K. S. Institute of Technology

Department of Computer Science and Engineering

Advanced Computer Architecture – 18CS733

Faculty Name: Dr. Vijayalaxmi Mekali

Associate Professor, Dept. of CSE

KSIT, Bangalorehttps://www.knowledgeadda.com/2019/06/vtu-cse-notes-for-7th-semester-2015.html

Department Vision and Mission

VISION

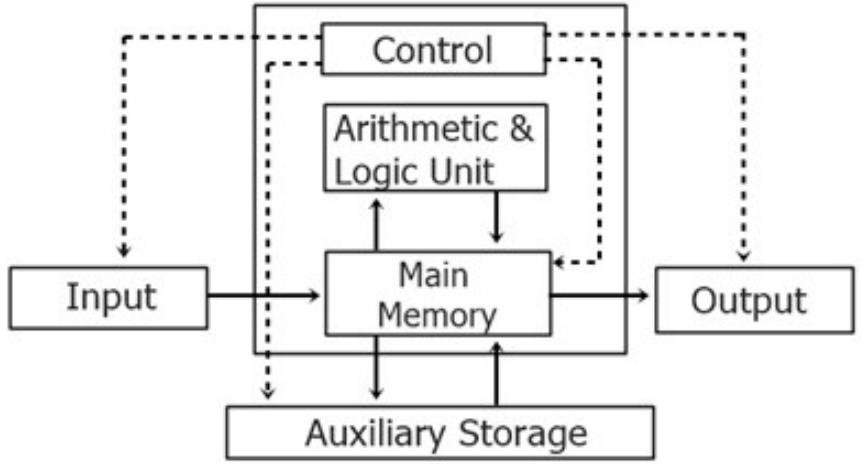
To create competent professionals in Computer Science and Engineering with adequate skills to drive the IT industry”

MISSION

- **Impart sound technical knowledge and quest for continuous learning.**
- **To equip students to furnish Computer Applications for the society through experiential learning and research with professional ethics.**
- **Encourage team work through inter-disciplinary project and evolve as leaders with social concerns.**

What is Computer?

- **Computer is a system.**
- **Computer is an electronic device, provides the solution to given problem**



Block Diagram of Computer

Why Computer?

- **To solve the given problems.**
- **It is very much accurate, fast and can accomplish many tasks easily (Otherwise to complete those tasks manually much more time is required).**
- **It can do very big calculations in just a fraction of a second. Ex: Factorial of 1223**
- **Moreover it can store huge amount of data in it.**

Advanced Computer Architecture

Current age is called as the ERA OF INFORMATION TECHNOLOGY. And now we cannot imagine a world without computers.

It is made up of two things

All physical components of computer like memory, wire, IC's, keyboard, mouse, monitor etc comes under the hardware whereas all the programs and languages used by the computer are called software.

Computer is a Tool for not only engineers and scientists but also they are being used by millions of people around the world. In almost every field even where it is most unexpected.

Advanced Computer Architecture

Computer Architecture?

- Computer architecture is a set of rules and methods that describe the functionality, organization, and implementation of computer systems.

- Computer architecture is a specification detailing how a set of software and hardware technology standards interact to form a computer system or platform.

- Computer architecture refers to how a computer system is designed and what technologies it is compatible with.

- Computer architecture is art of determining the needs of the user/system/technology, and creating a logical design and standards based on those requirements.

Computer Architecture?

- **System Design: Designing of**

Hardware components in the system such as data processors, CPU, graphics processing unit and direct memory access. It also includes memory controllers, data paths and miscellaneous things like multiprocessing and virtualization.

- **Instruction Set Architecture (ISA):**
- **An ISA is defined as the design of a computer from the Programmer's Perspective.**
- **ISA describes the design of a Computer in terms of the basic operations it must support. The ISA is only concerned with the set or collection of basic operations the computer must support.**
- **For example the AMD Athlon and the Core 2 Duo processors have entirely different implementations but they support more or less the same set of basic operations as defined in the x86 Instruction Set**

ISA is Embedded programming language of the CPU.

This includes the word size, processor register types, memory addressing modes, data formats and the instruction set that programmers use.

Ex: MIPS (Microprocessor without Interlocked Pipelined Stages) is a Reduced Instruction Set Computer (RISC) instruction set architecture (ISA) developed by MIPS Technologies.

Two types of ISA

Reduced Instruction Set Architecture (RISC) –

- **Simpler Hardware**
- **An instruction set composed of a few basic steps for loading, evaluating, and storing operations just like a load command will load data, store command will store the data.**

Complex Instruction Set Architecture (CISC) –

Complex Hardware

Large set of instructions

ISA is Embedded programming language of the CPU.

Two types of ISA

Reduced Instruction Set Architecture (RISC) –

Complex Instruction Set Architecture (CISC) –

RISC vs. CISC

CISC	RISC
Emphasis on hardware	Emphasis on software
Multiple instruction sizes and formats	Instructions of same set with few formats
Less registers	Uses more registers
More addressing modes	Fewer addressing modes
Extensive use of microprogramming	Complexity in compiler
Instructions take a varying amount of cycle time	Instructions take one cycle time
Pipelining is difficult	Pipelining is easy

Why Computer Architecture? Is it most important RESEARCH AREA

YES/NO

YES

Why?

To improve the performance.

What is Performance?

- Time
- Speed
- Accuracy
- Number of Instructions/sec

Advanced Computer Architecture

Performance $\propto$ speed

Performance $\propto$ 1/Time

Advanced Computer Architecture - Syllabus

ADVANCED COMPUTER ARCHITECTURES (Effective from the academic year 2018 -2019) SEMESTER – VIII			
Course Code	**18CS733**	**CIE Marks**	40
Number of Contact Hours/Week	3:0:0	**SEE Marks**	60
Total Number of Contact Hours	40	**Exam Hours**	03
CREDITS –3			

Course Learning Objectives: This course (18CS733) 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

Module 1	**Contact Hours**
Theory of Parallelism: Parallel Computer Models, The State of Computing, Multiprocessors and Multicomputer, Multivector and SIMD Computers, PRAM and VLSI Models, Program and Network Properties, Conditions of Parallelism, Program Partitioning and Scheduling, Program Flow Mechanisms, System Interconnect Architectures, Principles of Scalable Performance, Performance Metrics and Measures, Parallel Processing Applications, Speedup Performance Laws. For all Algorithm or mechanism any one example is sufficient. **Chapter 1 (1.1to 1.4), Chapter 2(2.1 to 2.4) Chapter 3 (3.1 to 3.3)** **RBT: L1, L2**	08
Module 2	
Hardware Technologies 1: Processors and Memory Hierarchy, Advanced Processor Technology, Superscalar and Vector Processors, Memory Hierarchy Technology, Virtual Memory Technology. For all Algorithms or mechanisms any one example is sufficient. **Chapter 4 (4.1 to 4.4)** **RBT: L1, L2, L3**	08

Advanced Computer Architecture - Syllabus

Module 3	
Hardware Technologies 2: Bus Systems, Cache Memory Organizations, Shared Memory Organizations, Sequential and Weak Consistency Models, Pipelining and Superscalar Techniques, Linear Pipeline Processors, Nonlinear Pipeline Processors. For all Algorithms or mechanisms any one example is sufficient. **Chapter 5 (5.1 to 5.4) Chapter 6 (6.1 to 6.2)** **RBT: L1, L2, L3**	08
Module 4	
Parallel and Scalable Architectures: Multiprocessors and Multicomputers, Multiprocessor System Interconnects, Cache Coherence and Synchronization Mechanisms, Message-Passing Mechanisms, Multivector and SIMD Computers, Vector Processing Principles, Multivector Multiprocessors, Compound Vector Processing, Scalable, Multithreaded, and Dataflow Architectures, Latency-Hiding Techniques, Principles of Multithreading, Fine-Grain Multicomputers. For all Algorithms or mechanisms any one example is sufficient. **Chapter 7 (7.1,7.2 and 7.4) Chapter 8(8.1 to 8.3) Chapter 9(9.1 to 9.3)** **RBT: L1, L2, L3**	08
Module 5	
Software for parallel programming: Parallel Models, Languages, and Compilers ,Parallel Programming Models, Parallel Languages and Compilers, Dependence Analysis of Data Arrays. Instruction and System Level Parallelism, Instruction Level Parallelism, Computer Architecture, Contents, Basic Design Issues, Problem Definition, Model of a Typical	08

Advanced Computer Architecture - Syllabus

Processor, Compiler-detected Instruction Level Parallelism ,Operand Forwarding ,Reorder Buffer, Register Renaming ,Tomasulo's Algorithm. For all Algorithms or mechanisms any one example is sufficient. **Chapter 10(10.1 to 10.3) Chapter 12(12.1 to 12.9)** **RBT: L1, L2, L3**	

Advanced Computer Architecture - Syllabus

Textbooks

Kai Hwang and Naresh Jotwani, "Advanced Computer Architecture (SIE): Parallelism, Scalability, Programmability", McGraw Hill Education 3/e. 2015

Reference Books:

John L. Hennessy and David A. Patterson, "Computer Architecture: A quantitative approach", 5th edition, Morgan Kaufmann Elseveir, 2013

1.1.2 Elements of Modern Computers

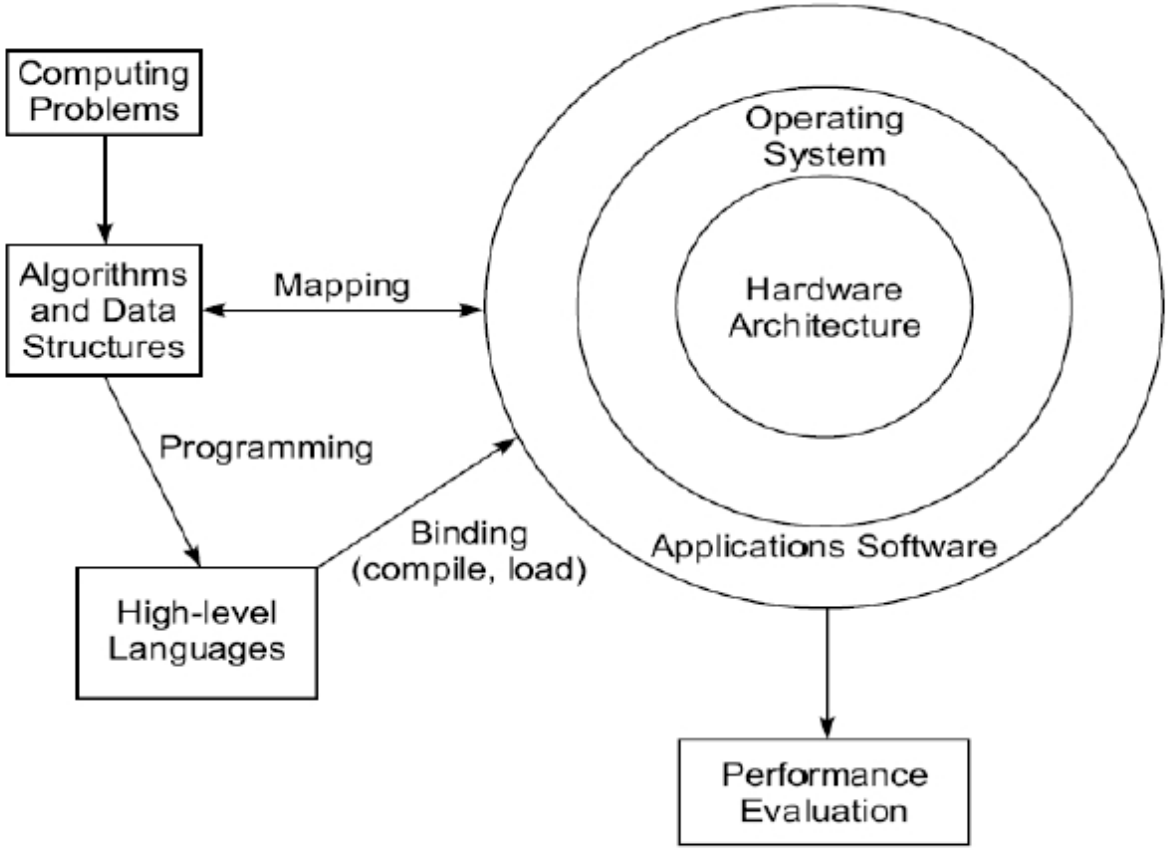


Fig. 1.1 Elements of a modern computer system

1. Computing Problems

The use of a computer is driven by real-life problems demanding fast and accurate solutions. Depending on the nature of the problems, the solutions may require different computing resources.

- ***Numerical computing:*** For numerical problems in science and technology, the solutions demand complex mathematical formulations and tedious integer or floating-point computations.
- ***Transaction processing:*** For numerical problems in business and government, the solutions demand accurate transactions, large database management, and information retrieval operations.
- ***Logical reasoning:*** For artificial intelligence (AI) problems, the solutions demand logic inferences and symbolic manipulations.
- Some complex problems may demand a combination of these processing modes.

Theory of Parallelism
Elements of Modern Computers contd…

2. Hardware Resources

- A modern computer system demonstrates its power through coordinated efforts by **hardware resources, an operating system**, and **application** software.
- Hardware core of a computer system are **Processors, memory, and peripheral devices.**
- **Special hardware interfaces** are often built into I/O devices, such as terminals, workstations, optical page scanners, magnetic ink character recognizers, modems, file servers, voice data entry, printers, and plotters.
- These peripherals are connected to mainframe computers directly or through local or wide-area networks.

3. Operating System

- An effective operating system manages the allocation and deallocation of resources during the execution of user programs.
- Beyond the OS, application software must be developed to benefit the users.
- Standard benchmark programs are needed for performance evaluation.
- Mapping is a bidirectional process matching algorithmic structure with hardware architecture, and vice versa.
- Efficient mapping will benefit the programmer and produce better source codes.
- The mapping of algorithmic and data structures onto the machine architecture includes processor scheduling, memory maps, interprocessor communications, etc.
- These activities are usually architecture-dependent.

Theory of Parallelism
Elements of Modern Computers contd…

System Software Support

- Software support is needed for the development of efficient programs in high-level languages. The source code written in a HLL must be first translated into object code by an optimizing compiler.
- The compiler assigns variables to registers or to memory words and reserves functional units for operators.
- An assembler is used to translate the compiled object code into machine code which can be recognized by the machine hardware. A loader is used to initiate the program execution through the OS kernel.

Compiler Support

There are three compiler upgrade approaches:

- **Preprocessor:** A preprocessor uses a sequential compiler and a low-level library of the target computer to implement high-level parallel constructs.
- **Precompiler**: The precompiler approach requires some program flow analysis, dependence checking, and limited optimizations toward parallelism detection.
- **Parallelizing Compiler**: This approach demands a fully developed parallelizing or vectorizing compiler which can automatically detect parallelism in source code and transform sequential codes into parallel constructs.

1.1.3 Evolution of Computer Architecture

- The study of computer architecture involves both hardware organization and programming/software requirements.
- As seen by an assembly language programmer, computer architecture is abstracted by its instruction set, which includes opcode (operation codes), addressing modes, registers, virtual memory, etc.
- From the hardware implementation point of view, the abstract machine is organized with CPUs, caches, buses, microcode, pipelines, physical memory, etc.
- Therefore, the study of architecture covers both instruction-set architectures and machine implementation organizations.

Theory of Parallelism

Evolution of Computer Architecture contd…

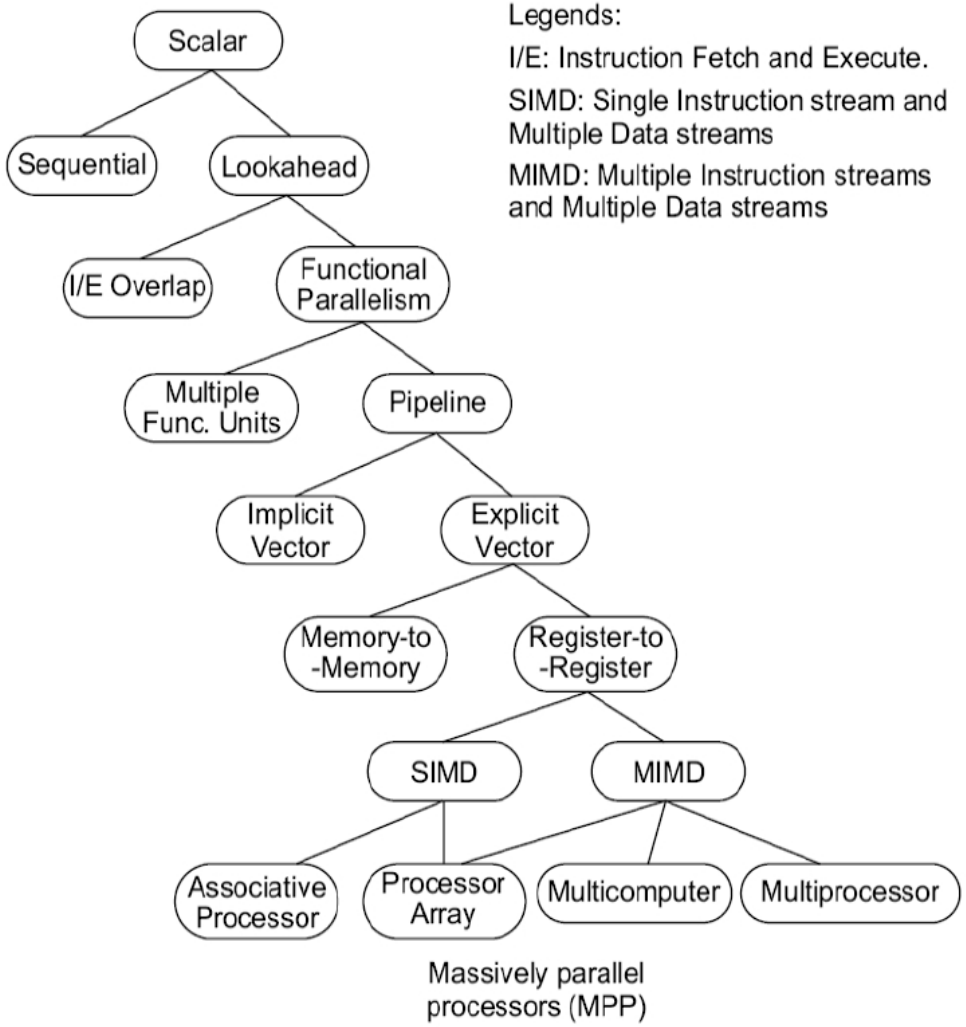


Fig. 1.2 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers

Flynn's Classification

Michael Flynn (1972) introduced a classification of various computer architectures based on notions of instruction and data streams.

