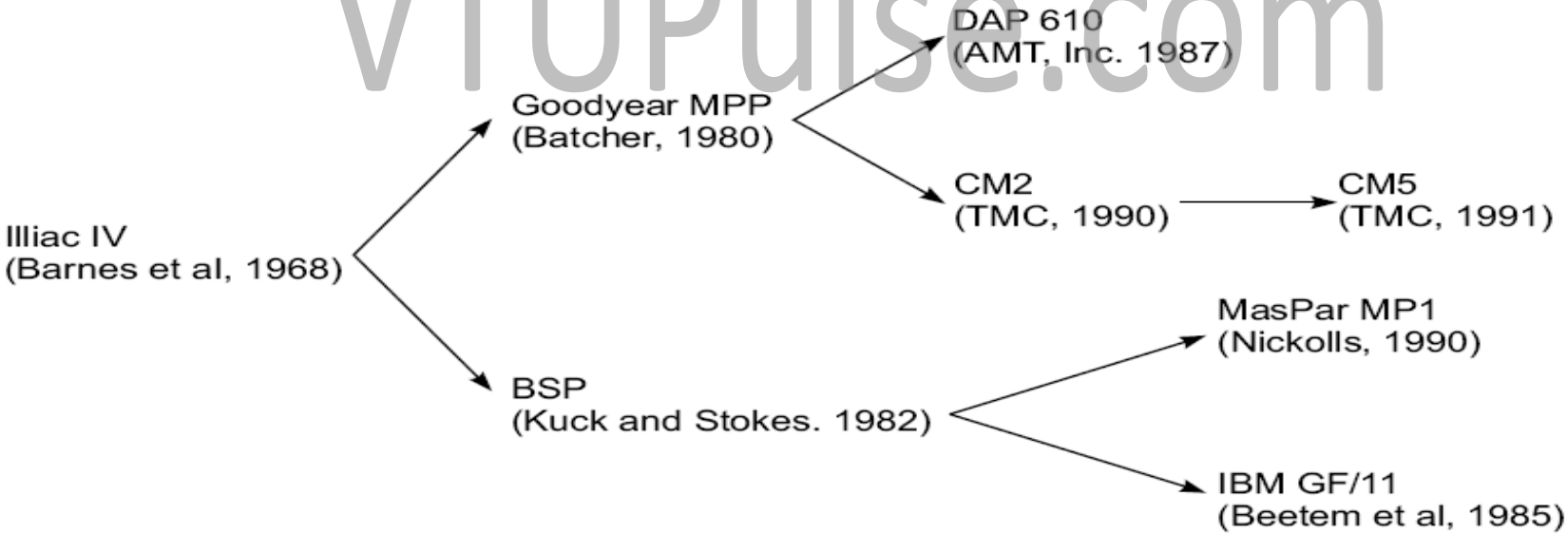
(a) Shared-memory track



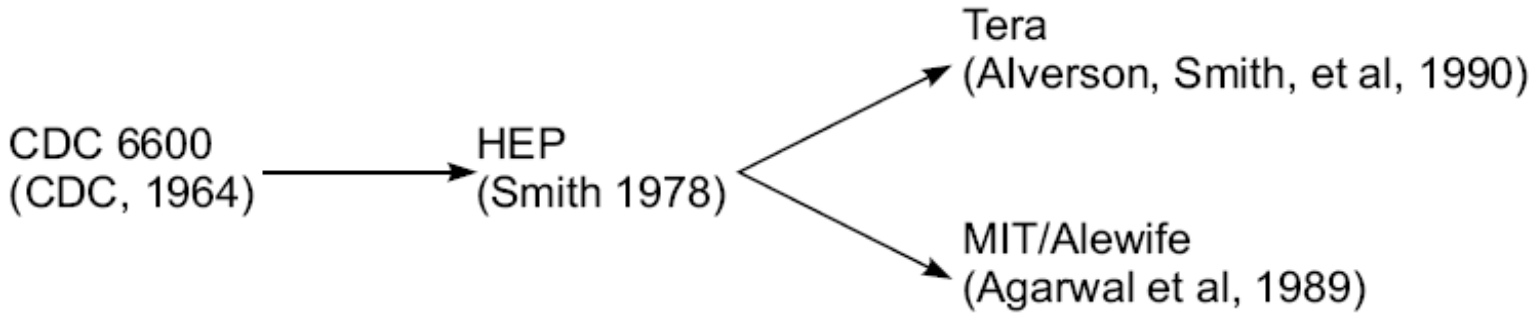
(b) Message-passing track



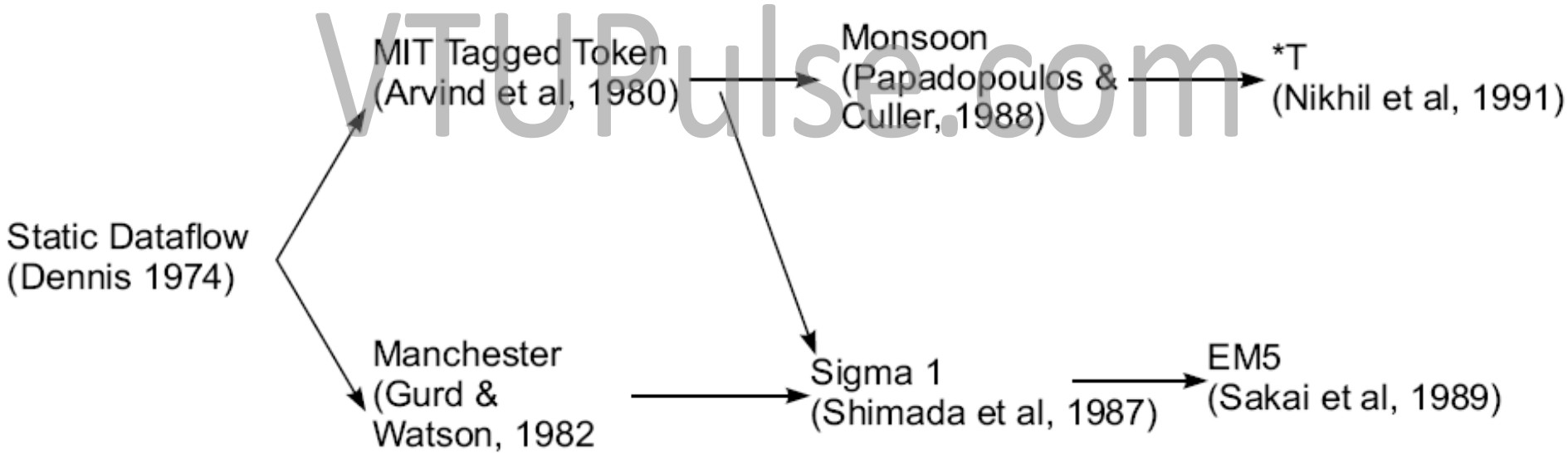
(a) Multivector track



(b) SIMD track



(a) Multithreaded track



(b) Dataflow track

Chapter 2

Program and Network Properties

Book: “Advanced Computer Architecture – Parallelism, Scalability, Programmability”, Hwang & Jotwani

In this chapter…

- CONDITIONS OF PARALLELISM
- PROBLEM PARTITIONING AND SCHEDULING
- PROGRAM FLOW MECHANISMS
- SYSTEM INTERCONNECT ARCHITECTURES

Conditions of Parallelism

- Key areas which need significant progress in parallel processing:
  - **Computational Models** for parallel computing
  - **Inter-processor communication** for parallel architectures
  - **System Integration** for incorporating parallel systems into general computing environment
- Various forms of parallelism can be attributed to:
  - Levels of Parallelism
  - Computational Granularity
  - Time and Space Complexities
  - Communication Latencies
  - Scheduling Policies
  - Load Balancing

CONDITIONS OF PARALLELISM
Data and Resource Dependencies

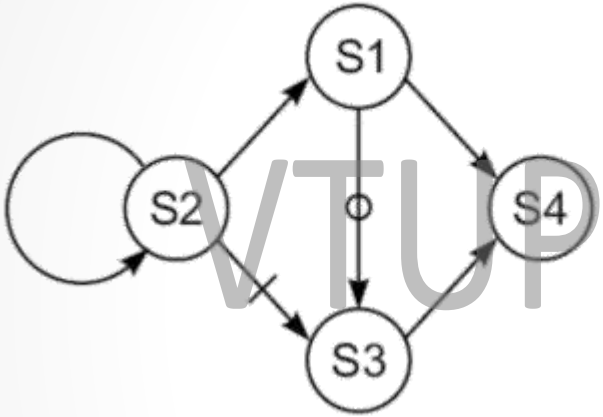
- Data Dependence
  - Flow Dependence
  - Anti-dependence
  - Output Dependence
  - I/O Dependence
  - Unknown Dependence
- Control Dependence
  - Control-dependence and control-independence
- Resource Dependence
  - ALU Dependence, Storage Dependence, etc.
- Bernstein's Conditions for Parallelism
- Dependence Graph

CONDITIONS OF PARALLELISM
Data and Resource Dependencies

- Consider the following examples and determine the types of dependencies and draw corresponding dependence graphs
- **Example 2.1(a):**
  - S1: R1 ← M[A]
  - S2: R2 ← R1 + R2
  - S3: R1 ← R3
  - S4: M[B] ← R1
- **Example 2.1(b):**
  - S1: Read array A from file F
  - S2: Process data
  - S3: Write array B into file F
  - S4: Close file F

CONDITIONS OF PARALLELISM

Data and Resource Dependencies



(a) Dependence graph

(b) I/O dependence caused by accessing the same file by the read and write state-ments

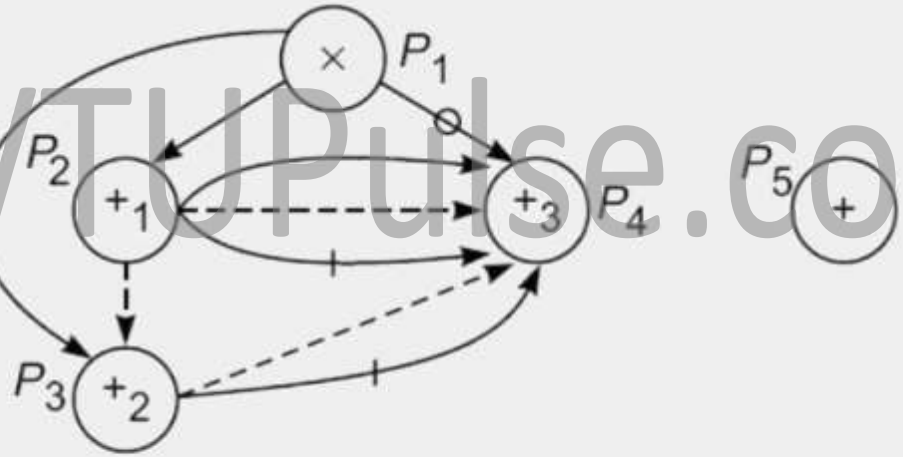
Fig. 2.1 Data and I/O dependences in the program of Example 2.1

CONDITIONS OF PARALLELISM
Data and Resource Dependencies

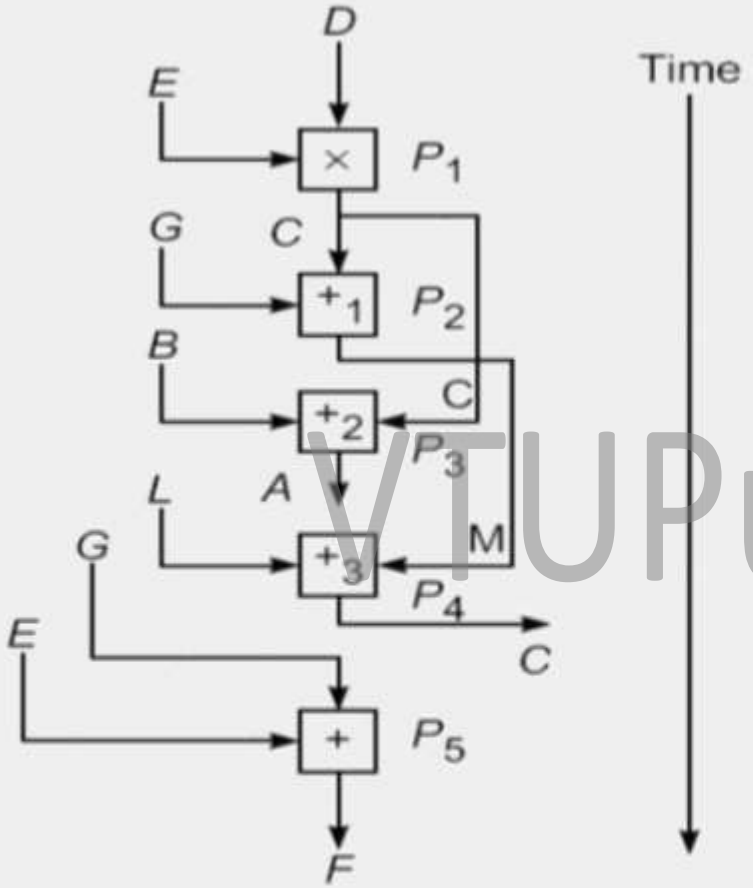
- **Example 2.2:** Determine the parallelism embedded in the following statements and draw the dependency graphs.
  - P1: C = D x E
  - P2: M = G + C
  - P3: A = B + C
  - P4: C = L + M
  - P5: F = G + E
- Also analyse the statements against Bernstein's Conditions.

CONDITIONS OF PARALLELISM

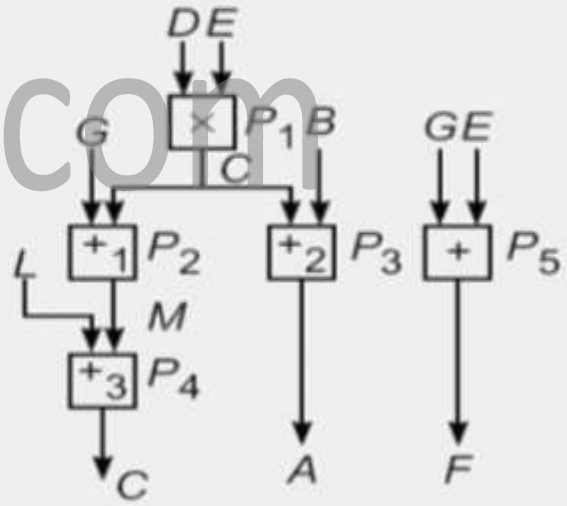
Data and Resource Dependencies



(a) A dependence graph showing both data dependence (solid arrows) and resource dependence (dashed arrows)



(b) Sequential execution in five steps, assuming one step per statement (no pipelining)



(c) Parallel execution in three steps, assuming two adders are available per step

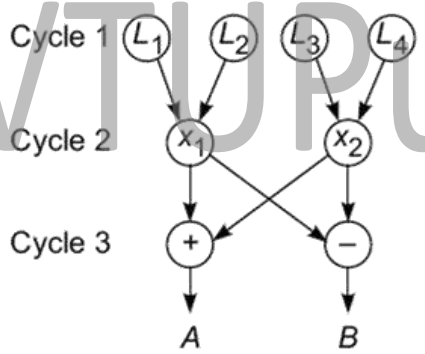
CONDITIONS OF PARALLELISM

Hardware and Software Parallelism

- Hardware Parallelism
  - Cost and Performance Trade-offs
  - Resource Utilization Patterns of simultaneously executable operations
  - **k-issue** processors and **one-issue** processor
  - **Representative systems:** Intel i960CA (3-issue), IBM RISC System 6000 (4-issue)
- Software Parallelism
  - Algorithm **x** Programming Style **x** Program Design
  - Program Flow Graphs
  - **Control Parallelism**
  - **Data Parallelism**
- Mismatch between Software Parallelism and Hardware Parallelism
  - **Role of Compilers** in resolving the mismatch