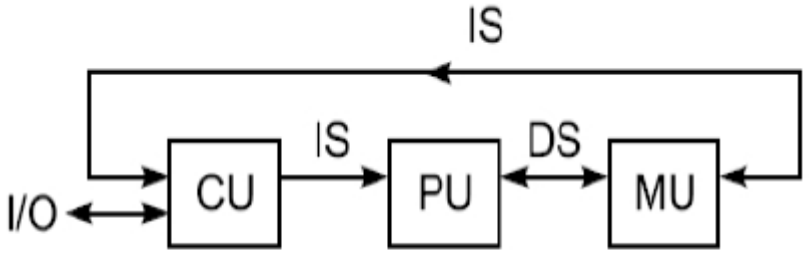
1. **SISD** (Single Instruction stream over a Single Data stream) computers
2. **SIMD** (Single Instruction stream over Multiple Data streams) machines
3. **MIMD** (Multiple Instruction streams over Multiple Data streams) machines.
4. **MISD** (Multiple Instruction streams and a Single Data stream) machines

Captions:

CU = Control Unit
PU = Processing Unit
MU = Memory Unit
IS = Instruction Stream
DS = Data Stream
PE = Processing Element
LM = Local Memory

Flynn's Classification

SISD (Single Instruction stream over a Single Data stream) computers

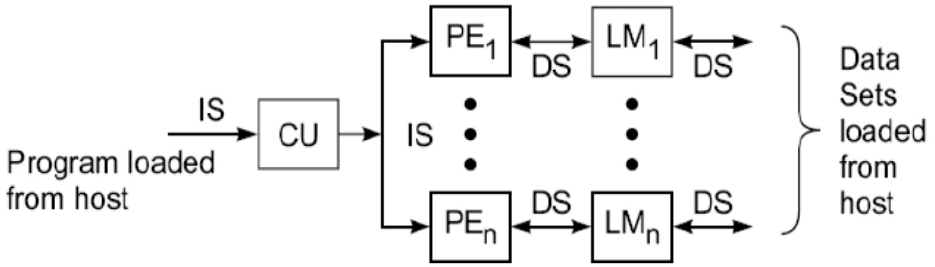


(a) SISD uniprocessor architecture

- Conventional sequential machines are called SISD computers.
- They are also called scalar processor i.e., one instruction at a time and each instruction have only one set of operands.
- **Single instruction:** only one instruction stream is being acted on by the CPU during any one clock cycle
- **Single data:** only one data stream is being used as input during any one clock cycle
- Deterministic execution
- Instructions are executed sequentially.
- This is the oldest and until recently, the most prevalent form of computer
- Examples: most PCs, single CPU workstations and mainframes

Module 1 - Theory of Parallelism
Flynn's Classification

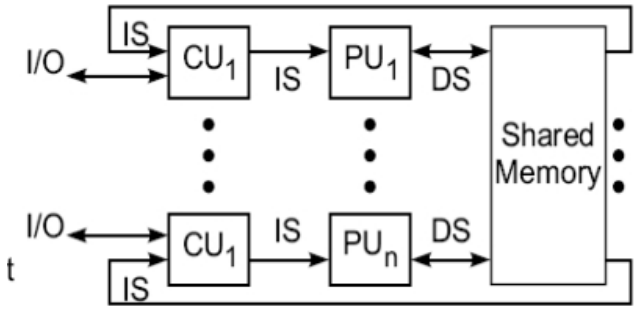
SIMD (Single Instruction stream over Multiple Data streams) machines



(b) SIMD architecture (with distributed memory)

- A type of parallel computer
- **Single instruction:** All processing units execute the same instruction issued by the control unit at any given clock cycle.
- **Multiple data**: Each processing unit can operate on a different data element. The processors are connected to shared memory or interconnection network providing multiple data to processing unit.
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Thus single instruction is executed by different processing unit on different set of data.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing and vector computation.
- Synchronous (lockstep) and deterministic execution.
- Two varieties: Processor Arrays e.g., Connection Machine CM-2, Maspar MP-1, MP-2 and Vector Pipelines processor e.g., IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820

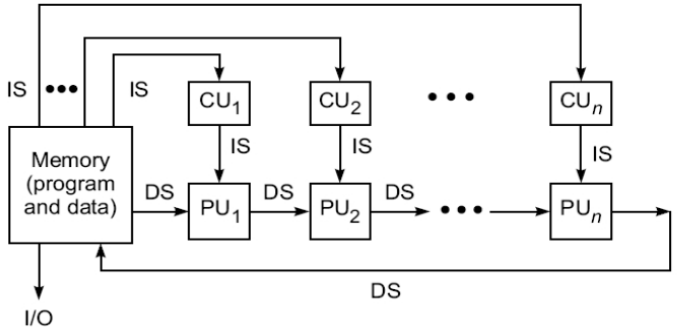
MIMD (Multiple Instruction streams over Multiple Data streams) machines.



(c) MIMD architecture (with shared memory)

- **Multiple Data:** Every processor may be working with a different data stream, multiple data stream is provided by shared memory.
- Can be categorized as loosely coupled or tightly coupled depending on sharing of data and control.
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- There are multiple processors each processing different tasks.
- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.

MISD (Multiple Instruction streams and a Single Data stream) machines



(d) MISD architecture (the systolic array)

Fig. 1.3 Flynn's classification of computer architectures (Derived from Michael Flynn,

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- A single data stream is forwarded to different processing unit which are connected to different control unit and execute instruction given to it by control unit to which it is attached.
- Thus in these computers same data flow through a linear array of processors executing different instruction streams.
- This architecture is also known as Systolic Arrays for pipelined execution of specific instructions.
- Some conceivable uses might be:

1. multiple frequency filters operating on a single signal stream

2. multiple cryptography algorithms attempting to crack a single coded message. Multiple Instructions: Every processor may be executing a different instruction stream

Development Layers

A layered development of parallel computers is illustrated in Fig. 1.4.

Hardware configurations differ from machine to machine, even those of the same model.

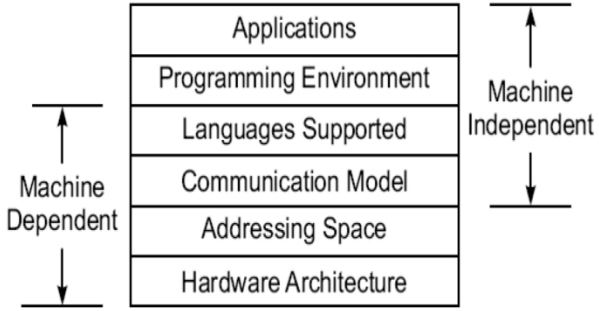


Fig. 1.4 Six layers for computer system development (Courtesy of Lionel Ni, 1990)

- **Address Space** of a processor in a computer system varies among different architectures. **It depends on the memory organization, which is machine-dependent.** These features are up to the designer and should match the target application domains.
- **Application Programs and Programming Environments** which are **machine-independent.** Independent of machine architecture, the user programs can be ported to many computers with minimum conversion costs.
- **High-level languages and Communication Models** **depend on the architectural choices made in a computer system.** From a programmer's viewpoint, these two layers should be architecture-transparent.

- However, the Communication Models, shared variables versus message passing, are mostly **machine-dependent.** The Linda approach using tuple spaces offers architecture transparent Communication model for parallel computers.
- Application programmers prefer more architectural transparency. However, kernel programmers have to explore the opportunities supported by hardware.
- As a good computer architect, one has to approach the problem from both ends.
- The compilers and OS support should be designed to remove as many architectural constraints as possible from the programmer.

System Attributes affecting Performance

Clock Rate and CPI (Cycles Per Instruction)

- The CPU (or simply the processor) of today's digital computer is driven by a clock with a constant cycle time ((τ – tau) in nanoseconds).
- The inverse of the cycle time is the clock rate (/ = 1/ τ in megahertz).
- The size of a program is determined by its instruction count (IC), in terms of the number of machine instructions to be executed in the program.
- Different machine instructions may require different numbers of clock cycles to execute.
  - Therefore, the Cycles Per Instruction (CPI) becomes an important parameter for measuring the time needed to execute each instruction.
- For a given instruction set, we can calculate an average CPI over all instruction types, provided we know their frequencies of appearance in the program.
- An accurate estimate of the average CPI requires a large amount of program code to be traced over a long period of time.
- Unless specifically focusing on a single instruction type, we simply use the term CPI to mean the average value with respect to a given instruction set and a given program mix.

Performance Factors

- Let IC be the number of instructions in a given program, or the instruction count.
- The CPU time *(T* in seconds/program) needed to execute the program is estimated by finding the product of three contributing factors:

$$T = IC \times CPI \times \tau \qquad (1.1)$$

- The execution of an instruction requires going through a cycle of events involving the
  - Instruction Fetch (MEMORY ACCESS)
  - Decode (WITHIN CPU)
  - Operand(s) fetch (MEMORY ACCESS)
  - Execution (WITHIN CPU)
  - Store results (MEMORY ACCESS)
- In this cycle, only the instruction **decode and execution phases are carried out in the CPU**.
- The remaining three operations may be required to access the memory. We define a ***memory cycle* as the time needed to complete one memory reference.**

Performance Factors condt...

Performance Factors

- Memory cycle is the time **needed to complete one memory reference**.
- Usually, a memory cycle is **k** times the processor cycle **τ**.
- The value of **k** depends on the speed of the memory technology and processor-memory interconnection scheme used.
- The CPI of an instruction type can be divided into two component to complete the execution of the instruction.
  - Total processor cycles
  - Memory cycles needed
- **Depending on the instruction type, the complete instruction cycle may involve one to four memory references**
  - **one memory cycle for instruction fetch**
  - **two memory cycle for operand fetch**
  - **one memory cycle for store results.**

Therefore we can rewrite Eq. 1.1 as follows;

$$T = IC \times (p + m \times k) \times \tau \qquad (1.2)$$

where **p** is the number of processor cycles needed for the instruction decode and execution,

m is the number of memory references needed,

k is the ratio between memory cycle and processor cycle,

IC is the instruction count,

τ is the processor cycle time.

p is the number of processor cycles needed for the instruction decode and execution,

m is the number of memory references needed,

k is the ratio between memory cycle and processor cycle,

IC is the instruction count,

τ is the processor cycle time.

System Attributes

- The above five performance factors (IC, p, *m, k,* τ) **are influenced by four system attributes:**
  - Instruction-Set Architecture
  - Compiler technology
  - CPU implementation and control
  - Cache and memory hierarchy, as specified in Table 1.2.
- The instruction-set architecture affects the **program length (IC) and processor cycle needed (p).**
- The compiler technology affects the values of IC, p and the memory reference count (m).
- The CPU implementation and control determine the total processor time *(p. τ)* needed.
- Finally, the memory technology and hierarchy design affect the **memory access latency (k. τ).** The above CPU time can be used as a basis in estimating the execution rate of a processor.

System Attributes

Table 1.2 Performance Factors Versus System Attributes

System Attributes	Performance Factors				
	Instr. Count,	Average Cycles per Instruction, CP1			Processor Cycle Time, T
		Processor Cycles per Instruction, p	Memory References per Instruction, m	Memory-Access Latency, k	
Instruction-set Architecture	X	X			
Compiler Technology	X	X	X		
Processor Implementation and Control		X			X
Cache and Memory Hierarchy				X	X

System Attributes

MIPS Rate (Million Instructions Per Second)

- Let C be the total number of clock cycles needed to execute a given program.
- Then the CPU time in Eq. 1.2 can be estimated as

$$T = C \times \tau = C/f. \quad (\tau = 1/f \text{ clock rate})$$

- Furthermore,

$$CPI = C/IC \text{ and}$$

$$T = IC \times CPI \times \tau = IC \times CPI/f.$$

The processor speed is often measured in terms of Million Instructions Per Second (MIPS).

- We simply call it the MIPS rate of a given processor. It should be emphasized that the MIPS rate varies with respect to a number of factors such as
- Clock rate (f)
- Instruction Count (IC)
- CPI of a given machine

System Attributes

$T = IC \times CPI \times \tau = IC \times CPI/f.$

$T = C/f$

$CPI = C/IC$

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{\text{CPI} \times 10^6} = \frac{f \times I_c}{C \times 10^6} \qquad (1.3)$$

$$T = IC \times (p + m \times k) \times \tau \qquad (1.2)$$

- **Based on Eq. 1.3, the CPU time in Eq. 1.2 can also be written as $T = IC \times 10^6 / MIPS$.**
- **Based on the system attributes identified in Table 1.2 and the above derived expressions, we conclude by indicating the fact that the**

"MIPS rate of a given computer is directly proportional to the clock rate and inversely proportional to the CPI"

- **All four system attributes, instruction set, compiler, processor, and memory technologies, affect the MIPS rate, which varies also from program to program.**

Throughput Rate

- System throughput Ws (unit for Ws is programs/second) -- Number of programs a system can execute per unit time, called the System throughput
- In a multiprogrammed system, the system throughput is often lower than the CPU throughput Wp defined by:

$$W_p = \frac{f}{I_c \times \text{CPI}} \qquad (1.4)$$

- Note that Wp = (MIPS) X 10^6/IC from Eq. 1.3- The unit for Wp is programs/second.
- The CPU throughput is a measure of how many programs can be executed per second, based on the MIPS rate and average program length (IC).
- The reason why Ws < Wp is due to the additional system overheads caused by the I/O, compiler, and OS when multiple programs are interleaved for CPU execution by multiprogramming or timesharing operations.
- If the CPU is kept busy in a perfect program-interleaving fashion, then Ws = Wp. This will probably never happen, since the system overhead often causes an extra delay and the CPU may be left idle for some cycles.

Programming Environments

- Programmability depends on the programming environment provided to the users.
- Conventional computers are used in a sequential programming environment with tools developed for a uniprocessor computer.
- Parallel computers need parallel tools that allow specification or easy detection of parallelism and operating systems that can perform parallel scheduling of concurrent events, shared memory allocation, and shared peripheral and communication links.

Programming Environments

- Implicit Parallel
- Explicit Parallel

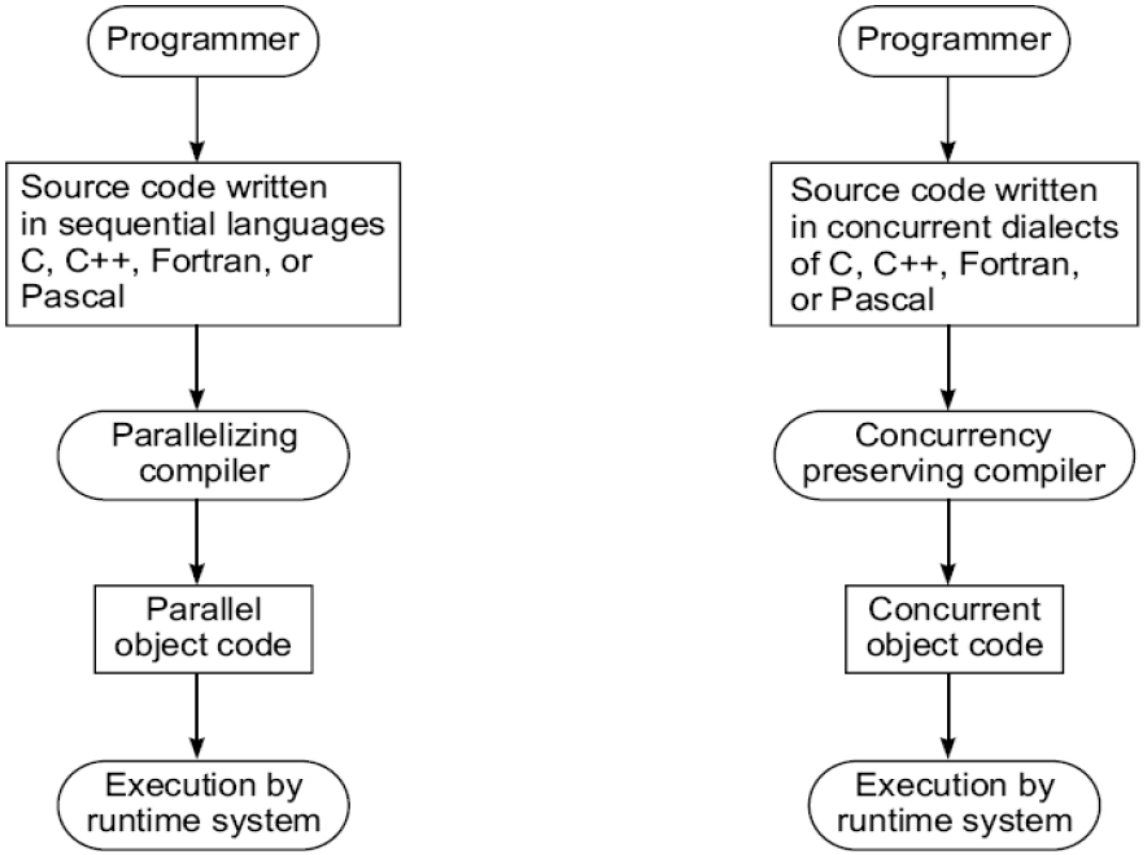


Fig. 1.5 Two approaches to parallel programming (Courtesy of Charles Seitz; adapted with permission from

Implicit Parallelism

- An implicit approach uses a conventional language, such as C, Fortran, Lisp, or Pascal, to write the source program.
- The sequentially coded source program is translated into parallel object code by a parallelizing compiler.
- The compiler must be able to detect parallelism and assign target machine resources. This compiler approach has been applied in programming shared-memory multiprocessors.
- With parallelism being implicit, success relies heavily on the "intelligence" of a parallelizing compiler.
- This approach requires less effort on the part of the programmer.

Explicit Parallelism

- The second approach (Fig. 1.5b) requires more effort by the programmer to develop a source program using parallel dialects of C, Fortran, Lisp, or Pascal.
- Parallelism is explicitly specified in the user programs.
- This will significantly reduce the burden on the compiler to detect parallelism.
- Instead, the compiler needs to preserve parallelism and, where possible, assigns target machine resources.

Two categories of parallel computers are architecturally modeled below. These physical models are distinguished by having a shared common memory or unshared distributed memories.

1. Shared-Memory Multiprocessors

There are 3 shared-memory multiprocessor models:

i. Uniform Memory-Access (UMA) model
ii. Non Uniform-Memory-Access (NUMA) model
iii. Cache-Only Memory Architecture (COMA) model

These models differ in how the memory and peripheral resources are shared or Distributed.

a. Uniform Memory-Access (UMA) model

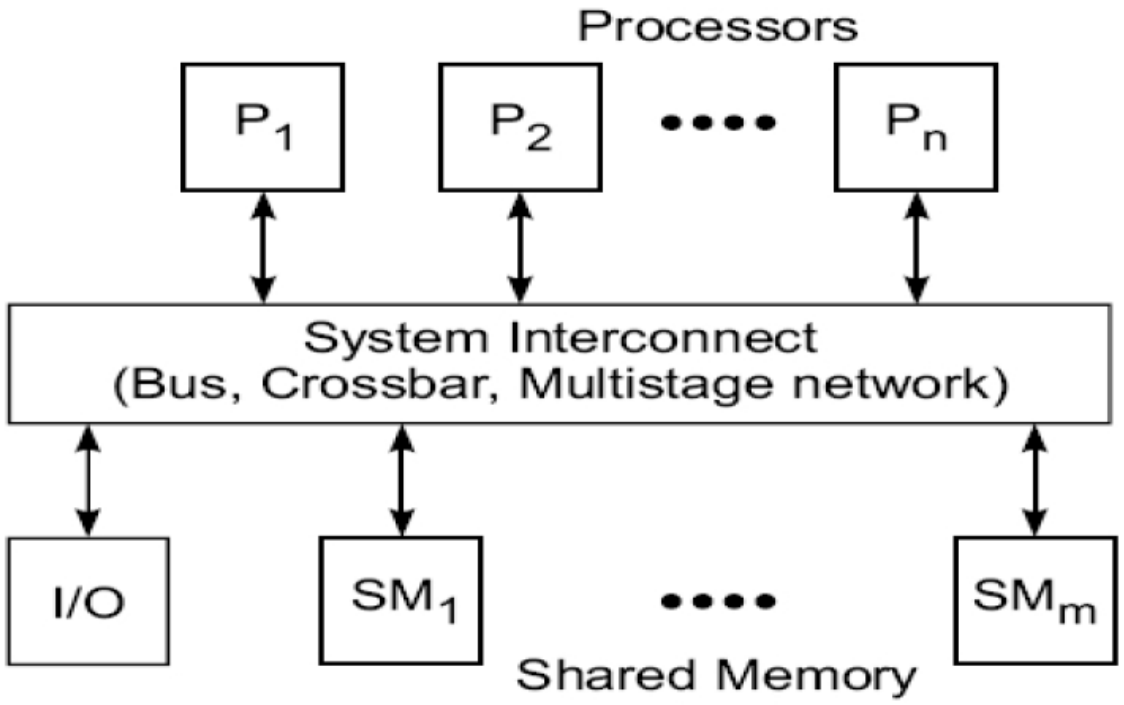
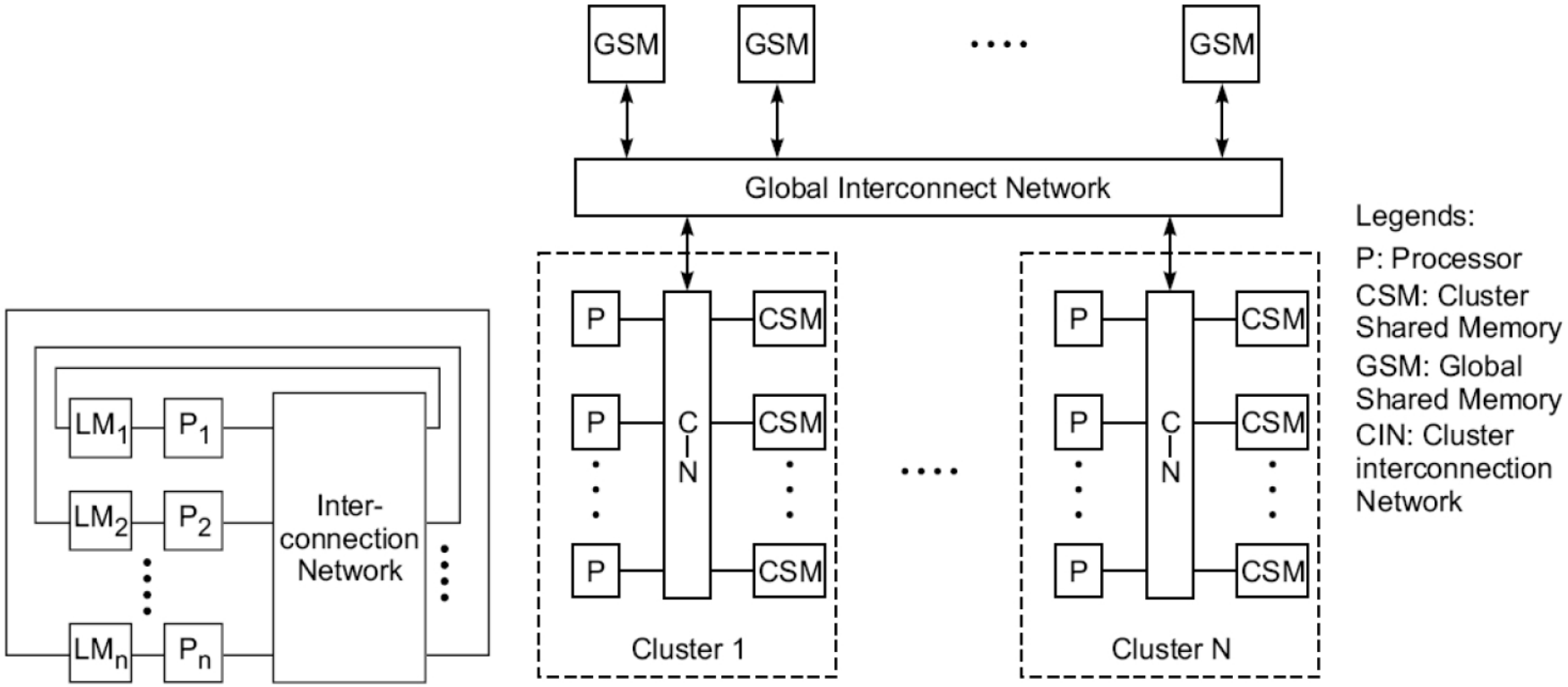


Fig. 1.6 The UMA multiprocessor model

a. Uniform Memory-Access (UMA) model contd…

- In a UMA multiprocessor model (Fig. 1.6), the physical memory is uniformly shared by all the processors. these systems are also called a symmetric multiprocessor.
- All processors have equal access time to all memory words, which is why it is called uniform memory access.
- Each processor may use a private cache. Peripherals are also shared in some fashion.
- Multiprocessors are called tightly coupled systems due to the high degree of resource sharing. The system interconnect takes the form of a common bus, a crossbar switch, or a multistage network.
- Most computer manufacturers have *multiprocessor* (MP) extensions of their *uniprocessor* (UP) product line.
- The UMA model is suitable for general-purpose and timesharing applications by multiple users.
- It can be used to speed up the execution of a single large program in time-critical applications. To coordinate parallel events, synchronization and communication among processors are done through using shared variables in the common memory.
- When all processors have equal access to all peripheral devices, the system is called a symmetric multiprocessor. In this case, all the processors are equally capable of running the executive programs, such as the OS kernel and I/O service routines.

b. Non uniform-Memory-Access (NUMA) 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)

Fig. 1.7 Two NUMA models for multiprocessor systems

b. Non uniform-Memory-Access (NUMA) model contd…

- A NUMA multiprocessor is a shared-memory system in which the access time varies with the location of the memory word.
- Two NUMA machine models are depicted in Fig. 1.7.
- The shared memory is physically distributed to all processors, called *local memories.*
- The collection of all local memories forms a global address space accessible by all processors.
- It is faster to access a local memory with a local processor. The access of remote memory attached to other processors takes longer due to the added delay through the interconnection network.
- The BBN TC-2000 Butterfly multiprocessor assumes the configuration shown in Fig. 1.7a.

c. Cache-Only Memory Architecture (COMA) model

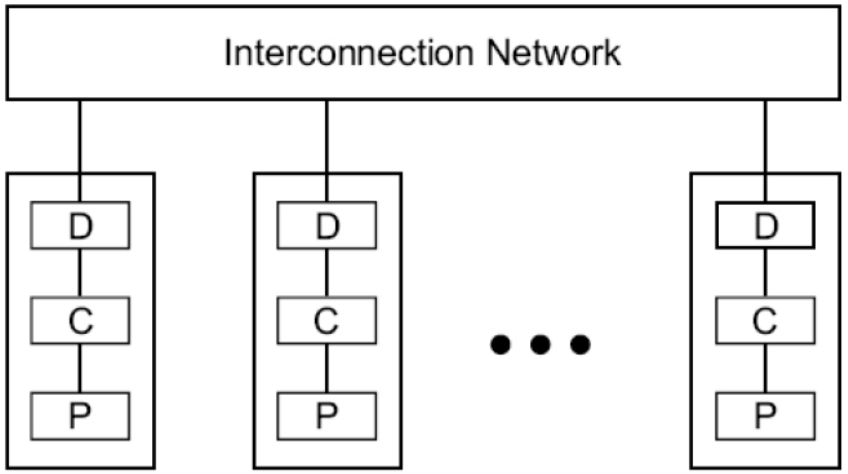


Fig. 1.8 The COMA model of a multiprocessor (P: Processor, C: Cache, D: Directory; e.g. the KSR-1)

c. Cache-Only Memory Architecture (COMA) model condt…

- A multiprocessor using cache-only memory assumes the COMA model.
- The COMA model is a special case of a NUMA machine, in which the distributed main memories are converted to caches.
- There is no memory hierarchy at each processor node. All the caches form a global address space.
- Remote cache access is assisted by the distributed cache directories (D in Fig. 1.8).
- Depending on the interconnection network used, sometimes hierarchical directories may be used to help locate copies of cache blocks.
- Initial data placement is not critical because data will eventually migrate to where it will be used.

2. Distributed-Memory Multicomputers

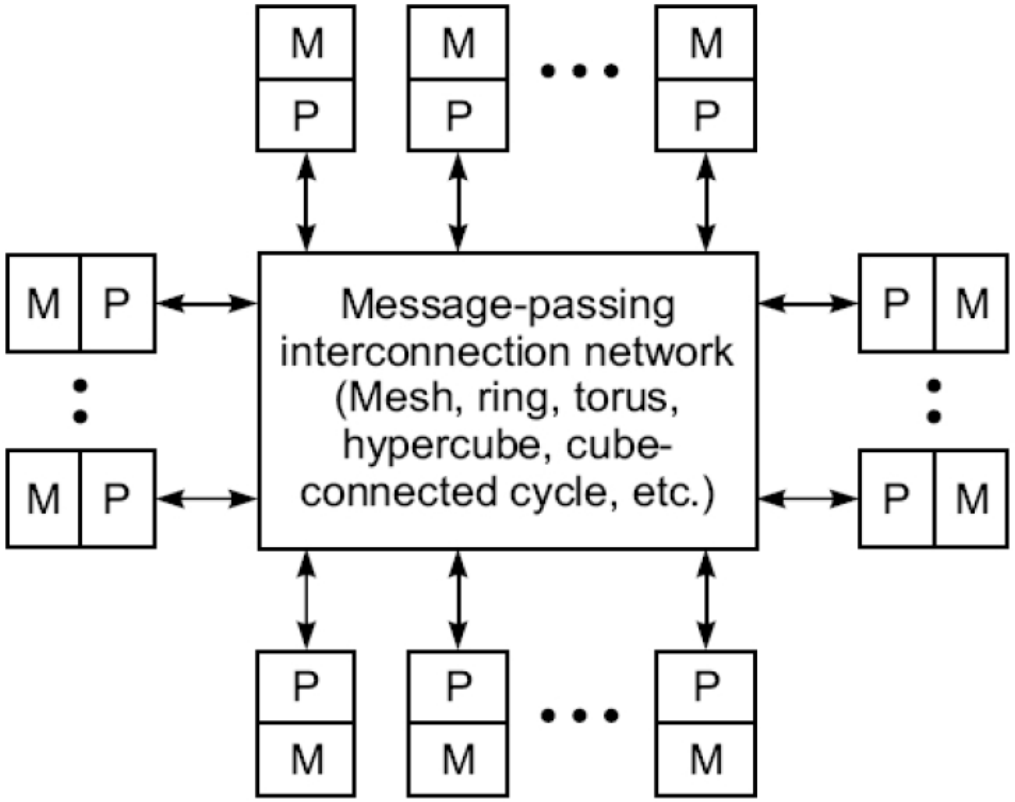


Fig. 1.9 Generic model of a message-passing multicomputer

2. Distributed-Memory Multicomputers condt…

- A distributed-memory multicomputer system is modeled in the above figure consists of multiple computers, often called *nodes*, interconnected by a message-passing network.
- Each node is an autonomous computer consisting of a processor, local memory, and sometimes attached disks or I/O peripherals.
- The message-passing network provides point-to-point static connections among the nodes.
- All local memories are private and are accessible only by local processors.
- For this reason, traditional multicomputers have been called *No-Remote-Memory-Access* (NORMA) machines.
- However, this restriction will gradually be removed in future multi computers with distributed shared memories. Internode communication is carried out by passing messages through the static connection network.

Multivector and SIMD Computers

We can classify super computers as:

i. Pipelined vector machines using a few powerful processors equipped with vector hardware

ii. SIMD computers emphasizing massive data parallelism

Vector Supercomputers

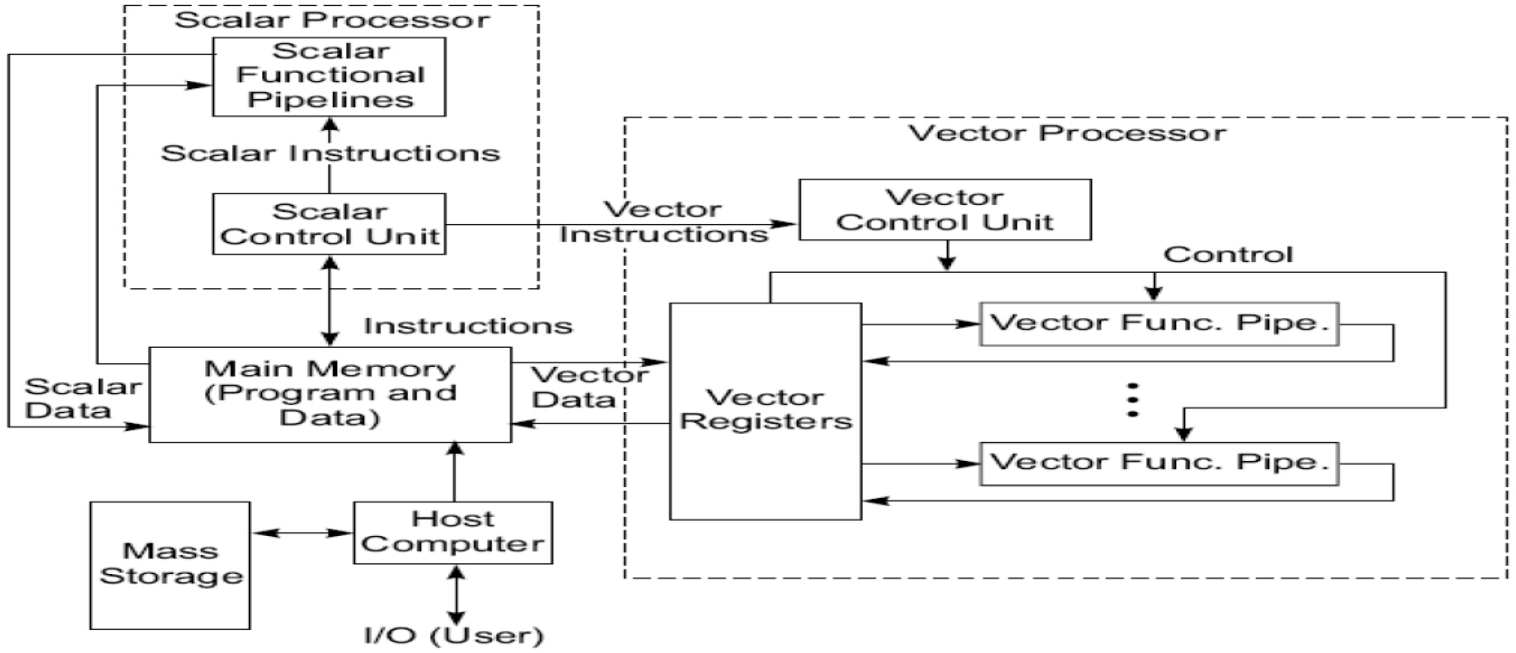


Fig. 1.11 The architecture of a vector supercomputer

Multivector and SIMD Computers

- A vector computer is often built on top of a scalar processor.
- As shown in Fig. 1.11, the vector processor is attached to the scalar processor as an optional feature.
- Program and data are first loaded into the main memory through a host computer.
- All instructions are first decoded by the scalar control unit.
- If the decoded instruction is a scalar operation or a program control operation, it will be directly executed by the scalar processor using the scalar functional pipelines.
- If the instruction is decoded as a vector operation, it will be sent to the vector control unit.
- This control unit will supervise the flow of vector data between the main memory and vector functional pipelines.
- The vector data flow is coordinated by the control unit. A number of vector functional pipelines may be built into a vector processor.

Multivector and SIMD Computers

Vector Processor Models

- Figure l.ll shows a register-to-register architecture.
- Vector registers are used to hold the vector operands, intermediate and final vector results.
- The vector functional pipelines retrieve operands from and put results into the vector registers.
- All vector registers are programmable in user instructions.
- Each vector register is equipped with a component counter which keeps track of the component registers used in successive pipeline cycles.
- The length of each vector register is usually fixed, say, sixty-four 64-bit component registers in a vector register in a Cray Series supercomputer.
- Other machines, like the Fujitsu VP2000 Series, use reconfigurable vector registers to dynamically match the register length with that of the vector operands.

Multivector and SIMD Computers

SIMD Super computers

SIMD computers have a single instruction stream over multiple data streams.

An operational model of an SIMD computer is specified by a 5-tuple: M = (N,C, I,M, R) where

1. N is the number of processing elements (PEs) in the machine. For example, the Illiac IV had 64 PEs and the Connection Machine CM-2 had 65,536 PEs.

2. C is the set of instructions directly executed by the control unit (CU), including scalar and program flow control instructions.

3. I is the set of instructions broadcast by the CU to all PEs for parallel execution. These include arithmetic, logic, data routing, masking, and other local operations executed by each active PE over data within that PE.

4. M is the set of masking schemes, where each mask partitions the set of PEs into enabled and disabled subsets.

R is the set of data-routing functions, specifying various patterns to be set up in the interconnection network for inter-PE communications.