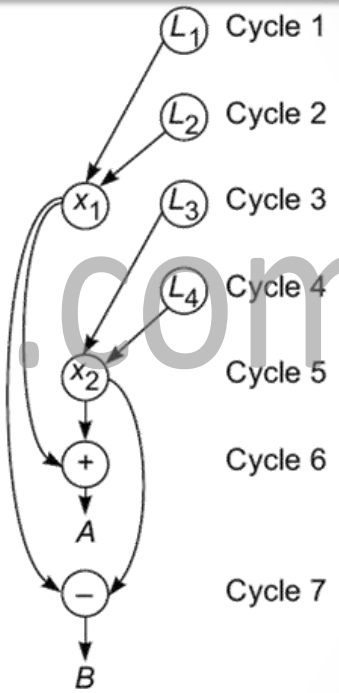
CONDITIONS OF PARALLELISM

Hardware and Software Parallelism



L_i : Load operation
X_i : Multiply operation

(a) Software parallelism



(b) Hardware parallelism

Fig. 2.3 Executing an example program by a two-issue superscalar processor

CONDITIONS OF PARALLELISM

Hardware and Software Parallelism

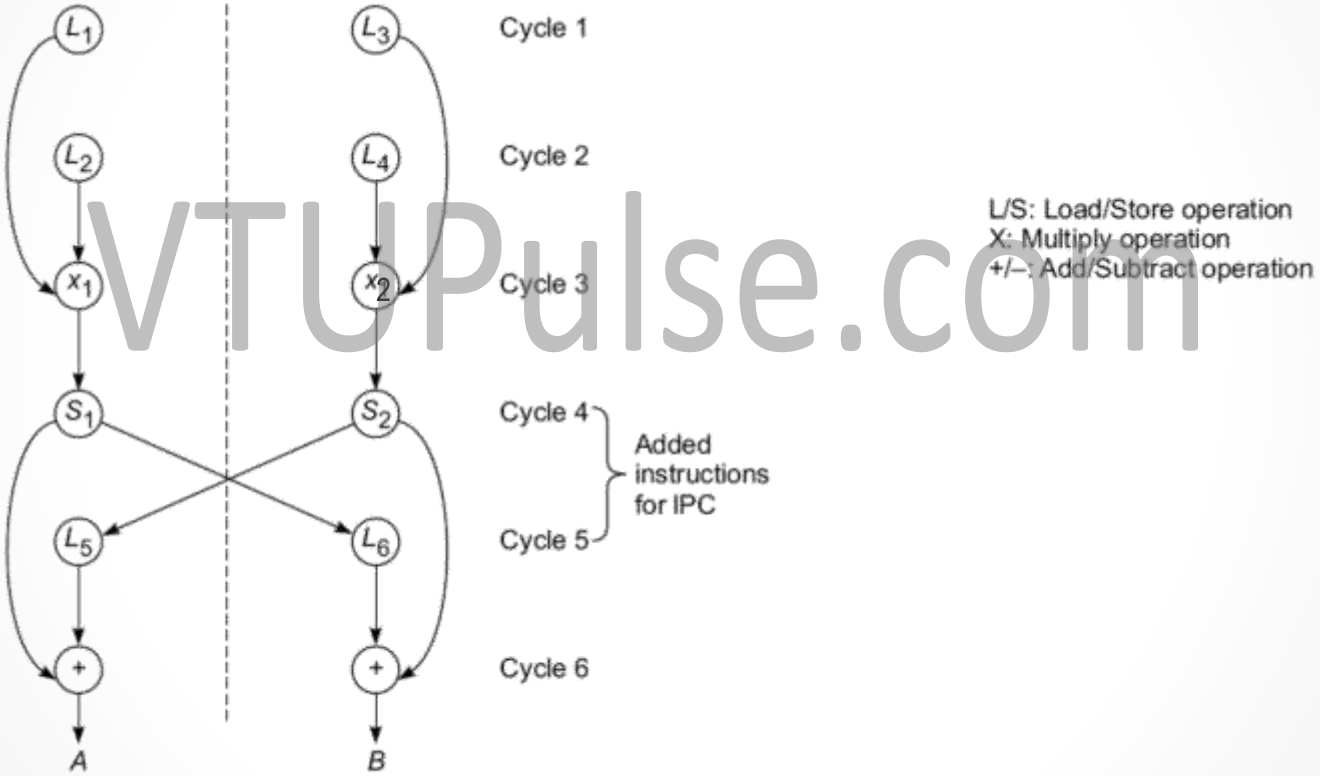


Fig. 2.4 Dual-processor execution of the program in Fig. 2.3a

PROGRAM PARTITIONING AND SCHEDULING

Grain Packing and Scheduling

- Fundamental Objectives
  - To partition program into parallel branches, program modules, micro-tasks or grains to yield shortest possible execution time
  - Determining the optimal size of grains in a computation
- Grain-size problem
  - To determine both the size and number of grains in parallel program
- Parallelism and scheduling/synchronization overhead trade-off
- **Grain Packing** approach
  - Introduced by Kruatrachue and Lewis (1988)
  - Grain size is measured by number of basic machine cycles (both processor and memory) needed to execute all operations within the node
  - A program graph shows the structure of a program

PROGRAM PARTITIONING AND SCHEDULING

Grain Packing and Scheduling

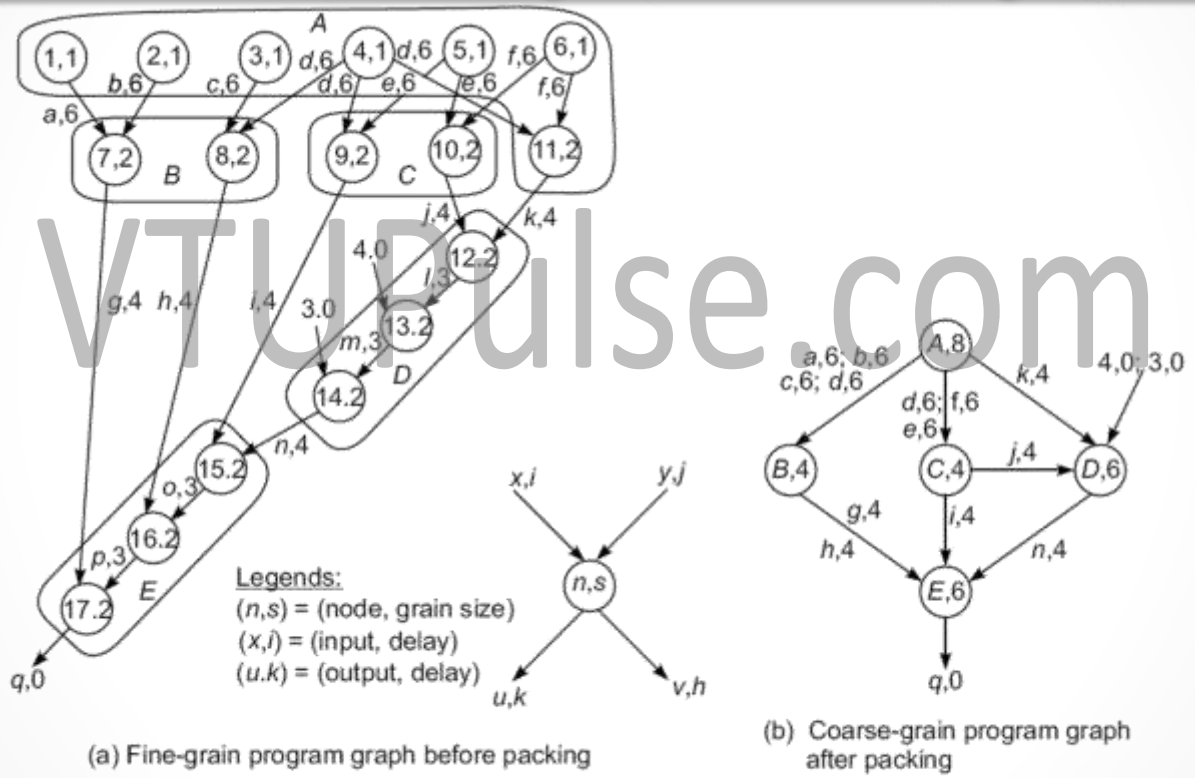
- Example 2.4: Consider the following program
  - **VAR** a, b, c, d, e, f, g, h, I, j, k, l, m, n, o, p, q
  - **BEGIN**

1) a = 1
2) b = 2
3) c = 3
4) d = 4
5) e = 5
6) f = 6
7) g = a x b
8) h = c x d
9) i = d x e
10) j = e x f
11) k = d x f
12) l = j x k
13) m =4 x 1
14) n = 3 x m
15) o = n x i
16) p = o x h
17) q = p x q

END

PROGRAM PARTITIONING AND SCHEDULING

Grain Packing and Scheduling



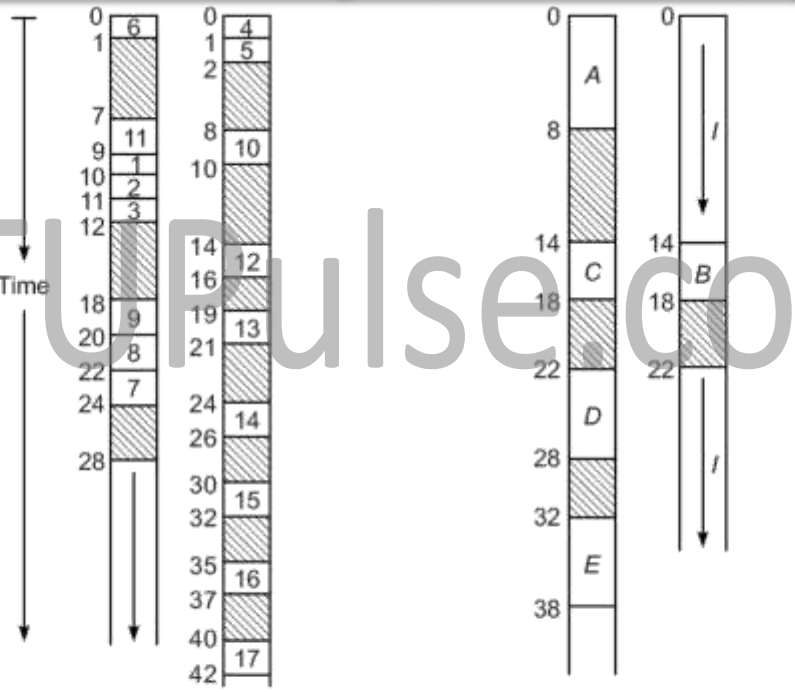
(a) Fine-grain program graph before packing

(b) Coarse-grain program graph after packing

Fig. 2.6 A program graph before and after grain packing in Example 2.4 (Modified from Kruatrachue and Lewis, *IEEE Software*, Jan. 1988)

PROGRAM PARTITIONING AND SCHEDULING

Grain Packing and Scheduling



(a) Fine grain (Fig. 2.6a) (b) Coarse grain (Fig. 2.6b)

Fig. 2.7 Scheduling of the fine-grain and coarse-grain programs (arrows: idle time; shaded areas: communication delays)

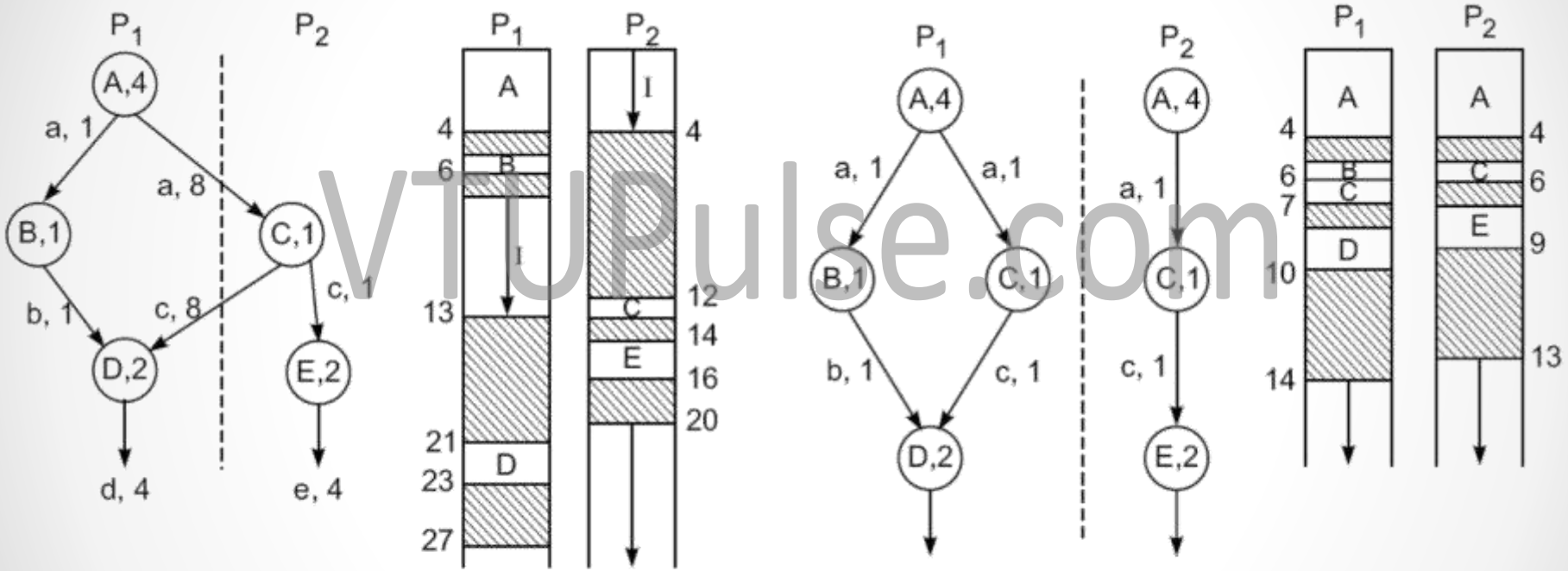
PROGRAM PARTITIONING AND SCHEDULING

Static Multiprocessor Scheduling

- **Node Duplication** scheduling
  - To eliminate idle time and further reduce communication delays among processor, nodes can be duplicated in more than one processors
- Grain Packing and Mode duplication are often used jointly to determine the best grain size and corresponding schedule.
- Steps involved in grain determination and process of scheduling optimization:
  - **Step 1:** Construct a fine-grain program graph
  - **Step 2:** Schedule the fine-grain computation
  - **Step 3:** Perform grain packing to produce the coarse grains
  - **Step 4:** Generate a parallel schedule based on the packed graph

PROGRAM PARTITIONING AND SCHEDULING

Static Multiprocessor Scheduling



(a) Schedule without node duplication

(b) Schedule with node duplication (A → A and A′; C → C and C′)

Fig. 2.8 Node-duplication scheduling to eliminate communication delays between processors (I: idle time; shaded areas: communication delays)

PROGRAM FLOW MECHANISM

- Control Flow v/s Data Flow
  - Control Flow Computer
  - Data Flow Computer
  - Example: **The MIT tagged-token data flow computer (Arvind and his associates), 1986**
- Demand-Driven Mechanisms
  - Reduction Machine
  - Demand-driven computation
  - Eager Evaluation and Lazy Evaluation
  - Reduction Machine Models – **String Reduction** Model and **Graph Reduction** Model

PROGRAM FLOW MECHANISM

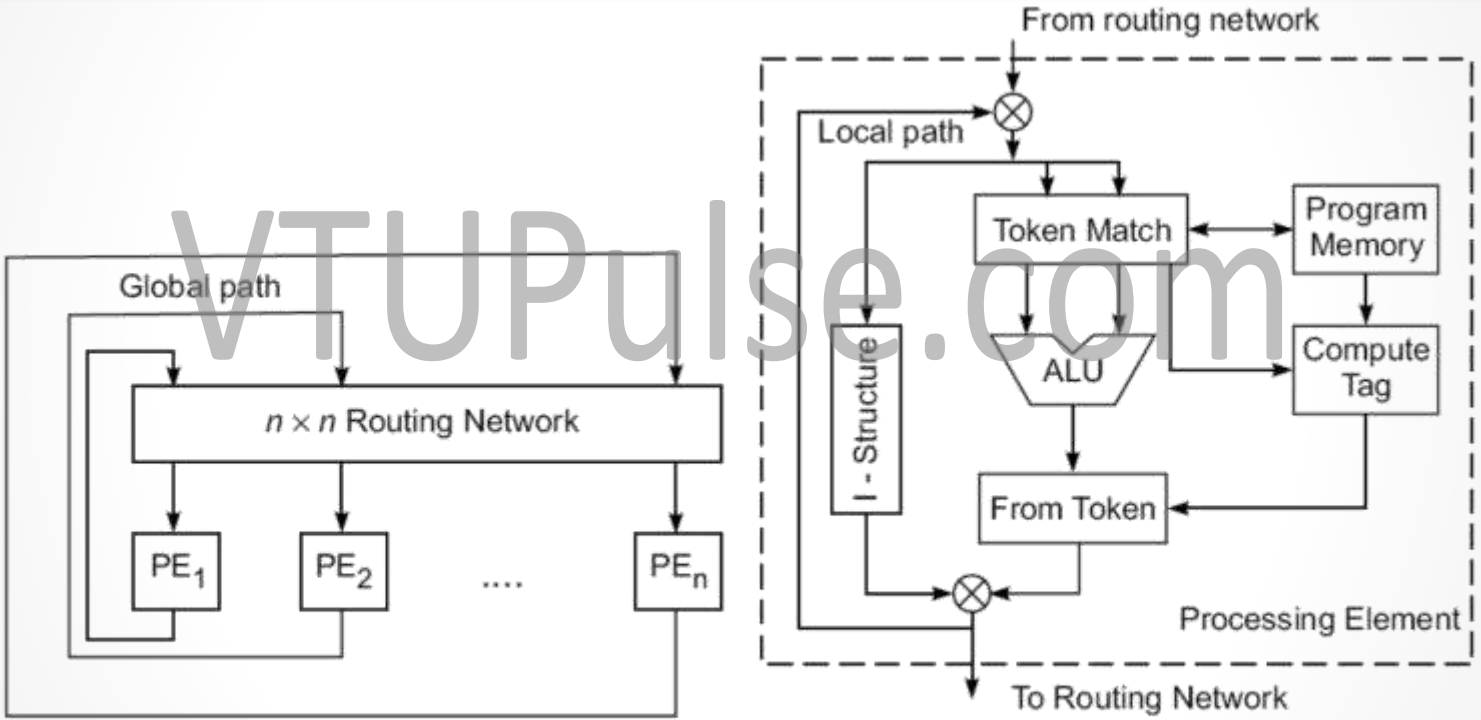
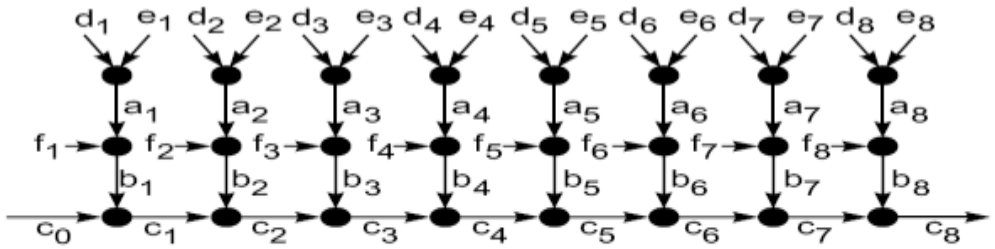


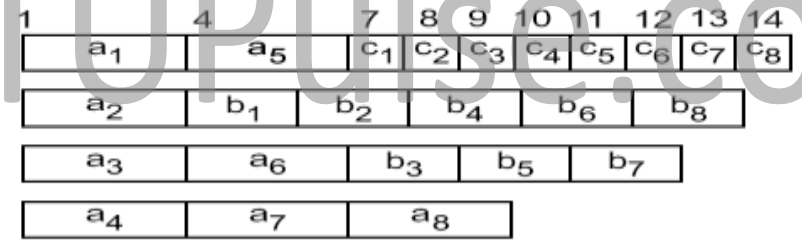
Fig. 2.12 The MIT tagged-token dataflow computer (adapted from Arvind and Iannucci, 1986 with permission)

```
input d, e, f
  c0 = 0
for i from 1 to 8 do
  begin
    ai := di + ei
    bi := ai * fi
    ci := bi + ci-1
  end
output a, b, c
```

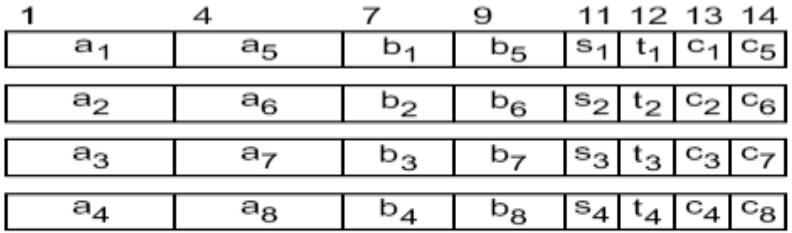


(a) A sample program and its dataflow graph

(b) Sequential execution on a uniprocessor in 48 cycles



(c) Data-driven execution on a 4-processor dataflow computer in 14 cycles



$s_1 = b_2 + b_1,\ t_1 = b_3 + s_1,\ c_1 = b_1 + c_0,\ c_5 = b_5 + c_4$

$s_2 = b_4 + b_3,\ t_2 = s_1 + s_2,\ c_2 = s_1 + c_0,\ c_6 = s_3 + c_4$

$s_3 = b_6 + b_5,\ t_3 = b_7 + s_3,\ c_3 = t_1 + c_0,\ c_7 = t_3 + c_4$

$s_4 = b_8 + b_7,\ t_4 = s_4 + s_3,\ c_4 = t_2 + c_0,\ c_8 = t_4 + c_4$

(d) Parallel execution on a shared-memory 4-processor system in 14 cycles

SYSTEM INTERCONNECT ARCHITECTURES

- Direct Networks for static connections
- Indirect Networks for dynamic connections
- Network Properties and Routing
  - Static Networks and Dynamic Networks
  - Network Size
  - Node Degree (in-degree and out-degree)
  - Network Diameter
  - Bisection Width
    - Channel width (w), Channel bisection width (b) , wire bisection width (B = b.w)
    - Wire length
    - Symmetric networks

SYSTEM INTERCONNECT ARCHITECTURES

- Network Properties and Routing
  - Data Routing Functions
    - Shifting
    - Rotation
    - Permutation
    - Broadcast
    - Multicast
    - Shuffle
    - Exchange
  - Hypercube Routing Functions

SYSTEM INTERCONNECT ARCHITECTURES

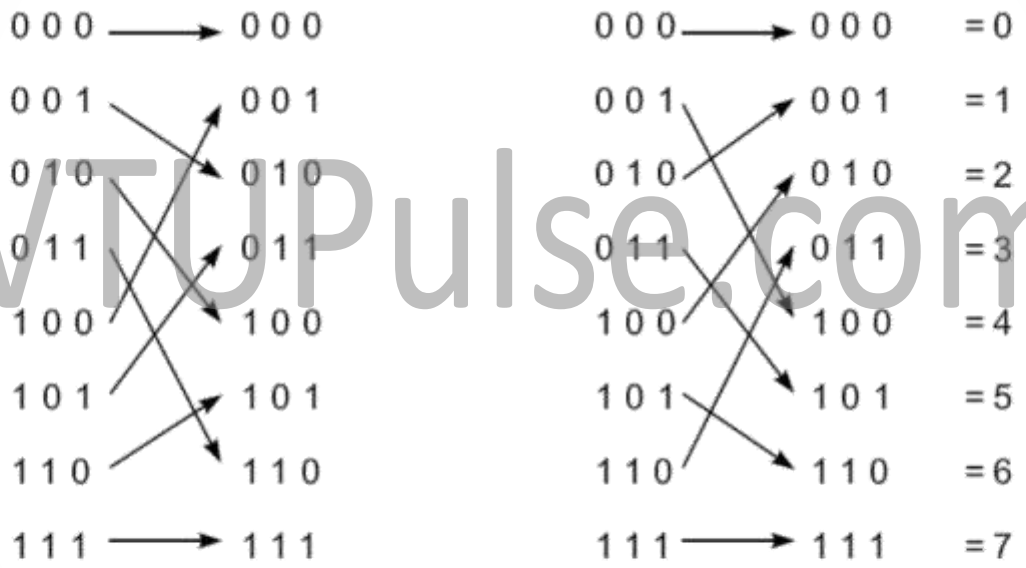
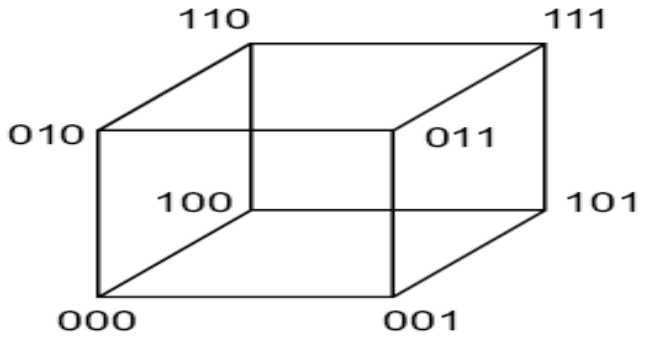
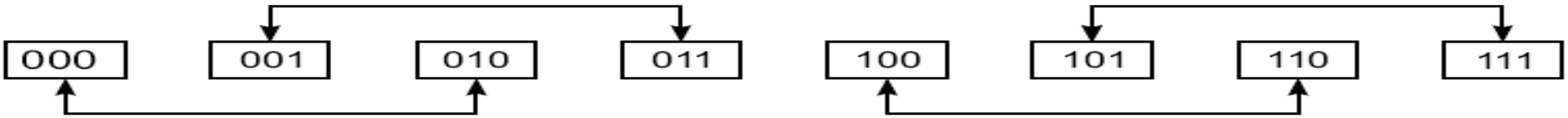


Fig. 2.14 Perfect shuffle and its inverse mapping over eight objects (Courtesy of H. Stone; reprinted with permission from *IEEE Trans. Computers*, 1971)

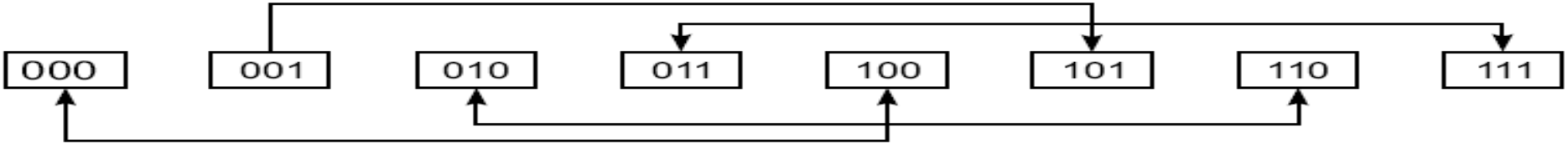


(a) A 3-cube with nodes denoted as $C_2C_1C_0$ in binary

(b) Routing by least significant bit, C_0



(c) Routing by middle bit, C_1



(d) Routing by most significant bit, C_2

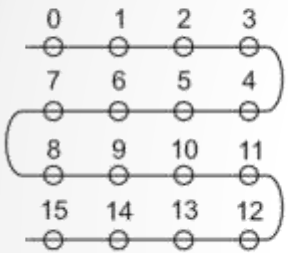
SYSTEM INTERCONNECT ARCHITECTURES

- Network Performance
  - Functionality
  - Network Latency
  - Bandwidth
  - Hardware Complexity
  - Scalability
- Network Throughput
  - Total no. of messages the network can handle per unit time
  - Hot spot and Hot spot throughput

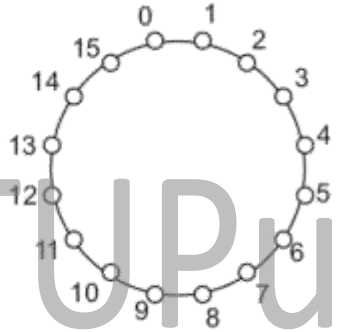
SYSTEM INTERCONNECT ARCHITECTURES

- Static Connection Networks
  - Liner Array
  - Ring and Chordal Ring
  - Barrel Shifter
  - Tree and Star
  - Fat Tree
  - Mesh and Torus
  - Systolic Arrays
  - Hypercubes
  - Cube-connected Cycles
  - K-ary n-Cube Networks
- Summary of Static Network Characteristics
  - Refer to **Table 2.2** (on page 76) of textbook

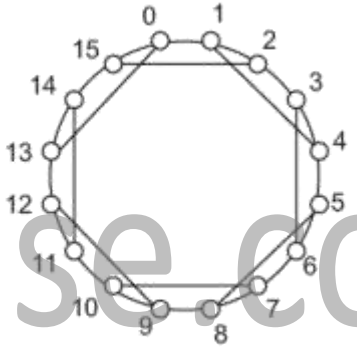
SYSTEM INTERCONNECT ARCHITECTURES



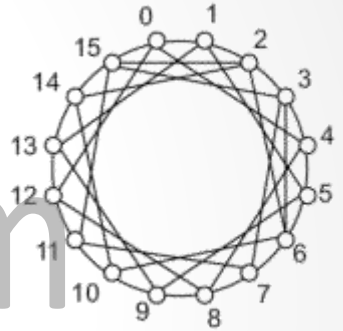
(a) Linear array



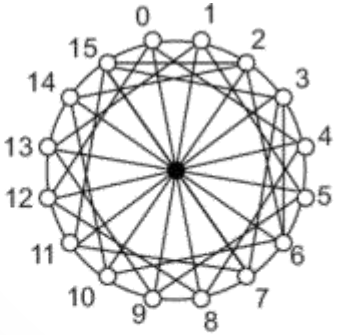
(b) Ring



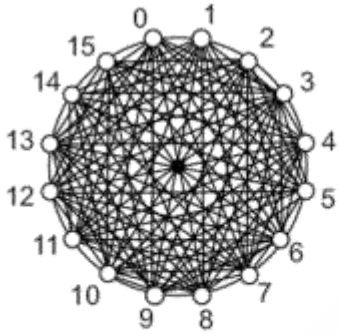
(c) Chordal ring of degree 3



(d) Chordal ring of degree 4
(same as Illiac mesh)