Multivector and SIMD Computers

SIMD Super computers

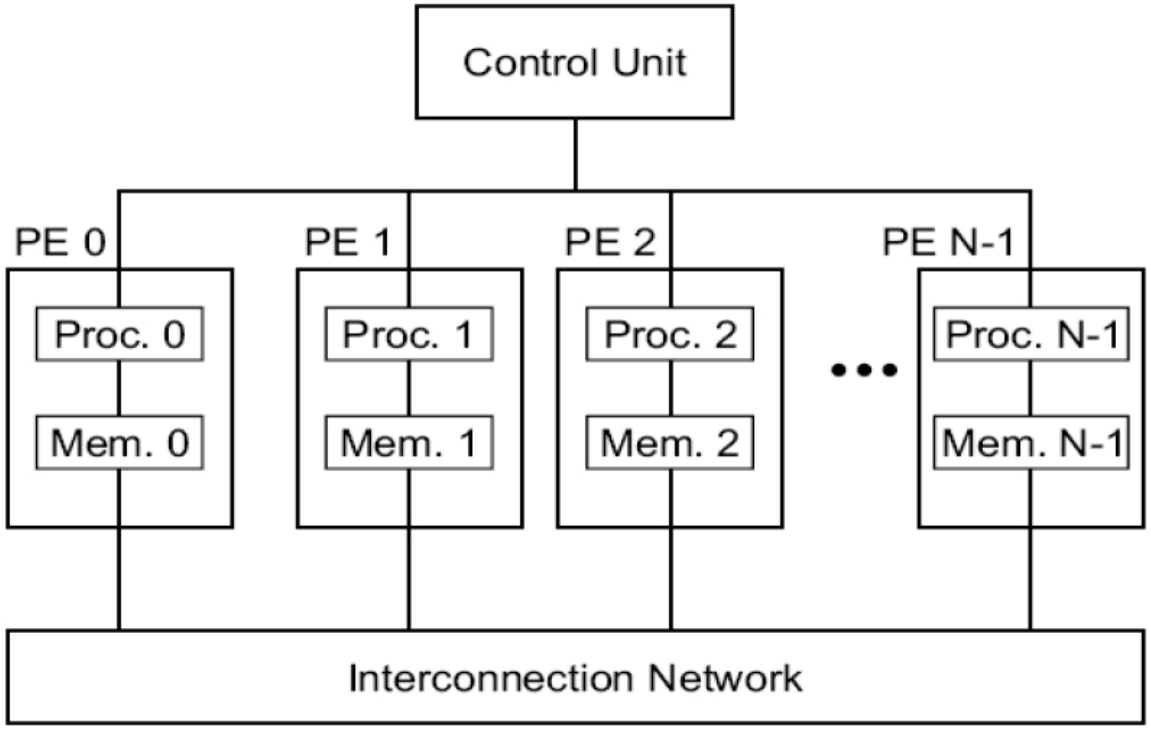


Fig. 1.12 Operational model of SIMD computers

PRAM AND VLSI MODELS

PRAM model (Parallel Random Access Machine)

- PRAM is a theoretical model of parallel computation in which an arbitrary but finite number of processors can access any value in an arbitrarily large shared memory in a single time step.
- Processors may execute different instruction streams, but work synchronously. This model assumes a shared memory, multiprocessor machine as shown:
- The machine size n can be arbitrarily large
- The machine is synchronous at the instruction level. That is, each processor is executing it's own series of instructions, and the entire machine operates at a basic time step (cycle). Within each cycle, each processor executes exactly one operation or does nothing, i.e. it is idle.
- An instruction can be any random access machine instruction, such as: fetch some operands from memory, perform an ALU operation on the data, and store the result back in memory.
- All processors implicitly synchronize on each cycle and the synchronization overhead is assumed to be zero.
- Communication is done through reading and writing of shared variables.
- Memory access can be specified to be UMA, NUMA, EREW, CREW, or CRCW with a defined conflict policy.
- The PRAM model can apply to SIMD class machines if all processors execute identical instructions on the same cycle or to MIMD class machines if the processors are executing different instructions.
- Load imbalance is the only form of overhead in the PRAM model.

PRAM AND VLSI MODELS

PRAM model (Parallel Random Access Machine)

•

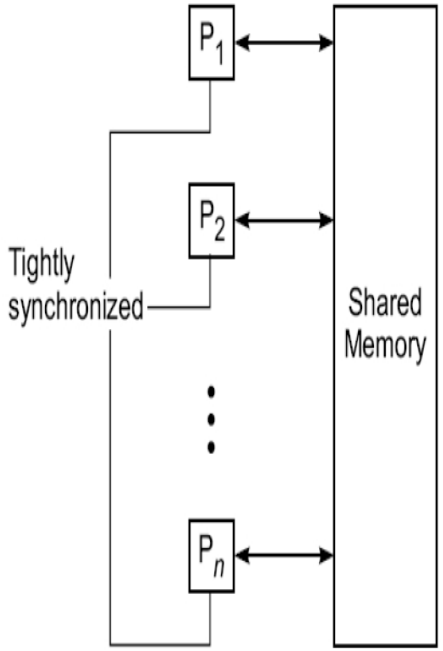


Fig. 1.14 PRAM model of a multiprocessor system with shared memory, on which all *n* processors operate in lockstep in memory access and program execution operations. Each processor can access any memory location in unit time

PRAM AND VLSI MODELS

PRAM model (Parallel Random Access Machine)

An n-processor PRAM (Fig. 1.14) has a globally addressable memory.
The shared memory can be distributed among the processors or centralized in one place. The n processors operate on a synchronized read-memory, compute, and write-memory cycle. With shared memory, the model must specify how concurrent read and concurrent write of memory are handled.
Four memory-update options are possible:

- Exclusive Read (ER) — This allows at most one processor to read from any memory location in each cycle, a rather restrictive policy.
- Exclusive Write (EW) — This allows at most one processor to write into a memory location at a time.
- Concurrent Read (CR) — This allows multiple processors to read the same information from the same memory cell in the same cycle.

Concurrent Write (CW) — This allows simultaneous writes to the same memory location.
In order to avoid confusion, some policy must be set up to resolve the write conflicts. Various combinations of the above options lead to several variants of the PRAM model as specified below.

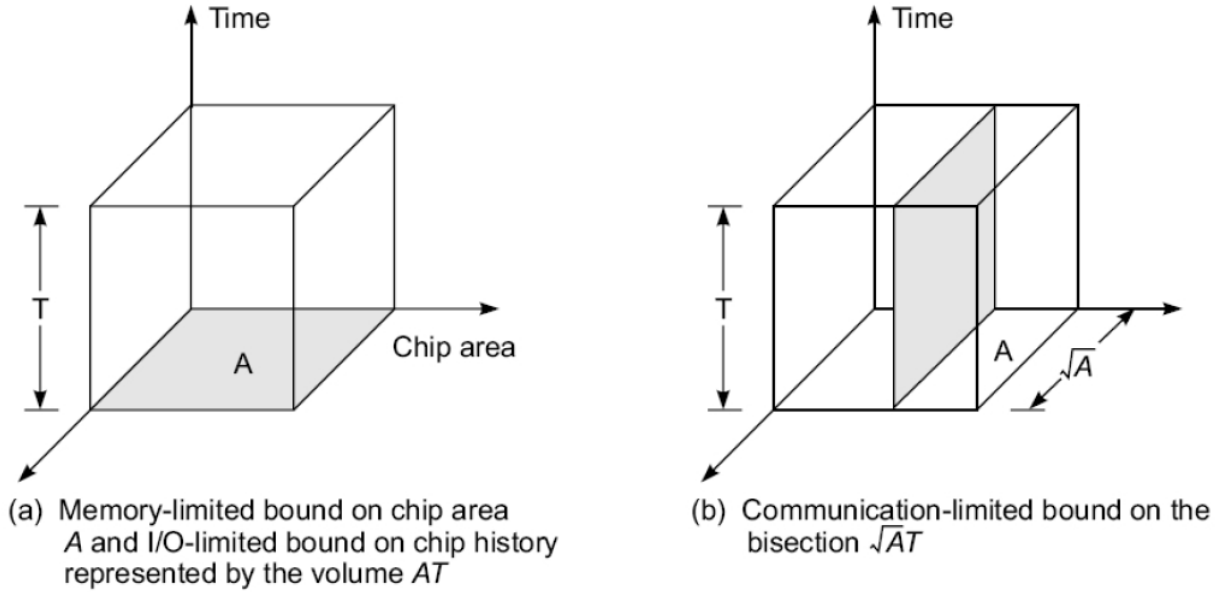
PRAM AND VLSI MODELS

PRAM model (Parallel Random Access Machine)

PRAM Variants

- There are 4 variants of the PRAM model, depending on how the memory reads and writes are handled.
- EREW – PRAM Model (Exclusive Read, Exclusive Write): This model forbids more than one processor from reading or writing the same memory cell simultaneously. This is the most restrictive PRAM model proposed.
- CREW – PRAM Model (Concurrent Read, Exclusive Write); The write conflicts are avoided by mutual exclusion. Concurrent reads to the same memory location arc allowed.
- ERCW – PRAM Model – This allows exclusive read or concurrent writes to the same memory location.
- CRCW – PRAM Model (Concurrent Read, Concurrent Write); This model allows either concurrent reads or concurrent writes to the same memory location.
-

- 1.4.2 VLSI Model
- **Parallel computers rely on the use of VLSI chips to fabricate the major components such as processor arrays memory arrays and large scale switching networks. The rapid advent of very large scale intergrated (VSLI) technology now computer architects are trying to implement parallel algorithms directly in hardware. An AT2 model is an example for two dimension VLSI chips**
-



(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

Condition of parallelism

Data and Resource Dependence

- The ability to execute several program segments in parallel requires each segment to be independent of the other segments. We use a dependence graph to describe the relations.
- The nodes of a dependence graph correspond to the program statement (instructions), and directed edges with different labels are used to represent the ordered relations among the statements.
- The analysis of dependence graphs shows where opportunity exists for parallelization and vectorization.

Condition of parallelism

1. Data dependence:

- The ordering relationship between statements is indicated by the data dependence. Five type of data dependence are defined below:

a. Flow dependence:

A statement S2 is flow dependent on S1 if an execution path exists from s1 to S2 and if at least one output (variables assigned) of S1 feeds in as input (operands to be used) to S2 also called RAW (Read After Write) hazard a $S_1 \rightarrow S_2$ s

Condition of parallelism

b. Antidependence:

Statement S2 is antidependent on the statement S1 if S2 follows S1 in the program order and if the output of S2 overlaps the input to S1 also called WAR (Write After Read) hazard and denoted as $S_1 \mapsto S_2$

3. Output dependence: Two statements are output dependent if they produce (write) the same output variable. Also called WAW (Write After Write) hazard and denoted as $S_1 \mapsto S_2$

4. I/O dependence: Read and write are I/O statements. I/O dependence occurs not because the same variable is involved but because the same file referenced by both I/O statement.

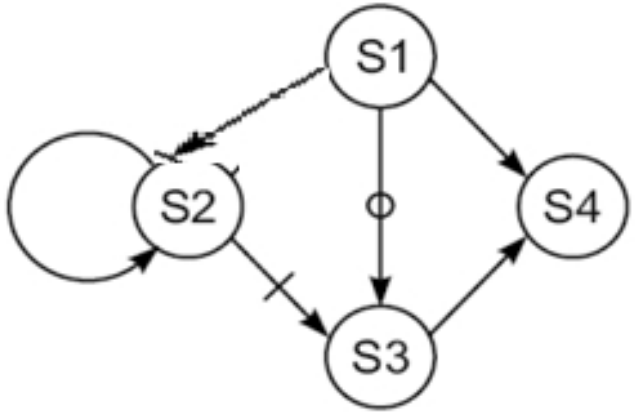
5. Unknown dependence: The dependence relation between two statements cannot be determined in the following situations:

- **The subscript of a variable is itself subscribed (indirect addressing)**
- **The subscript does not contain the loop index variable.**

Con

Example: Consider the following fragment of a program:

S1:	**Load**	**R1, A**	**/R1 ← Memory(A) /**
S2:	**Add**	**R2, R1**	**/R2 ← (R1) + (R2)/**
S3:	**Move**	**R1, R3**	**/R1 ← (R3)/**
S4:	**Store**	**B, R1**	**/Memory(B) ← (R1)/**



(a) Dependence graph

- Here the Flow dependency from **S1 to S2**, **S3 to S4**, **S2 to S2**
- Anti-dependency from **S2 to S3**
- Output dependency **S1 toS3**

S2:	**Rewind (4)**	/Process data/
S3:	**Write (4), B(I)**	/Write array B into file 4/
S4:	**Rewind (4)**	/Close file 4/

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

The read/write statements S1 and S2 are I/O dependent on each other because they both access the same file.

2. Control Dependence:

- This refers to the situation where the order of the execution of statements cannot be determined before run time.
- For example all condition statement, where the flow of statement depends on the output.
- Different paths taken after a conditional branch may depend on the data hence we need to eliminate this data dependence among the instructions.
- This dependence also exists between operations performed in successive iterations of looping procedure. Control dependence often prohibits parallelism from being exploited.

Control-independent example:

```
for (i=0; i<n; i++)
{
        a[i] = c[i];
        if (a[i] < 0) a[i] = 1;
}
```

Control-dependent example:

```
for (i=1; i<n; i++)
{
        if (a[i-1] < 0) a[i] = 1;
}
```

Control dependence also avoids parallelism to being exploited. Compilers are used to eliminate this control dependence and exploit the parallelism.

3. Resource dependence:

- Data and control dependencies are based on the independence of the work to be done.
- Resource independence is concerned with conflicts in using shared resources, such as registers, integer and floating point ALUs, etc. ALU conflicts are called ALU dependence.
- Memory (storage) conflicts are called storage dependence.

Bernstein's Conditions

- Bernstein's conditions are a set of conditions which must exist if two processes can execute in parallel.

Notation of Bernstein's Conditions

- Ia is the set of all input variables for a process Pa. Ia is also called the read set or domain of Pa. Oa is the set of all output variables for a process Pa . Oa is also called write set.
- If P_1 and P_2 can execute in parallel (which is written as $P_1 \parallel P_2$), then:

$$I_1 \cap O_2 = \varnothing$$

$$I_2 \cap O_1 = \varnothing$$

$$O_1 \cap O_2 = \varnothing$$

3. Resource dependence:

Bernstein's Conditions contd…

- In terms of data dependencies, Bernstein's conditions imply that two processes can execute in parallel if they are flow-independent, anti-independent, and output-independent.
- The parallelism relation || is commutative (P_i || P_j implies P_j || P_i), but not transitive (P_i || P_j and P_j || P_k does not imply P_i || P_k) .
- Therefore, || is not an equivalence relation. Intersection of the input sets is allowed.
- Example: Detection of parallelism in a program using Bernstein's conditions
- Consider the simple case in which each process is a single HLL statement. We want to detect the parallelism embedded in the following 5 statements labeled P1, P2, P3, P4, P5 in program order.

3. Resource dependence:

Bernstein's Conditions contd…

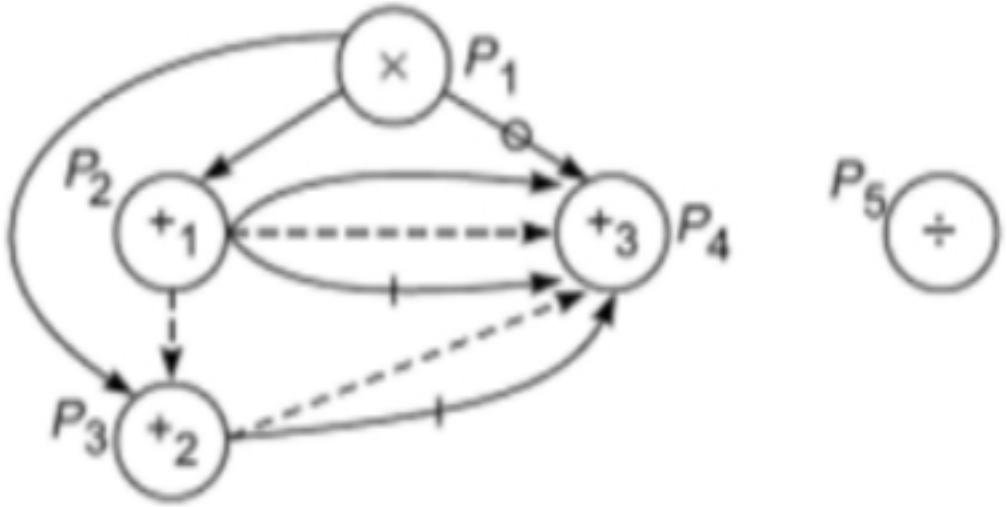
P_1: $C = D \times E$

P_2: $M = G + C$

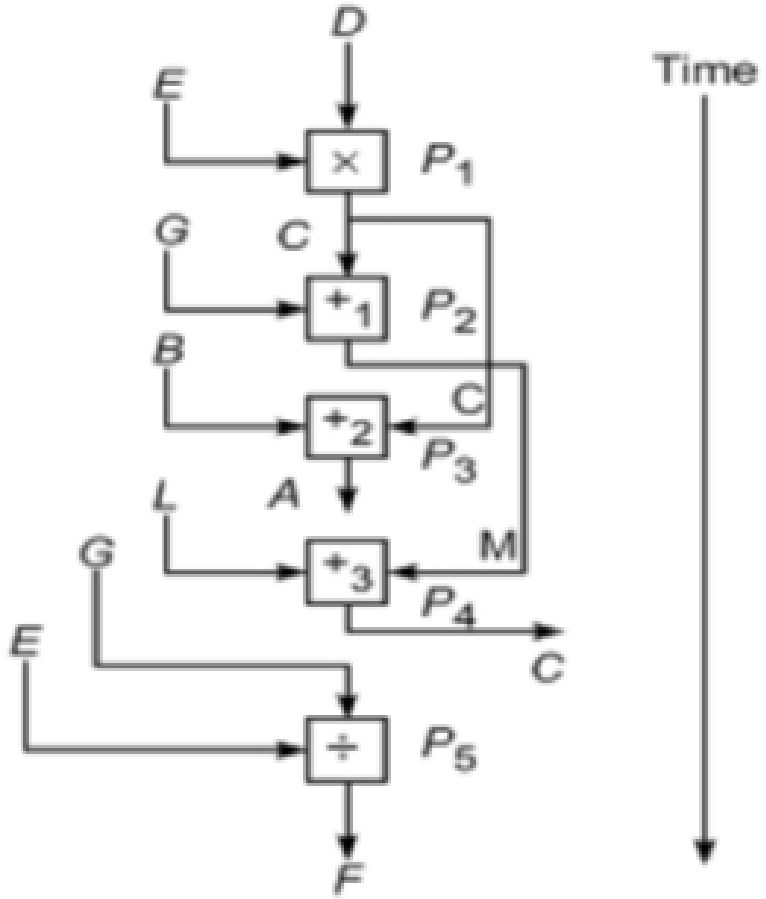
P_3: $A = B + C$

P_4: $C = L + M$

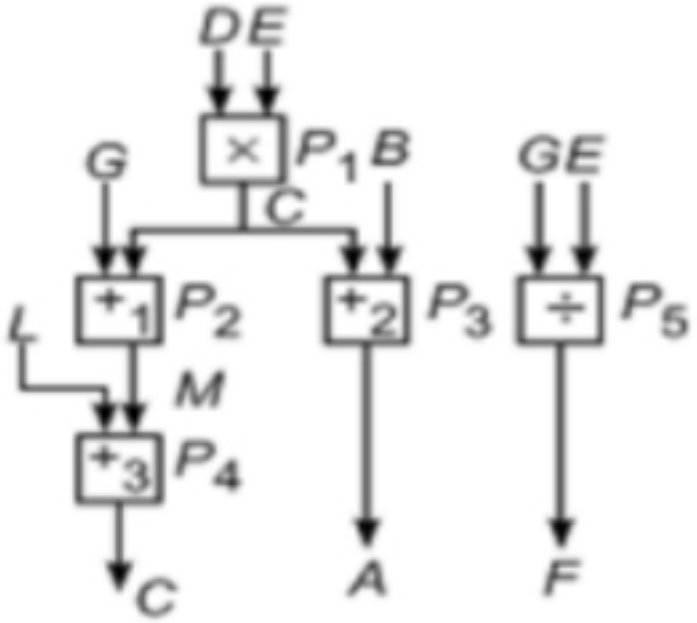
P_5: $F = G / E$



(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

Fig. 2.2 Detection of parallelism in the program of Example 2.2

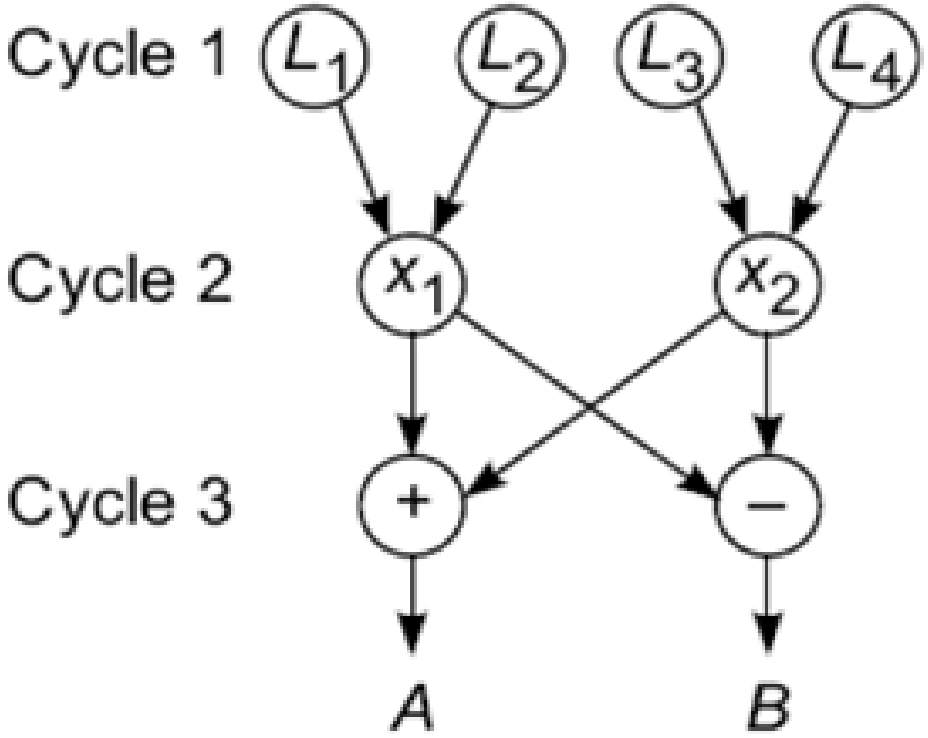
Hardware and software parallelism

Hardware parallelism

- Hardware parallelism is defined by machine architecture and hardware multiplicity i.e., functional parallelism times the processor parallelism.
- It can be characterized by the number of instructions that can be issued per machine cycle.
- If a processor issues k instructions per machine cycle, it is called a k-issue processor.
- Conventional processors are one-issue machines.
- This provide the user the information about peak attainable performance.
- Examples: Intel i960CA is a three-issue processor (arithmetic, memory access, branch).
- IBM RS -6000 is a four-issue processor (arithmetic, floating-point, memory access, branch).
- A machine with n k-issue processors should be able to handle a maximum of nk threads simultaneously.