(e) Barrel shifter



(f) Completely connected

SYSTEM INTERCONNECT ARCHITECTURES

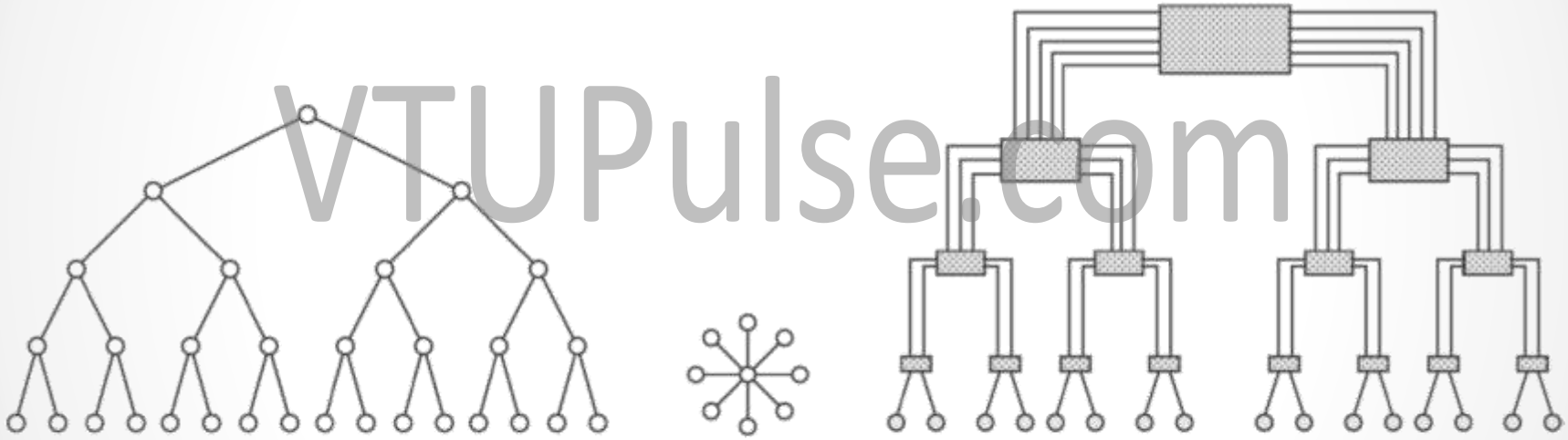


Fig. 2.17 Tree, star, and fat tree

SYSTEM INTERCONNECT ARCHITECTURES

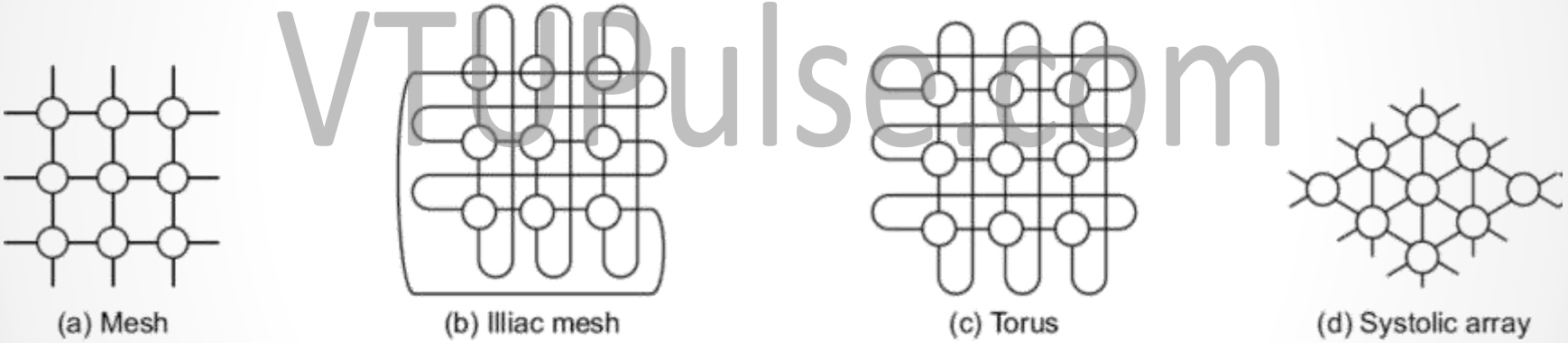
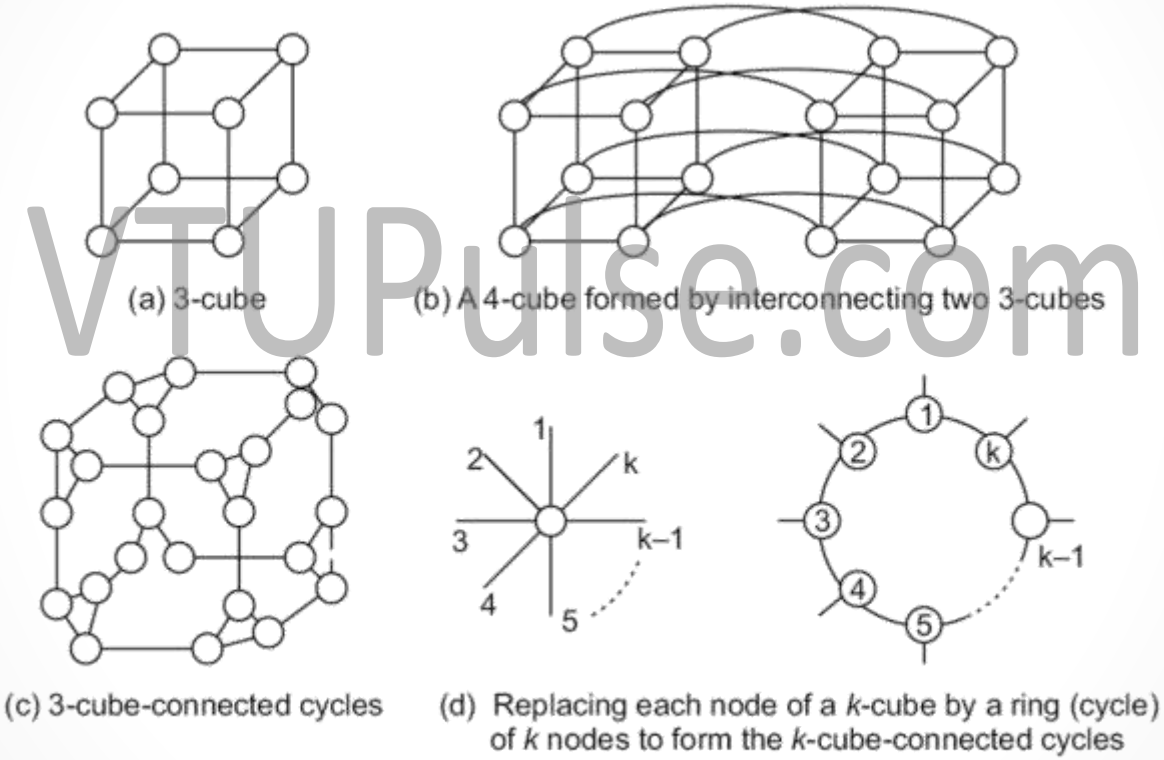


Fig. 2.18 Mesh, Illiac mesh, torus, and systolic array

SYSTEM INTERCONNECT ARCHITECTURES



(a) 3-cube (b) A 4-cube formed by interconnecting two 3-cubes

(c) 3-cube-connected cycles (d) Replacing each node of a *k*-cube by a ring (cycle) of *k* nodes to form the *k*-cube-connected cycles

Fig. 2.19 Hypercubes and cube-connected cycles

SYSTEM INTERCONNECT ARCHITECTURES

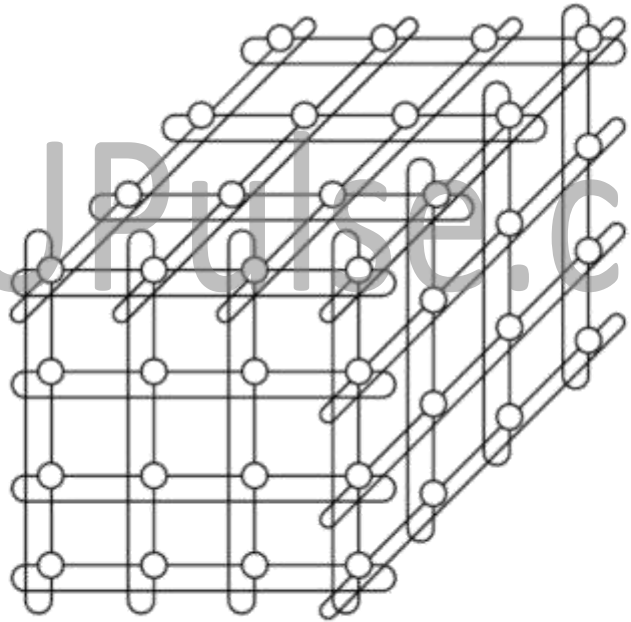
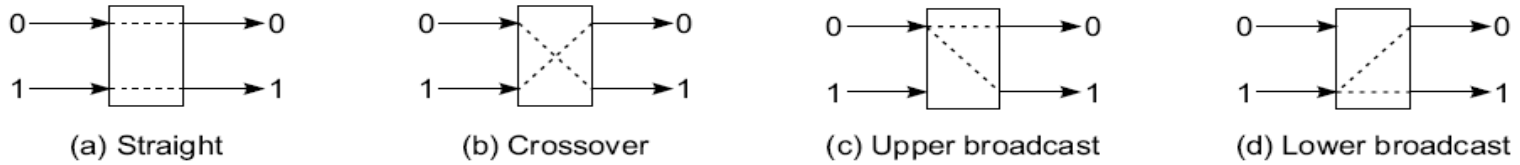
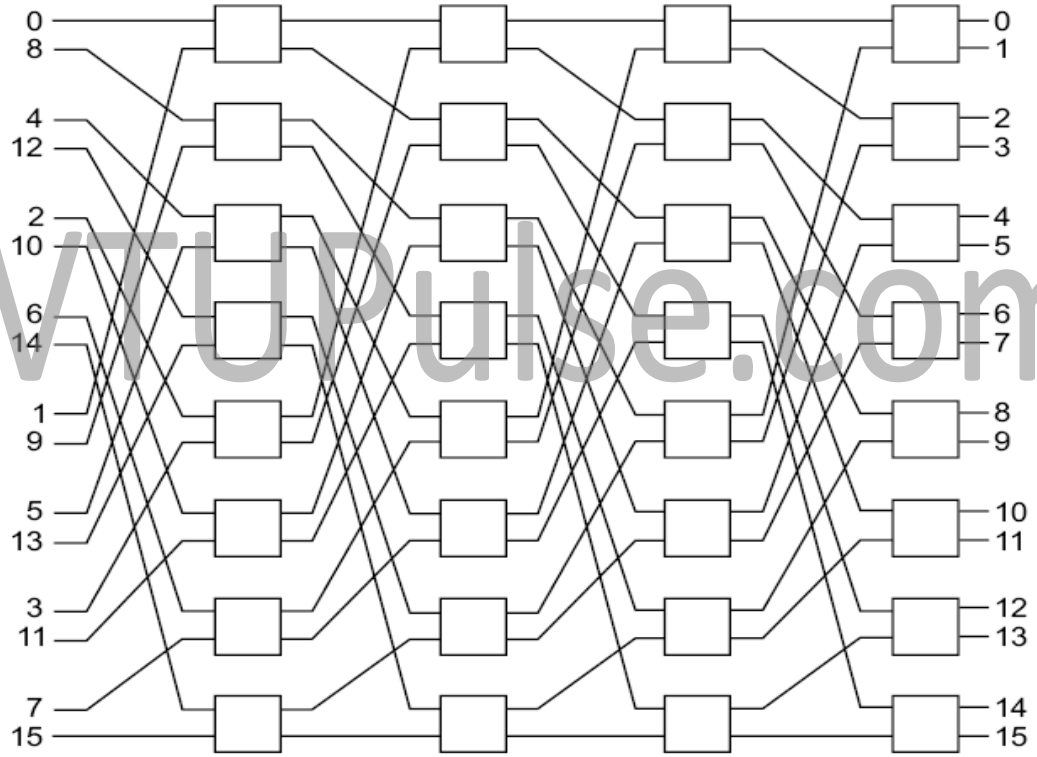


Fig. 2.20 The *k*-ary *n*-cube network shown with $k = 4$ and $n = 3$; hidden nodes or connections are not shown

SYSTEM INTERCONNECT ARCHITECTURES

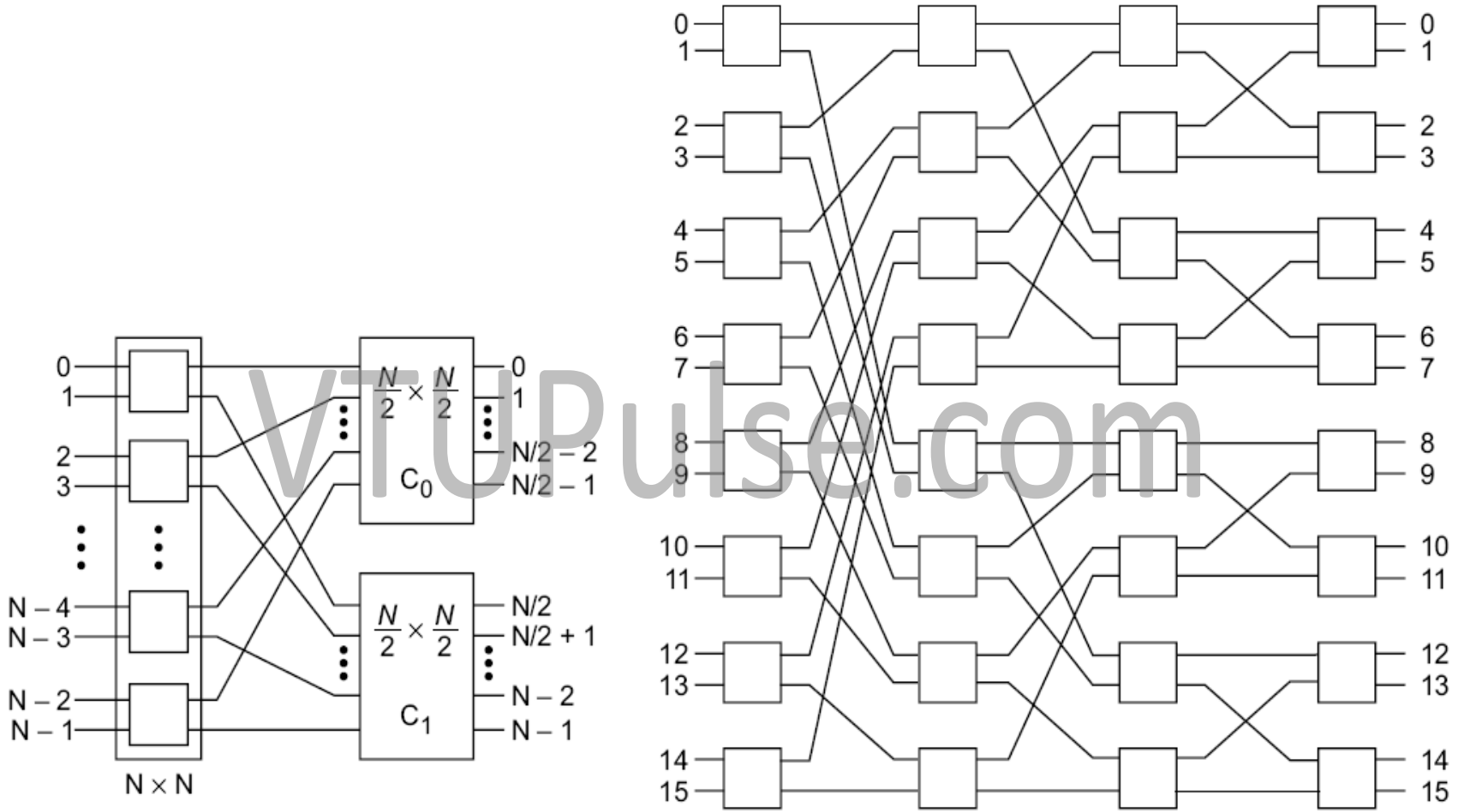
- Dynamic Connection Networks
  - Bus System – Digital or Contention or Time-shared bus
  - Switch Modules
    - Binary switch, straight or crossover connections
  - Multistage Interconnection Network (MIN)
    - Inter-Stage Connection (ISC) – *perfect shuffle, butterfly, multi-way shuffle, crossbar, etc.*
  - Omega Network
  - Baseline Network
  - Crossbar Network
- Summary of Dynamic Network Characteristics
  - Refer to **Table 2.4** (on page 82) of the textbook





(e) 16 × 16 Omega network

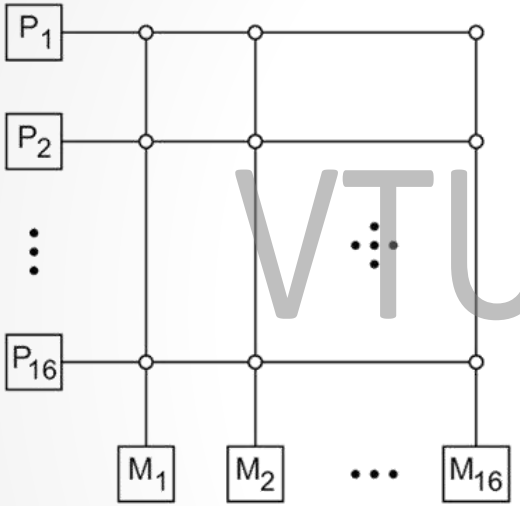
Fig. 2.24 The use of 2 × 2 switches and perfect shuffle as an interstage connection pattern to construct a



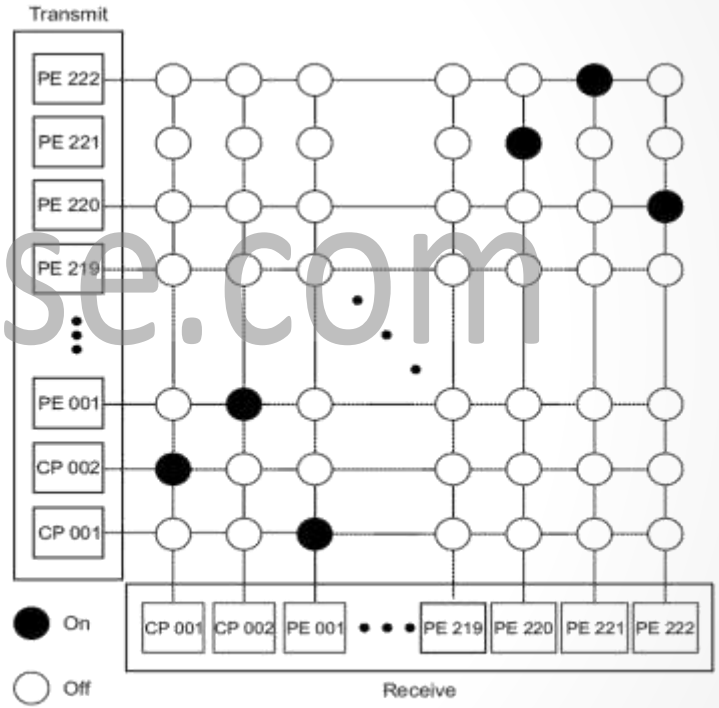
(a) Recursive construction

(b) A 16 × 16 Baseline network

SYSTEM INTERCONNECT ARCHITECTURES



(a) Interprocessor-memory crossbar network built in the C.mmp multiprocessor at Carnegie-Mellon University (1972)



(b) The interprocessor crossbar network built in the Fujitsu VPP500 vector parallel processor (1992)

Fig. 2.26 Two crossbar switch network configurations

PERFORMANCE EVALUATION OF PARALLEL COMPUTERS

CSE539

(Advanced Computer Architecture)

Sumit Mittu

(Assistant Professor, CSE/IT)

Lovely Professional University

BASICS OF PERFORMANCE EVALUATION

A sequential algorithm is evaluated in terms of its execution time which is expressed as a function of its input size.