Software Parallelism

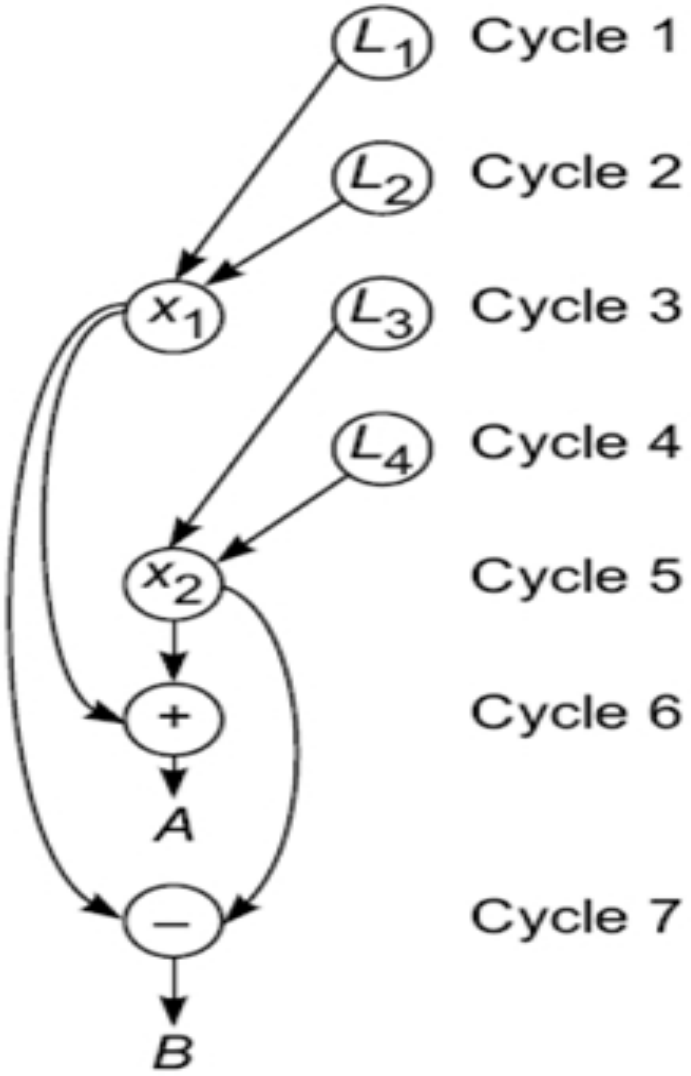
- Software parallelism is defined by the control and data dependence of programs, and is revealed in the program's flow graph i.e., it is defined by dependencies with in the code and is a function of algorithm, programming style, and compiler optimization.
- Example: Mismatch between Software parallelism and Hardware parallelism
  - Consider the example program graph in Fig. 2.3a. There are eight instructions (four loads and four arithmetic operations) to be executed in three consecutive machine cycles.
  - Four load operations are performed in the first cycle, followed by two multiply operations in the second cycle and two add/subtract operations in the third cycle.
  - Therefore, the parallelism varies from 4 to 2 in three cycles. The average software parallelism is equal to 8/3 = 2.67 instructions per cycle in this example program.
  - Now consider execution of the same program by a two-issue processor which can execute one memory access (load or write) and one arithmetic (add, subtract, multiply, etc.) operation simultaneously.
  - With this hardware restriction, the program must execute in seven machine cycles as shown in Fig. 2.3b. Therefore, the hardware parallelism displays an average value of 8/7 = 1.14 instructions executed per cycle.
  - This demonstrates a mismatch between the software parallelism and the hardware 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

- Let us try to match the software parallelism shown in Fig. 2.3a in a hardware platform of a dual-processor system, where single-issue processors are used.
- The achievable hardware parallelism is shown in Fig 2.4. Six processor cycles are needed to execute 12 instructions by two processors.
- S1 and S2 are two inserted store operations, l5 and l6 are two inserted load operations for interprocessor communication through the shared memory.

The Role of Compilers

- Compilers used to exploit hardware features to improve performance. Interaction between compiler and architecture design is a necessity in modern computer development.
- It is not necessarily the case that more software parallelism will improve performance in conventional scalar processors.
- The hardware and compiler should be designed at the same time.

Program Partitioning & Scheduling

Grain size and latency

- The size of the parts or pieces of a program that can be considered for parallel execution can vary.
- The sizes are roughly classified using the term —"granule size" or simply "granularity".
- The simplest measure, for example, is the number of instructions in a program part.
- Grain sizes are usually described as fine, medium or coarse, depending on the level of parallelism involved.

Program Partitioning & Scheduling

Latency

- Latency is the time required for communication between different subsystems in a computer.
- Memory latency, for example, is the time required by a processor to access memory.
- Synchronization latency is the time required for two processes to synchronize their execution.
- Computational granularity and communication latency are closely related.

- Latency and grain size are interrelated and some general observation are
  - As grain size decreases, potential parallelism increases, and overhead also increases.
  - Overhead is the cost of parallelizing a task. The principle overhead is communication latency.
  - As grain size is reduced, there are fewer operations between communication, and hence the impact of latency increases.
  - Surface to volume: inter to intra-node comm.

Levels of Parallelism

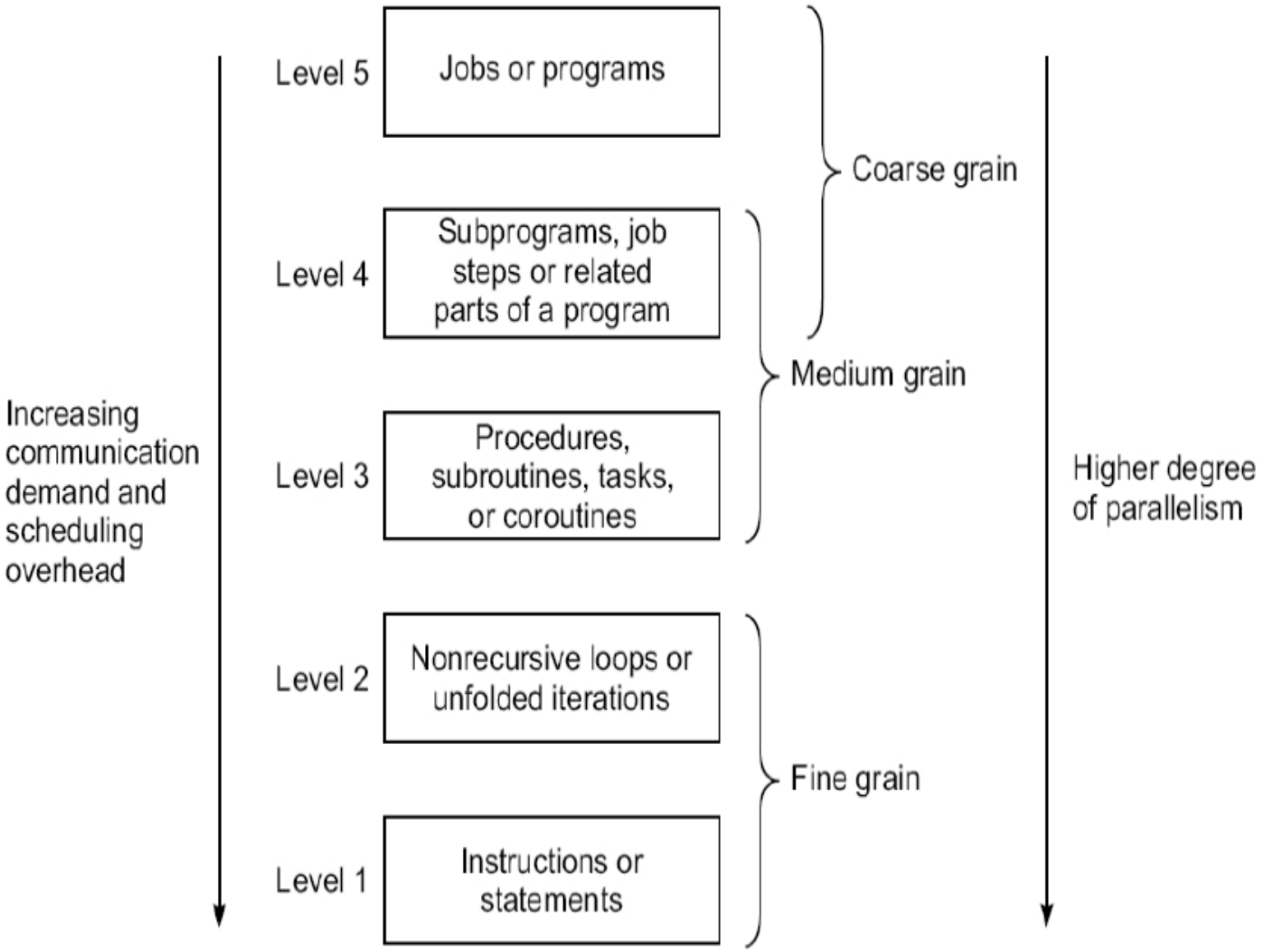


Fig. 2.5 Levels of parallelism in program execution on modern computers (Reprinted from Hwang, *Proc. IEEE*, October 1987)

1. Instruction Level Parallelism (ILP)

- This fine-grained, or smallest granularity level typically involves less than 20 instructions per grain.
- The number of candidates for parallel execution varies from 2 to thousands, with about five instructions or statements (on the average) being the average level of parallelism.
- Advantages of ILP
  - There are usually many candidates for parallel execution. Compilers can usually do a reasonable job of finding this parallelism

2. Loop-level Parallelism

- Typical loop has less than 500 instructions. If a loop operation is independent between iterations, it can be handled by a pipeline, or by a SIMD machine.
- Most optimized program construct to execute on a parallel or vector machine.
- Some loops (e.g. recursive) are difficult to handle. Loop-level parallelism is still considered fine grain computation.

3. Procedure-level Parallelism

- Medium-sized grain; usually less than 2000 instructions.
- Detection of parallelism is more difficult than with smaller grains; interprocedural dependence analysis is difficult and history-sensitive.
- Communication requirement less than instruction level SPMD (single procedure multiple data) is a special case Multitasking belongs to this level.

4. Subprogram-level Parallelism

- Job step level; grain typically has thousands of instructions; medium- or coarse-grain level.
- Job steps can overlap across different jobs. Multiprograming conducted at this level. No compilers available to exploit medium- or coarse-grain parallelism at present.

5. Job or Program-Level Parallelism

- Corresponds to execution of essentially independent jobs or programs on a parallel computer.
- This is practical for a machine with a small number of powerful processors, but impractical for a machine with a large number of simple processors (since each processor would take too long to process a single job).

Communication Latency

- Balancing granularity and latency can yield better performance. Various latencies attributed to machine architecture, technology, and communication patterns used.
- Latency imposes a limiting factor on machine scalability.
- Ex: Memory latency increases as memory capacity increases, limiting the amount of memory that can be used with a given tolerance for communication latency.
- Interprocessor Communication Latency
  - Needs to be minimized by system designer
- Affected by signal delays and communication patterns Ex: n communicating tasks may require

n*(n - 1)/2 communication links, and the complexity grows quadratically, effectively limiting the number of processors in the system.

Communication Patterns

- Determined by algorithms used and architectural support provided
- Patterns include permutations broadcast multicast conference
- Tradeoffs often exist between granularity of parallelism and communication demand.

Grain Packing and Scheduling

Two questions:

- **How can I partition a program into parallel "pieces" to yield the shortest execution time?**
- **What is the optimal size of parallel grains?**

- **There is an obvious tradeoff between the time spent scheduling and synchronizing parallel grains and the speedup obtained by parallel execution.**
- **One approach to the problem is called ―grain packing.‖**

Grain Packing and Scheduling cond…

- Program Graphs and Packing (Basic concept of Program Partitioning)
  - A program graph shows the structure of the program, similar to dependence graph. Each node in the program graph corresponds to a computational unit in the program.
  - Grain size is measured by the number of basic machine cycles needed to execute all the operations within the node.
  - Each node is denoted by, Nodes = { (n,s) }, where n = node name (id), s = grain size (larger s = larger grain size), Fine-grain nodes have a smaller grain size, and coarse-grain nodes have a larger grain size.
  - Edges = { (v,d) }, where v = variable being "communicated" and d = communication delay.
  - Packing two (or more) nodes produces a node with a larger grain size and possibly more edges to other nodes.

Packing is done to eliminate unnecessary communication delays or reduce overall scheduling overhead.

Grain Packing and Scheduling cond…

- **Example: Basic concept of Program Partitioning**

Var $a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q$

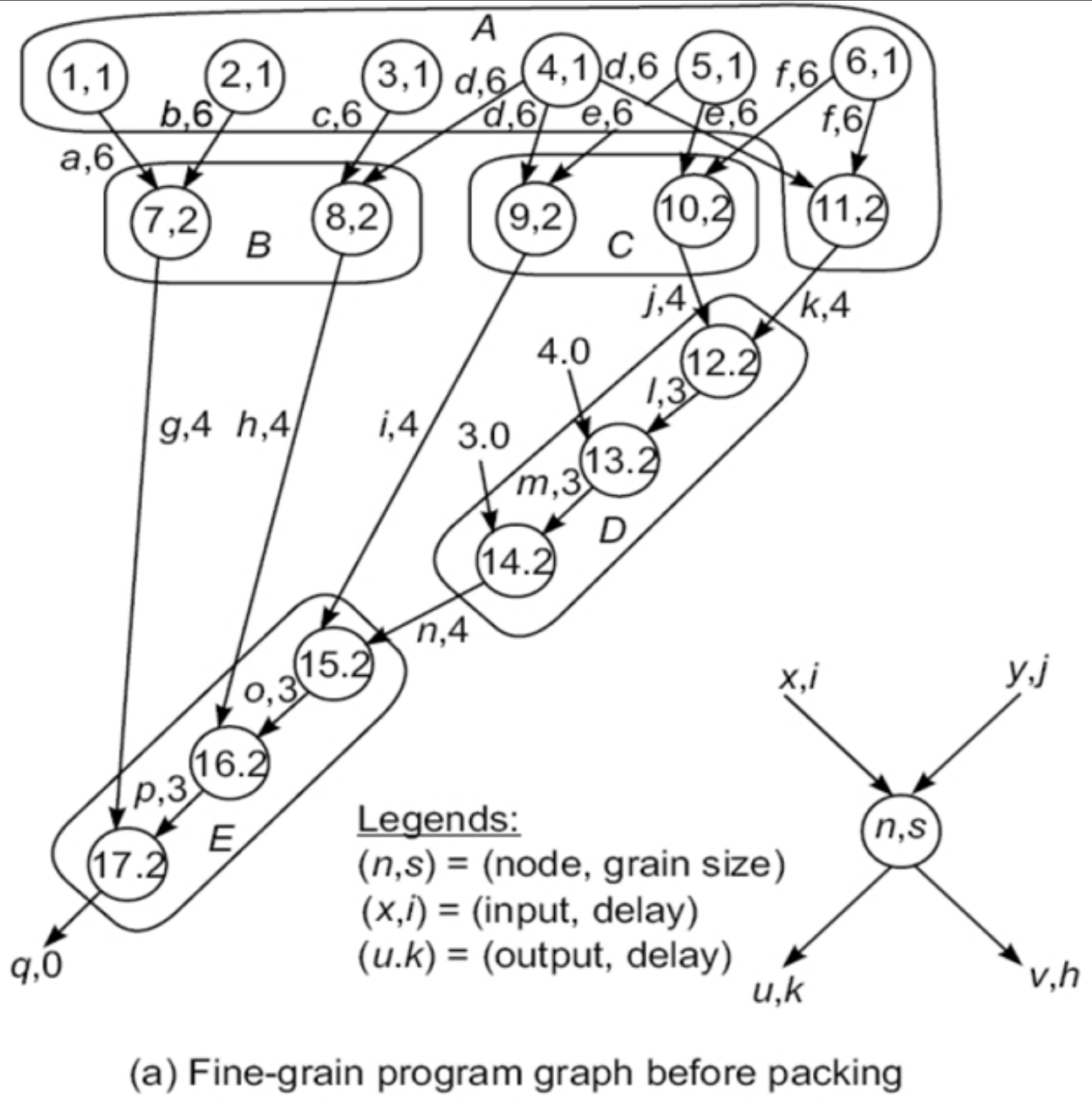
Begin

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

End

arallelism

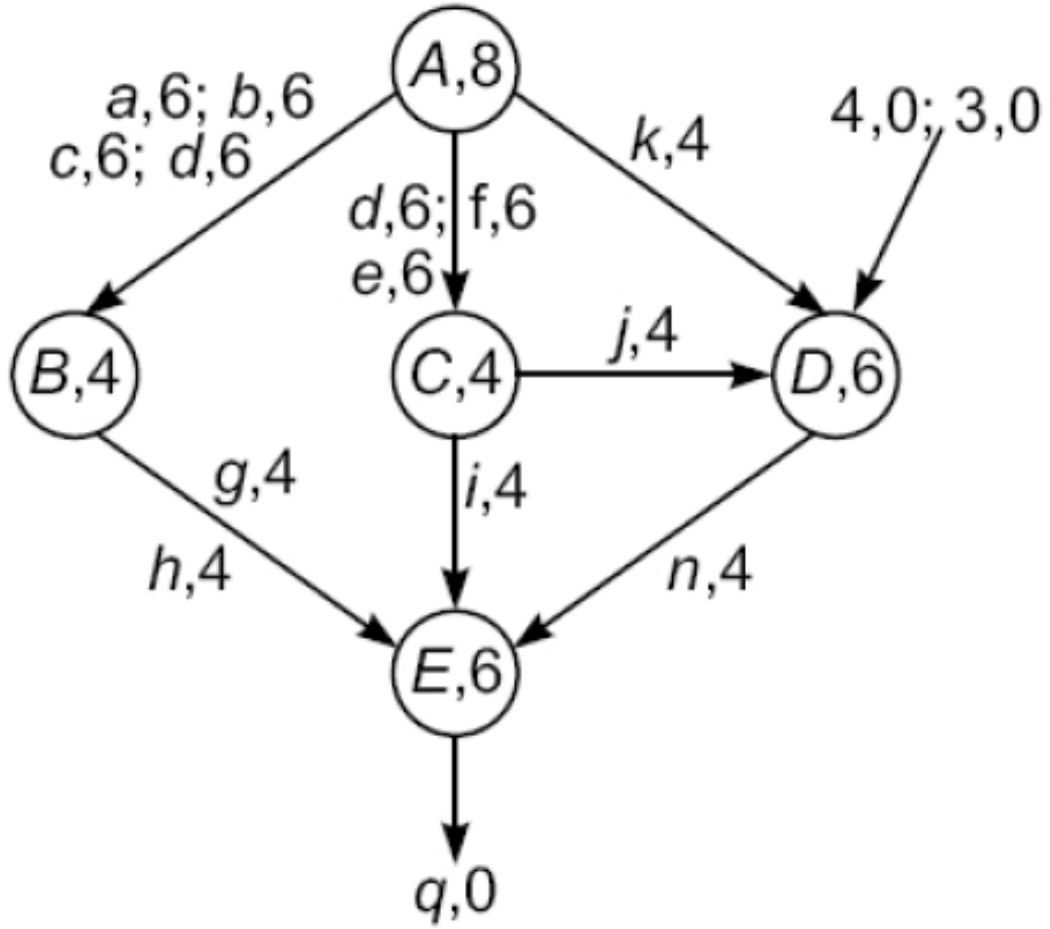
- Nodes 1, 2, 3, 4, 5, and 6 are memory reference (data fetch) operations.
- Each node takes one clock cycle to address and six cycles to fetch from memory.
- All other operations (7 to 17) are CPU operations, each requiring two cycles to complete
- Delay includes all the path delays and memory latency involved