For a parallel algorithm, the execution time depends not only on input size but also on factors such as parallel architecture, no. of processors, etc.

Performance Metrics

Parallel Run Time Speedup Efficiency

Standard Performance Measures

Peak Performance Sustained Performance

Instruction Execution Rate (in MIPS) Floating Point Capability (in MFLOPS)

PERFORMANCE METRICS

Parallel Runtime

The parallel run time **T(n)** of a program or application is the time required to run the program on an **n-processor** parallel computer.

When **n = 1**, **T(1)** denotes sequential runtime of the program on single processor.

Speedup

Speedup **S(n)** is defined as the ratio of time taken to run a program on a single processor to the time taken to run the program on a parallel computer with identical processors

$$S(n) = \frac{T(1)}{T(n)}$$

It measures how faster the program runs on a parallel computer rather than on a single processor.

PERFORMANCE METRICS

Efficiency

The Efficiency **E(n)** of a program on **n processors is** defined as the ratio of speedup achieved and the number of processor used to achieve it.

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{n.T(n)}$$

Relationship between Execution Time, Speedup and Efficiency and the number of processors used is depicted using the graphs in next slides.

In the ideal case:

Speedup is expected to be linear i.e. it grows linearly with the number of processors, but in most cases it falls due to parallel overhead.

PERFORMANCE METRICS

Graphs showing relationship b/w T(n) and no. of processors

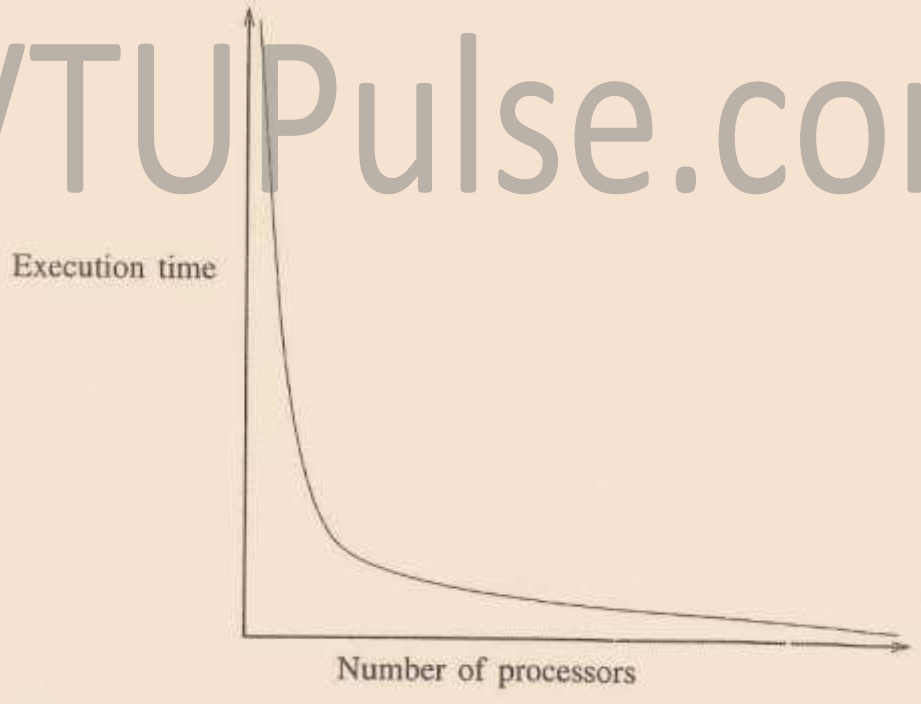


Fig. 9.3 Execution time versus number of processors employed.

PERFORMANCE METRICS

Graphs showing relationship b/w S(n) and no. of processors

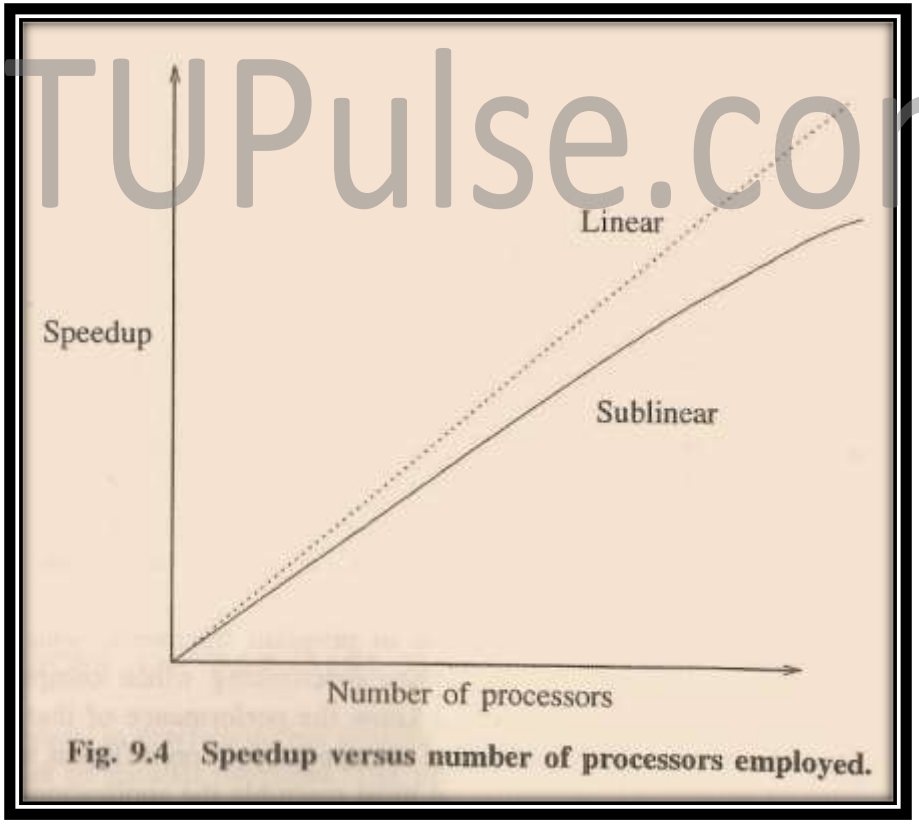


Fig. 9.4 Speedup versus number of processors employed.

PERFORMANCE METRICS

Graphs showing relationship b/w E(n) and no. of processors

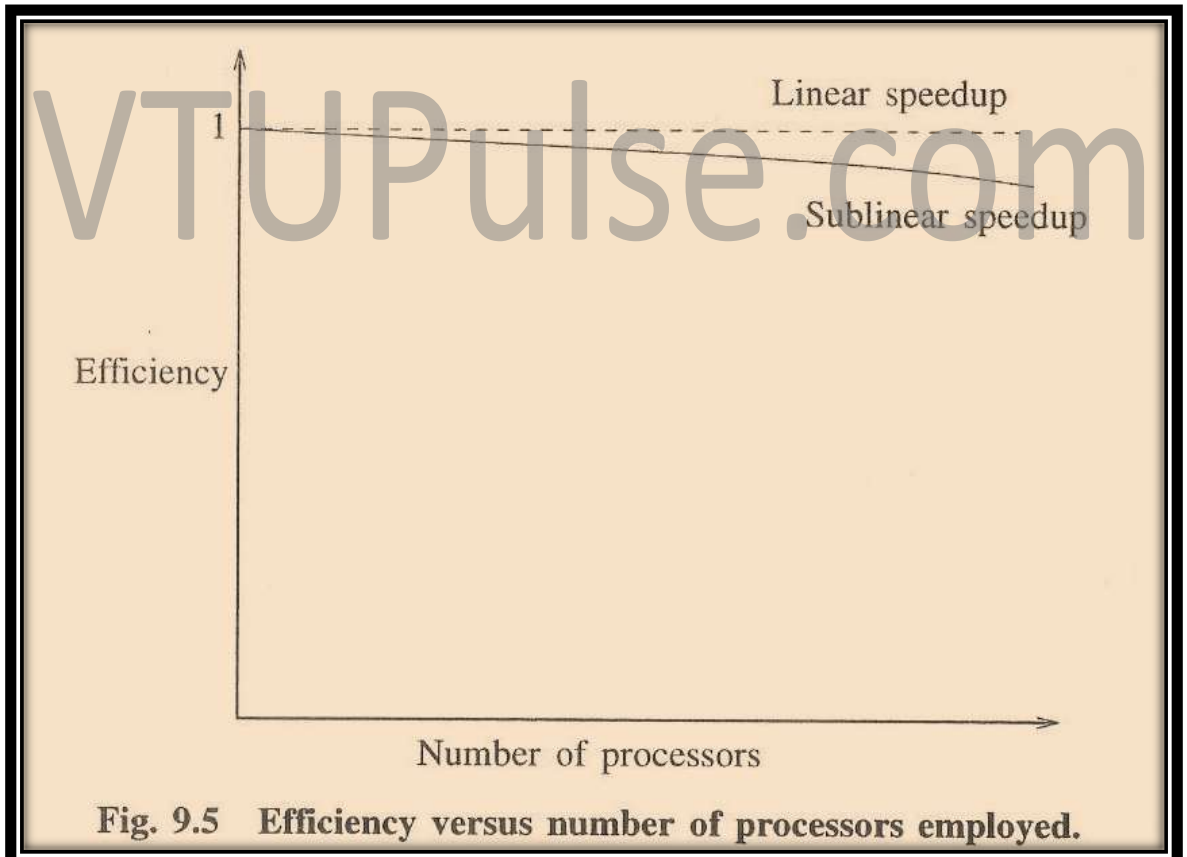


Fig. 9.5 Efficiency versus number of processors employed.

PERFORMANCE MEASURES

Standard Performance Measures

Most of the standard measures adopted by the industry to compare the performance of various parallel computers are based on the concepts of:

Peak Performance

[Theoretical maximum based on best possible utilization of all resources]

Sustained Performance

[based on running application-oriented benchmarks]

Generally measured in units of:

MIPS [to reflect instruction execution rate]

MFLOPS [to reflect the floating-point capability]

PERFORMANCE MEASURES

Benchmarks

Benchmarks are a set of programs of program fragments used to compare the performance of various machines.

Machines are exposed to these benchmark tests and tested for performance.

When it is not possible to test the applications of different machines, then the results of benchmark programs that most resemble the applications run on those machines are used to evaluate the performance of machine.

PERFORMANCE MEASURES

Benchmarks

Kernel Benchmarks

[Program fragments which are extracted from real programs]

[Heavily used core and responsible for most execution time]

Synthetic Benchmarks

[Small programs created especially for benchmarking purposes]

[These benchmarks do not perform any useful computation]

EXAMPLES

LINPACK *LAPACK* *Livermore Loops* *SPECmarks*

NAS Parallel Benchmarks *Perfect Club Parallel Benchmarks*

PARALLEL OVERHEAD

Sources of Parallel Overhead

Parallel computers in practice do not achieve linear speedup or an efficiency of 1 because of parallel overhead. The major sources of which could be:

- *Inter-processor Communication*
- *Load Imbalance*
- *Inter-Task Dependency*
- *Extra Computation*
- *Parallel Balance Point*

SPEEDUP PERFORMANCE LAWS

Speedup Performance Laws

Amdahl's Law

[based on fixed problem size or fixed work load]

Gustafson's Law

[for scaled problems, where problem size increases with machine size i.e. the number of processors]

Sun & Ni's Law

[applied to scaled problems bounded by memory capacity]

SPEEDUP PERFORMANCE LAWS

Amdahl's Law (1967)

For a given problem size, the speedup does not increase linearly as the number of processors increases. In fact, the speedup tends to become saturated. This is a consequence of Amdahl's Law.

According to Amdahl's Law, a program contains two types of operations:

Completely sequential

Completely parallel

Let, the time **Ts** taken to perform sequential operations be a fraction **α** ($0<\alpha\leq1$) of the total execution time **T(1)** of the program, then the time **Tp** to perform parallel operations shall be **(1-α)** of **T(1)**

SPEEDUP
PERFORMANCE LAWS

Amdahl's Law

Thus, **Ts = α.T(1)** and **Tp = (1-α).T(1)**

Assuming that the parallel operations achieve linear speedup (i.e. these operations use 1/n of the time taken to perform on each processor), then

$$\mathbf{T(n) = Ts + Tp/n = \alpha.T(1) + \frac{(1-\alpha).T(1)}{n}}$$

Thus, the speedup with **n** processors will be:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{\alpha.T(1) + \frac{(1-\alpha).T(1)}{n}} = \frac{1}{\alpha + \frac{(1-\alpha)}{n}}$$

SPEEDUP
PERFORMANCE LAWS

Amdahl's Law

Sequential operations will tend to dominate the speedup as n becomes very large.

As **n → ∞, S(n) → 1/α**

This means, no matter how many processors are employed, the speedup in this problem is limited to 1/α.

This is known as **sequential bottleneck** of the problem.

Note: *Sequential bottleneck* ***cannot*** *be removed just by increasing the no. of processors.*

SPEEDUP PERFORMANCE LAWS

Amdahl's Law

A major shortcoming in applying the Amdahl's Law: (is its own characteristic)

The total work load or the problem size is fixed

Thus, execution time decreases with increasing no. of processors

Thus, a successful method of overcoming this shortcoming is ***to increase the problem size!***

SPEEDUP
PERFORMANCE LAWS

Amdahl's Law

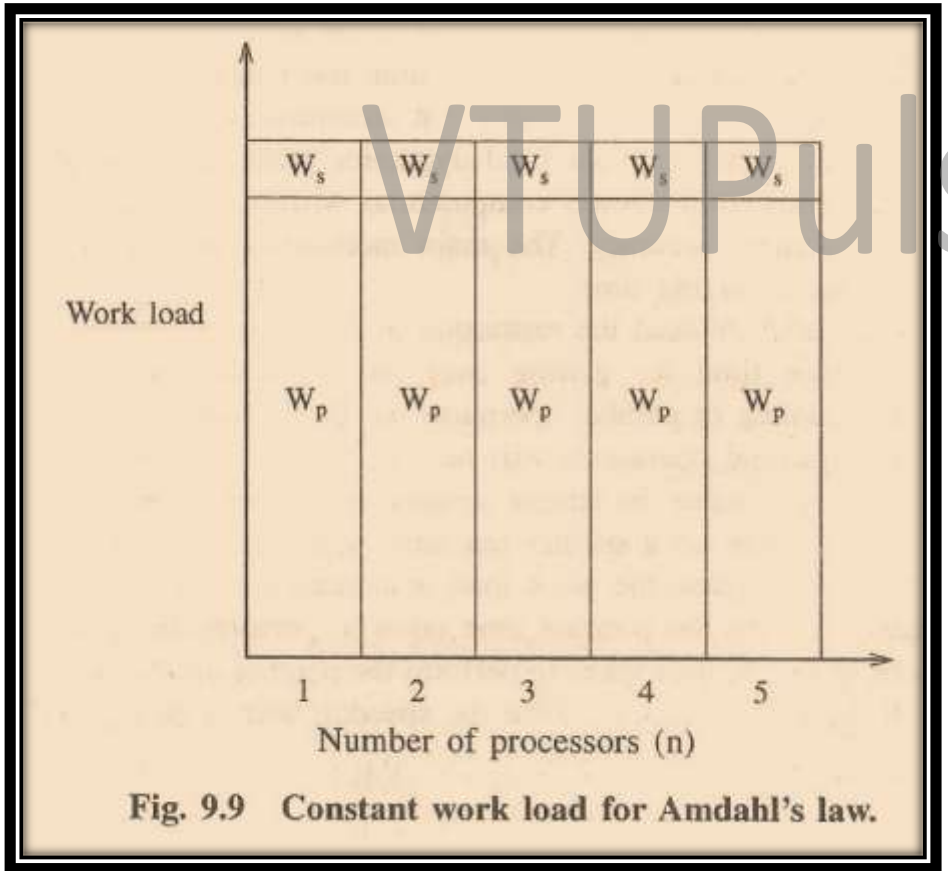
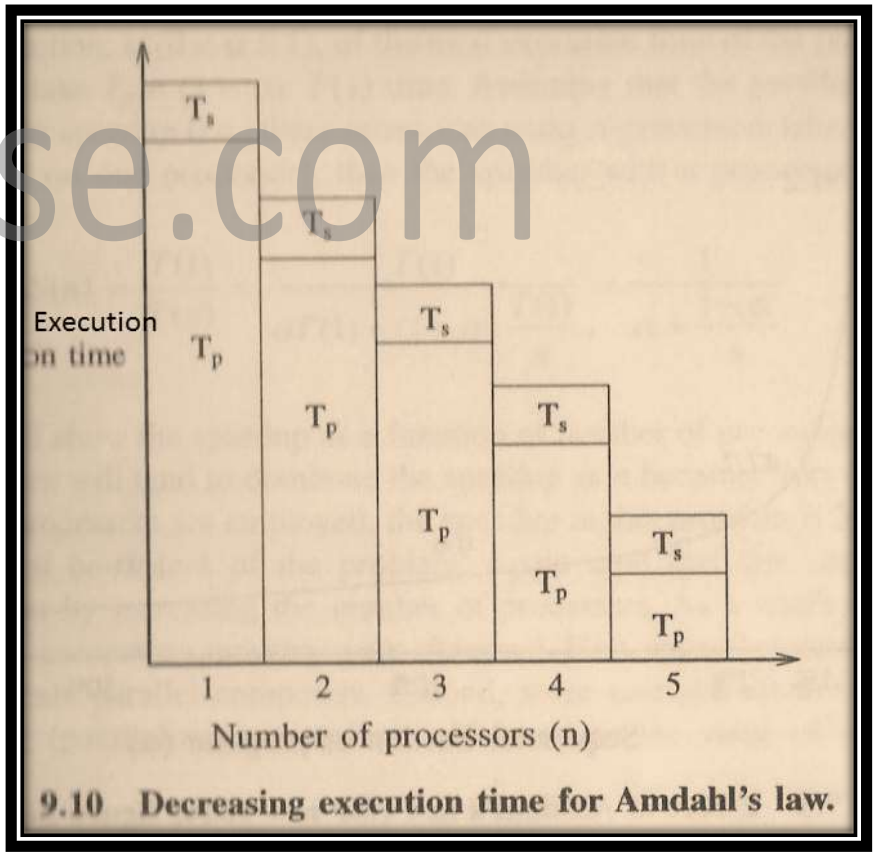


Fig. 9.9 Constant work load for Amdahl's law.



9.10 Decreasing execution time for Amdahl's law.

SPEEDUP PERFORMANCE LAWS

Gustafson's Law (1988)

It relaxed the restriction of fixed size of the problem and used the notion of fixed execution time for getting over the sequential bottleneck.

According to Gustafson's Law,

If *the number of parallel operations in the problem is increased (or scaled up) sufficiently,*

Then *sequential operations will no longer be a bottleneck.*

In accuracy-critical applications, it is desirable to solve the largest problem size on a larger machine rather than solving a smaller problem on a smaller machine, with almost the same execution time.

SPEEDUP PERFORMANCE LAWS

Gustafson's Law

As the machine size increases, the work load (or problem size) is also increased so as to keep the fixed execution time for the problem.

Let, **Ts** be the constant time tank to perform sequential operations; and

Tp(n,W) be the time taken to perform parallel operation of problem size or workload **W** using **n** processors;

Then the speedup with n processors is:

$$S'(n) = \frac{Ts + Tp(1,W)}{Ts + Tp(n,W)}$$

SPEEDUP
PERFORMANCE LAWS

Gustafson's Law

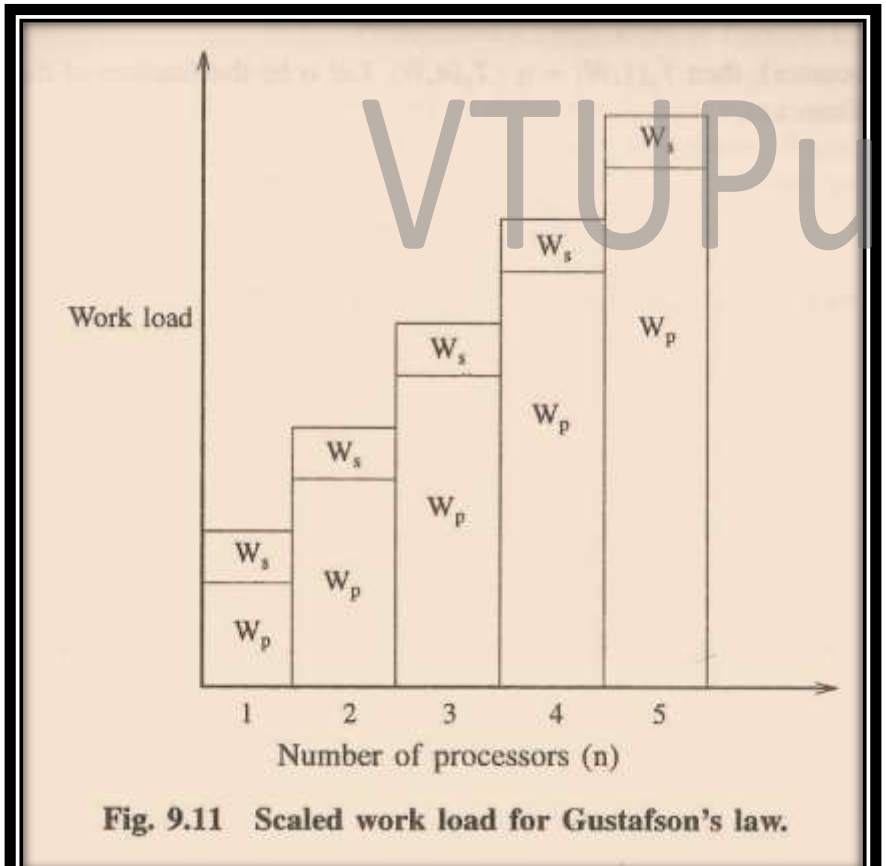


Fig. 9.11 Scaled work load for Gustafson's law.

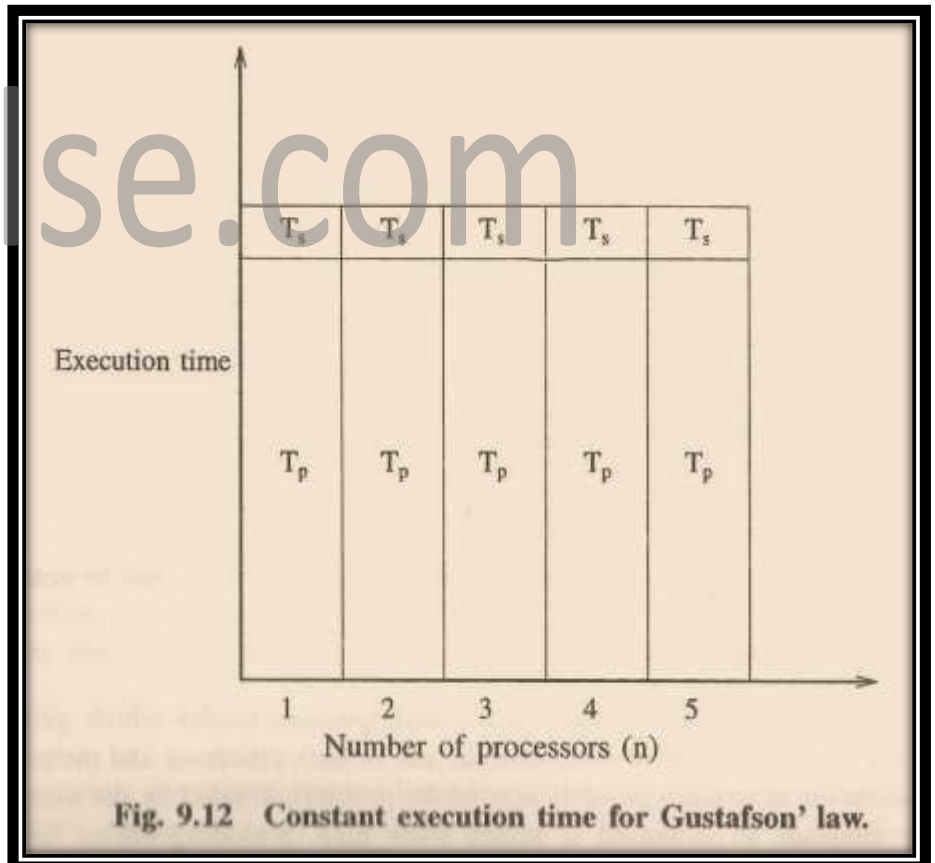


Fig. 9.12 Constant execution time for Gustafson' law.

SPEEDUP
PERFORMANCE LAWS

Gustafson's Law

Assuming that parallel operations achieve a linear speedup
(i.e. these operations take **1/n** of the time to perform on one processor)
Then, **Tp(1,W) = n. Tp(n,W)**

Let α be the fraction of sequential work load in problem, i.e.

$$\alpha = \frac{Ts}{Ts + Tp(n, W)}$$

Then the speedup can be expressed as : with n processors is:

$$S'^{(n)} = \frac{Ts + n.Tp(n, W)}{Ts + Tp(n, W)} = \alpha + n(1 - \alpha) = n - \alpha(n - 1)$$

SPEEDUP PERFORMANCE LAWS

Sun & Ni's Law (1993)

This law defines a ***memory bounded speedup*** *model which generalizes both Amdahl's Law and Gustafson's Law to maximize the use of both processor and memory capacities.*

The idea is to solve maximum possible size of problem, limited by memory capacity

This inherently **demands** *an increased or scaled work load*,

providing *higher speedup*,

Higher efficiency, and

Better resource (processor & memory) utilization

But may result in slight increase in execution time to achieve this scalable speedup performance!

SPEEDUP PERFORMANCE LAWS

Sun & Ni's Law

According to this law, the speedup **S*(n)** in the performance can be defined by:

$$S^{*(n)} = \frac{Ts + G(n).Tp(n,W)}{Ts + \frac{G(n)}{n}.Tp(n,W)} = \frac{\alpha + G(n).(1-\alpha)}{\alpha + \frac{G(n)}{n}.(1-\alpha)}$$

Assumptions made while deriving the above expression:

- A global address space is formed from all individual memory spaces i.e. there is a distributed shared memory space
- All available memory capacity of used up for solving the scaled problem.

SPEEDUP PERFORMANCE LAWS

Sun & Ni's Law

Special Cases:

- **G(n) = 1**

Corresponds to where we have fixed problem size i.e. Amdahl's Law

- **G(n) = n**

Corresponds to where the work load increases **n** times when memory is increased **n** times, i.e. for scaled problem or Gustafson's Law

- **G(n) ≥ n**

Corresponds to where computational workload (time) increases faster than memory requirement.

Comparing speedup factors S*(n), S'(n) and S'(n), we shall find **S*(n) ≥ S'(n) ≥ S(n)**

SPEEDUP
PERFORMANCE LAWS

Sun & Ni's Law

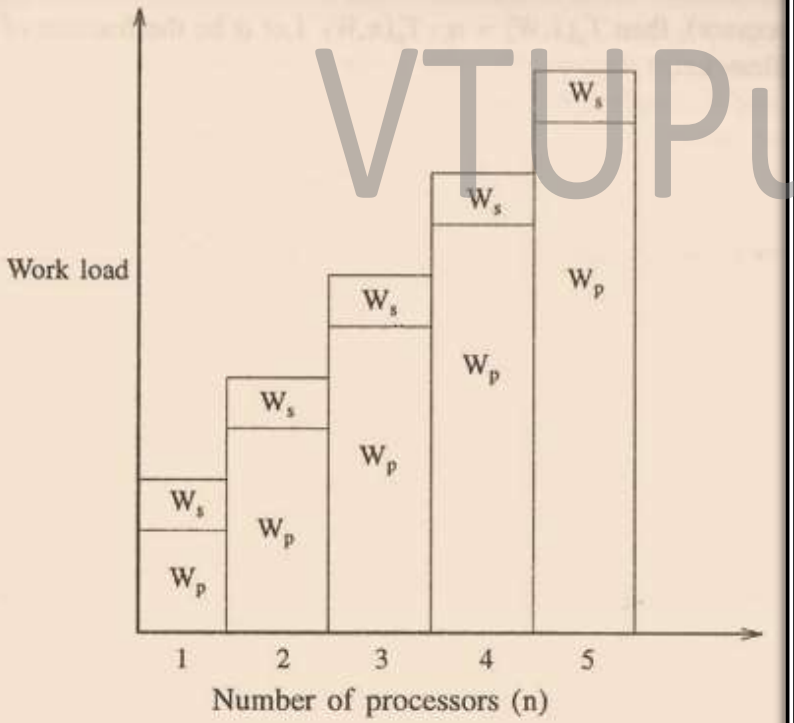


Fig. 9.11 Scaled work load for Sun and Ni's law.

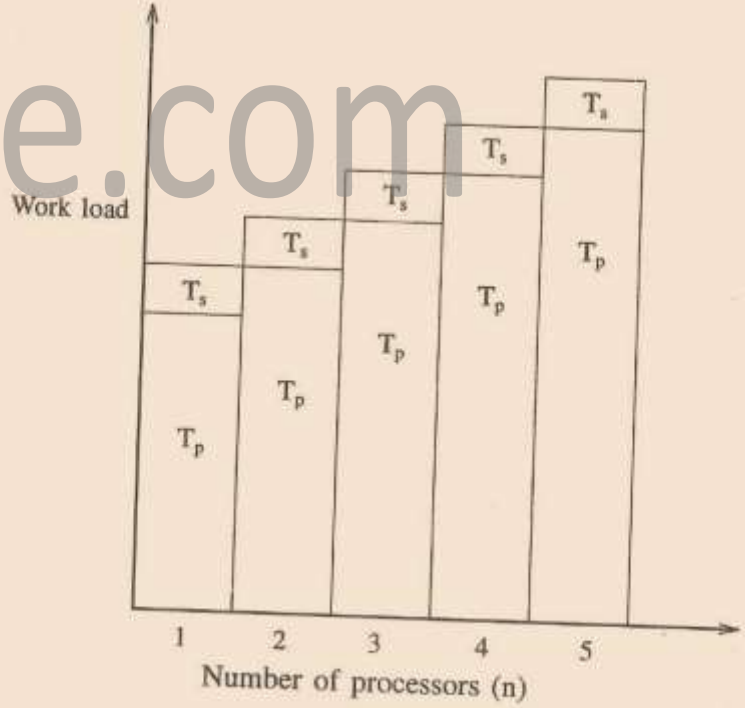


Fig. 9.15 Slightly increasing execution time for Sun and Ni's law.