(a) Fine-grain program graph before packing

```
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 × b
    8.  h := c × d
    9.  i := d × e
   10.  j := e × f
   11.  k := d × f
   12.  l := j × k
   13.  m := 4 × 1
   14.  n := 3 × m
   15.  o := n × i
   16.  p := o × h
   17.  q := p × q
End
```

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

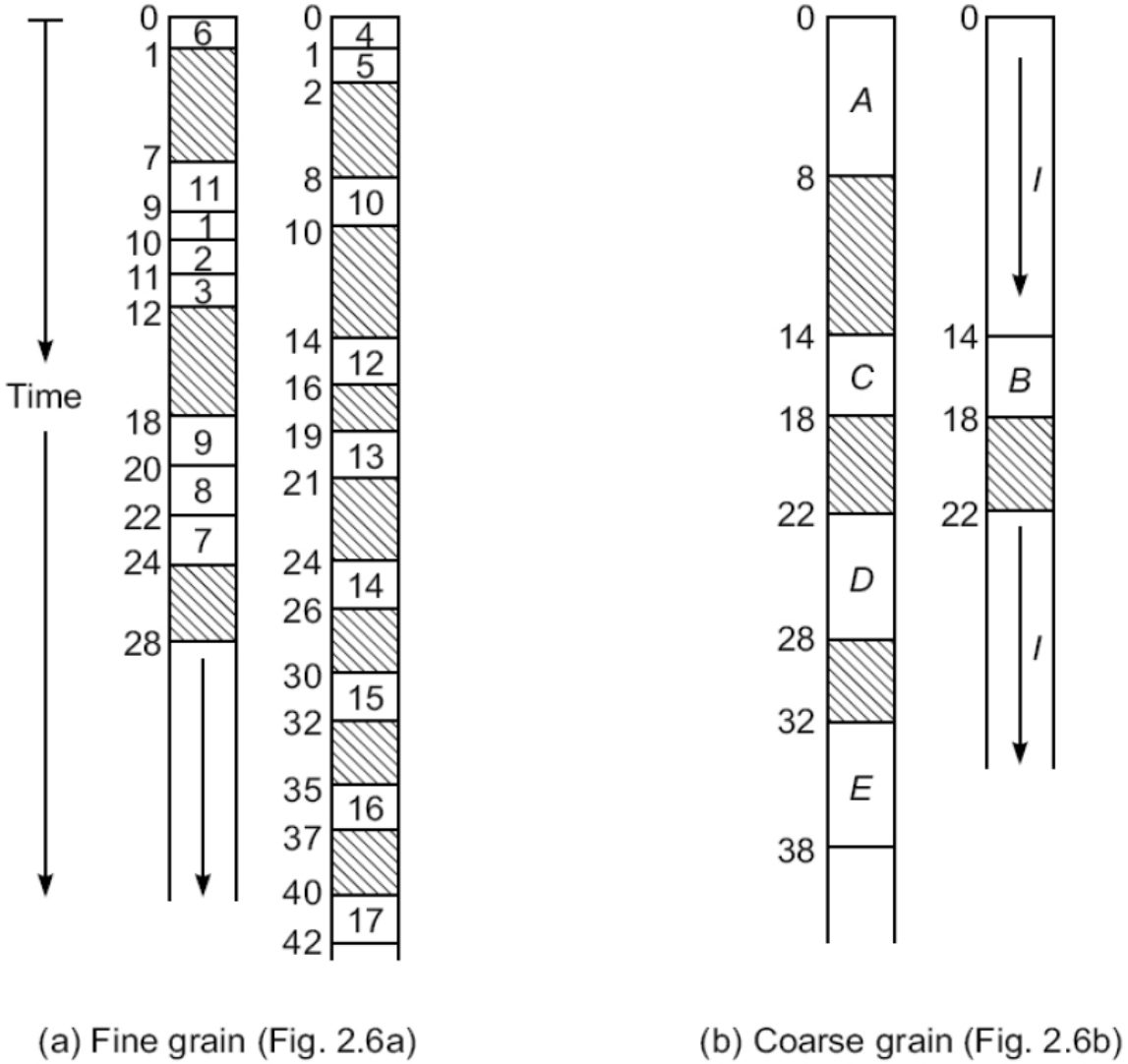


(b) Coarse-grain program graph after packing

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

- **Grains (Nodes) are combined (packed) to generate coarse grain nodes.**
- **Node (A, 8) is obtained by combining the nodes (1,1), (2,1), (3,1), (4,1), (5,1), (6,1), and (11,2). The obtained size of the grain A is 8 which is the summation of all grain sizes (1+1+1+1+1+1+2=8).**
- **Apply the fine grains first in order to achieve the higher degree of parallelism.**
- **Then combine (pack) multiple fine grains into coarse grain if it can eliminate unnecessary communication delays.**

Grain Packing and Scheduling cond…



(a) Fine grain (Fig. 2.6a)

(b) Coarse grain (Fig. 2.6b)

- All fine grain operations within the single coarse grain node are assigned to same processor for execution to reduce the communication delay.
- Grain Packing offers the trade off between the parallelism and scheduling/communication overhead.

- With respect to the fine-grain versus coarse-grain program graphs in Fig. 2.6, two multiprocessor schedules are shown in Fig. 2.7. The fine-grain schedule is longer (42 time units) because more communication delays were included as shown by the shaded area.
- The coarse-grain schedule is shorter (38 time units) because communication delays among nodes 12, 13 and 14 within the same node D (and also the delays among 15, 16 and 17 within the node E) are eliminated after grain packing.
-

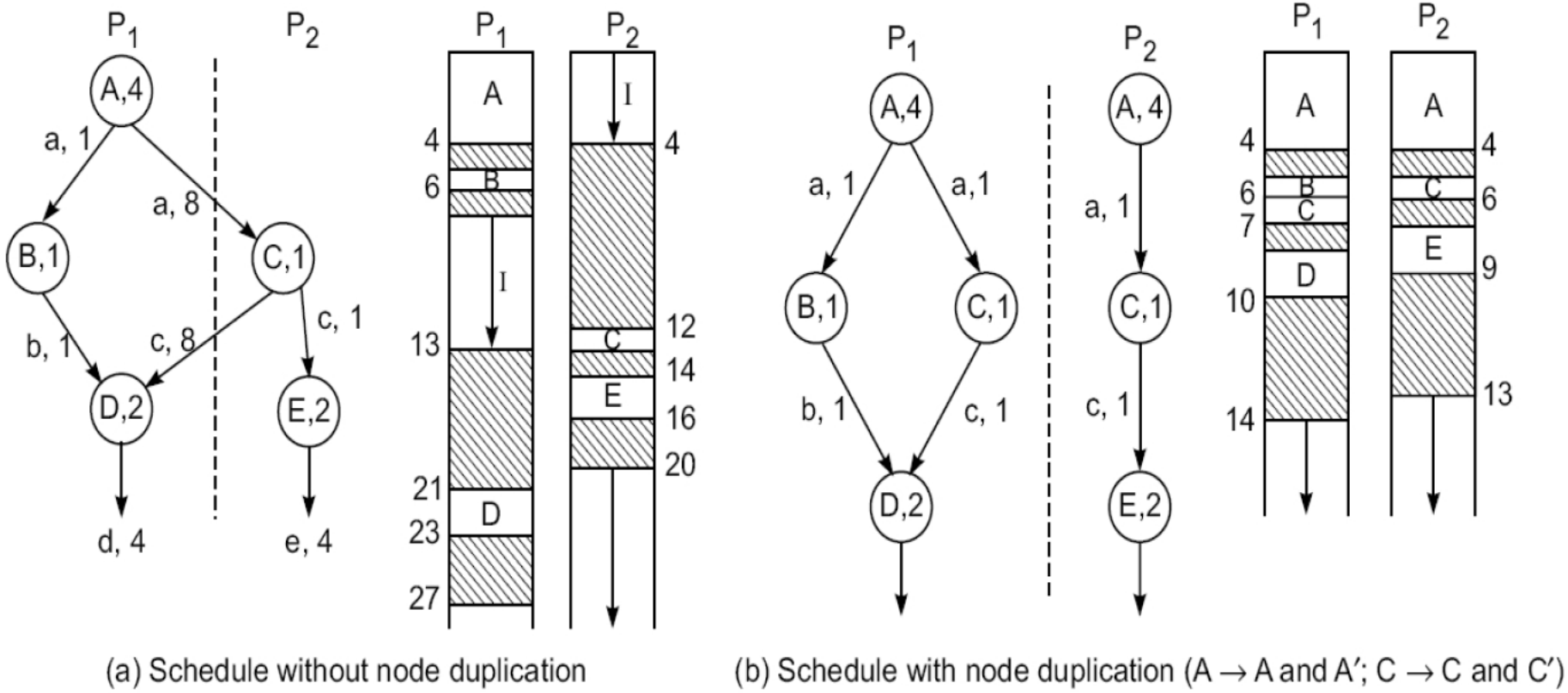
Static Multiprocessor Scheduling

Node duplication

- Grain packing may potentially eliminate interprocessor communication, but it may not always produce a shorter schedule.
- By duplicating nodes (that is, executing some instructions on multiple processors), we may eliminate some interprocessor communication, and thus produce a shorter schedule.

Static Multiprocessor Scheduling

Node duplication



(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)

- Grain packing and node duplication are often used jointly to determine the best grain size and corresponding schedule.
- Four major steps are involved in the grain determination and the 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 Flow Mechanisms

Program Flow Mechanisms

1. Control flow Mechanisms
2. Data flow Mechanisms
3. Demand-driven Mechanisms

1. Control flow Mechanisms

- Conventional computer are based on a control flow mechanism by which the order of program execution is explicitly stated in the user programs.
- Von Neumann computers use a Program Counter (PC) to sequence the execution of instructions in a program. The PC is sequenced by instruction flow in a program. This sequential execution style has been called control-driven as a program flow is explicitly controlled by programmers.
- Uniprocessor computer is inherently sequential, due to the use of control driven mechanism.

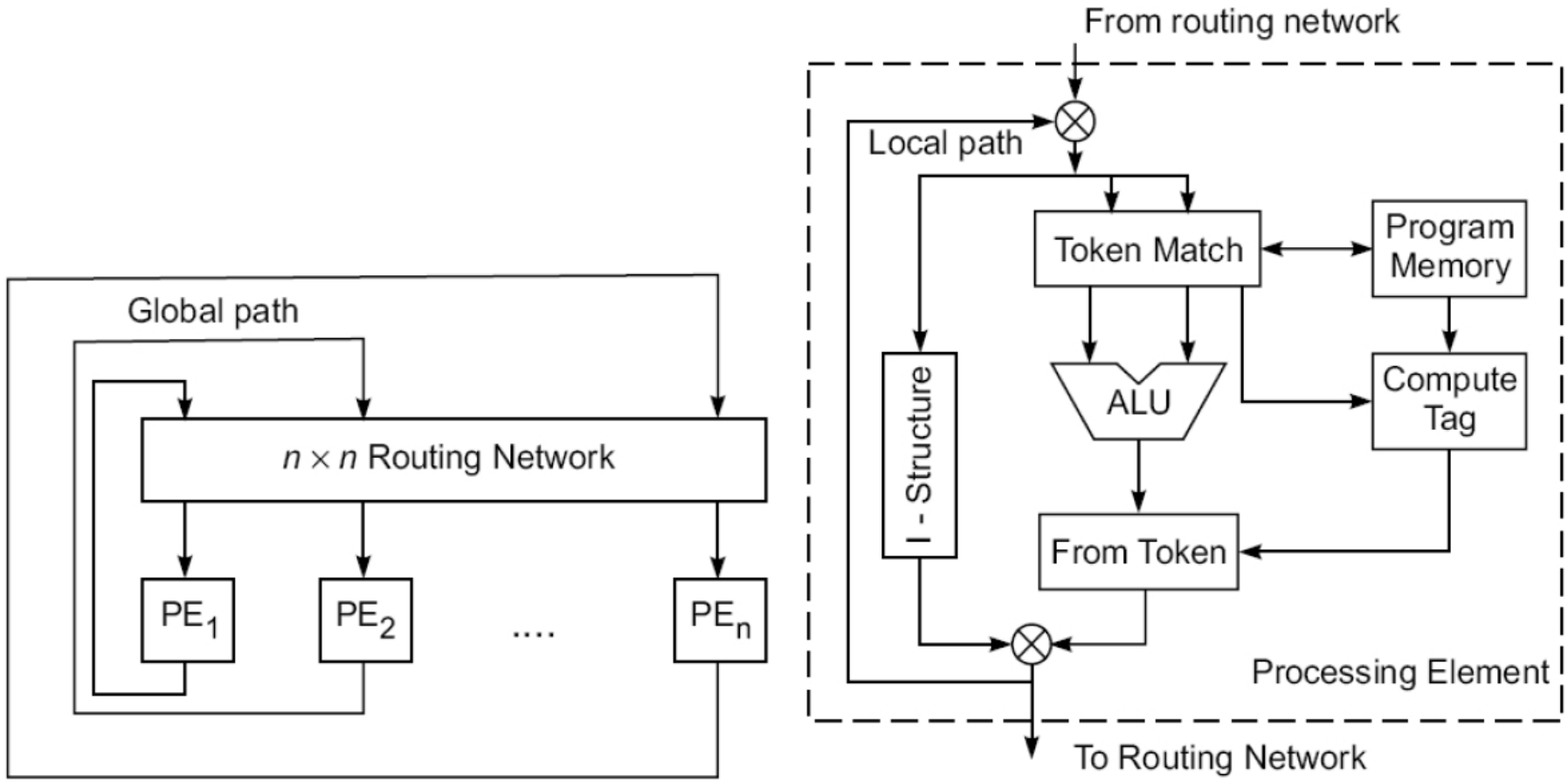
Program Flow Mechanisms

2. Data flow Mechanisms

- Data flow computers are based on a data-driven mechanism which allows the execution of any instruction to be driven by data (operands) availability
- These computers emphasize a high degree of parallelism at the fine grain instructional level.
- Any instruction should be ready for the execution whenever operands become available. Ie instructions are not ordered in any way, and are being stored separately in a main memory, data are directly held inside instructions.
- Computational results (data tokens) are passed directly between instructions. The data generated by an instruction will be duplicated into many copies and forwarded directly to all needy instructions. Data tokens once consumed by an instruction, will no longer be available for reuse by other instructions.
- Data flow Mechanisms requires no program counter and no control sequence.
- But requires special mechanism to detect the data availability, to match data tokens with needy instructions and to enable the chain reaction of asynchronous instruction execution.
- In this type computer no memory sharing between instructions results in no side effects.

Program Flow Mechanisms

Data flow architecture (Tagged-token dataflow computer)



(a) The global architecture

(b) Interior design of a processing element

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

Program Flow Mechanisms

Data flow architecture (Tagged-token dataflow computer) contd..

- The Arvind machine (MIT) has N PEs and an n-by-n interconnection network.
- Each PE has a token-matching mechanism that dispatches only instructions with data tokens available.
- Each datum is tagged with
  - address of instruction to which it belongs
  - context in which the instruction is being executed
- Tagged tokens enter PE through local path (pipelined), and can also be communicated to other PEs through the routing network.
- Instead of program counter, Instruction address(es) effectively used.
- Context identifier effectively used instead of frame base register. (Program counter and frame base registers are used in a control flow machine)
- Since the dataflow machine matches the data tags from one instruction with successors, synchronized instruction execution is implicit.
- An I-structure in each PE is provided to eliminate excessive copying of data structures.
- Each word of the I-structure has a two-bit tag indicating whether the value is empty, full or has pending read requests.
- This is a retreat from the pure dataflow approach.
- Special compiler technology needed for dataflow machines.

3. Demand driven mechanism

- Reduction computers are based on demand–driven mechanism which initiates the an operation based on the demand for its results by other computation.
- Demand-driven machines take a top-down approach, attempting to execute the instruction (a demander) that yields the final result.
- This triggers the execution of instructions that yield its operands, and so forth.
- The demand-driven approach matches naturally with functional programming languages (e.g. LISP and SCHEME).
- Example: Consider the evaluation of a nested arithmetic expression

  a=((b+1)*c-(d/e))

- In demand driven computation chooses top-down approach by first demanding the value of a
- Demanding value of a triggers the demand for evaluating the next level expression (b+1)*c and d/e
- To evaluate (b+1)*c triggers the evaluation of (b+1) innermost expression.
- Result are then returned to nested demander in the reverse order before a is evaluated.