SCALABILITY METRIC

Scalability

– Increasing the no. of processors decreases the efficiency!

+ Increasing the amount of computation per processor, increases the efficiency!

To keep the efficiency fixed, both the size of problem and the no. of processors must be increased simultaneously.

A parallel computing system is said to be scalable if its efficiency can be fixed by simultaneously increasing the machine size and the problem size.

Scalability of a parallel system is the measure of its capacity to increase speedup in proportion to the machine size.

SCALABILITY METRIC

Isoefficiency Function

The isoefficiency function can be used to measure scalability of the parallel computing systems.

It shows how the size of problem must grow as a function of the number of processors used in order to maintain some constant efficiency.

The general form of the function is derived using an equivalent definition of efficiency as follows:

$$E(p) = \frac{U}{U+O} = \frac{1}{1+\frac{O}{U}}$$

Where, **U** is the time taken to do the useful computation (essential work), and **O** is the parallel overhead. (Note: O is zero for sequential execution).

SCALABILITY METRIC

Isoefficiency Function

If the efficiency is fixed at some constant value **K** then

$$U = \frac{K}{1-K} O = K'O \qquad K' = \frac{K}{1-K}$$

Where, **K'** is a constant for fixed efficiency **K**.

This function is known as the isoefficiency function of parallel computing system.

A small isoefficiency function means that small increments in the problem size (U), are sufficient for efficient utilization of an increasing no. of processors, indicating high scalability.

A large isoeffcicnecy function indicates a poorly scalable system.

SCALABILITY METRIC

Isoefficiency Function

the isoefficiency function does not exist for unscalable parallel computing systems. This is because the efficiency in such systems cannot be kept at any constant value as machine size increases, no matter how fast the problem size is increased.

PERFORMANCE MEASUREMENT TOOLS

Performance Analysis

Search Based Tools

Visualization

Utilization Displays

[Processor (utilization) count, Utilization Summary, Gantt charts, Concurrency Profile, Kiviat Diagrams]

Communication Displays

[Message Queues, Communication Matrix, Communication Traffic, Hypercube]

Task Displays

[Task Gantt, Task Summary]

PERFORMANCE MEASUREMENT TOOLS

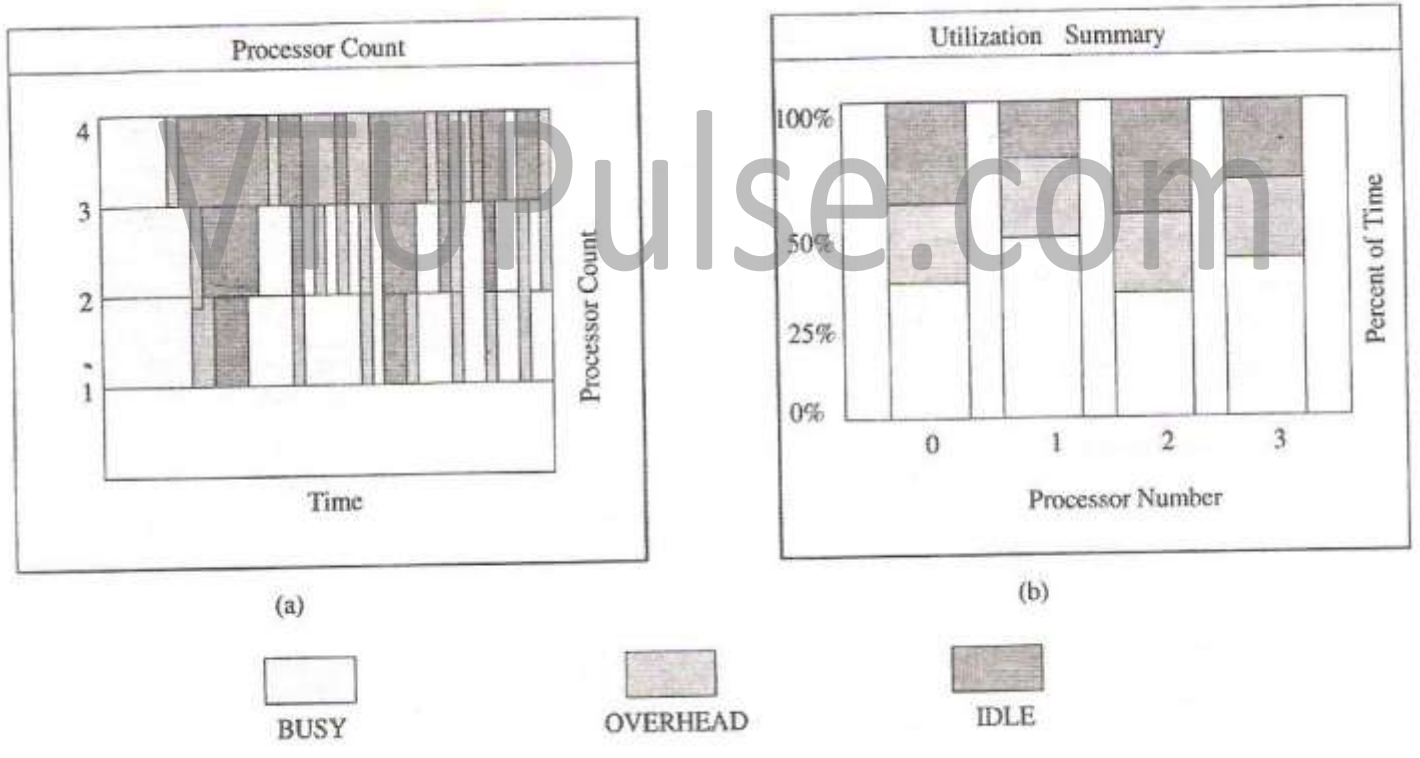


Fig. 9.17 (a) Processor count and (b) Utilization summary.

PERFORMANCE MEASUREMENT TOOLS

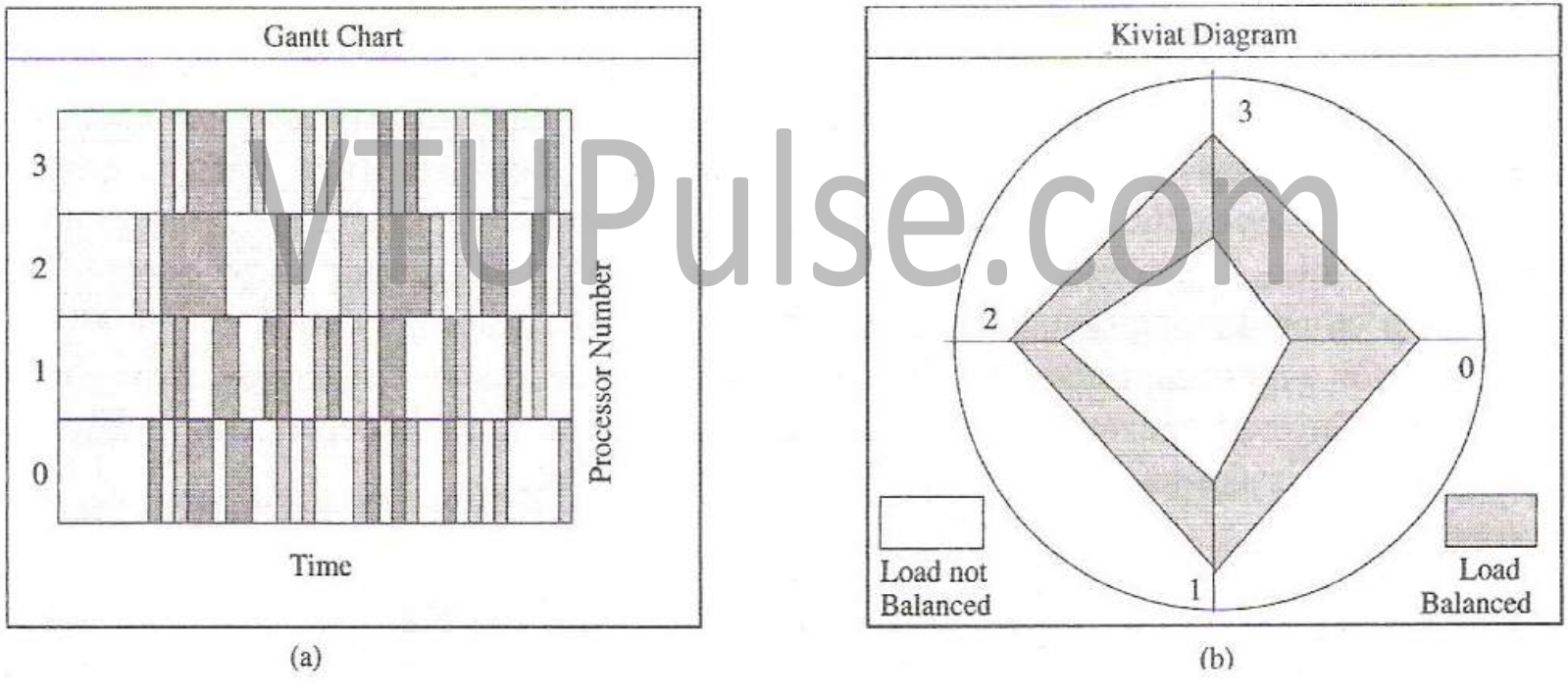


Fig. 9.18 (a) Gantt chart and (b) Kiviat diagram.

PERFORMANCE MEASUREMENT TOOLS

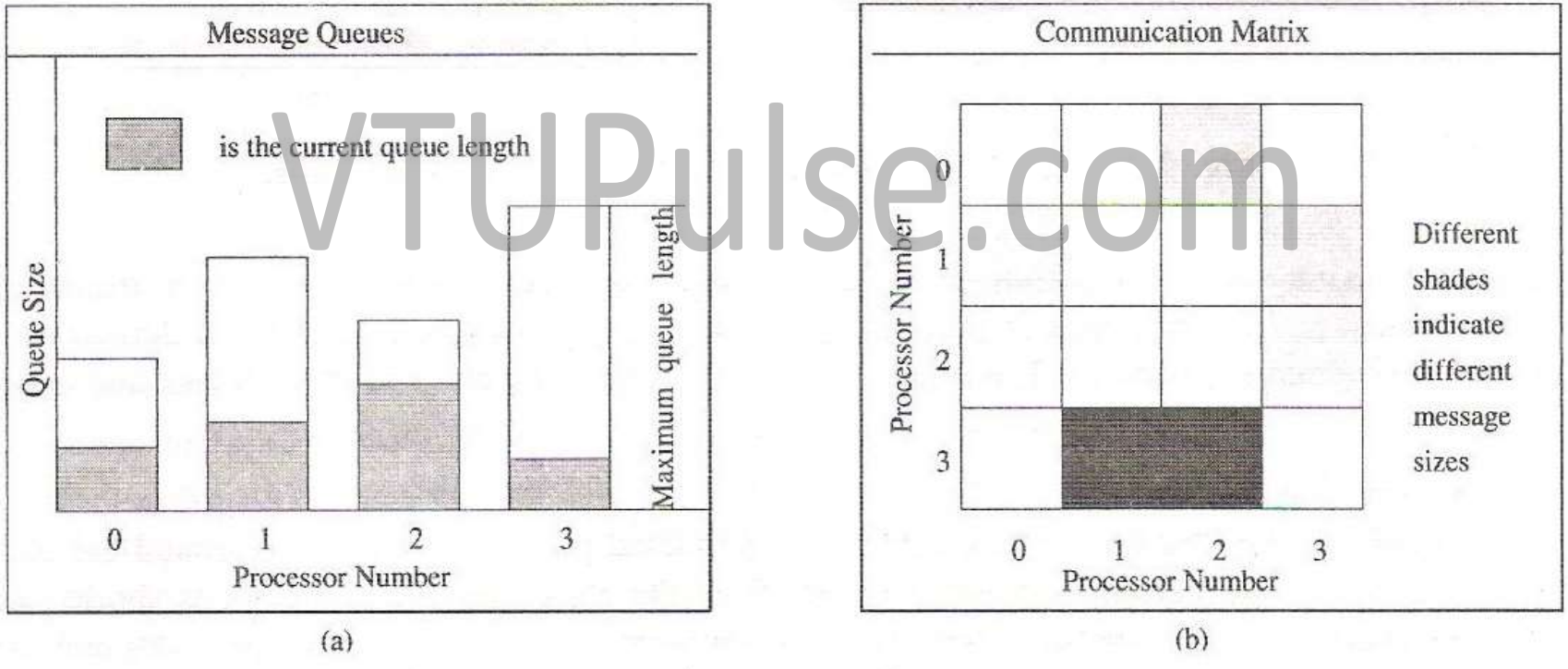


Fig. 9.19 (a) Message queues and (b) Communication matrix.