3. Demand driven mechanism

- Reduction computers are based on demand–driven mechanism which initiates the an operation based on the demand for its results by other computation.
- Demand-driven machines take a top-down approach, attempting to execute the instruction (a demander) that yields the final result.
- This triggers the execution of instructions that yield its operands, and so forth.
- The demand-driven approach matches naturally with functional programming languages (e.g. LISP and SCHEME).
- Example: Consider the evaluation of a nested arithmetic expression

a=((b+1*c-(d/e))

- In demand driven computation chooses top-down approach by first demanding the value of a
- Demanding value of a triggers the demand for evaluating the next level expression (b+1)*c and d/e
- To evaluate (b+1)*c triggers the evaluation of (b+1) innermost expression.
- Result are then returned to nested demander in the reverse order before a is evaluated.

Program Flow Mechanisms

3. Demand driven mechanism cond…

- Demand driven computation corresponds to lazy evaluation, because operations are executed only when their results are required by another instruction. Demand driven computers are reduction machine model. Two types of Demand Driven mechanism

A. String-reduction model:

- Each demander gets a separate copy of the expression string to evaluate.
- Long string expression is reduced to a single value in a recursive fashion
- Each reduction step has an operator and embedded reference to demand the corresponding input operands.
- The operator is suspended while its input arguments are being evaluated.
- An expression is said to be fully reduced when all the arguments have been replaced by literals values.

B. Graph-reduction model:

- An expression is represented by as a direct graph.
- Expression graph is reduced by evaluation of branches or subgraphs, possibly in parallel, with demanders given pointers to results of reductions.
- Based on sharing of pointers to arguments; traversal and reversal of pointers continues until constant arguments are encountered.

Program Flow Mechanisms

Comparision of Flow Mechanisms

Machine Model	Control Flow (control-driven)	Dataflow (data-driven)	Reduction (demand-driven)
Basic Definition	Conventional computation; token of control indicates when a statement should be executed	Eager evaluation; statements are executed when all of their operands are available	Lazy evaluation; statements are executed only when their result is required for another computation
Advantages	Full control The most successful model for commercial products	Very high potential for parallelism	Only required instructions are executed
	Complex data and control structures are easily implemented	High throughput	High degree of parallelism
		Free from side effects	Easy manipulation of data structures
Disadvantages	In theory, less efficient than the other two	Time lost waiting for unneeded arguments	Does not support sharing of objects with changing local state
	Difficult in preventing run-time errors	High control overhead	Time needed to propagate demand tokens
		Difficult in manipulating data structures	

System interconnect architecture

- Various types of interconnection networks have been suggested for SIMD computers. These are basically have been classified on network topologies into two categories namely
  1. Static Networks
  2. Dynamic Networks
- Static and dynamic networks are used for interconnecting computer subsystems or for constructing multiprocessor and multicomputer.
- Direct networks for static connections
- Indirect networks for dynamic connections
- Networks are used for
  - Internal connections in a centralized system among
    - processors
    - memory modules
    - I/O disk arrays
  - Distributed networking of multicomputer nodes

System interconnect architecture

- The goals of an interconnection network are to provide
  - low-latency
  - high data transfer rate
  - wide communication bandwidth
- Analysis includes
  - latency
  - bisection bandwidth
  - data-routing functions
  - scalability of parallel architecture

- The topology of an interconnection network can be either static or dynamic.
  - Static networks are formed of point-to-point direct connections which will not change during program execution.
  - Static networks are used for fixed connections among the sub systems of centralized system or multiple computing nodes of a distributed systems
- Dynamic networks are implemented with switched channels,
  - Which are dynamically configured to match the communication demand in user programs.
  - Dynamic network includes the buses, crossbar switches, multistage network and routers which are often used in shred memory multiprocessor.
- Packet switching and routing is playing an important role in modern multiprocessor architecture.

Network properties and Routing

1. Node Degree and Network Diameter:

- **node degree d : The number of edges (links or channels) incident on a node is called the.**
- **In the case of unidirectional channels, the number of channels into a node is the in degree, and that out of a node is the out degree.**
- **Then the node degree is the sum of the two. The node degree reflects the number of IO ports required per node, and thus the cost of a node.**
- **Therefore, the node degree should be kept a (small) constant, in order to reduce cost.**

- **The Diameter D of a network is the maximum shortest path between any two nodes.**
  - **The network diameter indicates the maximum number of distinct hops between any two nodes, thus providing a figure of communication merit for the network.**
  - **The path length is measured by the number of links traversed.**
  - **Therefore, the network diameter should be as small as possible from a communication point of view.**

Network properties and Routing

2. Bisection Width:

- **bisection width b:** When a given network is cut into two equal halves, the minimum number of edges (channels) along the cut is called the bisection width b. In the case of a communication network, each edge may correspond to a channel with **w bit wires**.
- Wire bisection width is B=bw. B indicates wiring density of a network.
- To summarize the above discussions, the performance of an interconnection network is affected by the following factors:
  - **Functionality:** refers to how the network supports data routing, interrupt handling, synchronization, request-"message combining, and coherence.
  - **Network Latency:-** This refers to the worst-ease time delay for a unit message to be transferred through the network.
  - **Bandwidth:** This refers to the maximum data transfer rate, in terms of Mbps or Gbps transmitted through the network.
  - **Hardware Complexity**: This refers to implementation costs such as those for wires, switches, connectors, arbitration, and interface logic.
  - **Scalability:** This refers to the ability of a network to be modularly expandable with a scalable performance with increasing machine resources.

Network properties and Routing

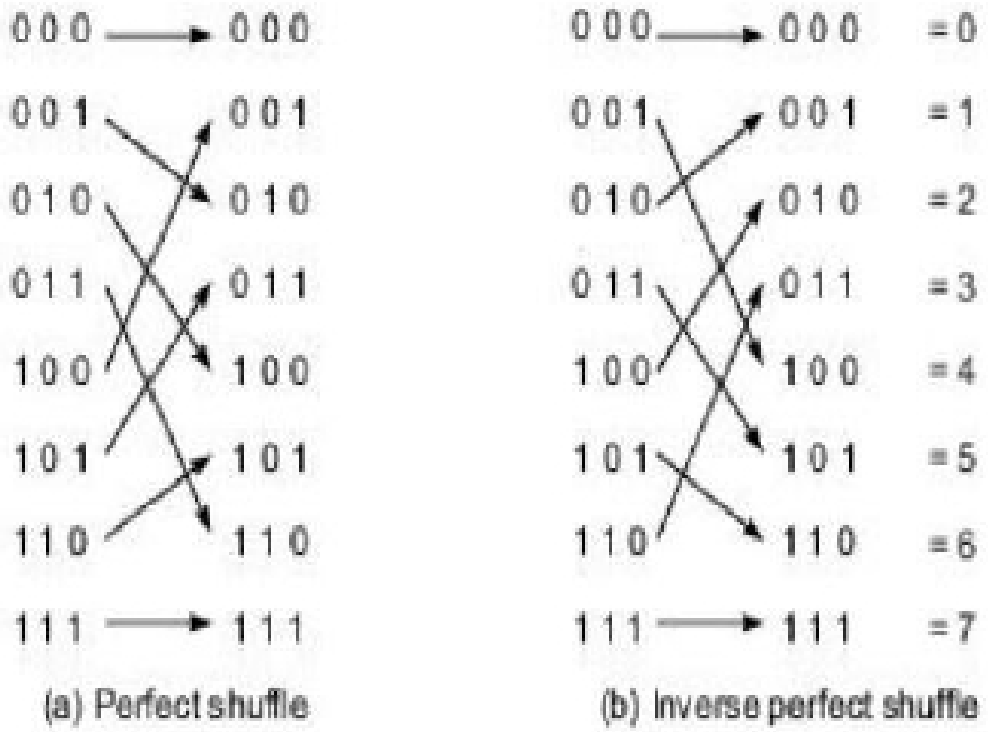
3. Data Routing Functions

- Data Routing network is used for inter-PE data exchange. This routing network can be static or dynamic.
- In multicomputer network data routing is achieved with through message passing. Hardware routers are used to route the message among the multiple computer nodes.
- Common data routing functions are permutation, shifting, rotation, broadcast(one to all), multicast (one to many), exchange.

Network properties and Routing

4. Perfect shuffle.

Perfect Shuffle and Exchange Perfect shuffle is a special permutation function suggested by Harold Stone (1971) for parallel processing applications. The mapping corresponding to a perfect shuffle is shown in Fig. 2.14a. Its inverse is shown on the right-hand side (Fig. 2.14b).



(a) Perfect shuffle (b) Inverse perfect shuffle

Network properties and Routing

4. Perfect shuffle.

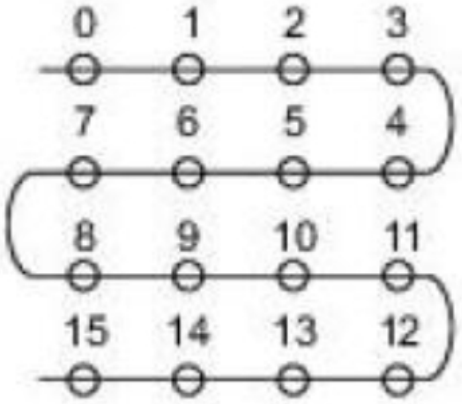
- In general to shuffle $n=2^k$ objects, the objects can be expressed as k bit binary number.
- Perfect shuffle maps x to y where $y=(x_{k-2},.....x_1, x_0, x_{k-1})$ and x is $(x_{k-1},.....x_1, x_0)$
- This obtained from x by shifting 1 bit to the left and wrapping around the most significant to the least significant position.
- Inverse of this operation results in inverse perfect shuffle.

System interconnect architecture

Static connection network

Static network use direct links which are fixed once built.

1. Liner array



(a) Linear array

- This one dimensional network in which N nodes are connected by N-1 links in a line.
- Internal nodes have degree 2, and the terminal nodes have degree 1.
- The diameter is N-1 and bisectional width is b=1
- This type network posses communication inefficiency when N becomes very large.
- Simplest connection topology.
- Poses communication inefficiency when N becomes very large.

2. Ring and Chordal Ring

Ring is obtained by connecting the two terminal nodes of a linear array with one extra link.

A ring can be unidirectional or bidirectional.

It is symmetric with a constant node degree of 2.

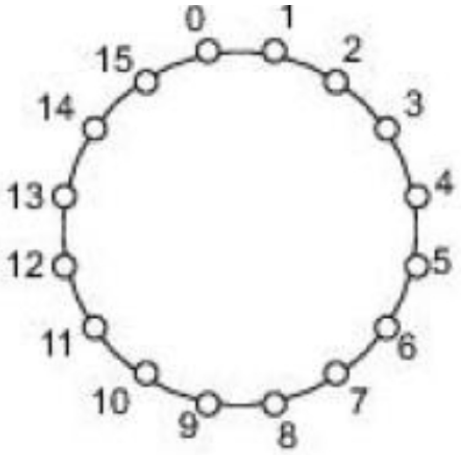
The diameter is N/2 in bidirectional ring and N for unidirectional ring.

Static connection network

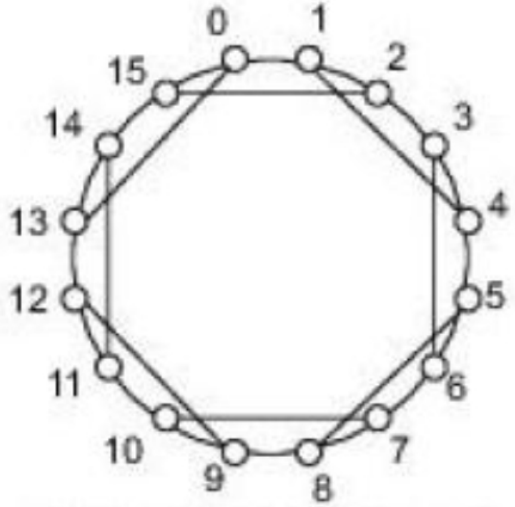
2. Ring and Chordal Ring

Increasing the node degree from 2 to 3 or 4 results in two chordal rings.

More links added higher the node degree and shorter the network diameter.

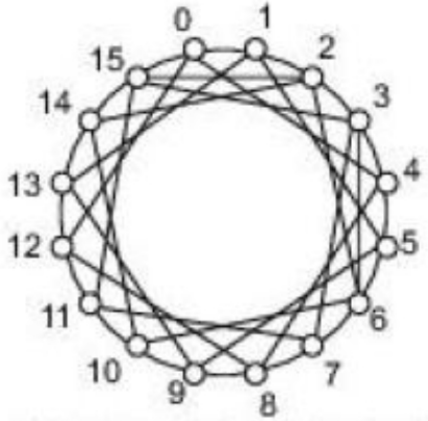


(b) Ring



(c) Chordal ring of degree 3

(b) Ring

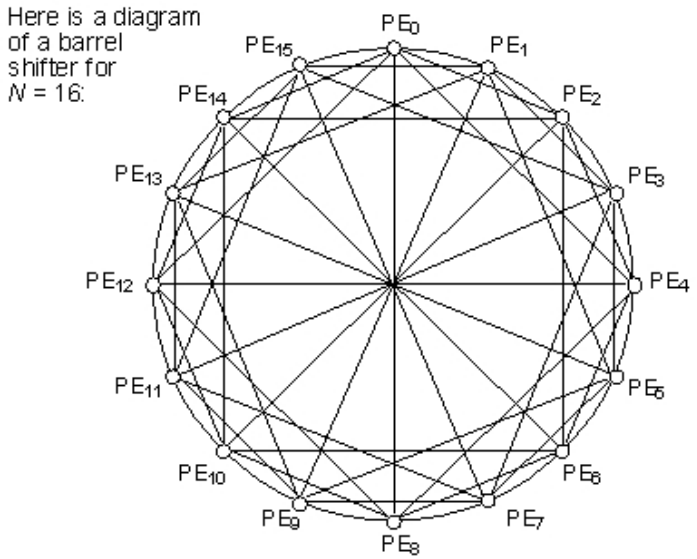


(d) Chordal ring of degree 4
(same as Illiac mesh)

Static connection network

3. Barrel shifter

- Barrel shifter ring is obtained with N=16 nodes. Barrel shifter is obtained from the ring by adding extra links from each node to those nodes having a distance equal to an interger power of 2.
- Node is i is connected to node j if $|j-1|=2^r$ for some r = 0, 1, 2, n-1



System interconnect architecture

Static connection network

4. Tree and Star.

Binary tree of 31 nodes in five levels is shown in fig.

In general a k level completely balanced binary tree should have $N = 2^{k-1}$ nodes.

The maximum node degree is 3 and the diameter is 2(k-1)

5. Fat tree

The channel width of the tree increases as ascend from level to the root.

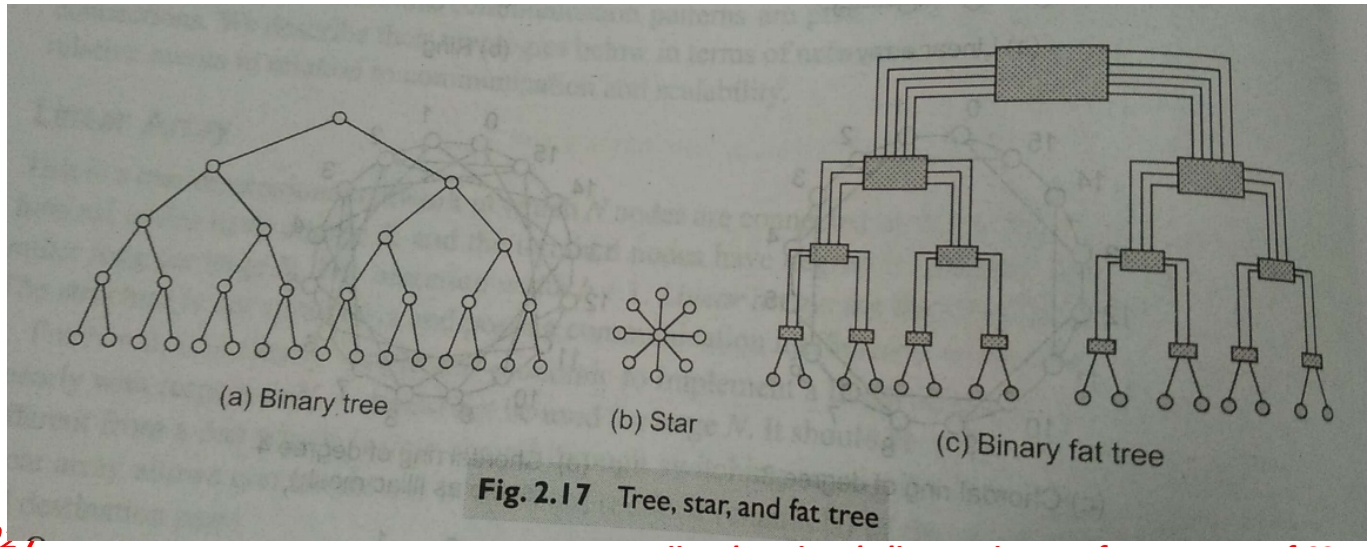


Fig. 2.17 Tree, star, and fat tree

System interconnect architecture

Static connection network

4. Mesh and torus

A 3* 3 mesh is as shown in fig.

The mesh is a frequently used architecture which has been implemented in the Intel variations.

In general k- dimention mesh is with $N = n^k$ node has an interior node degree of 2k and the network diameter of k(n-1).

Ittiac IV assumed an 8*8 mesh with a constant degree of 4 and diameter of 7.

Torus is another variant of mesh with shorter diameter.

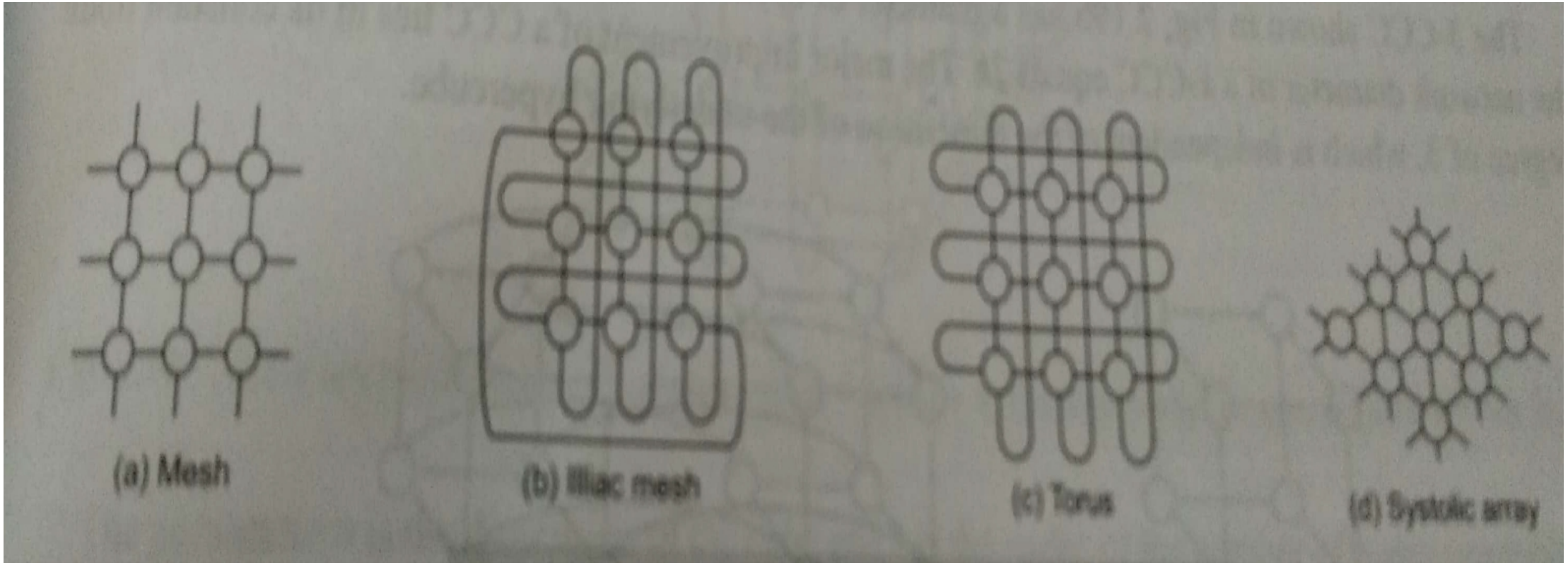
5. Systolic Arrays

This is class of multidimensional pipelined architecture designed for implementing fixed algorithms.