PERFORMANCE MEASUREMENT TOOLS

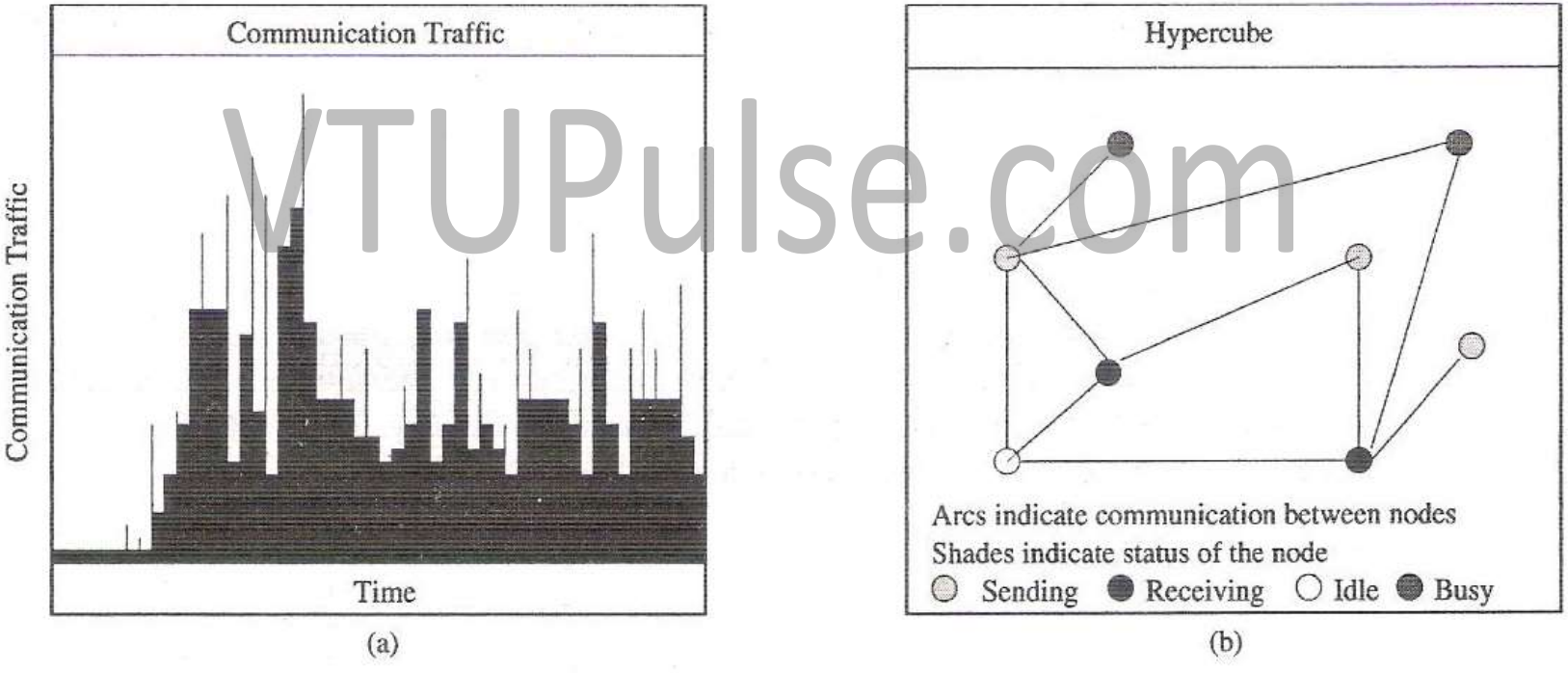
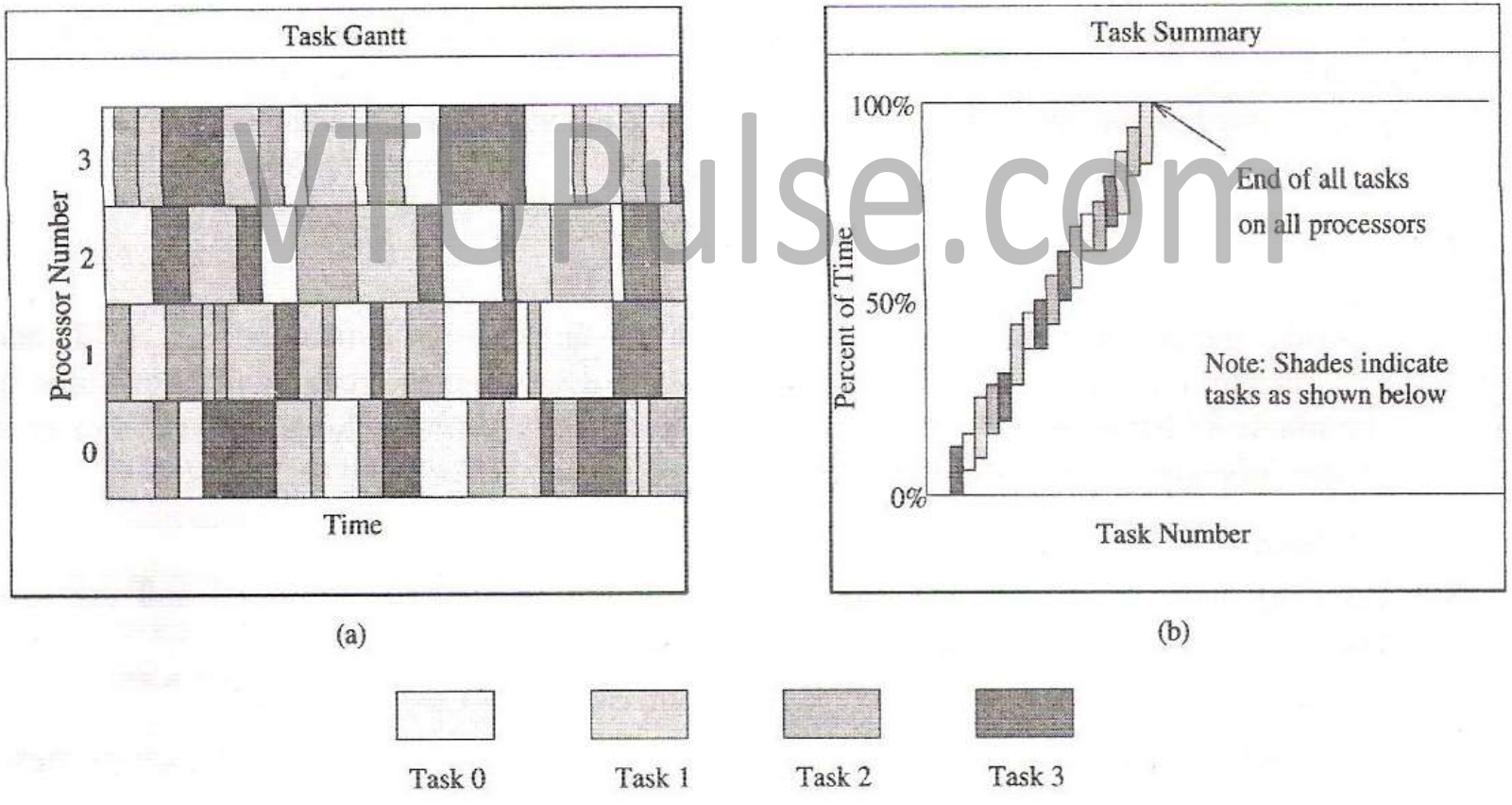


Fig. 9.20 (a) Communication traffic and (b) Hypercube.

PERFORMANCE MEASUREMENT TOOLS



PERFORMANCE MEASUREMENT TOOLS

Instrumentation

A way to collect data about an application is to *instrument* the application executable so that when it executes, *it generates the required information as a side-effect.*

Ways to do instrumentation:

By inserting it into the application source code directly, or

By placing it into the runtime libraries, or

By modifying the linked executable, etc.

Doing this, some perturbation of the application program will occur

(i.e. **intrusion problem**)

PERFORMANCE MEASUREMENT TOOLS

Instrumentation

Intrusion includes both:

Direct contention for resources (e.g. CPU, memory, communication links, etc.)

Secondary interference with resources (e.g. interaction with cache replacements or virtual memory, etc.)

To address such effects, you may adopt the following approaches:

Realizing that intrusion affects measurement, treat the resulting data as an approximation

Leave the added instrumentation in the final implementation.

Try to minimize the intrusion.

Quantify the intrusion and compensate for it!

CSE539: Advanced Computer Architecture

Chapter 3
Principles of Scalable Performance

Book: "Advanced Computer Architecture – Parallelism, Scalability, Programmability", Hwang & Jotwani

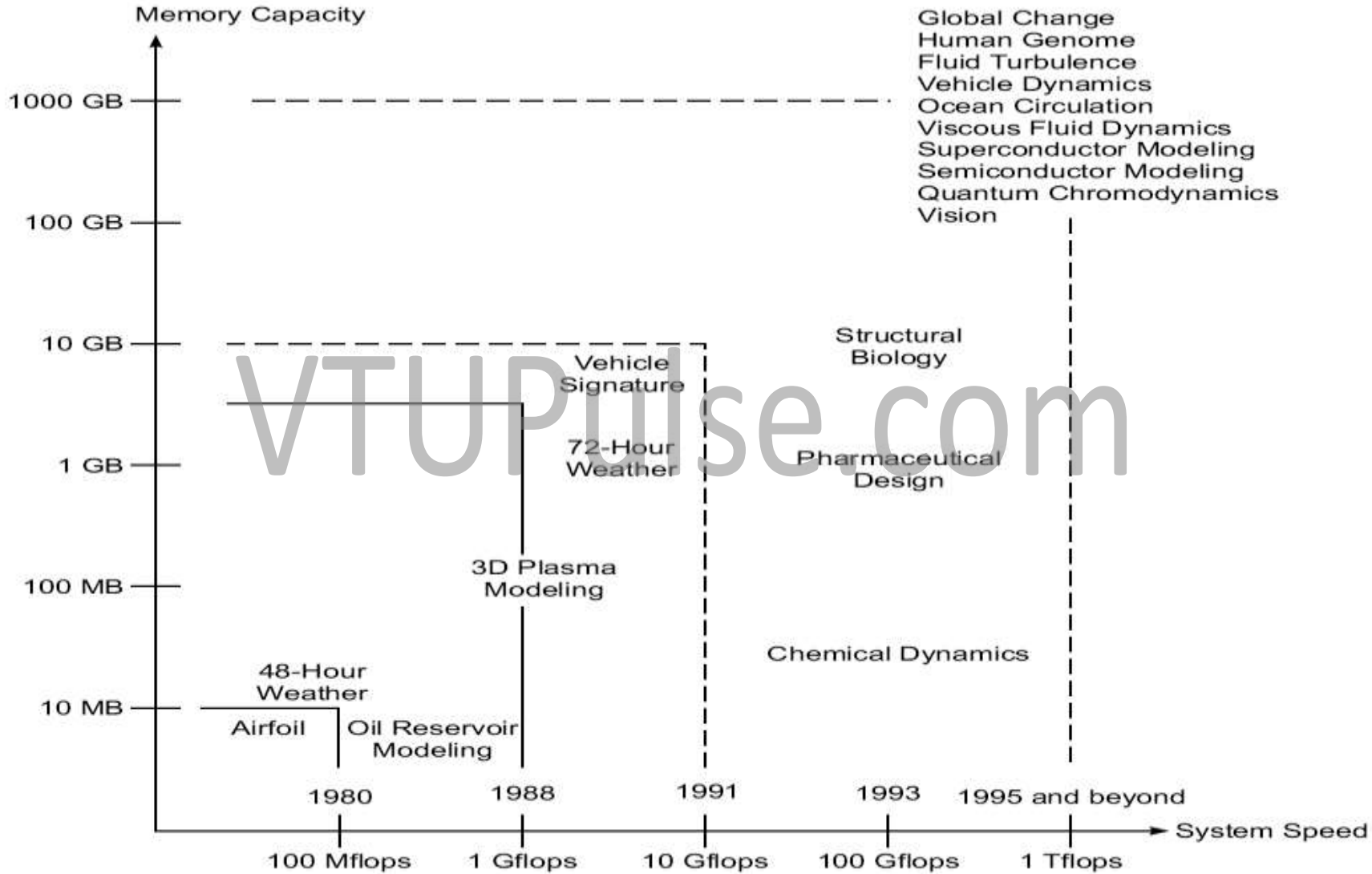
Sumit Mittu
Assistant Professor, CSE/IT
Lovely Professional University
sumit.12735@lpu.co.in

In this chapter…

- Parallel Processing Applications
- Sources of Parallel Overhead*
- Application Models for Parallel Processing
- Speedup Performance Laws

PARALLEL PROCESSING APPLICATIONS
Massive Parallelism for Grand Challenges

- Massively Parallel Processing
- High-Performance Computing and Communication (HPCC) Program
- Grand Challenges
  - Magnetic Recording Industry
  - Rational Drug Design
  - Design of High-speed Transport Aircraft
  - Catalyst for Chemical Reactions
  - Ocean Modelling and Environment Modelling
  - Digital Anatomy in:
    - Medical Diagnosis, Pollution reduction, Image Processing, Education Research

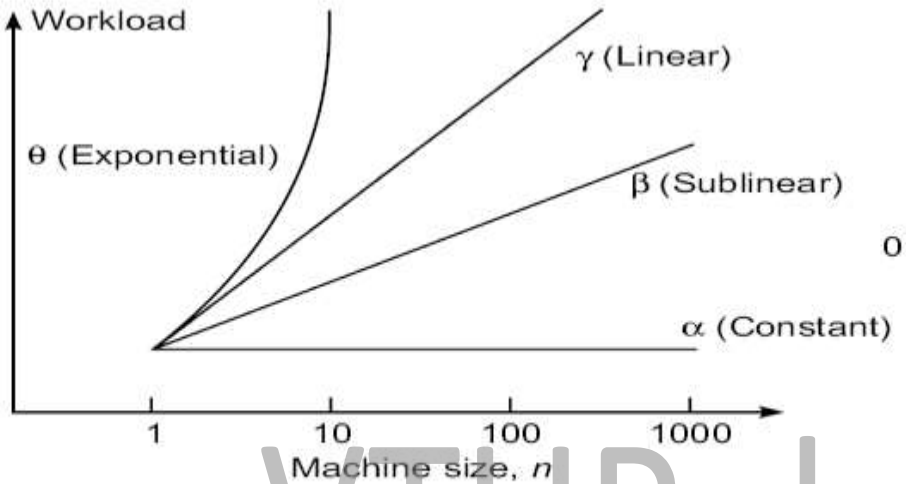


PARALLEL PROCESSING APPLICATIONS
Massive Parallelism for Grand Challenges

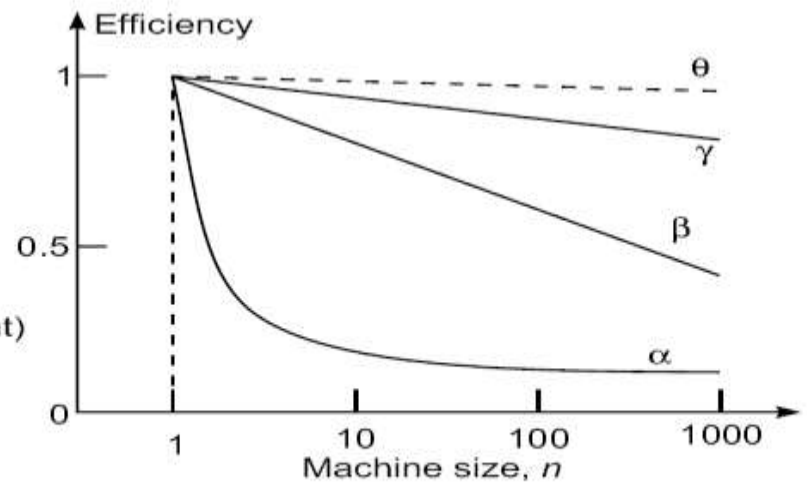
- Exploiting Massive Parallelism
  - Instruction Parallelism v/s Data Parallelism
- The Past and The Future
  - Early Representative Massively Parallel Processing Systems
    - Kendall Square KSR-1: All-cache ring multiprocessor; 43 Gflops peak performance
    - IBM MPP Model: 1024 IBM RISC/6000 processors; 50 Gflops peak performance
    - Cray MPP Model: 1024 DEC Alpha Processor chips; 150 Gflops peak performance
    - Intel Paragon: 2D mesh-connected multicomputer; 300 Gflops peak performance
    - Fujistu VPP-500: 222-PE MIMD vector system; 355 Gflops peak performance
    - TMC CM-5: 16K nodes of SPARC PEs, 2Tflops peak performance

PARALLEL PROCESSING APPLICATIONS
Application Models for Parallel Processing

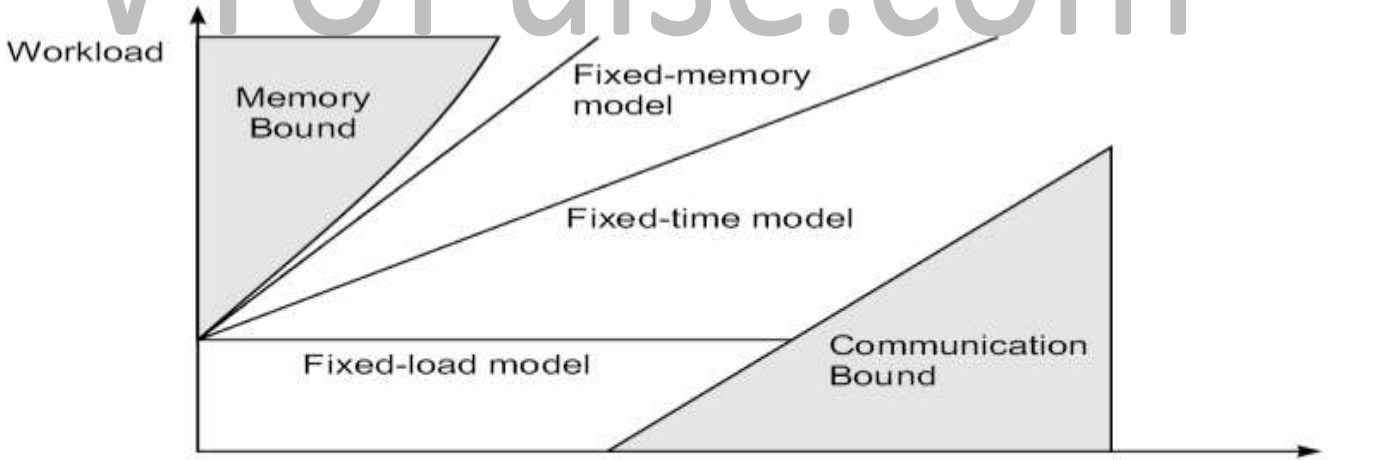
- Keeping the **workload** α as constant
  - **Efficiency E** decreases rapidly as **Machine Size n** increases
    - *Because:* **Overhead h** increases faster than machine size
- Scalable Computer
- Workload Growth and Efficiency Curves
- Application Models
  - Fixed Load Model
  - Fixed Time Model
  - Fixed Memory Model
- Trade-offs in Scalability Analysis



(a) Four workload growth patterns



(b) Corresponding efficiency curves



(c) Application models for parallel computers