Ex systolic array designed for matrix multiplication

Static connection network

4. Mesh and torus

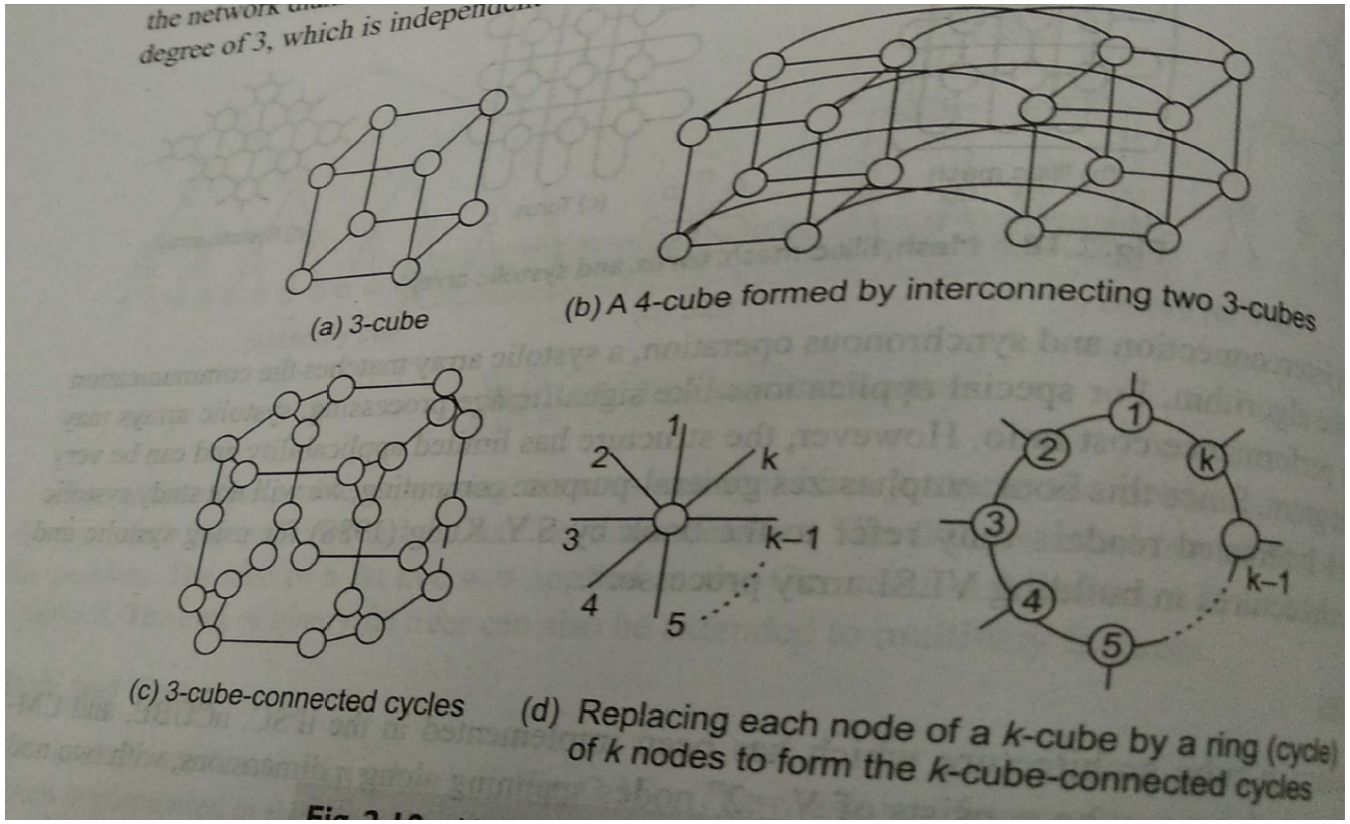


Static connection network

6. Hypercube

This is a binary n-cube architecture which has been implemented in CM-2 systems.

In general n-cube consists of $N=2^n$ nodes spanning along n dimension with two nodes per dimention. A 3 cube with 8 nodes is shown in fig



Dynamic Connection Networks

- Dynamic connection networks can implement all communication patterns based on program demands.
- In increasing order of cost and performance, these include
  - o bus systems
  - o multistage interconnection networks
  - o crossbar switch networks
- Price can be attributed to the cost of wires, switches, arbiters, and connectors.
- Performance is indicated by network bandwidth, data transfer rate, network latency, and communication patterns supported.

1. Digital Buses

- A bus system (contention bus, time-sharing bus) has
  - o a collection of wires and connectors
  - o multiple modules (processors, memories, peripherals, etc.) which connect to the wires
  - o data transactions between pairs of modules

1. Digital Buses

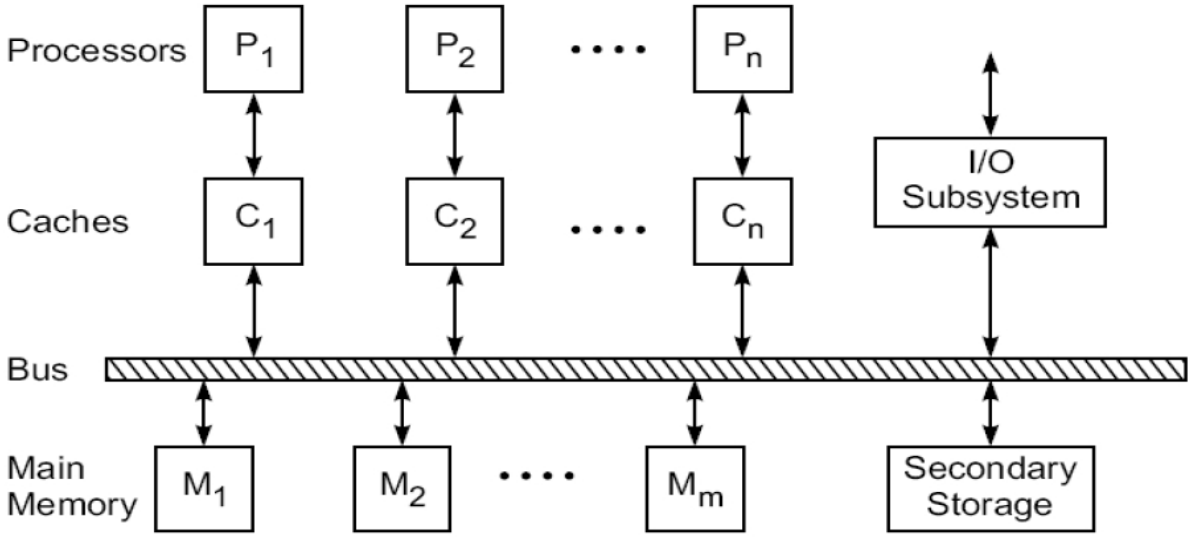


Fig. 2.22 A bus-connected multiprocessor system, such as the Sequent Symmetry S1

Bus supports only one transaction at a time.

- **Bus arbitration logic must deal with conflicting requests.**
- **Lowest cost and bandwidth of all dynamic schemes.**
- **Many bus standards are available.**

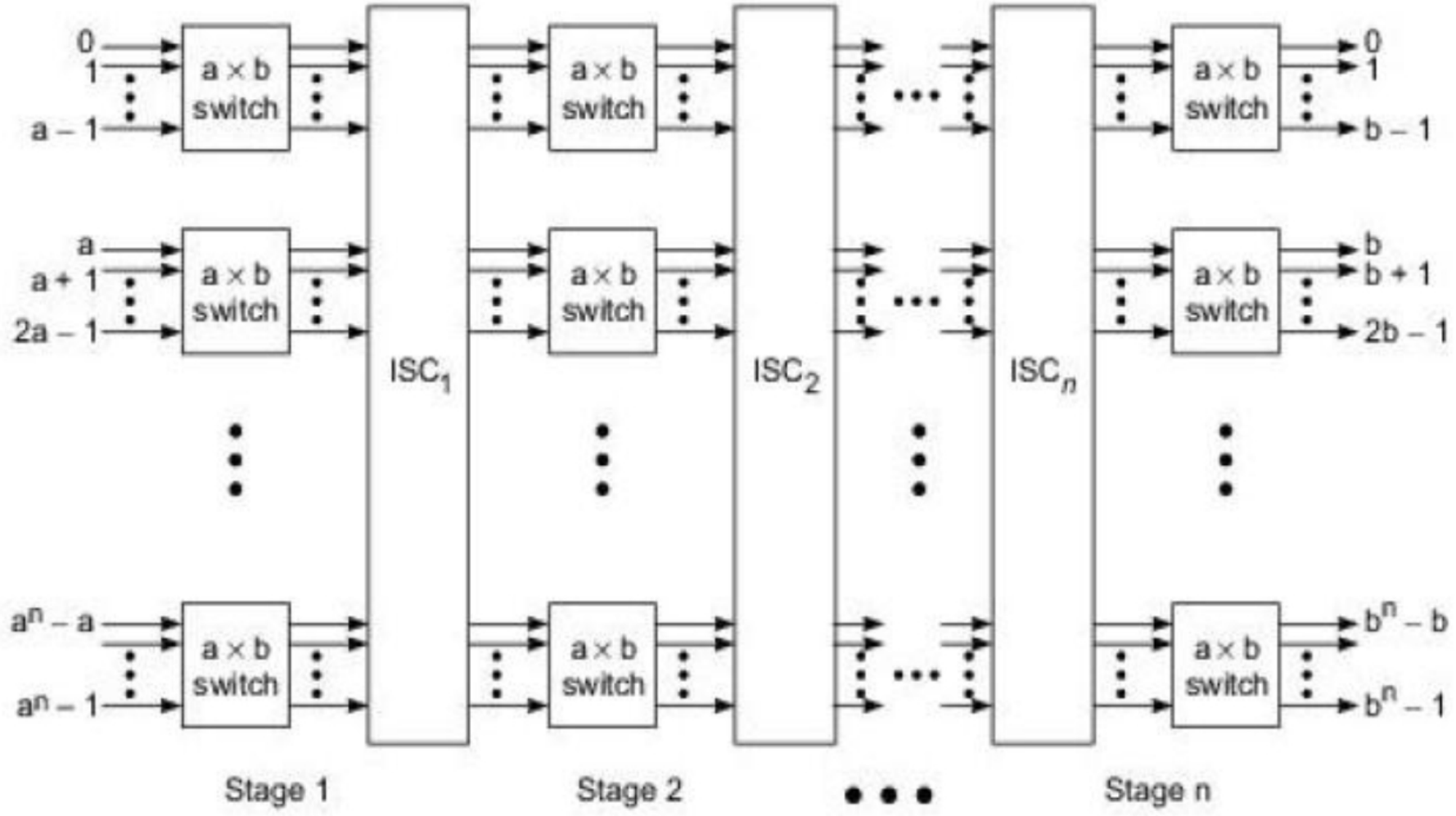
2. Switches

Switch Modules An $a \times b$ *switch module* has a inputs and b outputs. A *binary switch* corresponds to a 2×2 switch module in which $a = b = 2$. In theory, a and b do not have to be equal. However, in practice, a and b are often chosen as integer powers of 2; that is, $a = b = 2^k$ for some $k \geq 1$.

Multistage Interconnection Networks MINs have been used in both MIMD and SIMD computers. A generalized multistage network is illustrated in Fig. 2.23. A number of $a \times b$ switches are used in each stage. Fixed interstage connections are used between the switches in adjacent stages. The switches can be dynamically set to establish the desired connections between the inputs and outputs.

Different classes of MINs differ in the switch modules used and in the kind of *interstage connection* (ISC) patterns used. The simplest switch module would be the 2×2 switches ($a = b = 2$ in Fig. 2.23). The ISC

3. Multistage Networks



Principles of scalable performance-Scalability of parallel algorithms

Algorithm characteristics

1. **Deterministic vs nondeterministic**
   - Only deterministic algorithms are implementable on real machines
2. **Computational granularity**
   - Granularity decides the size of the data items and programs modules used in computation.
   - Algorithms can be classified as fine grain, medium grain and coarse grain modules.
3. **Parallel profile**
   - The distribution of the degree of parallelism in an algorithm reveals the opportunity for parallel processing. Often affects the effectiveness of parallel algorithms
4. **Communication patterns and synchronization requirements**
   - Communicational patterns address both the memory access and interprocessor communication.
   - The pattern can be static or dynamic depending on the algorithms.
   - Static algorithms are more suitable for SIMD or pipelined machines, while dynamic algorithms are for MIMD machines.
5. **Uniformity of the operations**
   - It refers to the type of fundamental operations to be performed . If the operations are uniform across the data set more the SIMD processing or pipelining may be more desirable.
   - Randomly structured algorithms are more suitable for MIMD processing. The other related issue includes data types and precision desired.

Principles of scalable performance-Scalability of parallel algorithms

6. Memory requirement and data structures

- In solving large scale programs Data set may requires large memory.
- Memory efficiency affected by data structures chosen and data movement pattern in the algorithm.
- Both time and space complexities are key measures of the granularity of the parallel algorithm.

Principles of scalable performance-Scalability metrics

1. Machine size(n)
   - The number of processor employed in a parallel computers. A large machine size implies more resource and more computing power.

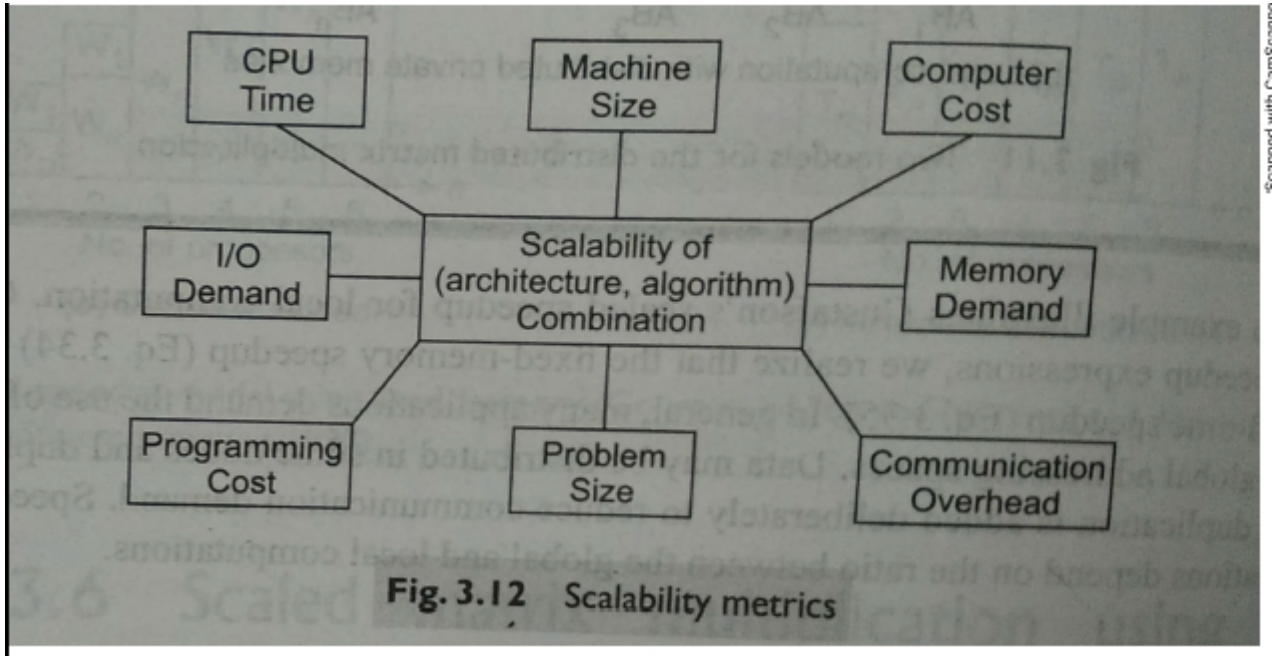


Fig. 3.12 Scalability metrics

Principles of scalable performance-Scalability metrics

2. **Clock rate (f)**
   - The clock rate determines the basic machine cycle, all components are driven by clock which can scale up with better technology
3. **Problem size (s)**
   - The amount of computational workload or the number of data points used to solve the given problem.
   - The problem size is directly proportional to the sequential execution time T(s, 1) for a uniprocessor system because each data point may demand one or more operation
4. **CPU Time (T)**
   - The actual CPU time (in sec) elapsed in executing a given program on parallel machine with a n processor collectively. This is the parallel execution time denoted as T(s, n) and is a function of both s and n
5. **I/O demand (d)**
   - The input/output demand in moving the program, data and results associated with a given application run. The I/O operations may overlap with the CPU operations in a multiprogrammed environment.
6. **Memory capacity(m)**
   - The amount of main memory (in bytes or words) used in a program execution. Note that the memory demand is affected by the problem size, the program size, the algorithm and the data structures used.
   - The memory demand varies dynamically during program execution. Here maximum number of memory words demanded. Virtual memory is almost unlimited with a 64 bit address space. It is the physical memory which may be limited in capacity.

Principles of scalable performance-Scalability metrics

7. Communication overhead (h)
   - The amount of time spent for interprocessor communication, synchronization, remote memory access etc.
   - This overhead also includes all noncompute operations which do not involve the CPU or I/O devices.
   - This overhead h(s, n) is a function of s and n and is not part of T(s, n). For a uniprocessor system the overhead h(s, n)=0
8. Computer cost ©
   - Total cost of hardware and software resources required to carry out the execution of a program

9. Programming overhead (p)
   - The development overhead associated with the application program.
   - Programming overhead may slowdown software productivity and thus implies a high cost.
   - Both computer and cost and programming cost are ignored in our scalability analysis.
   - Depending on computational objectives and resources constraints imposed one can fix some of the above parameters and optimize the remaining ones to achieve the highest performance with the lowest cost
   - The notion of scalability is tied to the notions of speedup and efficiency.
   - Scalability must able to express the effects of architecture's interconnection network, of the communication patterns inherent to algorithms of the physical constraints imposed by technology, and of the cost effectiveness or system efficiency.